Getting Started with cobrapy

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

```python
>>> import cobra.test
>>> model = cobra.test.create_test_model(cobra.test.salmonella_pickle)
```

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

```python
>>> print len(model.reactions)
2546
>>> print len(model.metabolites)
1802
>>> print len(model.genes)
1271
```

Just like a regular list, objects in the DictList can be retrieved by index. For example, to get the 30th reaction in the model:

```python
>>> print model.reactions[29]
2AGPA180tipp
```

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

```python
>>> atp_c = model.metabolites.get_by_id("atp_c")
>>> print atp_c.formula
C10H12N5O13P3
```

In the above example, atp_c is a Metabolite object, which has a formula attribute.

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:

```python
>>> model.reactions.EX_glc__D_e.lower_bound
0.0
```
The core classes of COBRApy are

- `cobra.core.Model.Model`
- `cobra.core.Reaction.Reaction`
- `cobra.core.Metabolite.Metabolite`

These objects can be used to create a model.

```python
# cobra/examples/01_create_model.py
# This simple example demonstrates how to create a model, create a reaction, and
# then add the reaction to the model.
# For this example, we’ll use the ‘3 oxoacyl acyl carrier protein synthase n C140’
# reaction from the STM_1.0 model which currently has the ID of ‘3OAS140’:
# 1.0 malACP[c] + 1.0 h[c] + 1.0 ddcaACP[c] -> 1.0 co2[c] + 1.0 ACP[c] + 1.0 3omrsACP[c]
# from cobra import Model, Reaction, Metabolite
cobra_model = Model('example_cobra_model')  # Best practise: SBML compliant IDs
reaction = Reaction('3OAS140')
reaction.name = '3 oxoacyl acyl carrier protein synthase n C140'
reaction.subsystem = 'Cell Envelope Biosynthesis'
reaction.lower_bound = 0.  # This is the default
reaction.upper_bound = 1000.  # This is the default
reaction.reversibility = 0  # This is the default
reaction.objective_coefficient = 0.  # this is the default

# Create the metabolites
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')
```

```python
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 can be added one at a time.
reaction.add_metabolites({
    'malACP_c': -1.0,
    'h_c': -1.0,
    'ddcaACP_c': -1.0,
    'co2_c': 1.0,
    'ACP_c': 1.0,
    'omrsACP_c': 1.0
})

# A string representation of the reaction
print reaction.reaction

# A boolean representation of the gene requirements for this reaction to be
gene_reaction_rule = '( STM2378 or STM1197 )'

# The next step will create cobra.Gene objects from the gene reaction rule,
# which will be used later by the cobra.Model to modulate reaction bounds after
# deleting genes.
reaction.add_gene_reaction_rule(gene_reaction_rule)

# The model is initially empty:
print '%i reactions in initial model' % len(cobra_model.reactions)
print '%i metabolites in initial model' % len(cobra_model.metabolites)
print '%i genes in initial model' % len(cobra_model.genes)

# Add the reaction to the model
cobra_model.add_reactions(reaction)

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

# Iterate through the the objects in the model
for x in cobra_model.reactions:
    print x.reaction
for x in cobra_model.metabolites:
    print 'x has formula %s' % (x, x.formula)
for x in cobra_model.genes:
    print 'gene %s is associated with reactions:' % x
    for y in x.get_reaction():
        print '\t%s' % y
# cobra/examples/02_read_simulate_write.py
#
# This example demonstrates reading in a COBRA SBML model xml file
# performing a simple optimization, and then saving the model as an SBML xml file.
#
# The model is from the Salmonella enterica Typhimurium LT2 publication in
# 2011 in BMC Sys Bio 5:8
#
# The simulated growth rate should be ~0.478.
#
from cobra.io.sbml import create_cobra_model_from_sbml_file
from cobra.io.sbml import write_cobra_model_to_sbml_file
from cobra.test import salmonella_sbml

sbml_out_file = 'salmonella.out.xml'
sbml_level = 2
sbml_version = 1  # Writing version 4 is not completely supported.

# Read in the sbml file.
cobra_model = create_cobra_model_from_sbml_file(salmonella_sbml, print_time=True)
# Run the optimization for the objective reaction and medium composition
# set in the file.
cobra_model.optimize(solver='glpk')  # 'gurobi' or 'cplex' are also supported

print '\nSimulated growth rate is %1.3f' % cobra_model.solution.f

# Save the model to an SBML file
print '\nConverting Model to SBML and saving as ' + sbml_out_file
write_cobra_model_to_sbml_file(cobra_model, sbml_out_file, sbml_level, sbml_version, print_time=True)
Single Gene Deletions

```python
# This example demonstrates a single gene deletion simulation
# Expected growth rates for the salmonella model after a deletions in LB medium

target_genes = ['STM4081', 'STM0247', 'STM3867', 'STM2952']
expected_growth_rates = {
    'STM4081': 2.41,
    'STM0247': 2.43,
    'STM3867': 1.87,
    'STM2952': 1.81
}
```

```
start_time = time()  # start timer

# Perform deletions for all genes in the list
rates, statuses, problems = single_deletion(cobra_model, target_genes)

```

```
total_time = time() - start_time  # stop timer

# print out results
passed_string = 'PASSED: %s simulation ($1.3f) == expectation ($1.2f)'
failed_string = 'FAILED: %s simulation ($1.3f) != expectation ($1.2f)'
```

```
for gene_locus, rate in rates.items():
    name = cobra_model.genes.get_by_id(gene_locus).name
    if statuses[gene_locus] != "optimal":
        print "deletion failed for %s (%s)" % (name, gene_locus)
        if abs(rate - expected_growth_rates[gene_locus]) > 0.01:
            print failed_string % (name, rate, expected_growth_rates[gene_locus])
        else:
```
print passed_string % (name, rate, expected_growth_rates[gene_locus])
print 'single deletion time: %f seconds' % (total_time)
# cobra/examples/04_change_objective_example.py
# This example changes the targeted objective for the optimization function.

```python
from cPickle import load
#Load in the example model file
from cobra.test import salmonella_pickle #This is the name of the test file
with open(salmonella_pickle) as in_file:
    cobra_model = load(in_file)

cobra_model.optimize()
print 'solution for old objective (should be approximately 0.32):'
print cobra_model.solution.f

my_objective = cobra_model.reactions.get_by_id('3OAS140')
print 'Changing objective to %s' % my_objective.id
cobra_model.optimize(new_objective=my_objective)
print 'solution for %s (should be approximately 2.5):' % my_objective.id
print cobra_model.solution.f
```

---

**Changing Objective Functions**

---

# cobra/examples/04_change_objective_example.py
# This example changes the targeted objective for the optimization function.

```python
from cPickle import load
#Load in the example model file
from cobra.test import salmonella_pickle #This is the name of the test file
with open(salmonella_pickle) as in_file:
    cobra_model = load(in_file)

cobra_model.optimize()
print 'solution for old objective (should be approximately 0.32):'
print cobra_model.solution.f

my_objective = cobra_model.reactions.get_by_id('3OAS140')
print 'Changing objective to %s' % my_objective.id
cobra_model.optimize(new_objective=my_objective)
print 'solution for %s (should be approximately 2.5):' % my_objective.id
print cobra_model.solution.f
```
Adding reactions

```python
# cobra/examples/05_create_model.py
# This example shows how to modify an existing cobra model and change the
# objective function. We'll use the '3 o xoacyl acyl carrier protein synthase n C140'
# reaction from the STM_1.0 model which currently has the ID of 'my_new_reaction':
# # 1.0 malACP[c] + 1.0 h[c] + 1.0 ddcaACP[c] -> 1.0 co2[c] + 1.0 ACP[c] + 1.0 3omrsACP[c]
# from cobra.flux_analysis import single_deletion
from cPickle import load, dump
from time import time
from cobra.manipulation import initialize_growth_medium
from cobra.test import salmonella_pickle #This is the name of the test file
with open(salmonella_pickle) as in_file:
    cobra_model = load(in_file)

from cobra import Reaction
reaction = Reaction('my_new_reaction')
reaction.name = '3 o xoacyl acyl carrier protein synthase n C140'
reaction.subsystem = 'Cell Envelope Biosynthesis'
reaction.lower_bound = 0. # This is the default
reaction.upper_bound = 1000. # This is the default
reaction.reversibility = 0 # This is the default
reaction.objective_coefficient = 0. # This is the default

# 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.

# Create the metabolites
from cobra import Metabolite
ACP_c = Metabolite('ACP_c', formula='C11H21N2O7PRS',
    name='acyl-carrier-protein', compartment='c')
3omrsACP_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')
```
# If the metabolites are already in Model.metabolites then it is permissible,
# but not necessary, to use them instead of creating new ones
ddcaACP_c = cobra_model.metabolites.get_by_id('ddcaACP_c')

# add the metabolites to the reaction:
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
})

# print the reaction to make sure it worked:
print(reaction.reaction)

print('%i reactions in original model' % len(cobra_model.reactions))
# Add the reaction to the model
cobra_model.add_reactions(reaction)
print('%i reaction in updated model' % len(cobra_model.reactions))

cobra_model.optimize()
print('solution for old objective (should be approximately 0.320):'
print(cobra_model.solution.f)

print('Changing objective to newly added reaction: %s' % reaction.id)
cobra_model.optimize(new_objective=reaction)
print('solution for %s (should be approximately 2.505):' % reaction.id)
print(cobra_model.solution.f)
CHAPTER 7

Mixed-Integer Linear Programming

# cobra/examples/06_ice_cream_milp.py
#
# Advanced user example showing how to set up and solve an MILP
#
from cobra import Model, Metabolite, Reaction

#solver = 'cplex'  #With libglpk-java there is an untraced memory bug.

cone_selling_price = 7.0
cone_production_cost = 3.0
popsicle_selling_price = 2.0
popsicle_production_cost = 1.0
starting_budget = 100.0

print('I can sell cones for $%.2f.'%cone_selling_price)
print('Cones cost me $%.2f to produce.'%cone_production_cost)
print('I can sell popsicles for $%.2f.'%popsicle_selling_price)
print('Popsicles cost me $%.2f to produce.'%popsicle_production_cost)
print('My total budget was capped at $%.2f today.'%starting_budget)

# problem is:
# max profit
# s.t.
# cone_production_cost*cone_production + popsicle_production_cost*popsicle_production <= starting_budget
# number of cones and popsicles has to be integer...

# first, we'll solve the continuous case just to make sure everything is
# working (it should only make cones)...then we'll tighten the constraints to
# integer... and it should make popsicles.

cobra_model = Model('MILP_implementation_test')
cone_out = Metabolite(id='cone_out', compartment='c')
cone_in = Metabolite(id='cone_in', compartment='c')
cone_consumed = Metabolite(id='cone_consumed', compartment='c')
popsicle_out = Metabolite(id='popsicle_out', compartment='c')
popsicle_in = Metabolite(id='popsicle_in', compartment='c')
popsicle_consumed = Metabolite(id='popsicle_consumed', compartment='c')
the_reactions = []

# SOURCE
Cone_source = Reaction(name='Cone_source')
temp_metabolite_dict = {cone_out: 1}
Cone_source.add_metabolites(temp_metabolite_dict)
the_reactions.append(Cone_source)

Popsicle_source = Reaction(name='Popsicle_source')
temp_metabolite_dict = {popsicle_out: 1}
Popsicle_source.add_metabolites(temp_metabolite_dict)
the_reactions.append(Popsicle_source)

## PRODUCTION
Cone_production = Reaction(name='Cone_production')
temp_metabolite_dict = {cone_out: -1,
                        cone_in: 1}
Cone_production.add_metabolites(temp_metabolite_dict)
the_reactions.append(Cone_production)

Popsicle_production = Reaction(name='Popsicle_production')
temp_metabolite_dict = {popsicle_out: -1,
                        popsicle_in: 1}
Popsicle_production.add_metabolites(temp_metabolite_dict)
the_reactions.append(Popsicle_production)

## CONSUMPTION
Cone_consumption = Reaction(name='Cone_consumption')
temp_metabolite_dict = {cone_in: -1,
                        cone_consumed: 1}
Cone_consumption.add_metabolites(temp_metabolite_dict)
the_reactions.append(Cone_consumption)

Popsicle_consumption = Reaction(name='Popsicle_consumption')
temp_metabolite_dict = {popsicle_in: -1,
                        popsicle_consumed: 1}
Popsicle_consumption.add_metabolites(temp_metabolite_dict)
the_reactions.append(Popsicle_consumption)

# SINK
Cone_consumed_sink = Reaction(name='Cone_consumed_sink')
temp_metabolite_dict = {cone_consumed: -1}
Cone_consumed_sink.add_metabolites(temp_metabolite_dict)
the_reactions.append(Cone_consumed_sink)

Popsicle_consumed_sink = Reaction(name='Popsicle_consumed_sink')
temp_metabolite_dict = {popsicle_consumed: -1}
Popsicle_consumed_sink.add_metabolites(temp_metabolite_dict)
the_reactions.append(Popsicle_consumed_sink)

## add all reactions
cobra_model.add_reactions(the_reactions)

# set objective coefficients
Cone_consumption.objective_coefficient = cone_selling_price
Popsicle_consumption.objective_coefficient = popsicle_selling_price
Cone_production.objective_coefficient = -1*cone_production_cost
Popsicle_production.objective_coefficient = -1*popsicle_production_cost

production_capacity_constraint = Metabolite(id='production_capacity_constraint')
production_capacity_constraint._constraint_sense = 'L'
production_capacity_constraint._bound = starting_budget;

Cone_production.add_metabolites({production_capacity_constraint: cone_production_cost })
Popsicle_production.add_metabolites({production_capacity_constraint: popsicle_production_cost })

print
print('Here is what happens in the continuous (LP) case...')

the_program = cobra_model.optimize(objective_sense='maximize')
print
print('Status is: %s' %cobra_model.solution.status)
print('Objective value is: %1.2f' %cobra_model.solution.f)

for the_reaction, the_value in cobra_model.solution.x_dict.items():
    print ' $s$: %1.2f' % (the_reaction, the_value)

print
print('Who wants 1/3 of a cone? Cones and popsicles are units aka integers, reformulate as MILP')
Cone_production.variable_kind = 'integer'
Popsicle_production.variable_kind = 'integer'

the_program = cobra_model.optimize(objective_sense='maximize')
print
print('Status is: %s' %cobra_model.solution.status)
print('Objective value is: %1.2f' %cobra_model.solution.f)

for the_reaction, the_value in cobra_model.solution.x_dict.items():
    print ' $s$: %1.2f' % (the_reaction, the_value)

print
print('We now make full items')
8.1 cobra Package

8.2 Subpackages

8.2.1 core Package

core Package

ArrayBasedModel Module

class cobra.core.ArrayBasedModel.ArrayBasedModel(description=None, copy_model=False)
    Bases: cobra.core.Model.Model

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

S [scipy.sparse.lil_matrix (in CPython)] Stoichiometric matrix of the model
lower_bounds
upper_bounds
objective_coefficients
b
constraint_sense

add_metabolites (metabolite_list, expand_stoichiometric_matrix=True)
    Will add a list of metabolites to the the object, if they do not exist and then expand the stoichiometric matrix
    metabolite_list: A list of Metabolite objects
    expand_stoichiometric_matrix: Boolean. If True and self.S is not None then it will add rows to self.S. self.S must be created after adding reactions and metabolites to self before it can be expanded. Trying to expand self.S when self only contains metabolites is ludacris.
add_reactions (reaction_list, update_matrices=False)
    Will add a cobra.Reaction object to the model, if reaction.id is not in self.reactions.

    reaction_list: A Reaction object or a list of them

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

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

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

    print_time: Boolean used for debugging

update ()
    Regenerates the stoichiometric matrix and vectors

DictList Module

class cobra.core.DictList.DictList (*args, **kwargs)

Bases: list

    A combined dict and list that feels like a list, but has the speed benefits of a dict. This may be eventually replaced
    by collections.OrderedDict.

    This was written to address the performance issues associated with searching, accessing, or iterating over a list
    in python that resulted in notable performance decays with COBRA for python.

append (object)

extend (iterable)

get_by_id (id)
    return the element with a matching id

index (id)
    id: A string or a Object

insert (index, object)

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

pop (*args, **kwargs)

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

    search_function: this will be used to select which objects to return This can be:
      • a string, in which case any object.attribute containing the string will be returned
      • a compiled regular expression
      • a boolean function which takes one argument and returns True for desired values

    attribute: the attribute to be searched for (default is 'id'). If this is None, the object itself is used.
    returns: a list of objects which match the query
**cobra Documentation, Release 0.2.1**

- `remove(*args, **kwargs)`
- `reverse(*args, **kwargs)`
- `sort(*args, **kwargs)`
- `union(iterable)`

  adds elements with id’s not already in the model

`cobra.core.DictList.get_id(object)`

return an id for the object

This allows the function to be generalize to non-cobra.core objects, however, this added function call slows things down.

### Formula Module

**class** `cobra.core.Formula.Formula(formula=None)`

**Bases:** `cobra.core.Object.Object`

Formula is a class for holding information regarding a Metabolite formula. This will replace the current way of dealing with molecular formulae in cobra.Metabolite objects

Legal formula string characters include letters, numbers, and `*`. If a formula string starts with a number then it is assumed that all element counts are multiplied by this number.

- `calculate_formula_weight(weight_dict=None)`
  
  Calculate the formula weight.

  weight_dict: None or a dictionary of elements and weights.

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

  TODO: Find a stable python package for parsing chemical formulas.

### Gene Module

**class** `cobra.core.Gene.Gene(id, formula=None, name=None, compartment=None, strand=’+’, locus_start=0, locus_end=0, functional=True)`

**Bases:** `cobra.core.Metabolite.Metabolite`

A Gene is a special class of metabolite.

TODO: Make design decisions about TUs and such

- `guided_copy(the_model)`
  
  Trying to make a faster copy procedure for cases where large numbers of metabolites might be copied. Such as when copying reactions.

- `remove_from_model(the_model, make_dependent_reactions_nonfunctional=True)`
  
  Removes the association

  **the_model:** Model object. Remove the reaction from this model.

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

  TODO: Better handling of the gene association

---

**8.2. Subpackages**
Metabolite Module

class cobra.core.Metabolite.Metabolite(id=None, formula=None, name=None, compartment=None)

Bases: cobra.core.Object.Object

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

copy()

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

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

This should be fixed with self.__deecopy__ if possible

get_model()

Returns the Model object that contain this Object

get_reaction()

Returns a list of Reactions that contain this Object

guided_copy(the_model)

Trying to make a faster copy procedure for cases where large numbers of metabolites might be copied. Such as when copying reactions.

parse_composition()

Breaks the chemical formula down by element. Useful for making sure Reactions are balanced.

remove_from_model(model=None, method='subtractive')

Removes the association

model: None or Model object. Remove the reaction from this model.

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

Model Module

class cobra.core.Model.Model(description=None)

Bases: cobra.core.Object.Object

Model is a class for analyzing metabolic models with the COBRA toolbox developed in the Palsson Lab at UCSD. Make all of the objects (Reaction, Metabolite, ...) to make OOP easier.

add_metabolites(metabolite_list)

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

metabolite_list: A list of Metabolite objects

add_reaction(reaction)

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

reaction: A Reaction object

add_reactions(reaction_list)

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

reaction_list: A Reaction object or a list of them

copy(print_time=False)

Provides a partial ‘deepcopy’ of the Model. All of the Metabolite, Gene, and Reaction objects are created anew but in a faster fashion than deepcopy
print_time: Boolean used for debugging

**optimize**

new_objective=None, objective_sense='maximize', the_problem=None, solver='glpk', error_reporting=None, quadratic_component=None, tolerance_optimality=1e-06, tolerance_feasibility=1e-06, tolerance_barrier=1e-10, **kwargs)

Optimize self for self._objective_coefficients or new_objective.

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

new_objective: Reaction, String, or Integer referring to a reaction in cobra_model.reactions to set as the objective. Currently, only supports single objective coefficients. Will expand to include mixed objectives.

objective_sense: ‘maximize’ or ‘minimize’

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

solver: ‘glpk’, ‘gurobi’, or ‘cplex’

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

quadratic_component: None or scipy.sparse.dok of dim(len(cobra_model.reactions),len(cobra_model.reactions))

If not None:

Solves quadratic programming problems for cobra_models of the form: cobra_model = ArrayBasedModel(cobra_model) minimize: 0.5 * x' * quadratic_component * x + cobra_model._objective_coefficients' * x such that,

cobra_model._lower_bounds <= x <= cobra_model._upper_bounds cobra_model._S * x (cobra_model._constraint_sense) cobra_model._b

#See cobra.flux_analysis.solvers for more info on the following parameters. Also, refer to your solver's manual

tolerance_optimality: Solver tolerance for optimality.

tolerance_feasibility: Solver tolerance for feasibility.

tolerance_barrier: Solver tolerance for barrier method

lp_method: Solver method to solve the problem

#End solver parameters

**kwargs: See additional parameters for your specific solver module in cobra.solvers

**remove_reactions**(the_reactions)

the_reactions: instance or list of cobra.Reactions or strings of self.reactions[:,:].id.

**to_array_based_model**(deepcopy_model=False)

Makes a ArrayBasedModel from a cobra.Model which may be used to perform linear algebra operations with the stoichiometric matrix.

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

**update**()

Non functional. Model.update is moved to ArrayBasedModel. Please use the to_array_based_model property to create an ArrayBasedModel.

**Object Module**

class cobra.core.Object.Object (id=None)

Bases: object

8.2. Subpackages
Guided_copy()  
Trying to make a faster copy procedure for cases where large numbers of metabolites might be copied.  
Such as when copying reactions.  
This function allows us to manipulate how specific attributes are copied.

Reaction Module

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

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

add_gene_reaction_rule (the_rule)
    This adds a gene reaction rule.
    
    the_rule: 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.
    
    Note that this method currently replaces any pre-existing rules

add_metabolites (the_metabolites, combine=True, add_to_container_model=True)
    Add metabolites and stoichiometric coefficients to the reaction. If the final coefficient for a metabolite is 0 then it is removed from the reaction.
    
    the_metabolites: A dict of cobra.Metabolites and their coefficient
    
    combine: Boolean. If True and a metabolite already exists in the reaction then the coefficients will be added. If False the old metabolite will be discarded and the new one added.
    
    add_to_container_model: Boolean. If True and this reaction is contained within a cobra.Model (i.e., self._model is a cobra.Model) then add the metabolite to the model.

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

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

copy ()
    When copying a reaction, it is necessary to deepcopy the components so the list references aren’t carried over.
    
    Additionally, a copy of a reaction is no longer in a cobra.Model.
    
    This should be fixed with self.__deecopy__ if possible

delete ()
    Removes all associations between a reaction and its container _model and metabolites and genes.
    
    TODO: Decide whether orphan metabolites should automatically be removed from the model.

get_coefficient (the_metabolite)
    Return the stoichiometric coefficient for a metabolite in the reaction.
    
    the_metabolite: A metabolite Id.

get_coefficients (the_metabolites)
    Return the stoichiometric coefficients for a list of metabolites in the reaction.
the_metabolites: A list of metabolite Ids.

get_compartments()

get_gene()
   Return a list of reactants for the reaction.

get_model()
   Returns the Model object that this Reaction is associated with.

get_products()
   Return a list of products for the reaction

get_reactants()
   Return a list of reactants for the reaction.

guided_copy (the_model, metabolite_dict, gene_dict=None)
   Trying to make a faster copy procedure for cases where large numbers of metabolites might be copied. Such as when copying reactions.

parse_gene_association (the_type='gene')
   Extract all genes from the Boolean Gene_Association string.
   #Formerly, update_names

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

   the_metabolite: A cobra.Metabolite that is in the reaction

print_values()
   Prints most of the contents of a reaction as a series of strings.

process_prefixed_reaction()
   Deal with reaction names that have a prefix.

   DEPRECATED This is necessary when parsing text files. It is better to get the reactions from SBML files.
   This can be moved to a tools section

reconstruct_reaction()
   Generate a human readable reaction string.

remove_from_model (model=None)
   Removes the association

   model: cobra.Model object. remove the reaction from this model.

subtract_metabolites (metabolites)
   This function will ‘subtract’ cobra.metabolites from a reaction, which means add the metabolites with -1*coefficient. If the final coefficient for a metabolite is 0 then the metabolite is removed from the reaction.

   Note: That a final coefficient < 0 implies a reactant.

   metabolites: dict of [Metabolite: coefficient] These metabolites will be added to the reaction

   Note: This function uses deepcopy in case the reaction is being subtracted from itself.
**Solution Module**

```python
class Solution:
    def __init__(self, the_f, x=None, x_dict=None, y=None, y_dict=None, the_solver=None, the_time=0, status='NA',):
        Bases: Object

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

**NOTE:** This class might be deprecated in favor of associating the values with the Reactions and Metabolites in the cobra.Model.

- **f:** The objective value
- **the_time:** Float. Sometimes indicates how long it took to solve a problem. As this is typically negligible and not used in cobra pie, it might be deprecated.
- **the_solver:** A string indicating which solver package was used.
- **x:** List or Array of the values from the primal.
- **x_dict:** A dictionary of reaction ids that maps to the primal values.
- **y:** List or Array of the values from the dual.
- **y_dict:** A dictionary of reaction ids that maps to the dual values.
```

### 8.2.2 external Package

**ppmap Module**

Very basic parallel processing support

Implements a work-alike of the builtin map() function that distributes work across many processes. As it uses Parallel Python to do the actual multi-processing, code using this must conform to the usual PP restrictions (arguments must be serializable, etc.)

```python
cobra.external.ppmap.ppmap(processes, function, sequence, *sequences)
```

Split the work of ‘function’ across the given number of processes. Set ‘processes’ to None to let Parallel Python autodetect the number of children to use.

Although the calling semantics should be identical to __builtin__.map (even using __builtin__.map to process arguments), it differs in that it returns a generator instead of a list. This enables lazy evaluation of the results so that other work can be done while the subprocesses are still running.

```python
>>> def rangetotal(n):
    return n, sum(range(n))
>>> list(map(rangetotal, range(1, 6)))
[(1, 0), (2, 1), (3, 3), (4, 6), (5, 10)]
>>> list(ppmap(1, rangetotal, range(1, 6)))
[(1, 0), (2, 1), (3, 3), (4, 6), (5, 10)]
```
8.2.3 flux_analysis Package

flux_analysis Package

double_deletion Module

cobra.flux_analysis.double_deletion(double_deletion)(cobra_model, element_list_1=None, element_list_2=None, method='fba', single_deletion_growth_dict=None, the_problem='return', element_type='gene', solver='glpk', error_reporting=None, number_of_processes=1)

Wrapper for double_gene_deletion and the currently unimplemented double_reaction_deletion functions

cobra_model: a cobra.Model object

element_list_1: Is None or a list of elements (genes or reactions)
element_list_2: Is None or a list of elements (genes or reactions)

method: ‘fba’ or ‘moma’ to run flux balance analysis or minimization of metabolic adjustments.
single_deletion_growth_dict: A dictionary that provides the growth rate information for single gene knock outs. This can speed up simulations because nonviable single deletion strains imply that all double deletion strains will also be nonviable.

the_problem: Is None or ‘reuse’
element_type: ‘gene’ or ‘reaction’
solver: ‘glpk’, ‘gurobi’, or ‘cplex’.

error_reporting: None or True

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

cobra.flux_analysis.double_deletion(double_gene_deletion)(cobra_model, gene_list_1=None, gene_list_2=None, method='fba', single_deletion_growth_dict=None, the_problem='return', solver='glpk', growth_tolerance=1e-08, error_reporting=None)

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

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

cobra_model: a cobra.Model object

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

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

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

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

solver: ‘glpk’, ‘gurobi’, or ‘cplex’.

error_reporting: None or True

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

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

**essentiality Module**

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

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

cobra_model: A Model object.

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

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

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

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

solver: ‘glpk’, ‘gurobi’, or ‘cplex’

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

returns: essentiality_dict: A dictionary providing the maximum growth rate accessible when the respective component is removed from the medium.
Performs single and/or double deletion analysis on all the genes in the model. Provides an interface to parallelize the deletion studies.

**moma Module**

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


wt_model: A cobra.Model object

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

objective_sense: ‘maximize’ or ‘minimize’

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

tolerance_optimality: Solver tolerance for optimality.

tolerance_feasibility: Solver tolerance for feasibility.

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

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

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

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

**objective Module**

cobra.flux_analysis.objective.assess_objective(cobra_model, the_objective=None, objective_cutoff=0.001, growth_medium=None)

Assesses the ability of the model to produce all reactants in the_objective on an individual basis. Returns True if the_objective can be realized to exceed objective_cutoff. Otherwise, determines which components of the_objective are lagging and returns a dict of the components and their required and realized values.

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

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

**single_deletion Module**

cobra.flux_analysis.single_deletion.single_deletion(cobra_model, element_list=None, method='fba', the_problem='return', element_type='gene', solver='glpk', error_reporting=None)

Wrapper for single_gene_deletion and the single_reaction_deletion functions

cobra_model: a cobra.Model object

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

method: ‘fba’ or ‘moma’

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

element_type: ‘gene’ or ‘reaction’

solver: ‘glpk’, ‘gurobi’, or ‘cplex’.

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

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

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

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

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

cobra_model: a cobra.Model object

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

method: ‘fba’ or ‘moma’

the_problem: Is None or ‘reuse’.

solver: ‘glpk’, ‘gurobi’, or ‘cplex’.

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

TODO: Add in a section that allows copying and collection of problem for debugging purposes.
cobra.flux_analysis.single_deletion.single_reaction_deletion(cobra_model, element_list=None, method='fba', the_problem='return', solver='glpk', error_reporting=None, discard_problems=True)

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

cobra_model: a cobra.Model object

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

method: ‘fba’ is the only option at the moment.

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

solver: ‘glpk’, ‘gurobi’, or ‘cplex’.

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

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

**variability Module**

cobra.flux_analysis.variability.find_blocked_reactions(cobra_model, the_reactions=None, allow_loops=True, solver='glpk', the_problem='return', tolerance_optimality=1e-09, open_exchanges=False)

Finds reactions that cannot carry a flux with the current exchange reaction settings for cobra_model, using flux variability analysis.
cobra.flux_analysis.variability.flux_variability_analysis(cobra_model, fraction_of_optimum=1.0, objective_sense='maximize', the_reactions=None, allow_loops=True, solver='glpk', the_problem='return', tolerance_optimality=1e-06, tolerance_feasibility=1e-06, tolerance_barrier=1e-08, lp_method=1, lp_parallel=0, new_objective=None, relax_b=None, error_reporting=None, number_of_processes=1, copy_model=True)

Runs flux variability analysis on a cobra.Model object

cobra_model: a Model object

fraction_of_optimum: fraction of the optimal solution that must be realized

the_reactions: list of reactions to run FVA on. if None then run on all reactions in the Model

allow_loops: Not Implemented. If false then run the simulations with the loop law method to remove loops.

the_problem: If ‘return’ or an LP model object for the specified solver then the optimizations will be sped up by attempting to use a previous solution as a starting point to optimize the current problem. Can reduce simulation time by over an order of magnitude.

solver: ‘glpk’, ‘gurobi’, or ‘cplex’.

the_problem: a problem object for the corresponding solver, ‘return’, or a float representing the wt_solution

number_of_processes: If greater than 1 then this function will attempt to parallelize the problem. NOTE: Currently not functional

returns a dictionary: {reaction.id: {‘maximum’: float, ‘minimum’: float}}

TODO: update how Metabolite._bound is handled so we can set a range instead of just a single value. This will be done in cobra.flux_analysis.solvers.

cobra.flux_analysis.variability.flux_variability_analysis_wrapper(keywords)

Provides an interface to call flux_variability_analysis from ppmap

cobra.flux_analysis.variability.run_fva(solver='cplex')

For debugging purposes only
8.2.4 io Package

io Package

mat Module

cobra.io.mat.load_matlab_model(infile_path, variable_name=None)
Load a cobra model stored as a .mat file. .. warning:: INCOMPLETE, does not load GPR's

infile_path : str
variable_name : str, optional

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

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

This .mat file can be used directly in the MATLAB version of COBRA. .. note:: This function works best with scipy 0.11b1 or later

model : cobra.Model
file_name : str or file-like object

sbml Module

cobra.io.sbml.add_sbml_species(sbml_model, cobra_metabolite, note_start_tag, note_end_tag, boundary_metabolite=False)
A helper function for adding cobra metabolites to an sbml model.

sbml_model: sbml_model object
cobra_metabolite: a cobra.Metabolite object
note_start_tag: the start tag for parsing cobra notes. this will eventually be supplanted when COBRA is worked into sbml.
note_end_tag: the end tag for parsing cobra notes. this will eventually be supplanted when COBRA is worked into sbml.

cobra.io.sbml.create_cobra_model_from_sbml_file(sbml_filename, old_sbml=False, legacy_metabolite=False, print_time=False, use_hyphens=False)
convert an SBML XML file into a cobra.Model object. Supports SBML Level 2 Versions 1 and 4

sbml_filename: String.
old_sbml: Boolean. Set to True if the XML file has metabolite formula appended to metabolite names. This was a poorly designed artifact that persists in some models.
legacy_metabolite: Boolean. If True then assume that the metabolite id has the compartment id appended after an underscore (e.g. _c for cytosol). This has not been implemented but will be soon.
print_time: Boolean. Print the time requirements for different sections
use_hyphens: Boolean. If True, double underscores (_) in an SBML ID will be converted to hyphens

cobra.io.sbml.fix_legacy_id(id, use_hyphens=False)
cobra.io.sbml.parse_legacy_id(the_id, the_compartment=None, the_type='metabolite', use_hyphens=False)
Deals with a bunch of problems due to bigg.ucsd.edu not following SBML standards
the_id: String.
the_compartment: String.
the_type: String. Currently only 'metabolite' is supported
use_hyphens: Boolean. If True, double underscores (__ ) in an SBML ID will be converted to hyphens

cobra.io.sbml.parse_legacy_sbml_notes(note_string, note_delimiter=':')
Deal with legacy SBML format issues arising from the COBRA Toolbox for MATLAB and BiGG.ucsd.edu developers.

cobra.io.sbml.read_legacy_sbml(filename, use_hyphens=False)
read in an sbml file and fix the sbml id’s

cobra.io.sbml.write_cobra_model_to_sbml_file(cobra_model, filename, use_hyphens=False)
Write a cobra.Model object to an SBML XML file.
cobra_model: A cobra.Model object
filename: The file to write the SBML XML to.
sbm_level: 2 is the only level supported at the moment.
sbm_version: 1 is the only version supported at the moment.
print_time: Boolean. Print the time requirements for different sections

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

8.2.5 manipulation Package

manipulation Package

delete Module

cobra.manipulation.delete.delete_model_genes(cobra_model, gene_list, cumulative_deletions=False, disable_orphans=False)
delete_model_genes will set the upper and lower bounds for reactions catalyzed 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.
TODO: Rewrite to use dicts for _trimmed*
TODO: All this will be replaced by Boolean logic associated with #the cobra.Gene.functional in cobra.Reaction.gene_reaction_rule
TODO: Update this to refer to cobra.(Gene|Reaction) in the _trimmed_(genes|reactions) fields and remove _trimmed_indices

cobra.manipulation.delete.prune_unused_metabolites(cobra_model)
Removes metabolites that aren’t involved in any reactions in the model
cobra_model: A Model object.

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

cobra.manipulation.delete.undelete_model_genes(cobra_model)

Undoes the effects of a call to delete_model_genes. Modifies cobra_model in place.

cobra_model: A cobra.Model object

modify Module

cobra.manipulation.modify.convert_rule_to_boolean_rule(cobra_model, the_rule, return_gene_indices=False, index_offset=0)

Used to convert a text based gpr to an index based gpr. This will also update the cobra_model.

the_rule: A COBRA 2.0 compliant GPR string
return_gene_indices: Boolean return the indices for the genes
index_offset: Integer. Set to 1 if the rules need to be generated for base 1 software like MATLAB.

TODO: Test now that cobra.Gene is in use DEPRECATED: This should be moved to the mlab module

cobra.manipulation.modify.convert_to_irreversible(cobra_model)

Will break all of the reversible reactions into two separate irreversible reactions with different directions. Useful for some modeling problems.

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

TODO: Can we just use a -1*guided_copy or something else?

cobra.manipulation.modify.initialize_growth_medium(cobra_model, the_medium='MgM', external_boundary_compartment='e', external_boundary_reactions=None, reaction_lower_bound=0.0, reaction_upper_bound=1000.0, irreversible=False, reactions_to_disable=None)

Sets all of the input fluxes to the model to zero and then will initialize the input fluxes to the values specified in the_medium if it is a dict or will see if the model has a composition dict and use that to do the initialization.

cobra_model: A cobra.Model object.

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

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

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

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

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

irreversible: Boolean. If the model is irreversible then the medium composition is taken as the upper bound
reactions_to_disable: List of reactions for which the upper and lower bounds are disabled. This is superceded by the contents of media_composition

cobra.manipulation.modify.revert_to_reversible(cobra_model)

This function will convert a reversible model made by convert_to_irreversible into a reversible model.

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

NOTE: It might just be easiest to include this function in the Reaction class

### 8.2.6 mlab Package

**mlab Package**

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

To use MATLAB functions directly, see cobra.matlab

**mlab Module**

cobra.mlab.mlab.cobra_model_object_to_cobra_matlab_struct(cobra_model)

This function converts all of the object values to the corresponding model fields to update the mlab model.

cobra.mlab.mlab/init_matlab_toolbox(matlab_cobra_path=None, discover_functions=True)

initialize the matlab cobra toolbox, and load its functions into mlab’s namespace (very useful for ipython tab completion)

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

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

cobra.mlab.mlab.matlab_cell_to_python_list(the_cell)

cobra.mlab.mlab.matlab_cobra_struct_to_python_cobra_object(matlab_struct)

Converts a COBRA toolbox 2.0 struct into a cobra.Model object using the mlabwrap matlab proxy

cobra.mlab.mlab.matlab_logical_to_python_logical(the_logical)

cobra.mlab.mlab.matlab_sparse_to_numpy_array(matlab_sparse_matrix)

cobra.mlab.mlab.matlab_sparse_to_scipy_sparse(matlab_sparse_matrix)

cobra.mlab.mlab.numpy_array_to_mlab_sparse(numpy_array)

A more efficient method is needed for when the matrix is so big that making a dense version is a waste of computer effort.

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

cobra.mlab.mlab/scipy_sparse_to_mlab_sparse(scipy_sparse_matrix)

A more efficient method is needed for when the matrix is so big that making a dense version is a waste of computer effort.
8.2.7 solvers Package

solvers Package

cobra.solvers.optimize(cobra_model, solver='glpk', error_reporting=True, **kwargs)
   Wrapper to optimization solvers

cplex_solver Module

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

cobra.solvers.cplex_solver.format_solution(lp, cobra_model, **kwargs)
cobra.solvers.cplex_solver.get_objective_value(lp)
cobra.solvers.cplex_solver.get_status(lp)
cobra.solvers.cplex_solver.set_parameter(lp, parameter_name, parameter_value)
cobra.solvers.cplex_solver.solve(cobra_model, **kwargs)
cobra.solvers.cplex_solver.solve_problem(lp, **kwargs)
   A performance tunable method for solving a problem

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

lp: A gurobi problem object
cobra_model: the cobra.Model corresponding to ‘lp’

glpk_solver Module

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

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

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

cobra_model: a cobra.Model

returns a dict: {'the_problem': solver specific object, 'the_solution': cobra.Solution for the optimization problem'
cobra.solvers.glpk_solver.solve_problem(lp, **kwars)

A performance tunable method for updating a model problem file

lp: a pyGLPK 0.3 problem

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

cobra.solvers.glpk_solver.update_problem(lp, cobra_model, **kwars)

A performance tunable method for updating a model problem file

lp: A gurobi problem object

cobra_model: the cobra.Model corresponding to ‘lp’

gurobi_solver Module

cobra.solvers.gurobi_solver.create_problem(cobra_model, **kwars)

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

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

cobra.solvers.gurobi_solver.get_objective_value(lp)

cobra.solvers.gurobi_solver.get_status(lp)

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

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

A performance tunable method for updating a model problem file

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

A performance tunable method for updating a model problem file

lp: A gurobi problem object

cobra_model: the cobra.Model corresponding to ‘lp’

legacy Module

cobra.solvers.legacy.test_solvers()

parameters Module

8.2.8 stats Package

stats Package

stats Module

cobra.stats.stats.combine_p_values(the_p_values, method='z', default_quantile=7.0)

Combines p-values from repeat measurements into a single p-value.

the_p_values: a list of p-values.
method: String. ‘z’|’fisher’. ‘z’ for using the weighted z-score. ‘fisher’ for using fisher’s combined probability test.

default_quantile: Float. Only used for z method. The quantile to use when the software’s normal inverse cdf(p-value) is infinite

cobra.stats.stats.error_weighted(the_means, the_stds)
Calculate the error-weighted mean and standard deviation.

the_means: A list or numpy array of floats.
the_stds: A list or numpy array of floats.

cobra.stats.stats.p_adjust(the_p_values, correction_method='bh')
Adjusts the p-values in a list for multiple hypothesis testing.

the_p_values: a list of p-values

NOTE: ‘bh’ and ‘by’ require R and rpy2 installed

8.2.9 test Package

test Package

cobra.test.create_test_model(test_pickle=’/var/build/user_builds/cobrapy/checkouts/0.2.1/cobra/test/data/salmonella.pickle’)
Returns a cobra model for testing. The default model is the up to date version of the Salmonella enterica Typhimurium LT2 model published in Thiele et al. 2011 BMC Sys Bio 5:8

test_pickle: The complete file name of a pickled cobra.Model or SBML XML file to be read. We currently provide Salmonella enterica Typhimurium and Escherichia coli models whose paths are stored in cobra.test.salmonella_pickle and cobra.test.ecoli_pickle, respectively. The ecoli model is a variant of the model published in Orth et al. 2011 Mol Syst Biol 7:535

cobra.test.create_test_suite()
create a unittest.TestSuite with available tests

cobra.test.test_all()
###running unit tests on cobra py###

flux_analysis Module

class cobra.test.flux_analysis.TestCobraFluxAnalysis(methodName=’runTest’)
Bases: unittest.case.TestCase
Test the simulation functions in cobra.flux_analysis

TODO: Add in tests for: MOMA

setUp()

test_double_deletion()

test_flux_variability()

test_single_deletion()
cobra.test.flux_analysis.test_all()
parallel Module

Future test suite for parallel features

solvers Module

class cobra.test.solvers.TestCobraSolver (methodName='runTest')
    Bases: unittest.case.TestCase

    setUp()

    test_cplex_attributes()
    test_cplex_creation()
    test_cplex_independent_creation()
    test_cplex_set_objective_sense()
    test_cplex_solve_feasible()
    test_cplex_solve_infeasible()
    test_cplex_solve_minimize()
    test_cplex_solve_mip()

    test_glpk_attributes()
    test_glpk_creation()
    test_glpk_independent_creation()
    test_glpk_set_objective_sense()
    test_glpk_solve_feasible()
    test_glpk_solve_infeasible()
    test_glpk_solve_minimize()
    test_glpk_solve_mip()

    test_gurobi_attributes()
    test_gurobi_creation()
    test_gurobi_independent_creation()
    test_gurobi_set_objective_sense()
    test_gurobi_solve_feasible()
    test_gurobi_solve_infeasible()
    test_gurobi_solve_minimize()
    test_gurobi_solve_mip()

cobra.test.solvers.add_legacy_test (TestCobraSolver, solver_name, solver_function)
    Creates a test set for each of the installed solvers using the legacy interface.

cobra.test.solvers.add_new_test (TestCobraSolver, solver_name, solver)
    Creates a test set for each of the solvers that are installed using the modular interface.

cobra.test.solvers.add_test (TestCobraSolver, solver_name, solver)
    Creates a test set for each of the solvers that are installed using the modular interface.
cobra.test.solvers.test_all()

**unit_tests Module**

class cobra.test.unit_tests.CobraTestCase (methodName='runTest')
    Bases: unittest.case.TestCase
    setUp()

class cobra.test.unit_tests.TestCobraCore (methodName='runTest')
    Bases: cobra.test.unit_tests.CobraTestCase
    test core cobra functions
    test_add_reaction()
        test adding and deleting reactions
    test_copy()
        modifying copy should not modify the original

class cobra.test.unit_tests.TestCobraIO (methodName='runTest')
    Bases: cobra.test.unit_tests.CobraTestCase
    save_matlab_model (model, file_name)
        Save the cobra model as a .mat file.
        This .mat file can be used directly in the MATLAB version of COBRA. .. note:: This function works best with scipy 0.11b1 or later
        model : cobra.Model file_name : str or file-like object
    test_mat_read_write()
    test_sbml_read()
    test_sbml_write()

class cobra.test.unit_tests.TestDictList (methodName='runTest')
    Bases: unittest.case.TestCase
    setUp()
    testAdd()
    testAppend()
    testDeepcopy()
    testExtend()
    testIadd()
8.2.10 topology Package

topology Package

reporter_metabolites Module

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

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

cobra_model: A cobra.Model object

tODO: CHANGE TO USING DICTIONARIES for the_reactions: the_scores

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

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

number_of_layers: 1 is the only option supported

scoring_metric: default means divide by k**0.5

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

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

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

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

cobra.topology.reporter_metabolites.ppmap_identify_reporter_metabolites (keywords)

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

• genindex
• modindex
• search
Python Module Index

C
  cobra.__init__,
  cobra.core,
  cobra.core.ArrayBasedModel,
  cobra.core.DictList,
  cobra.core.Formula,
  cobra.core.Gene,
  cobra.core.DictBasedModel,
  cobra.core.Model,
  cobra.core.Object,
  cobra.core.Reaction,
  cobra.core.Solution,
  cobra.external.ppmap,
  cobra.flux_analysis,
  cobra.flux_analysis.double_deletion,
  cobra.flux_analysis.essentiality,
  cobra.flux_analysis.moma,
  cobra.flux_analysis.objective,
  cobra.flux_analysis.single_deletion,
  cobra.flux_analysis.variability,
  cobra.io,
  cobra.io.mat,
  cobra.io.sbml,
  cobra.manipulation,
  cobra.manipulation.delete,
  cobra.manipulation.modify,
  cobra.mlab,
  cobra.mlab.mlab,
  cobra.solvers,
  cobra.solvers.cplex_solver,
  cobra.solvers.glpk_solver,
  cobra.solvers.gurobi_solver,
  cobra.solvers.legacy,
  cobra.solvers.parameters,
  cobra.stats,
  cobra.stats.stats,
  cobra.test,
  cobra.test.flux_analysis,
  cobra.test.parallel,
  cobra.test.solvers,
  cobra.test.unit_tests,
  cobra.topology,
  cobra.topology.reporter_metabolites,