ETFL Documentation

Release 0.1

ETFL Team

Jul 11, 2022

CONTENTS:

ETFL is a framework to account for expression, resource allocation, and thermodynamic constraints on stoichiometric models.

You can have a look at our preprint[1] on BiorXiv for more information on the formulation and results on an *E. coli* model.

[1] Salvy, Pierre, and Vassily Hatzimanikatis. "ETFL: A formulation for flux balance models accounting for expression, thermodynamics, and resource allocation constraints." bioRxiv (2019): 590992.

CHAPTER

ONE

QUICK START

These tutorial files detail typical usages of the ETFL package. They can be found at:

```
etfl
└── tutorials
    ├── benchmark_models.py
    ├── paper_iJO1366_models.py
    └── blah.py
```

blah.py details how to blah with *Escherichia coli*.

We use optlang.

We recommend you to get a commercial solver, as it has been seen that GLPK's lack of parallelism significantly increases solving time

Cheers,

The ETFL team

CHAPTER

TWO

SOP FOR CREATING AN ETFL MODEL

2.1 Checklist

Here is a summarized checklist of the material needed to turn a COBRA model into ETFL:

- A working installation of ETFL
- A Cobra model with:
  - Gene identifiers (IDs),
  - All nucleotides triphosphates(NTPs), deoxynucleotides triphosphate(dNTP), nucleotides diphosphate (NMP), aminoacids.
- (Optional) Gene reaction rules
- Gene sequences indexed by their gene IDs
- Peptide stoichiometry of enzymes
- Enzyme assigments per reaction.
- Enzyme catalytic rate constants:
  - Forward
  - (Optional) Reverse
- Enzyme degradation rate constants
- mRNA degradation rate constants
- (Optional) Free ribosomes ratio
- (Optional) Free RNA Polymerase ratio
- (Optional) GC-content and length of the genome
- (Optional) Average aminoacid abundances
- (Optional) Average NTP abundances
- (Optional) Average mRNA length
- (Optional) Avergae peptide length
- (Optional) Growth-dependant mRNA, peptide, and DNA mass ratios.

2.2 Setup

Prerequisites

Make sure you have Git installed. Since ETFL is built upon pyTFA[1] we will clone both repositories. In a folder of your choice, download the source code from our repositories:

```
git clone https://github.com/EPFL-LCSB/pytfa
git clone https://github.com/EPFL-LCSB/etfl
# -- OR --
git clone https://gitlab.com/EPFL-LCSB/pytfa
git clone ttps://gitlab.com/EPFL-LCSB/etfl
```

Docker container (recommended)

We recommend the use of Docker containers as they provide a standardized, controlled and reproducible environment. The ETFL Docker is built upon the pyTFA Docker image. We recommend building it yourself as it is where your solvers can be installed.

Downloading Docker

If Docker is not yet installed on your machine, you can get it from [here]

Building and running the Docker container 37

```
# Build the pyTFA docker
cd pytfa/docker && . build
# Build and run the ETFL docker
cd ../../etfl/docker
. build
. run
```

Solvers

For installing the solvers, please refer to the pyTFA documentation

Python environment

Alternatively, you can install ETFL using either:

```
pip install -r etfl/requirements.txt
# -- OR --
pip install -e etfl
```

Make sure your solvers are also installed in the same environment if you are using a *virtualenv* or *pyenv*.

[1] Salvy P, Fengos G, Ataman M, Pathier T, Soh KC, Hatzimanikatis V. pyTFA and matTFA: A Python package and a Matlab toolbox for Thermodynamics-based Flux Analysis [Journal Article]. Bioinformatics. 2018;.

2.3 From COBRA to ETFL

ETFL models can be generated fairly easily from a COBRA model. In the following subsections, we detail the required information to add expression constraints to a COBRA model and turn it into an ETFL model.

Constraint-based model

You will need to start with a COBRA model including the following information:

- Genes and their gene ID (necessary to retrieve gene sequences)
- (Optional) Gene-protein rules: These are used to make approximated enzymes if peptide information is not enough

Additionally, you will need to build a dictionary of essential metabolites required in the model. It should follow this example structure (all fields mandatory):

```
dict(atp='atp_c', adp='adp_c', amp='amp_c', gtp='gtp_c',
     gdp='gdp_c', pi ='pi_c' , ppi='ppi_c', h2o='h2o_c', h ='h_c' )
```

A dictionary of RNA NTPs, DNA dNTPS, and aminoacids is also required, of the type:

```
aa_dict = {
    'A': 'ala L_c',
    # ...
    'V': 'val L_c', }
rna_nucleotides = {
    'u': 'utp_c',
    # ...
    'c': 'ctp_c'}

rna_nucleotides_mp = {
    'u': 'ump_c',
    # ...
    'c': 'cmp_c'}

dna_nucleotides = {
    't': 'dttp\_c',
    # ...
    'c': 'dctp\_c'}
```

From genes to peptides

In order to build the transcription and translation, it is necessary to provide ETFL with gene deoxynucleotide sequences. These will be automatically transcribed in RNA sequences and then translated into aminoacid peptide sequences. They must be fed to the function `model.add_nucleotides_sequences` in a dict-like object, indexed by gene IDs (model.genes.mygene.id property in COBRA).

We suggest the following sources for obtaining such information:

- KEGG Genes
- NCBI Gene DB
- MetaCyc Gene Search

ETFL will automatically synthesize the correct peptides from the nucleotides sequences. This is based on the Biopython package's `transcribe` and `translate` functions[2]

[2] Dalke A, Wilczynski B, Chapman BA, Cox CJ, Kauff F, Friedberg I, et al. Biopython: freely available Python tools for computational molecular

For each enzyme created by transcription, a degradation rate constant must be specified. These can be obtained through literature search, or using an average value.

From peptides to enzymes

A key part of the expression modeling is to properly represent the assembly of enzymes from peptides. For each enzyme of the model, a stoichiometry of the peptides necessary for its assembly is needed. These are stored as dictionnaries in the `Enzyme.composition` property under a form similar to :

```
>>> enzyme.composition
{'b2868': 1, 'b2866': 1, 'b2867': 1}
```

The keys match the IDs of genes coding for the peptide, and the value represent the stoichiometry of the peptide in the enzyme. These can be obtained from litterature search or specialized databases. In particular, we used for the paper the Metacyc/Biocyc database[3][4] using specialised SmartTables queries[5]

```
html-sort-ascending( html-table-headers (
[(f,genes,(protein-to-components f)):
f<-ECOLI^^Protein-Complexes,genes := (enzyme-to-genes f)
],
("Product Name", "Genes", "Component coefficients")),
1)
```

From enzymes back to the metabolism

Lastly, the enzymes must be assigned reactions and catalytic rate constants. Several enzymes can catalyze the same reactions. COBRA models can take this into account differently, usually having either (i) multiple reactions with a simple gene reaction rule; or (ii) one unique reaction with several isozymes in the gene reaction rule. Although not often applied consistently within the same model, these two formalisms are equivalent, and their ETFL counterparts will also behave equivalently.

For each enzyme, the information needed is the (forward) catalytic rate constant k_{cat}^{+} , facultatively the reverse catalytic rate constant k_{cat}^{-} (set equal to k_{cat}^{+} if none is provided), and a degradation rate constant.

This is done by calling the function `model.add_enzymatic_coupling(coupling_dict)` where `coupling_dict` is a dict-like object with reaction IDs as keys and a list of enzyme objects as values:

```
coupling_dict = {
    #...
    'AB6PGH': [ <Enzyme AB6PGH_G495_MONOMER at 0x7ff00e0f1b38>],
    'ABTA' :  [ <Enzyme ABTA_GABATRANSAM at 0x7ff00e0fda90>,
                <Enzyme ABTA_G6646 at 0x7ff00e0fd4e0>],
    'ACALD' : [ <Enzyme ACALD_MHPF at 0x7ff00e0fdcf8>],
    #...
    }
```

The catalytic rate constants can be obtained from several databases, such as:

- Rhea
- BRENDA
- SabioRK

biology and bioinformatics. Bioinformatics. 2009 03;25(11):1422–1423. Available from: https://dx.doi.org/10.1093/bioinformatics/btp163.

[3] Caspi R, Foerster H, Fulcher CA, Kaipa P, Krummenacker M, Latendresse M, et al. The MetaCyc Database of metabolic pathways and enzymes and the BioCyc collection of Pathway/Genome Databases. Nucleic acids research. 2007;36(suppl 1):D623–D631.

[4] Keseler IM, Collado-Vides J, Gama-Castro S, Ingraham J, Paley S, Paulsen IT, et al. EcoCyc: a comprehensive database resource for Escherichia coli. Nucleic acids research. 2005;33(suppl 1):D334–D337.

[5] Travers M, Paley SM, Shrager J, Holland TA, Karp PD. Groups: knowledge spreadsheets for symbolic biocomputing. Database. 2013;2013.

- Uniprot

Several enzymes can be assigned to a reaction. ETFL will try to match the gene reaction rule isozymes to the supplied enzymes. If the gene reaction rule shows several isozymes while only one enzyme is supplied, the enzyme can be replicated to match the number of isozymes in the gene reaction rule.

Given a reaction in the model, if no enzyme is supplied but the reaction possesses a gene reaction rule, it is possible to infer an enzyme from it. The rule expression is expanded, and each term seprated a by an OR boolean operator is interpreted as an isozyme, while terms separated by an AND boolean operators are interpreted as unit peptide stoichiometric requirements. The enzyme is then assigned an average catalytic rate constant and degradation rate constant.

Growth-dependant parameters

Accounting for growth-dependent RNA and protein content requires additional information. In particular:

- GC-content and length of the genome
- Average aminoacid abundances
- Average NTP abundances
- Average mRNA length
- Average peptide length
- Growth-dependant mRNA, peptide, and DNA mass ratios.

These values are usually obtained through litterature search. All of the last three ratios are optional, although using none defeats the purpose of accounting for growth-dependant parameters.

2.4 Additional documentation

Example

We encourage the reader to look at the script used to generate the models with which the paper's results were generated, available in `etfl/tutorials/helper_gen_models.py`. The data it takes in input has been generated in `etfl/etfl/data/ecoli.py`. These are good examples to start from in order to make a custom ETFL from a different COBRA model.

2.5 Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 722287.

2.6 References

CHAPTER

THREE

API REFERENCE

This page contains auto-generated API reference documentation[1].

3.1 etfl

3.1.1 Subpackages

etfl.analysis

Submodules

etfl.analysis.dynamic

ME-related Reaction subclasses and methods definition

Module Contents

Classes

EnzymeDeltaRHS	E(t+dt) - E(t) <= f(dt, E(t), mu)
mRNADeltaRHS	F(t+dt) - F(t) <= f(dt, F(t), mu)

[1] Created with sphinx-autoapi

Functions

add_enzyme_ref_variable(dmodel)	
add_mRNA_ref_variable(dmodel)	
add_enzyme_rhs_variable(dmodel)	
add_mRNA_rhs_variable(dmodel)	
get_mu_times_var(dmodel, macromolecule)	
add_enzyme_delta_constraint(dmodel, timestep, degradation, synthesis)	Adds the constraint
add_mRNA_delta_constraint(dmodel, timestep, degradation, synthesis)	Adds the constraint
add_dynamic_variables_constraints(dmodel, timestep, dynamic_constraints)	
apply_ref_state(dmodel, solution, timestep, has_mrna, has_enzymes, mode='backward')	**param dmodel**
update_sol(t, X, S, dmodel, obs_values, colname)	
update_medium(t, Xi, Si, dmodel, medium_fun, timestep)	
compute_center(dmodel, objective, provided_solution=None, revert_changes=True)	Fixes growth to be above computed lower bound, finds chebyshev center,
show_initial_solution(model, solution)	
run_dynamic_etfl(model, timestep, tfinal, uptake_fun, medium_fun, uptake_enz, S0, X0, step_fun=None, inplace=False, initial_solution=None, chebyshev_bigm=BIGM, chebyshev_variables=None, chebyshev_exclude=None, chebyshev_include=None, dynamic_constraints=DEFAULT_DYNAMIC_CONS, mode='backward')	**param model** the model to simulate
wrap_time_sol(var_solutions, obs_values)	

Attributes

mrna_length_avg
DEFAULT_DYNAMIC_CONS
BIGM

ETFL.**mrna_length_avg = 370**

ETFL.**DEFAULT_DYNAMIC_CONS**

class ETFL.**EnzymeDeltaRHS**

Bases: etfl.optim.variables.EnzymeVariable

E(t+dt) - E(t) <= f(dt, E(t), mu) E(t+dt) <= E(t) + f(dt, E(t), mu) E(t+dt) <= ERHS(t, E(t), mu)

prefix = ERHS_

class ETFL.**mRNADeltaRHS**

Bases: etfl.optim.variables.mRNAVariable

F(t+dt) - F(t) <= f(dt, F(t), mu) F(t+dt) <= F(t) + f(dt, F(t), mu) F(t+dt) <= FRHS(t, F(t), mu)

prefix = FRHS_

ETFL.**add_enzyme_ref_variable**(*dmodel*)

ETFL.**add_mRNA_ref_variable**(*dmodel*)

ETFL.**add_enzyme_rhs_variable**(*dmodel*)

ETFL.**add_mRNA_rhs_variable**(*dmodel*)

ETFL.**get_mu_times_var**(*dmodel*, *macromolecule*)

ETFL.**add_enzyme_delta_constraint**(*dmodel*, *timestep*, *degradation*, *synthesis*)

Adds the constraint

E-Eref <= t*v_assembly_max E-Eref-t*v_assembly_max <= 0

Parameters

- **dmodel** –
- **timestep** –

Returns

ETFL.**add_mRNA_delta_constraint**(*dmodel*, *timestep*, *degradation*, *synthesis*)

Adds the constraint

Parameters

- **dmodel** –
- **timestep** –

Returns

ETFL.**add_dynamic_variables_constraints**(*dmodel*, *timestep*, *dynamic_constraints*)

ETFL.**apply_ref_state**(*dmodel*, *solution*, *timestep*, *has_mrna*, *has_enzymes*, *mode='backward'*)

Parameters

- **dmodel** –
- **solution** –
- **timestep** –
- **has_mrna** –
- **has_enzymes** –

- **mode** – `forward' or `backward' for the integration scheme

Returns

ETFL.**update_sol**(*t*, *X*, *S*, *dmodel*, *obs_values*, *colname*)

ETFL.**update_medium**(*t*, *Xi*, *Si*, *dmodel*, *medium_fun*, *timestep*)

ETFL.**compute_center**(*dmodel*, *objective*, *provided_solution=None*, *revert_changes=True*)

Fixes growth to be above computed lower bound, finds chebyshev center, resets the model, returns solution data

Parameters

- **dmodel** –
- **objective** – the radius to maximize

Returns

ETFL.**show_initial_solution**(*model*, *solution*)

ETFL.**BIGM = 1000**

ETFL.**run_dynamic_etfl**(*model*, *timestep*, *tfinal*, *uptake_fun*, *medium_fun*, *uptake_enz*, *S0*, *X0*, *step_fun=None*, *inplace=False*, *initial_solution=None*, *chebyshev_bigm=BIGM*, *chebyshev_variables=None*, *chebyshev_exclude=None*, *chebyshev_include=None*, *dynamic_constraints=DEFAULT_DYNAMIC_CONS*, *mode='backward'*)

Parameters

- **model** – the model to simulate
- **timestep** – the time between each step of the integration
- **tfinal** – The stopping time
- **uptake_fun** – Functions that regulate the uptakes (Michaelis Menten etc.)
- **medium_fun** – Functions that regulates the medium concentrations (switches, bubbling diffusion, etc…)
- **uptake_enz** – If specified, will use the enzyme kcats for the uptake functions
- **S0** – Initial concentrations
- **X0** – Initial amount of cells
- **step_fun** – Function for additional operations on the model at each time step (extra kinetics, etc …)
- **inplace** –
- **initial_solution** – Used for setting growth rate lower bound
- **chebyshev_bigm** –
- **chebyshev_variables** –
- **chebyshev_exclude** –
- **chebyshev_include** –
- **dynamic_constraints** –
- **mode** – `forward' or `backward' for the Euler integration scheme

Returns

ETFL.**wrap_time_sol**(*var_solutions*, *obs_values*)

etfl.analysis.summary

Summarizes quantities in models

Module Contents

Functions

get_amino_acid_consumption(model, solution=None, trna_reaction_prefix='trna_ch_')	
get_ntp_consumption(model, solution=None)	
check_solution(model, solution)	
print_standard_sol(model, solution=None, flux_dict=None)	
_print_dict_items_vars(solution, the_dict, width)	
_print_dict_items_fluxes(solution, the_dict, width)	

ETFL.**get_amino_acid_consumption**(*model*, *solution=None*, *trna_reaction_prefix='trna_ch_'*)

ETFL.**get_ntp_consumption**(*model*, *solution=None*)

ETFL.**check_solution**(*model*, *solution*)

ETFL.**print_standard_sol**(*model*, *solution=None*, *flux_dict=None*)

ETFL.**_print_dict_items_vars**(*solution*, *the_dict*, *width*)

ETFL.**_print_dict_items_fluxes**(*solution*, *the_dict*, *width*)

etfl.analysis.utils

Analysis utilities

Module Contents

Functions

enzymes_to_peptides_conc(model, enzyme_conc)	**param enzyme_conc** dict or pandas.Series, with the key/index being enzyme

ETFL.**enzymes_to_peptides_conc**(*model*, *enzyme_conc*)

Parameters

enzyme_conc – dict or pandas.Series, with the key/index being enzyme variable names, and the value their concentration

Returns

etfl.core

Submodules

etfl.core.allocation

Core for the ME-part

Module Contents

Functions

fix_prot_ratio(model, mass_ratios)	To keep consistency between FBA and ETFL biomass compositions, we divide biomass
fix_RNA_ratio(model, mass_ratios)	To keep consistency between FBA and ETFL biomass compositions, we divide biomass
fix_DNA_ratio(model, mass_ratios, gc_ratio, chromosome_len, tol=0.05)	A function similar to fix_RNA_ratio. Used only in the case of adding vector
add_dummy_expression(model, aa_ratios, dummy_gene, dummy_peptide, dummy_protein, peptide_length)	
add_dummy_protein(model, dummy_peptide, enzyme_kdeg)	
add_dummy_peptide(model, aa_ratios, dummy_gene, peptide_length)	
add_dummy_mrna(model, dummy_gene, mrna_kdeg, mrna_length, nt_ratios)	
add_interpolation_variables(model)	
add_protein_mass_requirement(model, mu_values, p_rel)	Adds protein synthesis requirement
apply_prot_weight_constraint(model, p_ref, prot_ggdw, epsilon)	
define_prot_weight_constraint(model, prot_ggdw)	
add_rna_mass_requirement(model, mu_values, rna_rel)	Adds RNA synthesis requirement
apply_mrna_weight_constraint(model, m_ref, mrna_ggdw, epsilon)	
define_mrna_weight_constraint(model, mrna_ggdw)	
add_dna_mass_requirement(model, mu_values, dna_rel, gc_ratio, chromosome_len, dna_dict, ppi='ppi_c')	Adds DNA synthesis requirement
get_dna_synthesis_mets(model, chromosome_len, gc_ratio, ppi)	
apply_dna_weight_constraint(model, m_ref, dna_ggdw, epsilon)	
define_dna_weight_constraint(model, dna, dna_ggdw, gc_content, chromosome_len)	
add_lipid_mass_requirement(model, lipid_mets, mass_ratios, mu_values, lipid_rel, lipid_rxn=None)	In general, we have two main situations:
apply_lipid_weight_constraint(model, l_ref, lipid, epsilon)	
add_carbohydrate_mass_requirement(model, carbohydrate_mets, mass_ratios, mu_values, carbohydrate_rel, carbohydrate_rxn=None)	In general, we have two main situations:
apply_carbohydrate_weight_constraint(model, c_ref, carbohydrate, epsilon)	
add_ion_mass_requirement(model, ion_mets, mass_ratios, mu_values, ion_rel, ion_rxn=None)	In general, we have two main situations:
apply_ion_weight_constraint(model, i_ref, ion, epsilon)	

Attributes

MRNA_WEIGHT_CONS_ID
PROT_WEIGHT_CONS_ID
DNA_WEIGHT_CONS_ID
MRNA_WEIGHT_VAR_ID
PROT_WEIGHT_VAR_ID
DNA_WEIGHT_VAR_ID
DNA_FORMATION_RXN_ID
LIPID_FORMATION_RXN_ID
LIPID_WEIGHT_VAR_ID
LIPID_WEIGHT_CONS_ID
ION_FORMATION_RXN_ID
ION_WEIGHT_VAR_ID
ION_WEIGHT_CONS_ID
CARBOHYDRATE_FORMATION_RXN_ID
CARBOHYDRATE_WEIGHT_VAR_ID
CARBOHYDRATE_WEIGHT_CONS_ID

ETFL.**MRNA_WEIGHT_CONS_ID = mRNA_weight_definition**

ETFL.**PROT_WEIGHT_CONS_ID = prot_weight_definition**

ETFL.**DNA_WEIGHT_CONS_ID = DNA_weight_definition**

ETFL.**MRNA_WEIGHT_VAR_ID = mrna_ggdw**

ETFL.**PROT_WEIGHT_VAR_ID = prot_ggdw**

ETFL.**DNA_WEIGHT_VAR_ID = dna_ggdw**

ETFL.**DNA_FORMATION_RXN_ID = DNA_formation**

ETFL.**LIPID_FORMATION_RXN_ID = Lipid_formation**

ETFL.**LIPID_WEIGHT_VAR_ID = lipid_ggdw**

ETFL.**LIPID_WEIGHT_CONS_ID = lipid_weight_definition**

ETFL.**ION_FORMATION_RXN_ID = ion_formation**

ETFL.**ION_WEIGHT_VAR_ID = ion_ggdw**

ETFL.**ION_WEIGHT_CONS_ID = ion_weight_definition**

ETFL.**CARBOHYDRATE_FORMATION_RXN_ID = Carbohydrate_formation**

ETFL.**CARBOHYDRATE_WEIGHT_VAR_ID = carbohydrate_ggdw**

ETFL.**CARBOHYDRATE_WEIGHT_CONS_ID = carbohydrate_weight_definition**

ETFL.**fix_prot_ratio**(*model*, *mass_ratios*)

To keep consistency between FBA and ETFL biomass compositions, we divide biomass into two parts: BS1 and BS2. BS1 includes variable parts of biomass (i.e. RNA and protein), while BS2 includes the other components that are not modeled explicitly. inputs:

model: ME-model mass_ratios: a dict of mass_ratios for biomass composition in the GEM

It must have ratios for 'RNA' and 'protein'. If 'total mass' is provided, it is used to scale ratios. Otherwise, it's assumed to be 1 gr.

outputs:

return a model with an additional constraint on sum of RNA and protein share

ETFL.**fix_RNA_ratio**(*model*, *mass_ratios*)

To keep consistency between FBA and ETFL biomass compositions, we divide biomass into two parts: BS1 and BS2. BS1 includes variable parts of biomass (i.e. RNA and protein), while BS2 includes the other components that are not modeled explicitly. inputs:

model: ME-model mass_ratios: a dict of mass_ratios for biomass composition in the GEM

It must have ratios for 'RNA' and 'protein'. If 'total mass' is provided, it is used to scale ratios. Otherwise, it's assumed to be 1 gr.

outputs:

return a model with an additional constraint on sum of RNA and protein share

ETFL.**fix_DNA_ratio**(*model*, *mass_ratios*, *gc_ratio*, *chromosome_len*, *tol=0.05*)

A function similar to fix_RNA_ratio. Used only in the case of adding vector and when variable biomass composition is not available. It adds a DNA species to the model that with a constant concentration, but this can be used for RNAP allocation constraints (to be compatible with those constraints). tol: a tolerance ration for the deviation of DNA from its mass ratio

ETFL.**add_dummy_expression**(*model*, *aa_ratios*, *dummy_gene*, *dummy_peptide*, *dummy_protein*, *peptide_length*)

ETFL.**add_dummy_protein**(*model*, *dummy_peptide*, *enzyme_kdeg*)

ETFL.**add_dummy_peptide**(*model*, *aa_ratios*, *dummy_gene*, *peptide_length*)

ETFL.**add_dummy_mrna**(*model*, *dummy_gene*, *mrna_kdeg*, *mrna_length*, *nt_ratios*)

ETFL.**add_interpolation_variables**(*model*)

ETFL.**add_protein_mass_requirement**(*model*, *mu_values*, *p_rel*)

Adds protein synthesis requirement

input of type:

..code

```
mu_values=[ 0.6,        1.0,        1.5,        2.0,        2.5     ]
p_rel   = [ 0.675676,   0.604651,   0.540416,   0.530421,   0.520231]

# mu_values in [h^-1]
# p_rel in [g/gDw]
```

Parameters

- **mu_values** –
- **p_rel** –

Returns

ETFL.**apply_prot_weight_constraint**(*model*, *p_ref*, *prot_ggdw*, *epsilon*)

ETFL.**define_prot_weight_constraint**(*model*, *prot_ggdw*)

ETFL.**add_rna_mass_requirement**(*model*, *mu_values*, *rna_rel*)

Adds RNA synthesis requirement

input of type:

```
mu_values = [   0.6,        1.0,        1.5,        2.0,        2.5     ]
rna_rel   = [   0.135135    0.151163    0.177829    0.205928    0.243931]

# mu_values in [h^-1]
# rna_rel in [g/gDw]
```

Parameters

- **mu_values** –
- **rna_rel** –

Returns

ETFL.**apply_mrna_weight_constraint**(*model*, *m_ref*, *mrna_ggdw*, *epsilon*)

ETFL.**define_mrna_weight_constraint**(*model*, *mrna_ggdw*)

ETFL.**add_dna_mass_requirement**(*model*, *mu_values*, *dna_rel*, *gc_ratio*, *chromosome_len*, *dna_dict*, *ppi='ppi_c'*)

Adds DNA synthesis requirement

input of type:

```
mu_values = [   0.6,        1.0,        1.5,        2.0,        2.5     ]
dna_rel   = [   0.135135    0.151163    0.177829    0.205928    0.243931]

# mu_values in [h^-1]
# dna_rel in [g/gDw]
```

Parameters

- **mu_values** –
- **dna_rel** –

Returns

ETFL.**get_dna_synthesis_mets**(*model*, *chromosome_len*, *gc_ratio*, *ppi*)

ETFL.**apply_dna_weight_constraint**(*model*, *m_ref*, *dna_ggdw*, *epsilon*)

ETFL.**define_dna_weight_constraint**(*model*, *dna*, *dna_ggdw*, *gc_content*, *chromosome_len*)

ETFL.**add_lipid_mass_requirement**(*model*, *lipid_mets*, *mass_ratios*, *mu_values*, *lipid_rel*, *lipid_rxn=None*)

In general, we have two main situations:

1) the lipid paripates in biomass formation as lumped metabolite.

2) the lipid components partipate in biomass formation individually.

In the first case, we should remove lipid metabolite from the model and replace it with a mcromolecule with a new mass balnce constraint. In the second case, after removing lipid metabolites from biomass rxn, we should define a new reaction to lump lipid metabolites. Then, it becomes similar to the first case.

model
[MeModel] ETFL model with variable biomass composition.

lipid_mets
[list] A list of lipid metabolite id(s)

mass_ratios
[dict] Keys are strings for biomass components and values are their ration in FBA model. The ratios should be consistent with the current stoichiometric coefficients.

mu_values
[list or DataFrame] Values of growth rates for which experimental data is available

lipid_rel
[ist or DataFrame] Different ratios of lipid for different growth rates

lipid_rxn
[string] the rxn id for lipid psedoreaction. If None, there is no such reaction.

None.

ETFL.**apply_lipid_weight_constraint**(*model*, *l_ref*, *lipid*, *epsilon*)

ETFL.**add_carbohydrate_mass_requirement**(*model*, *carbohydrate_mets*, *mass_ratios*, *mu_values*, *carbohydrate_rel*, *carbohydrate_rxn=None*)

In general, we have two main situations:

1) the carbohydrate paripates in biomass formation as lumped metabolite.

2) the carbohydrate components partipate in biomass formation individually.

In the first case, we should remove carbohydrate metabolite from the model and replace it with a mcromolecule with a new mass balnce constraint. In the second case, after removing carbohydrate metabolites from biomass rxn, we should define a new reaction to lump carbohydrate metabolites. Then, it becomes similar to the first case.

model
[MeModel] ETFL model with variable biomass composition.

carbohydrate_mets
[list] A list of carbohydrate metabolite id(s)

mass_ratios
[dict] Keys are strings for biomass components and values are their ration in FBA model. The ratios should be consistent with the current stoichiometric coefficients.

mu_values
[list or DataFrame] Values of growth rates for which experimental data is available

carbohydrate_rel
[ist or DataFrame] Different ratios of carbohydrate for different growth rates

carbohydrate_rxn
[string] the rxn id for carbohydrate psedoreaction. If None, there is no such reaction.

None.

ETFL.**apply_carbohydrate_weight_constraint**(*model*, *c_ref*, *carbohydrate*, *epsilon*)

ETFL.**add_ion_mass_requirement**(*model*, *ion_mets*, *mass_ratios*, *mu_values*, *ion_rel*, *ion_rxn=None*)

In general, we have two main situations:

1) the ion paripates in biomass formation as lumped metabolite.

2) the ion components partipate in biomass formation individually.

In the first case, we should remove ion metabolite from the model and replace it with a mcromolecule with a new mass balnce constraint. In the second case, after removing ion metabolites from biomass rxn, we should define a new reaction to lump ion metabolites. Then, it becomes similar to the first case.

model
[MeModel] ETFL model with variable biomass composition.

ion_mets
[list] A list of ion metabolite id(s)

mass_ratios
[dict] Keys are strings for biomass components and values are their ration in FBA model. The ratios should be consistent with the current stoichiometric coefficients.

mu_values
[list or DataFrame] Values of growth rates for which experimental data is available

ion_rel
[ist or DataFrame] Different ratios of ion for different growth rates

ion_rxn
[string] the rxn id for ion psedoreaction. If None, there is no such reaction.

None.

ETFL.**apply_ion_weight_constraint**(*model*, *i_ref*, *ion*, *epsilon*)

etfl.core.carbohydrate

Created on Tue Mar 17 18:56:43 2020

@author: Omid

Module Contents

Classes

Carbohydrate

class etfl.core.carbohydrate.**Carbohydrate**(*composition*, *mass_ratio*, *id='Carbohydrate'*, *kdeg=0*, **args*, ***kwargs*)

Bases: etfl.core.macromolecule.Macromolecule

init_variable(*self*, *queue=False*)

Attach a carbohydrateVariable object to the Species. Needs to have the object attached to a model

Returns

property molecular_weight(*self*)

To keep consistency

Returns

etfl.core.dna

ME-related Enzyme subclasses and methods definition

Module Contents

Classes

DNA

class ETFL.**DNA**(*dna_len*, *gc_ratio*, *id='DNA'*, *kdeg=0*, **args*, ***kwargs*)

Bases: etfl.core.macromolecule.Macromolecule

init_variable(*self*, *queue=False*)

Attach a DNAVariable object to the Species. Needs to have the object attached to a model

Returns

property molecular_weight(*self*)

Calculates the moleular weight of DNA based on the DNA GC-content and length

Returns

etfl.core.enzyme

ME-related Enzyme subclasses and methods definition

Module Contents

Classes

Enzyme	
Peptide	Subclass to describe peptides resulting from gene translation
Ribosome	
RNAPolymerase	

class ETFL.**Enzyme**(*id=None*, *kcat=None*, *kcat_fwd=None*, *kcat_bwd=None*, *kdeg=None*, *composition=None*, **args*, ***kwargs*)

Bases: etfl.core.macromolecule.Macromolecule

init_variable(*self*, *queue=False*)

Attach an EnzymeVariable object to the Species. Needs to have the object attached to a model

Returns

property molecular_weight(*self*)

class ETFL.**Peptide**(*id=None*, *gene_id=None*, *sequence=None*, ***kwargs*)

Bases: cobra.Metabolite

Subclass to describe peptides resulting from gene translation

property gene(*self*)

property peptide(*self*)

property molecular_weight(*self*)

static from_metabolite(*met*, *gene_id=None*)

class ETFL.**Ribosome**(*id=None*, *kribo=None*, *kdeg=None*, *composition=None*, *rrna=None*, **args*, ***kwargs*)

Bases: *Enzyme*

property kribo(*self*)

property molecular_weight(*self*)

class ETFL.**RNAPolymerase**(*id=None*, *ktrans=None*, *kdeg=None*, *composition=None*, **args*, ***kwargs*)

Bases: *Enzyme*

property ktrans(*self*)

etfl.core.expression

ME-related Reaction subclasses and methods definition

Module Contents

Functions

build_trna_charging(model, aa_dict, atp='atp_c', amp='amp_c', ppi='ppi_c', h2o='h2o_c', h='h_c')	Build th tRNA charging reactions, based on the amino acid dictionary
get_trna_charging_id(aa_id)	
make_stoich_from_aa_sequence(sequence, aa_dict, trna_dict, gtp, gdp, pi, h2o, h)	Makes the stoichiometry of the peptide synthesis reaction based on the
make_stoich_from_nt_sequence(sequence, nt_dict, ppi)	Makes the stoichiometry of the RNA synthesis reaction based on the
degrade_peptide(peptide, aa_dict, h2o)	Degrades a peptide in amino acids, based on its sequence
degrade_mrna(mrna, nt_dict, h2o, h)	Degrades a mRNA in nucleotides monophosphate, based on its sequence
is_me_compatible(reaction)	Check if a Cobra reaction has sufficient information to add expression coupling
enzymes_to_gpr(rxn)	Builds a Gene to Protein to Reaction association rules from the enzymes of
enzymes_to_gpr_no_stoichiometry(rxn)	Builds a Gene to Protein to Reaction association rules from the enzymes of
_extract_trna_from_reaction(aa_stoichiometry, rxn)	Read a stoichiometry dictionary, and replaces free aminoacids with tRNAs

ETFL.**build_trna_charging**(*model*, *aa_dict*, *atp='atp_c'*, *amp='amp_c'*, *ppi='ppi_c'*, *h2o='h2o_c'*, *h='h_c'*)

Build th tRNA charging reactions, based on the amino acid dictionary

Parameters

- **model** (etfl.core.memodel.MEModel) – An ETFL Model
- **aa_dict** – A dictionary of aminoacid letter to amicoacid met id

  Example :

```
aa_dict = {
            'A':'ala__L_c',
            'R':'arg__L_c',
            ...
        }
```

- **atp** – metabolite ID of the cytosolic ATP
- **amp** – metabolite ID of the cytosolic AMP
- **ppi** – metabolite ID of the cytosolic diphosphate
- **h2o** – metabolite ID of the cytosolic water
- **h** – metabolite ID of the cytosolic hydrogen ions

Returns
A dictionary of tRNAs, keys are the aminoacido letters, values the charging reactions

ETFL.**get_trna_charging_id**(*aa_id*)

ETFL.**make_stoich_from_aa_sequence**(*sequence*, *aa_dict*, *trna_dict*, *gtp*, *gdp*, *pi*, *h2o*, *h*)
Makes the stoichiometry of the peptide synthesis reaction based on the amino acid sequence

Parameters

- **sequence** (Bio.Seq or str) – sequence of aminoacids (letter form)
- **aa_dict** – A dictionary of aminoacid letter to amicoacid met id

  Example :

```
aa_dict = {
            'A':'ala__L_c',
            'R':'arg__L_c',
            ...
        }
```

- **trna_dict** – the dict returned by etfl.core.expression.build_trna_charging()
- **gtp** – metabolite ID for GTP
- **gdp** – metabolite ID for GDP
- **pi** – metabolite ID for phosphate
- **h2o** – metabolite ID for water
- **h** – metabolite ID for H+

Returns

ETFL.**make_stoich_from_nt_sequence**(*sequence*, *nt_dict*, *ppi*)
Makes the stoichiometry of the RNA synthesis reaction based on the nucleotides sequence

Parameters

- **sequence** (Bio.Seq or str) – sequence of RNA nucleotides
- **nt_dict** –

  A dictionary of RNA nucleotide triphosphate
  letter to nucleotideTP met id

  Example :

```
rna_nucleotides = {
            'A':'atp_c',
            'U':'utp_c',
            ...
        }
```

- **ppi** – metabolite ID for diphosphate

Returns

ETFL.**degrade_peptide**(*peptide*, *aa_dict*, *h2o*)
Degrades a peptide in amino acids, based on its sequence

Parameters

- **peptide** (etfl.core.enzyme.Peptide) – The peptide
- **aa_dict** – A dictionary of aminoacid letter to amicoacid met id

  ** Example : **

```
aa_dict = {
            'A':'ala__L_c',
            'R':'arg__L_c',
            ...
        }
```

- **h2o** – metabolite ID for water

Returns

ETFL.**degrade_mrna**(*mrna*, *nt_dict*, *h2o*, *h*)

Degrades a mRNA in nucleotides monophosphate, based on its sequence

Parameters

- **mrna** (etfl.core.rna.mRNA) – The peptide
- **nt_dict** – A dictionary of RNA nucleotide monophosphate letter to nucleotideMP met id

  Example :

```
rna_nucleotides_mp = {
            'A':'amp_c',
            'U':'ump_c',
            ...
        }
```

- **h2o** – metabolite ID for water
- **h** – metabolite ID for H+

Returns

ETFL.**is_me_compatible**(*reaction*)

Check if a Cobra reaction has sufficient information to add expression coupling

Parameters
reaction (cobra.core.Reaction) –

Returns

ETFL.**enzymes_to_gpr**(*rxn*)

Builds a Gene to Protein to Reaction association rules from the enzymes of an enzymatic reaction

Parameters
rxn –

Returns

ETFL.**enzymes_to_gpr_no_stoichiometry**(*rxn*)

Builds a Gene to Protein to Reaction association rules from the enzymes of an enzymatic reaction

Parameters
rxn –

Returns

ETFL.**_extract_trna_from_reaction**(*aa_stoichiometry*, *rxn*)

Read a stoichiometry dictionary, and replaces free aminoacids with tRNAs

Parameters

- **aa_stoichiometry** ((dict) {cobra.core.Metabolite: Number}) – the stoichiometry dict to edit
- **rxn** (cobra.core.Reaction) – the reaction whose stoichiometry is inspected

Returns

etfl.core.genes

ME-related Reaction subclasses and methods definition

Module Contents

Classes

ExpressedGene	This calss represents the genes that can be transcribed
CodingGene	This calss represents the genes that can be translated into protein

Functions

make_sequence(sequence)	seq_type must be an instance of DNAAphabet(), RNAAlphabet, or ProteinAlphabet

ETFL.**make_sequence**(*sequence*)

seq_type must be an instance of DNAAphabet(), RNAAlphabet, or ProteinAlphabet :param sequence: :param seq_type: :return:

class ETFL.**ExpressedGene**(*id*, *name*, *sequence*, *copy_number=1*, *transcribed_by=None*, *min_tcpt_activity=0*, **args*, ***kwargs*)

Bases: cobra.Gene

This calss represents the genes that can be transcribed

property copy_number(*self*)

property transcribed_by(*self*)

property rna(*self*)

property min_tcpt_activity(*self*)

class ETFL.**CodingGene**(*id*, *name*, *sequence*, *min_tnsl_activity=0*, *translated_by=None*, **args*, ***kwargs*)

Bases: *ExpressedGene*

This calss represents the genes that can be translated into protein

property translated_by(*self*)

property peptide(*self*)

property min_tnsl_activity(*self*)

static from_gene(*gene*, *sequence*)

This method clones a cobra.Gene object into an CodingGene, and attaches a sequence to it

Parameters

- **gene** (*cobra.Gene*) – the gene to reproduce
- **sequence** – a string-like dna sequence

Returns
an CodingGene object

etfl.core.ion

Created on Tue Mar 17 18:56:43 2020

@author: Omid

Module Contents

Classes

Ion	Helper class that provides a standard way to create an ABC using

class etfl.core.ion.**Ion**(*composition*, *mass_ratio*, *id='Ion'*, *kdeg=0*, **args*, ***kwargs*)

Bases: etfl.core.macromolecule.Macromolecule

Helper class that provides a standard way to create an ABC using inheritance.

init_variable(*self*, *queue=False*)

Attach a IonVariable object to the Species. Needs to have the object attached to a model

Returns

property molecular_weight(*self*)

To keep consistency

Returns

etfl.core.lipid

Created on Mon Mar 23 10:24:43 2020

@author: DELL

Module Contents

Classes

Lipid	

class etfl.core.lipid.**Lipid**(*composition*, *mass_ratio*, *id='Lipid'*, *kdeg=0*, **args*, ***kwargs*)

Bases: etfl.core.macromolecule.Macromolecule

init_variable(*self*, *queue=False*)

Attach a LipidVariable object to the Species. Needs to have the object attached to a model

Returns

property molecular_weight(*self*)

To keep consistency

Returns

etfl.core.macromolecule

ME-related macromolecule subclasses and methods definition

Module Contents

Classes

Macromolecule	Helper class that provides a standard way to create an ABC using

class ETFL.**Macromolecule**(*id=None*, *kdeg=0*, *scaling_factor=None*, **args*, ***kwargs*)

Bases: cobra.Species, abc.ABC

Helper class that provides a standard way to create an ABC using inheritance.

abstract init_variable(*self*, *queue=False*)

Attach an EnzymeVariable object to the Species. Needs to have the object attached to a model

Returns

property concentration(*self*)

Concentration variable of the macromolecule in the cell. :return:

property scaled_concentration(*self*)

Scaled concentration (scaling_factor*conc). If the scaling factor is the molecular weight, then this is similar to the mass fraction of the macromolecule in the cell, in g/gDW. :return:

property X(*self*)

Value of the concentration after optimization. :return:

property scaled_X(*self*)

Value of the scaled concentration (mass ratio) after optimization. :return:

property variable(*self*)

For convenience in the equations of the constraints

Returns

throw_nomodel_error(*self*)

property molecular_weight(*self*)

Necessary for scaling Use Biopython for this

Returns

property scaling_factor(*self*)

etfl.core.memodel

Core for the ME-part

Module Contents

Classes

MEModel

Functions

id_maker_rib_rnap(the_set)

ETFL.**id_maker_rib_rnap**(*the_set*)

class ETFL.**MEModel**(*model=Model()*, *name=None*, *growth_reaction=''*, *mu_range=None*, *n_mu_bins=1*, *big_M=1000*, **args*, ***kwargs*)

Bases: pytfa.core.model.LCSBModel, cobra.Model

init_etfl(*self*, *big_M*, *growth_reaction*, *mu_range*, *n_mu_bins*, *name*)

property mu(*self*)

property mu_max(*self*)

make_mu_bins(*self*)

property n_mu_bins(*self*)

init_mu_variables(*self*)

Necessary for the zeroth order approximation of mu:

$$mu \in [0.1, 0.9], nbins = 8 => mu = 0.15 OR mu = 0.25 OR ... OR mu = 0.85$$

Using binary expansion of the bins instead of a list of 0-1s described here

Returns

property mu_approx_resolution(*self*)

property growth_reaction(*self*)

Returns the growth reaction of the model. Useful because tied to the growth variable

Returns

add_nucleotide_sequences(*self*, *sequences*)

Parameters
sequences –

Returns

add_transcription_by(*self*, *transcription_dict*)

add_translation_by(*self*, *translation_dict*)

add_min_tcpt_activity(*self*, *min_act_dict*)

add_min_tnsl_activity(*self*, *min_act_dict*)

_make_peptide_from_gene(*self*, *gene_id*)

add_peptide_sequences(*self*, *aa_sequences*)

add_dummies(*self*, *nt_ratios*, *mrna_kdeg*, *mrna_length*, *aa_ratios*, *enzyme_kdeg*, *peptide_length*, *transcribed_by=None*, *translated_by=None*)

Create dummy peptide and mrna to enforce mrna and peptide production. Can be used to account for the missing data for all mrnas and proteins.

Parameters

- **nt_ratios** –
- **mrna_kdeg** –
- **mrna_length** –
- **aa_ratios** –
- **enzyme_kdeg** –
- **peptide_length** –
- **gtp** –
- **gdp** –
- **h2o** –
- **h** –

Returns

add_essentials(*self*, *essentials*, *aa_dict*, *rna_nucleotides*, *rna_nucleotides_mp*)

Marks important metabolites for expression

Parameters

- **essentials** – A dictionary of important metabolites to met id

  Example :

```
essentials = {
            'atp': 'atp_c',
            'adp': 'adp_c',
            'amp': 'amp_c',
            ...
            'h2o': 'h2o_c',
            'h': 'h_c'}
        }
```

- **aa_dict** – A dictionary of aminoacid letter to amicoacid met id

  Example :

```
aa_dict = {
            'A':'ala__L_c',
            'R':'arg__L_c',
            ...
        }
```

- **rna_nucleotides** – A dictionary of RNA nucleotide triphosphate letter to nucleotideTP met id

  Example :

```
rna_nucleotides = {
            'A':'atp_c',
            'U':'utp_c',
            ...
        }
```

- **rna_nucleotides_mp** – A dictionary of RNA nucleotide monophosphate letter to nucleotideMP met id

  Example :

```
rna_nucleotides_mp = {
            'A':'amp_c',
            'U':'ump_c',
            ...
        }
```

Returns

build_expression(*self*)

Given a dictionary from amino acids nucleotides to metabolite names, goes through the list of genes in the model that have sequence information to build transcription and translation reactions

Returns

express_genes(*self*, *gene_list*)

Adds translation and transcription reaction to the genes in the provided list

Parameters
gene_list (*Iterable of str or* ExpressedGene) –

Returns

_add_gene_translation_reaction(*self*, *gene*)

Parameters
gene (CodingGene) – A gene of the model that has sequence data

Returns

_add_gene_transcription_reaction(*self*, *gene*)

Adds the transcription reaction related to a gene

Parameters
gene (ExpressedGene) – A gene of the model that has sequence data

Returns

add_trna_mass_balances(*self*)

Once the tRNAs, transcription and translation reactions have been added, we need to add the constraints:

d/dt [charged_tRNA] = v_charging - sum(nu_trans*v_trans) - mu*[charged_tRNA] d/dt [uncharged_tRNA] = -v_charging + sum(nu_trans*v_trans) - mu*[uncharged_tRNA]

The stoichiometries are set from the reaction dict in _extract_trna_from_reaction

We also need to scale the tRNAs in mRNA space and unscale the translation:

```
d/dt _m * [*charged_tRNA] =    +- _m * v_charging
                                -+ _m/_p*sum(nu_tsl*_p*v_tr)
                                -  _m * mu*[*charged_tRNA]

d/dt [*charged_tRNA]_hat =      +- _m * v_charging
                                -+ _m/_p * sum( nu_tsl * v_tr_hat)
                              -  mu*[*charged_tRNA]_hat
```

Returns

add_enzymatic_coupling(*self*, *coupling_dict*)

Couples the enzymatic reactions maximal rates with the Enzyme availability The coupling dictionary looks like:

```
coupling_dict : {
                'reaction_id_1':[   enzyme_instance_1,
                                    enzyme_instance_2],
                'reaction_id_2':[   enzyme_instance_3,
                                    enzyme_instance_4,
                                    enzyme_instance_5],
```

Parameters
coupling_dict (*{str:list(*Enzyme*)}*) – A dictionary of reaction ids to enzyme lists

Returns

apply_enzyme_catalytic_constraint(*self*, *reaction*)

Apply a catalytic constraint using a gene-enzymes reaction rule (GPR)

Parameters
reaction –

Returns

add_mass_balance_constraint(*self*, *synthesis_flux*, *macromolecule=None*, *queue=False*)

Adds a mass balance constraint of the type

$$d[E]/dt = 0 <=> v_synthesis - k_deg * [M] - *[M] = 0$$

for a macromolecule (mRNA or enzyme)

Parameters

- **synthesis_flux** –
- **macromolecule** –

Returns

linearize_me(*self*, *macromolecule*, *queue=False*)

Performs Petersen linearization on *E to keep a MILP problem

Returns

get_ordered_ga_vars(*self*)

Returns in order the variables that discretize growth :return:

_prep_enzyme_variables(*self*, *enzyme*)

Reads Enzyme.composition to find complexation reaction from enzyme information

Parameters
reaction (*cobra.Reaction*) –

Returns

make_enzyme_complexation(*self*, *enzyme*)

Makes the complexation reaction and attached it to its enzyme

Parameters
enzyme –

Returns

add_enzymes(*self*, *enzyme_list*, *prep=True*)

Adds an Enzyme object, or iterable of Enzyme objects, to the model :param enzyme_list: :type enzyme_list:Iterable(Enzyme) or Enzyme :param prep: whether or not to add complexation, degradation, and mass

balance constraints (needs to be overridden for dummies for example)

Returns

add_mrnas(*self*, *mrna_list*, *add_degradation=True*)

Adds a mRNA object, or iterable of mRNA objects, to the model :param mrna_list: :type mrna_list:Iterable(mRNA) or mRNA :return:

add_trnas(*self*, *trna_list*)

Adds a tRNA object, or iterable of tRNA objects, to the model :param trna_list: :type trna_list:Iterable(tRNA) or tRNA :return:

add_dna(*self*, *dna*)

Adds a DNA object to the model

Parameters
dna (DNA) –

Returns

add_lipid(*self*, *lipid*)

Adds a lipid object to the model

Parameters
lipid (Lipid) –

Returns

add_ion(*self*, *ion*)

Adds a ion object to the model

Parameters
ion (*ion*) –

Returns

add_carbohydrate(*self*, *carbohydrate*)

Adds a carbohydrate object to the model

Parameters
carbohydrate (*carbohydrate*) –

Returns

remove_enzymes(*self*, *enzyme_list*)

Removes an Enzyme object, or iterable of Enzyme objects, from the model

Parameters
enzyme_list –

:type enzyme_list:Iterable(Enzyme) or Enzyme :return:

_add_enzyme_degradation(*self*, *enzyme*, *scaled=True*, *queue=False*)

Given an enzyme, adds the corresponding degradation reaction

Parameters

- **enzyme** (Enzyme) –
- **scaled** (*bool*) – Indicates whether scaling should be performed (see manuscript)
- **queue** (*bool*) – Indicates whether to add the variable directly or in the next batch

Returns

_add_mrna_degradation(*self*, *mrna*, *scaled=True*, *queue=False*)

Given an mRNA, adds the corresponding degradation reaction

Parameters

- **mrna** (mRNA) –
- **scaled** (*bool*) – Indicates whether scaling should be performed (see manuscript)

- **queue** (*bool*) – Indicates whether to add the variable directly or in the next batch

Returns

_make_degradation_reaction(*self*, *deg_stoich*, *macromolecule*, *kind*, *scaled*, *queue=False*)

given a degradation stoichiometry, makes the corresponding degradation reaction

Parameters

- **deg_stoich** (dict({cobra.core.Species:Number})) – stoichiometry of the degradation
- **macromolecule** (Macromolecule) – the macromalecule being degraded. Used for binding the degradation constraint
- **kind** (mRNADegradation *or* EnzymeDegradation) – kind of constraint
- **scaled** (*bool*) – Indicates whether scaling should be performed (see manuscript)
- **queue** (*bool*) – Indicates whether to add the variable directly or in the next batch

Returns

populate_expression(*self*)

Defines upper- and lower_bound for the RNAP and Ribosome binding capacities and define catalytic constraints for the RNAP and Ribosome

Returns

add_mrna_mass_balance(*self*, *the_mrna*)

_constrain_polysome(*self*, *the_mrna*, *basal_fraction=0*)

Add the coupling between mRNA availability and ribosome charging The number of ribosomes assigned to a mRNA species is lower than the number of such mRNA times the max number of ribosomes that can sit on the mRNA: [RPi] <= loadmax_i*[mRNAi]

loadmax is : len(peptide_chain)/size(ribo) Their distance from one another along the mRNA is at least the size of the physical footprint of a ribosome (20 nm, BNID 102320, 100121) which is the length of about 60 base pairs (length of nucleotide 0.3 nm, BNID 103777), equivalent to 20 aa. also 28715909 "http://book.bionumbers.org/how-many-proteins-are-made-per-mrna-molecule/"

Hence: [RPi] <= L_nt/Ribo_footprint * [mRNA]

In addition, it also adds a minimal binding activity for ribosome to the mRNA. We modeled it as a Fraction of the maximum loadmax and the Fraction depends on the affinity of ribosome to the mRNA: [RPi] >= Fraction * L_nt/Ribo_footprint * [mRNA]

Returns

_constrain_polymerase(*self*, *the_gene*, *basal_fraction=0*)

Add the coupling between DNA availability and RNAP charging The number of RNAP assigned to a gene locus is lower than the number of such loci times the max number of RNAP that can sit on the locus: [RNAPi] <= loadmax_i*[# of loci]*[DNA]

loadmax is : len(nucleotide chain)/size(RNAP)

"The footprint of RNAP II [. . .] covers approximately 40 nt and is nearly symmetrical [. . .]." BNID 107873 Range ~40 Nucleotides

Hence: [RNAPi] <= loadmax_i*[# of loci]*[DNA]

In addition, it also adds a minimal binding activity for RNAP to the gene. We modeled it as a Fraction of the maximum loadmax and the Fraction depends on the affinity of RNAP to the gene, i.e. the strength of the promoter: [RNAPi] >= Fraction * L_nt/RNAP_footprint * [# of loci]*[DNA]

Returns

edit_gene_copy_number(*self*, *gene_id*)

Edits the RNAP allocation constraints if the copy number of a gene changes.

Parameters
gene_id –

Returns

recompute_translation(*self*)

Returns

recompute_transcription(*self*)

Returns

recompute_allocation(*self*)

Returns

_get_transcription_name(*self*, *the_mrna_id*)

Given an mrna_id, gives the id of the corresponding transcription reaction :param the_mrna_id: :type the_mrna_id: str :return: str

_get_translation_name(*self*, *the_peptide_id*)

Given an mrna_id, gives the id of the corresponding translation reaction :param the_peptide_id: :type the_peptide_id: str :return: str

get_translation(*self*, *the_peptide_id*)

Given an peptide_id, gives the translation reaction :param the_peptide_id: :type the_peptide_id: str :return: TranslationReaction

get_transcription(*self*, *the_peptide_id*)

Given an mrna_id, gives corresponding transcription reaction :param the_mrna_id: :type the_mrna_id: str :return: TranscriptionReaction

add_rnap(*self*, *rnap*, *free_ratio=0*)

Adds the RNA Polymerase used by the model.

Parameters
rnap (Ribosome) –

Returns

_populate_rnap(*self*)

Once RNAP have been assigned to the model, we still need to link them to the rest of the variables and constraints. This function creates the mass balance constraint on the RNAP, as well as the total RNAP capacity constraint :return:

_sort_rnap_assignment(*self*)

_get_rnap_total_capacity(*self*, *rnap_ids*, *genes*)

apply_rnap_catalytic_constraint(*self*, *reaction*, *queue*)

Given a translation reaction, apply the constraint that links it with RNAP usage :param reaction: a TranscriptionReaction :type reaction: TranscriptionReaction :return:

_add_free_enzyme_ratio(*self*, *enzyme*, *free_ratio*)

Adds free enzyme variables to the models /!A total capacity constraint still needs to be added # TODO: Make that more user friendly :return:

add_ribosome(*self*, *ribosome*, *free_ratio*)

Adds the ribosome used by the model.

Parameters
ribosome (Ribosome) –

Returns

add_rrnas_to_rib_assembly(*self*, *ribosome*)

Adds the ribosomal RMAs to the composition of the ribosome. This has to be done after the transcription reactions have been added, so that the rRNAs synthesis reactions exist for the mass balance

Returns

property Rt(*self*)

_populate_ribosomes(*self*)

Once ribosomes have been assigned to the model, we still need to link them to the rest of the variables and constraints. This function creates the mass balance constraint on the ribosomes, as well as the total ribosome capacity constraint :return:

couple_rrna_synthesis(*self*)

_sort_rib_assignment(*self*)

_get_rib_total_capacity(*self*, *rib_ids*, *genes*)

apply_ribosomal_catalytic_constraint(*self*, *reaction*)

Given a translation reaction, apply the constraint that links it with ribosome usage :param reaction: a TranslationReaction :type reaction: TranslationReaction :return:

add_genes(*self*, *genes*)

Oddly I could not find this method in cobra. Adds one or several genes to the model.

Parameters
genes (*Iterable(Gene) or Gene*) –

Returns

_add_gene(*self*, *gene*)

sanitize_varnames(*self*)

Makes variable name safe for the solvers. In particular, variables whose name start with :return:

print_info(*self*, *specific=False*)

Print information and counts for the cobra_model :return:

__deepcopy__(*self*, *memo*)

Calls self.copy() to return an independant copy of the model

Parameters
memo –

Returns

copy(*self*)

Pseudo-smart copy of the model using dict serialization. This builds a new model from the ground up, with independwnt variables, solver, etc.

Returns

etfl.core.reactions

ME-related Reaction subclasses and methods definition

Module Contents

Classes

ExpressionReaction	
EnzymaticReaction	Subclass to describe reactions that are catalyzed by an enzyme.
TranscriptionReaction	Class describing transcription - Assembly of amino acids into peptides
TranslationReaction	Class describing translation - Assembly of amino acids into peptides
ProteinComplexation	Describes the assembly of peptides into an enzyme
DegradationReaction	Describes the degradation of macromolecules
DNAFormation	Describes the assembly of NTPs into DNA

class ETFL.**ExpressionReaction**(*scaled*, ***kwargs*)

Bases: cobra.Reaction

classmethod from_reaction(*cls*, *reaction*, *scaled=False*, ***kwargs*)

This method clones a cobra.Reaction object into a expression-related type of reaction

Parameters
reaction – the reaction to reproduce

Returns
an EnzymaticReaction object

add_metabolites(*self*, *metabolites*, *rescale=True*, ***kwargs*)

We need to override this method if the reaction is scaled

v_hat = v/vmax

dM/dt = n1*v1 + ...

dM/dt = n1*vmax1 * v1_hat + ...

Parameters
metabolites –

Returns

property scaling_factor(*self*)

property net(*self*)

property scaled_net(*self*)

class ETFL.EnzymaticReaction(*enzymes=None*, *scaled=False*, **args*, ***kwargs*)

Bases: *ExpressionReaction*

Subclass to describe reactions that are catalyzed by an enzyme.

add_enzymes(*self*, *enzymes*)

` Method to add the enzymes to the reaction. :param enzymes: iterable of or single Enzyme object :return:

property scaling_factor(*self*)

class ETFL.TranscriptionReaction(*id*, *name*, *gene_id*, *enzymes*, ***kwargs*)

Bases: *EnzymaticReaction*

Class describing transcription - Assembly of amino acids into peptides

property gene(*self*)

property nucleotide_length(*self*)

add_rnap(*self*, *rnap*)

By definition this reaction will be catalyzed by RNA polymerase :param ribosome: :type ribosome: pytfa.me.RNAPolymerase :return:

property scaling_factor(*self*)

class ETFL.TranslationReaction(*id*, *name*, *gene_id*, *enzymes*, *trna_stoich=None*, ***kwargs*)

Bases: *EnzymaticReaction*

Class describing translation - Assembly of amino acids into peptides

property gene(*self*)

property aminoacid_length(*self*)

add_peptide(*self*, *peptide*)

According to the scaling rules, the coefficient of the scaled translation reaction for the peptide balance is 1:

dPep/dt = v_tsl - sum(j * vj_asm) = 0

v_tsl_hat - sum(j * L_aa/(krib * R_max) * kdegj * Ej_max * vj_asm_max)

Parameters

peptide –

Returns

add_ribosome(*self*, *ribosome*)

By definition this reaction will be catalyzed by a ribosome :param ribosome: :type ribosome: pytfa.me.Ribosome :return:

property scaling_factor(*self*)

class ETFL.ProteinComplexation(*target*, **args*, ***kwargs*)

Bases: *ExpressionReaction*

Describes the assembly of peptides into an enzyme

property scaling_factor(*self*)

add_peptides(*self*, *peptides*)

/!Reaction must belong to a model

According to the scaling rules, the coefficient of the scaled complexation reaction for the peptide balance is L_aa/(krib * R_max):

dPep/dt = v_tsl - sum(j * vj_asm) = 0

v_tsl_hat - sum(j * L_aa/(krib * R_max) * kdegj * Ej_max * vj_asm_max)

Parameters

peptides – dict(Peptide: int)

Returns

class ETFL.**DegradationReaction**(*macromolecule*, **args*, ***kwargs*)

Bases: *ExpressionReaction*

Describes the degradation of macromolecules

property scaling_factor(*self*)

class ETFL.**DNAFormation**(*dna*, *mu_sigma=1*, **args*, ***kwargs*)

Bases: *ExpressionReaction*

Describes the assembly of NTPs into DNA

property scaling_factor(*self*)

etfl.core.rna

ME-related Enzyme subclasses and methods definition

Module Contents

Classes

RNA
mRNA
rRNA
tRNA

class ETFL.**RNA**(*id=None*, *kdeg=None*, *gene_id=None*, **args*, ***kwargs*)

Bases: etfl.core.macromolecule.Macromolecule

property rna(*self*)

property gene(*self*)

init_variable(*self*, *queue=False*)

Attach an mRNAVariable object to the Species. Needs to have the object attached to a model

Returns

property molecular_weight(*self*)

class ETFL.**mRNA**(*id=None*, *kdeg=None*, *gene_id=None*, **args*, ***kwargs*)

Bases: *RNA*

property peptide(*self*)

class ETFL.**rRNA**(*id=None*, *ribosomes=[]*, ***kwargs*)

Bases: cobra.Metabolite

property ribosomes(*self*)

static from_metabolite(*met*)

class ETFL.**tRNA**(*aminoacid_id*, *charged*, **args*, ***kwargs*)

Bases: etfl.core.macromolecule.Macromolecule

init_variable(*self*, *queue=False*)

Attach a tRNAVariable object to the Species. Needs to have the object attached to a model

Returns

property aminoacid(*self*)

property molecular_weight(*self*)

etfl.core.thermomemodel

Fusion for Thermo and Me Models

Module Contents

Classes

ThermoMEModel

Attributes

BIGM
BIGM_THERMO
BIGM_DG
BIGM_P
EPSILON
MAX_STOICH

ETFL.**BIGM**

ETFL.**BIGM_THERMO**

ETFL.**BIGM_DG**

ETFL.**BIGM_P**

ETFL.**EPSILON**

ETFL.**MAX_STOICH = 10**

class ETFL.**ThermoMEModel**(*thermo_data*, *model=Model()*, *name=None*, *growth_reaction=''*, *mu=None*, *mu_error=0*, *mu_range=None*, *n_mu_bins=1*, *big_M=1000*, *temperature=std.TEMPERATURE_0*, *min_ph=std.MIN_PH*, *max_ph=std.MAX_PH*, *prot_scaling=1000*, *mrna_scaling=None*)

Bases: etfl.core.memodel.MEModel, pytfa.thermo.ThermoModel

print_info(*self*)

Print information and counts for the cobra_model :return:

__deepcopy__(*self*, *memodict={}*)

Calls self.copy() to return an independant copy of the model

Parameters
memo –

Returns

copy(*self*)

Pseudo-smart copy of the model using dict serialization. This builds a new model from the ground up, with independwnt variables, solver, etc.

Returns

Package Contents

Classes

Enzyme
Ribosome
RNAPolymerase
ThermoMEModel
MEModel

class etfl.core.**Enzyme**(*id=None*, *kcat=None*, *kcat_fwd=None*, *kcat_bwd=None*, *kdeg=None*, *composition=None*, **args*, ***kwargs*)

Bases: etfl.core.macromolecule.Macromolecule

init_variable(*self*, *queue=False*)

Attach an EnzymeVariable object to the Species. Needs to have the object attached to a model

Returns

property molecular_weight(*self*)

class etfl.core.**Ribosome**(*id=None*, *kribo=None*, *kdeg=None*, *composition=None*, *rrna=None*, **args*, ***kwargs*)

Bases: *Enzyme*

property kribo(*self*)

property molecular_weight(*self*)

class etfl.core.**RNAPolymerase**(*id=None*, *ktrans=None*, *kdeg=None*, *composition=None*, **args*, ***kwargs*)

Bases: *Enzyme*

property ktrans(*self*)

class etfl.core.**ThermoMEModel**(*thermo_data*, *model=Model()*, *name=None*, *growth_reaction=''*, *mu=None*, *mu_error=0*, *mu_range=None*, *n_mu_bins=1*, *big_M=1000*, *temperature=std.TEMPERATURE_0*, *min_ph=std.MIN_PH*, *max_ph=std.MAX_PH*, *prot_scaling=1000*, *mrna_scaling=None*)

Bases: etfl.core.memodel.MEModel, pytfa.thermo.ThermoModel

print_info(*self*)

Print information and counts for the cobra_model :return:

__deepcopy__(*self*, *memodict={}*)

Calls self.copy() to return an independant copy of the model

Parameters

memo –

Returns

copy(*self*)

Pseudo-smart copy of the model using dict serialization. This builds a new model from the ground up, with independwnt variables, solver, etc.

Returns

class etfl.core.**MEModel**(*model=Model()*, *name=None*, *growth_reaction=''*, *mu_range=None*, *n_mu_bins=1*, *big_M=1000*, **args*, ***kwargs*)

Bases: pytfa.core.model.LCSBModel, cobra.Model

init_etfl(*self*, *big_M*, *growth_reaction*, *mu_range*, *n_mu_bins*, *name*)

property mu(*self*)

property mu_max(*self*)

make_mu_bins(*self*)

property n_mu_bins(*self*)

init_mu_variables(*self*)

Necessary for the zeroth order approximation of mu:

$$mu \in [0.1, 0.9], nbins = 8 => mu = 0.15 OR mu = 0.25 OR ... OR mu = 0.85$$

Using binary expansion of the bins instead of a list of 0-1s described here

Returns

property mu_approx_resolution(*self*)

property growth_reaction(*self*)

Returns the growth reaction of the model. Useful because tied to the growth variable

Returns

add_nucleotide_sequences(*self*, *sequences*)

Parameters
sequences –

Returns

add_transcription_by(*self*, *transcription_dict*)

add_translation_by(*self*, *translation_dict*)

add_min_tcpt_activity(*self*, *min_act_dict*)

add_min_tnsl_activity(*self*, *min_act_dict*)

_make_peptide_from_gene(*self*, *gene_id*)

add_peptide_sequences(*self*, *aa_sequences*)

add_dummies(*self*, *nt_ratios*, *mrna_kdeg*, *mrna_length*, *aa_ratios*, *enzyme_kdeg*, *peptide_length*, *transcribed_by=None*, *translated_by=None*)

Create dummy peptide and mrna to enforce mrna and peptide production. Can be used to account for the missing data for all mrnas and proteins.

Parameters

- **nt_ratios** –
- **mrna_kdeg** –
- **mrna_length** –
- **aa_ratios** –
- **enzyme_kdeg** –
- **peptide_length** –
- **gtp** –
- **gdp** –
- **h2o** –
- **h** –

Returns

add_essentials(*self*, *essentials*, *aa_dict*, *rna_nucleotides*, *rna_nucleotides_mp*)

Marks important metabolites for expression

Parameters

- **essentials** – A dictionary of important metabolites to met id

  Example :

```
essentials = {
            'atp': 'atp_c',
            'adp': 'adp_c',
            'amp': 'amp_c',
            ...
            'h2o': 'h2o_c',
            'h': 'h_c'}
        }
```

- **aa_dict** – A dictionary of aminoacid letter to amicoacid met id

  Example :

```
aa_dict = {
            'A':'ala__L_c',
            'R':'arg__L_c',
            ...
        }
```

- **rna_nucleotides** – A dictionary of RNA nucleotide triphosphate letter to nucleotideTP met id

  Example :

```
rna_nucleotides = {
            'A':'atp_c',
            'U':'utp_c',
            ...
        }
```

- **rna_nucleotides_mp** – A dictionary of RNA nucleotide monophosphate letter to nucleotideMP met id

  Example :

```
rna_nucleotides_mp = {
            'A':'amp_c',
            'U':'ump_c',
            ...
        }
```

Returns

build_expression(*self*)

Given a dictionary from amino acids nucleotides to metabolite names, goes through the list of genes in the model that have sequence information to build transcription and translation reactions

Returns

express_genes(*self*, *gene_list*)

Adds translation and transcription reaction to the genes in the provided list

Parameters
gene_list (*Iterable of str or* ExpressedGene) –

Returns

_add_gene_translation_reaction(*self*, *gene*)

Parameters
gene (CodingGene) – A gene of the model that has sequence data

Returns

_add_gene_transcription_reaction(*self*, *gene*)

Adds the transcription reaction related to a gene

Parameters
gene (ExpressedGene) – A gene of the model that has sequence data

Returns

add_trna_mass_balances(*self*)

Once the tRNAs, transcription and translation reactions have been added, we need to add the constraints:

d/dt [charged_tRNA] = v_charging - sum(nu_trans*v_trans) - mu*[charged_tRNA] d/dt [uncharged_tRNA] = -v_charging + sum(nu_trans*v_trans) - mu*[uncharged_tRNA]

The stoichiometries are set from the reaction dict in _extract_trna_from_reaction

We also need to scale the tRNAs in mRNA space and unscale the translation:

```
d/dt _m * [*charged_tRNA] =     +- _m * v_charging
                                -+ _m/_p*sum(nu_tsl*_p*v_tr)
                                -  _m * mu*[*charged_tRNA]

d/dt [*charged_tRNA]_hat =      +- _m * v_charging
                                -+ _m/_p * sum( nu_tsl * v_tr_hat)
                              -  mu*[*charged_tRNA]_hat
```

Returns

add_enzymatic_coupling(*self*, *coupling_dict*)

Couples the enzymatic reactions maximal rates with the Enzyme availability The coupling dictionary looks like:

```
coupling_dict : {
                'reaction_id_1':[   enzyme_instance_1,
                                    enzyme_instance_2],
                'reaction_id_2':[   enzyme_instance_3,
                                    enzyme_instance_4,
                                    enzyme_instance_5],
```

Parameters

coupling_dict (*{str:list(Enzyme)}*) – A dictionary of reaction ids to enzyme lists

Returns

apply_enzyme_catalytic_constraint(*self*, *reaction*)

Apply a catalytic constraint using a gene-enzymes reaction rule (GPR)

Parameters

reaction –

Returns

add_mass_balance_constraint(*self*, *synthesis_flux*, *macromolecule=None*, *queue=False*)

Adds a mass balance constraint of the type

$$d[E]/dt = 0 <=> v_synthesis - k_deg * [M] - *[M] = 0$$

for a macromolecule (mRNA or enzyme)

Parameters

- **synthesis_flux** –
- **macromolecule** –

Returns

linearize_me(*self*, *macromolecule*, *queue=False*)

Performs Petersen linearization on *E to keep a MILP problem

Returns

get_ordered_ga_vars(*self*)

Returns in order the variables that discretize growth :return:

_prep_enzyme_variables(*self*, *enzyme*)

Reads Enzyme.composition to find complexation reaction from enzyme information

Parameters

reaction (*cobra.Reaction*) –

Returns

make_enzyme_complexation(*self*, *enzyme*)

Makes the complexation reaction and attached it to its enzyme

Parameters

enzyme –

Returns

add_enzymes(*self*, *enzyme_list*, *prep=True*)

Adds an Enzyme object, or iterable of Enzyme objects, to the model :param enzyme_list: :type enzyme_list:Iterable(Enzyme) or Enzyme :param prep: whether or not to add complexation, degradation, and mass

balance constraints (needs to be overridden for dummies for example)

Returns

add_mrnas(*self*, *mrna_list*, *add_degradation=True*)

Adds a mRNA object, or iterable of mRNA objects, to the model :param mrna_list: :type mrna_list:Iterable(mRNA) or mRNA :return:

add_trnas(*self*, *trna_list*)

Adds a tRNA object, or iterable of tRNA objects, to the model :param trna_list: :type trna_list:Iterable(tRNA) or tRNA :return:

add_dna(*self*, *dna*)

Adds a DNA object to the model

Parameters
dna (DNA) –

Returns

add_lipid(*self*, *lipid*)

Adds a lipid object to the model

Parameters
lipid (Lipid) –

Returns

add_ion(*self*, *ion*)

Adds a ion object to the model

Parameters
ion (*ion*) –

Returns

add_carbohydrate(*self*, *carbohydrate*)

Adds a carbohydrate object to the model

Parameters
carbohydrate (*carbohydrate*) –

Returns

remove_enzymes(*self*, *enzyme_list*)

Removes an Enzyme object, or iterable of Enzyme objects, from the model

Parameters
enzyme_list –

:type enzyme_list:Iterable(Enzyme) or Enzyme :return:

_add_enzyme_degradation(*self*, *enzyme*, *scaled=True*, *queue=False*)

Given an enzyme, adds the corresponding degradation reaction

Parameters

- **enzyme** (Enzyme) –
- **scaled** (*bool*) – Indicates whether scaling should be performed (see manuscript)
- **queue** (*bool*) – Indicates whether to add the variable directly or in the next batch

Returns

_add_mrna_degradation(*self*, *mrna*, *scaled=True*, *queue=False*)

Given an mRNA, adds the corresponding degradation reaction

Parameters

- **mrna** (mRNA) –
- **scaled** (*bool*) – Indicates whether scaling should be performed (see manuscript)
- **queue** (*bool*) – Indicates whether to add the variable directly or in the next batch

Returns

_make_degradation_reaction(*self*, *deg_stoich*, *macromolecule*, *kind*, *scaled*, *queue=False*)

given a degradation stoichiometry, makes the corresponding degradation reaction

Parameters

- **deg_stoich** (dict({cobra.core.Species:Number})) – stoichiometry of the degradation
- **macromolecule** (Macromolecule) – the macromalecule being degraded. Used for binding the degradation constraint
- **kind** (mRNADegradation *or* EnzymeDegradation) – kind of constraint
- **scaled** (*bool*) – Indicates whether scaling should be performed (see manuscript)
- **queue** (*bool*) – Indicates whether to add the variable directly or in the next batch

Returns

populate_expression(*self*)

Defines upper- and lower_bound for the RNAP and Ribosome binding capacities and define catalytic constraints for the RNAP and Ribosome

Returns

add_mrna_mass_balance(*self*, *the_mrna*)

_constrain_polysome(*self*, *the_mrna*, *basal_fraction=0*)

Add the coupling between mRNA availability and ribosome charging The number of ribosomes assigned to a mRNA species is lower than the number of such mRNA times the max number of ribosomes that can sit on the mRNA: [RPi] <= loadmax_i*[mRNAi]

loadmax is : len(peptide_chain)/size(ribo) Their distance from one another along the mRNA is at least the size of the physical footprint of a ribosome (20 nm, BNID 102320, 100121) which is the length of about 60 base pairs (length of nucleotide 0.3 nm, BNID 103777), equivalent to 20 aa. also 28715909 "http://book.bionumbers.org/how-many-proteins-are-made-per-mrna-molecule/"

Hence: [RPi] <= L_nt/Ribo_footprint * [mRNA]

In addition, it also adds a minimal binding activity for ribosome to the mRNA. We modeled it as a Fraction of the maximum loadmax and the Fraction depends on the affinity of ribosome to the mRNA: [RPi] >= Fraction * L_nt/Ribo_footprint * [mRNA]

Returns

`_constrain_polymerase`(*self*, *the_gene*, *basal_fraction=0*)

Add the coupling between DNA availability and RNAP charging The number of RNAP assigned to a gene locus is lower than the number of such loci times the max number of RNAP that can sit on the locus: [RNAPi] <= loadmax_i*[# of loci]*[DNA]

loadmax is : len(nucleotide chain)/size(RNAP)

"The footprint of RNAP II [. . .] covers approximately 40 nt and is nearly symmetrical [. . .]." BNID 107873 Range ~40 Nucleotides

Hence: [RNAPi] <= loadmax_i*[# of loci]*[DNA]

In addition, it also adds a minimal binding activity for RNAP to the gene. We modeled it as a Fraction of the maximum loadmax and the Fraction depends on the affinity of RNAP to the gene, i.e. the strength of the promoter: [RNAPi] >= Fraction * L_nt/RNAP_footprint * [# of loci]*[DNA]

Returns

`edit_gene_copy_number`(*self*, *gene_id*)

Edits the RNAP allocation constraints if the copy number of a gene changes.

Parameters
`gene_id` –

Returns

`recompute_translation`(*self*)

Returns

`recompute_transcription`(*self*)

Returns

`recompute_allocation`(*self*)

Returns

`_get_transcription_name`(*self*, *the_mrna_id*)

Given an mrna_id, gives the id of the corresponding transcription reaction :param the_mrna_id: :type the_mrna_id: str :return: str

`_get_translation_name`(*self*, *the_peptide_id*)

Given an mrna_id, gives the id of the corresponding translation reaction :param the_peptide_id: :type the_peptide_id: str :return: str

`get_translation`(*self*, *the_peptide_id*)

Given an peptide_id, gives the translation reaction :param the_peptide_id: :type the_peptide_id: str :return: TranslationReaction

`get_transcription`(*self*, *the_peptide_id*)

Given an mrna_id, gives corresponding transcription reaction :param the_mrna_id: :type the_mrna_id: str :return: TranscriptionReaction

add_rnap(*self*, *rnap*, *free_ratio=0*)

Adds the RNA Polymerase used by the model.

Parameters
rnap (Ribosome) –

Returns

_populate_rnap(*self*)

Once RNAP have been assigned to the model, we still need to link them to the rest of the variables and constraints. This function creates the mass balance constraint on the RNAP, as well as the total RNAP capacity constraint :return:

_sort_rnap_assignment(*self*)

_get_rnap_total_capacity(*self*, *rnap_ids*, *genes*)

apply_rnap_catalytic_constraint(*self*, *reaction*, *queue*)

Given a translation reaction, apply the constraint that links it with RNAP usage :param reaction: a TranscriptionReaction :type reaction: TranscriptionReaction :return:

_add_free_enzyme_ratio(*self*, *enzyme*, *free_ratio*)

Adds free enzyme variables to the models /!A total capacity constraint still needs to be added # TODO: Make that more user friendly :return:

add_ribosome(*self*, *ribosome*, *free_ratio*)

Adds the ribosome used by the model.

Parameters
ribosome (Ribosome) –

Returns

add_rrnas_to_rib_assembly(*self*, *ribosome*)

Adds the ribosomal RMAs to the composition of the ribosome. This has to be done after the transcription reactions have been added, so that the rRNAs synthesis reactions exist for the mass balance

Returns

property Rt(*self*)

_populate_ribosomes(*self*)

Once ribosomes have been assigned to the model, we still need to link them to the rest of the variables and constraints. This function creates the mass balance constraint on the ribosomes, as well as the total ribosome capacity constraint :return:

couple_rrna_synthesis(*self*)

_sort_rib_assignment(*self*)

_get_rib_total_capacity(*self*, *rib_ids*, *genes*)

apply_ribosomal_catalytic_constraint(*self*, *reaction*)

Given a translation reaction, apply the constraint that links it with ribosome usage :param reaction: a TranslationReaction :type reaction: TranslationReaction :return:

add_genes(*self*, *genes*)

Oddly I could not find this method in cobra. Adds one or several genes to the model.

Parameters
genes (*Iterable(Gene) or Gene*) –

Returns

_add_gene(*self*, *gene*)

sanitize_varnames(*self*)

Makes variable name safe for the solvers. In particular, variables whose name start with :return:

print_info(*self*, *specific=False*)

Print information and counts for the cobra_model :return:

__deepcopy__(*self*, *memo*)

Calls self.copy() to return an independant copy of the model

Parameters

memo –

Returns

copy(*self*)

Pseudo-smart copy of the model using dict serialization. This builds a new model from the ground up, with independwnt variables, solver, etc.

Returns

etfl.data

Submodules

etfl.data.ecoli

Module Contents

Functions

clean_string(s)

get_model(solver)

get_thermo_data()

get_essentials()

get_neidhardt_data()

get_nt_sequences()

get_ecoli_gen_stats()

get_ratios()

get_monomers_dict()

continues on next page

Table 1 – continued from previous page

remove_from_biomass_equation(model, nt_dict, aa_dict, essentials_dict)	
get_mrna_metrics()	
get_enz_metrics()	
is_gpr(s)	
get_homomer_coupling_dict(model, mode='kcat')	
get_rate_constant(reaction, k_info, k_column, n_column)	
ec2ecocyc(ec_number)	
score_against_genes(putative_genes, reaction_genes)	
match_ec_genes_ecocyc(ecocyc, genes, threshold=0.5)	
ecocyc2composition(ecocyc)	
complex2composition(complex_name)	
ec2kcat(ec_number)	
check_id_in_reaction_list(the_id, df)	
get_aggregated_coupling_dict(model, coupling_dict=dict())	
get_lloyd_keffs()	
get_keffs_from_complex_name(keffs, name)	
get_lloyd_coupling_dict(model, select=None)	
get_coupling_dict(model, mode, atps_name=None, infer_missing_enz=False)	
get_average_kcat()	
get_atp_synthase_coupling(atps_name)	ATP synthesis rate of F1F0 ATP synthase
get_dna_polymerase(dna_pol_name='DNAPol3')	https://en.wikipedia.org/wiki/DNA_polymerase_III_holoenzyme
get_transporters_coupling(model, additional_enz)	
get_mrna_dict(model)	
get_rib()	# Ribosome
get_rnap()	# RNAP
get_sigma_70(rnap)	# RNAP
read_growth_dependant_rnap_alloc()	Read table with data on , the fraction of RNAP holoenzyme. We define :

continues on next page

Table 1 – continued from previous page

get_growth_dependant_transformed_rnap_alloc()	We are given the active RNAP ratio , which we approximate to be

Attributes

file_dir
data_dir
nt_sequences
kcat_info_milo
kmax_info_milo
kcat_info_aggregated
ec_info_ecocyc
composition_info_ecocyc
reaction2complexes_info_obrien
complexes2peptides_info_obrien
reaction2complexes_info_lloyd
columns
complexes2peptides_info_lloyd
columns
gene_names
bernstein_ecoli_deg_rates
gc_ratio
chromosome_len
kdeg_enz
kdeg_mrna
mrna_length_avg
peptide_length_avg
comp_regex
kdeg_rib
rrna_genes
ktrans

etfl.data.ecoli.**clean_string**(*s*)

etfl.data.ecoli.**file_dir**

etfl.data.ecoli.**data_dir**

etfl.data.ecoli.**get_model**(*solver*)

etfl.data.ecoli.**get_thermo_data**()

etfl.data.ecoli.**get_essentials**()

etfl.data.ecoli.**get_neidhardt_data**()

etfl.data.ecoli.**nt_sequences**

etfl.data.ecoli.**get_nt_sequences**()

etfl.data.ecoli.**kcat_info_milo**

etfl.data.ecoli.**kmax_info_milo**

etfl.data.ecoli.**kcat_info_aggregated**

etfl.data.ecoli.**ec_info_ecocyc**

etfl.data.ecoli.**composition_info_ecocyc**

etfl.data.ecoli.**reaction2complexes_info_obrien**

etfl.data.ecoli.**complexes2peptides_info_obrien**

etfl.data.ecoli.**reaction2complexes_info_lloyd**

etfl.data.ecoli.**columns = ['Enzymes']**

etfl.data.ecoli.**complexes2peptides_info_lloyd**

etfl.data.ecoli.**columns = ['Gene composition']**

etfl.data.ecoli.**gene_names**

etfl.data.ecoli.**bernstein_ecoli_deg_rates**

etfl.data.ecoli.**gc_ratio = 0.5078**

etfl.data.ecoli.**chromosome_len = 4639675**

etfl.data.ecoli.**get_ecoli_gen_stats**()

etfl.data.ecoli.**get_ratios**()

etfl.data.ecoli.**get_monomers_dict**()

etfl.data.ecoli.**remove_from_biomass_equation**(*model*, *nt_dict*, *aa_dict*, *essentials_dict*)

etfl.data.ecoli.**kdeg_enz**

etfl.data.ecoli.**kdeg_mrna**

etfl.data.ecoli.**mrna_length_avg = 1000**

etfl.data.ecoli.**peptide_length_avg**

etfl.data.ecoli.**get_mrna_metrics**()

etfl.data.ecoli.**get_enz_metrics**()

etfl.data.ecoli.**is_gpr**(*s*)

etfl.data.ecoli.**get_homomer_coupling_dict**(*model*, *mode='kcat'*)

etfl.data.ecoli.**get_rate_constant**(*reaction*, *k_info*, *k_column*, *n_column*)

etfl.data.ecoli.**ec2ecocyc**(*ec_number*)

etfl.data.ecoli.**score_against_genes**(*putative_genes*, *reaction_genes*)

etfl.data.ecoli.**match_ec_genes_ecocyc**(*ecocyc*, *genes*, *threshold=0.5*)

etfl.data.ecoli.**ecocyc2composition**(*ecocyc*)

etfl.data.ecoli.**comp_regex**

etfl.data.ecoli.**complex2composition**(*complex_name*)

etfl.data.ecoli.**ec2kcat**(*ec_number*)

etfl.data.ecoli.**check_id_in_reaction_list**(*the_id*, *df*)

etfl.data.ecoli.**get_aggregated_coupling_dict**(*model*, *coupling_dict=dict()*)

etfl.data.ecoli.**get_lloyd_keffs**()

etfl.data.ecoli.**get_keffs_from_complex_name**(*keffs*, *name*)

etfl.data.ecoli.**get_lloyd_coupling_dict**(*model*, *select=None*)

etfl.data.ecoli.**get_coupling_dict**(*model*, *mode*, *atps_name=None*, *infer_missing_enz=False*)

etfl.data.ecoli.**get_average_kcat**()

etfl.data.ecoli.**get_atp_synthase_coupling**(*atps_name*)

ATP synthesis rate of F1F0 ATP synthase Range at room temperature 0.060-0.10 mol/min/mg of membrane protein : at 37°C 0.20 mol/min/mg of membrane protein Organism Bacteria Escherichia coli Reference Tomashek JJ, Glagoleva OB, Brusilow WS. The Escherichia coli F1F0 ATP synthase displays biphasic synthesis kinetics. J Biol Chem. 2004 Feb 6 279(6):4465-70 DOI: 10.1074/jbc.M310826200 p.4467 right column bottom paragraphPubMed ID14602713 Primary Source [18] Etzold C, Deckers-Hebestreit G, Altendorf K. Turnover number of Escherichia coli F0F1 ATP synthase for ATP synthesis in membrane vesicles. Eur J Biochem. 1997 Jan 15 243(1-2):336-43.PubMed ID9030757 Method Luciferase assay Comments P.4467 right column bottom paragraph: "Previously, Etzold et al. (primary source) used the luciferase assay to measure the turnover number of the ATP synthase during synthesis by membrane vesicles of E. coli. They measured ATP synthesis rates of 0.060-0.10 mol/min/mg of membrane protein at room temperature and 0.20 mol/min/mg of membrane protein at 37 °C." Entered by Uri M ID 115175 :return:

etfl.data.ecoli.**get_dna_polymerase**(*dna_pol_name='DNAPol3'*)

https://en.wikipedia.org/wiki/DNA_polymerase_III_holoenzyme

The replisome is composed of the following:

2 DNA Pol III enzymes, each comprising , and subunits. (It has been proven that there is a third copy of Pol III at the replisome.[1])

the subunit (encoded by the dnaE gene) has the polymerase activity. the subunit (dnaQ) has 3'→5' exonuclease activity. the subunit (holE) stimulates the subunit's proofreading.

2 units (dnaN) which act as sliding DNA clamps, they keep the polymerase bound to the DNA. 2 units (dnaX) which act to dimerize two of the core enzymes (, , and subunits). 1 unit (also dnaX) which acts as a clamp loader for the lagging strand Okazaki fragments, helping the two subunits to form a unit and bind to DNA. The unit is made up of 5 subunits which include 3 subunits, 1 subunit (holA), and 1 ' subunit (holB). The is involved in copying of the lagging strand. (holC) and (holD) which form a 1:1 complex and bind to or . X can also mediate the switch from RNA primer to DNA.[2] :return:

etfl.data.ecoli.**get_transporters_coupling**(*model*, *additional_enz*)

etfl.data.ecoli.**get_mrna_dict**(*model*)

etfl.data.ecoli.**kdeg_rib**

etfl.data.ecoli.**rrna_genes = ['b3851', 'b3854', 'b3855']**

etfl.data.ecoli.**get_rib**()

Ribosome

rRNA: b3851: K01977 16S ribosomal RNA | (RefSeq) rrsA; 16S ribosomal RNA of rrnA operon b3854: K01980 23S ribosomal RNA | (RefSeq) rrlA; 23S ribosomal RNA of rrnA operon b3855: K01985 5S ribosomal RNA | (RefSeq) rrfA; 5S ribosomal RNA of rrnA operon # rPeptides: See file ribosomal_proteins_ecoli.tsv

Returns

etfl.data.ecoli.**ktrans = 85**

etfl.data.ecoli.**get_rnap**()

RNAP

b3295: K03040 DNA-directed RNA polymerase subunit alpha [EC:2.7.7.6] | (RefSeq) rpoA; RNA polymerase, alpha subunit b3649: K03060 DNA-directed RNA polymerase subunit omega [EC:2.7.7.6] | (RefSeq) rpoZ; RNA polymerase, omega subunit b3987: K03043 DNA-directed RNA polymerase subunit beta [EC:2.7.7.6] | (RefSeq) rpoB; RNA polymerase, beta subunit b3988: K03046 DNA-directed RNA polymerase subunit beta' [EC:2.7.7.6] | (RefSeq) rpoC; RNA polymerase, beta prime subunit

Returns

etfl.data.ecoli.**get_sigma_70**(*rnap*)

RNAP

b3067: rpoD :return:

etfl.data.ecoli.**read_growth_dependant_rnap_alloc**()

Read table with data on , the fraction of RNAP holoenzyme. We define : = holoRNAP / RNAP_total = holoRNAP / (holoRNAP + RNAP_free) :return:

etfl.data.ecoli.**get_growth_dependant_transformed_rnap_alloc**()

We are given the active RNAP ratio , which we approximate to be

= holoRNAP / RNAP_total = holoRNAP / (holoRNAP + RNAP_free)

For our calculations, we are interested in q = holoRNAP / RNAP_free

= holoRNAP / (holoRNAP + RNAP_free) <=> 1/ = 1 + 1/q <=> 1/ - 1 = 1/q <=> /(1 -) = q

Returns

etfl.data.ecoli_utils

Module Contents

Functions

infer_enzyme_from_gpr(reaction, default_kcat, default_kdeg)	
compositions_from_gpr(reaction)	*Warning*: Use this function only if you have no information on the enzymes.

etfl.data.ecoli_utils.**infer_enzyme_from_gpr**(*reaction*, *default_kcat*, *default_kdeg*)

etfl.data.ecoli_utils.**compositions_from_gpr**(*reaction*)

Warning: Use this function only if you have no information on the enzymes. Logically parses the GPR to automatically find isozymes (logical OR) and subunits (logical AND), and creates the necessary complexation reactions: 1 per isozyme, requiring the peptides of each subunit

Parameters

reaction (*cobra.Reaction*) –

Returns

etfl.debugging

Submodules

etfl.debugging.debugging

Module Contents

Functions

localize_exp(exp)	Takes an optlang expression, and replaces symbols (tied to variables) by
compare_expressions(exp1, exp2)	Check is the two given expressions are equal
find_different_constraints(model1, model2)	Given two models, find which expressions are different
find_translation_gaps(model)	For each translation constraint in the model, finds the value of each
find_essentials_from(model, met_dict)	Given a dictionnary of {met_id:uptake_reaction}, checks the value of the
get_model_argument(args, kwargs, arg_index=0)	Utility function to get the model object from the arguments of a function
save_objective_function(fun)	Decorator to restore the objective function after the execution of the
save_growth_bounds(fun)	Decorator to save the growth bound and restore them after the execution of
perform_iMM(model, uptake_dict, min_growth_coef=0.5, bigM=1000)	An implementation of the *in silico Minimal Media* methods, which uses MILP
check_production_of_mets(model, met_ids)	for each metabolite ID given, create a sink and maximize the production of
relax_catalytic_constraints(model, min_growth)	Find a minimal set of catalytic constraints to relax to meet a minimum
relax_catalytic_constraints_bkwd(model, min_growth)	Find a minimal set of catalytic constraints to relax to meet a minimum

etfl.debugging.debugging.**localize_exp**(*exp*)

Takes an optlang expression, and replaces symbols (tied to variables) by their string names, to compare expressions of two different models

Parameters

exp (optlang.symbolics.Expr) –

Returns

etfl.debugging.debugging.**compare_expressions**(*exp1*, *exp2*)

Check is the two given expressions are equal

Parameters

- **exp1** (optlang.symbolics.Expr) –
- **exp2** (optlang.symbolics.Expr) –

Returns

etfl.debugging.debugging.**find_different_constraints**(*model1*, *model2*)

Given two models, find which expressions are different

Parameters

- **model1** –
- **model2** –

Returns

etfl.debugging.debugging.**find_translation_gaps**(*model*)

For each translation constraint in the model, finds the value of each variable, and then evaluates the LHS of the constraint

Constraints look like v_tsl - ktrans/L [RNAP_i] <= 0

Parameters
model –

Returns

etfl.debugging.debugging.**find_essentials_from**(*model*, *met_dict*)

Given a dictionnary of {met_id:uptake_reaction}, checks the value of the objective function at optimality when the given uptake is closed.

Uptake reactions are expected to be aligned according to the consensus directionality for systems : met_e <=> []

Parameters

- **model** –
- **met_dict** –

Returns

etfl.debugging.debugging.**get_model_argument**(*args*, *kwargs*, *arg_index=0*)

Utility function to get the model object from the arguments of a function

Parameters

- **args** –
- **kwargs** –
- **arg_index** –

Returns

etfl.debugging.debugging.**save_objective_function**(*fun*)

Decorator to restore the objective function after the execution of the decorated function.

Parameters
fun –

Returns

etfl.debugging.debugging.**save_growth_bounds**(*fun*)

Decorator to save the growth bound and restore them after the execution of the decorated function.

Parameters
fun –

Returns

etfl.debugging.debugging.**perform_iMM**(*model*, *uptake_dict*, *min_growth_coef=0.5*, *bigM=1000*)

An implementation of the *in silico Minimal Media* methods, which uses MILP to find a minimum set of uptakes necessary to meet growth requirements

See:

Bioenergetics-based modeling of Plasmodium falciparum metabolism reveals its essential genes, nutritional requirements, and thermodynamic bottlenecks Chiappino-Pepe A, Tymoshenko S, Ataman M, Soldati-Favre D, Hatzimanikatis V (2017) PLOS Computational Biology 13(3): e1005397. https://doi.org/10.1371/journal.pcbi.1005397

Parameters

- **model** (etfl.core.memodel.MEModel) –
- **uptake_dict** – {met_id : <reaction object>}
- **min_growth_coef** – minimum fraction of optimal growth to be met
- **bigM** – a big-M value for the optimization problem

Returns

etfl.debugging.debugging.**check_production_of_mets**(*model*, *met_ids*)

for each metabolite ID given, create a sink and maximize the production of the metabolite

Parameters

- **model** (etfl.core.memodel.MEModel) –
- **met_ids** –

Returns

etfl.debugging.debugging.**relax_catalytic_constraints**(*model*, *min_growth*)

Find a minimal set of catalytic constraints to relax to meet a minimum growth criterion

Parameters

- **model** (etfl.core.memodel.MEModel) –
- **min_growth** –

Returns

etfl.debugging.debugging.**relax_catalytic_constraints_bkwd**(*model*, *min_growth*)

Find a minimal set of catalytic constraints to relax to meet a minimum growth criterion

Parameters

- **model** (etfl.core.memodel.MEModel) –
- **min_growth** –

Returns

Package Contents

Classes

CatalyticConstraint	Class to represent a enzymatic constraint
CatalyticActivator	Class to represent a binary variable that activates a catalytic constraint
ForwardCatalyticConstraint	Class to represent a enzymatic constraint
BackwardCatalyticConstraint	Class to represent a enzymatic constraint

Functions

localize_exp(exp)	Takes an optlang expression, and replaces symbols (tied to variables) by
compare_expressions(exp1, exp2)	Check is the two given expressions are equal
find_different_constraints(model1, model2)	Given two models, find which expressions are different
find_translation_gaps(model)	For each translation constraint in the model, finds the value of each
find_essentials_from(model, met_dict)	Given a dictionnary of {met_id:uptake_reaction}, checks the value of the
get_model_argument(args, kwargs, arg_index=0)	Utility function to get the model object from the arguments of a function
save_objective_function(fun)	Decorator to restore the objective function after the execution of the
save_growth_bounds(fun)	Decorator to save the growth bound and restore them after the execution of
perform_iMM(model, uptake_dict, min_growth_coef=0.5, bigM=1000)	An implementation of the *in silico Minimal Media* methods, which uses MILP
check_production_of_mets(model, met_ids)	for each metabolite ID given, create a sink and maximize the production of
relax_catalytic_constraints(model, min_growth)	Find a minimal set of catalytic constraints to relax to meet a minimum
relax_catalytic_constraints_bkwd(model, min_growth)	Find a minimal set of catalytic constraints to relax to meet a minimum

class etfl.debugging.**CatalyticConstraint**

Bases: pytfa.optim.ReactionConstraint

Class to represent a enzymatic constraint

prefix = CC_

class etfl.debugging.**CatalyticActivator**(*reaction*, ***kwargs*)

Bases: pytfa.optim.variables.ReactionVariable, pytfa.optim.variables.BinaryVariable

Class to represent a binary variable that activates a catalytic constraint or relaxes it

prefix = CA_

class etfl.debugging.**ForwardCatalyticConstraint**

Bases: pytfa.optim.ReactionConstraint

Class to represent a enzymatic constraint

prefix = FC_

class etfl.debugging.**BackwardCatalyticConstraint**

Bases: pytfa.optim.ReactionConstraint

Class to represent a enzymatic constraint

prefix = BC_

etfl.debugging.**localize_exp**(*exp*)

Takes an optlang expression, and replaces symbols (tied to variables) by their string names, to compare expressions of two different models

Parameters
exp (optlang.symbolics.Expr) –

Returns

etfl.debugging.**compare_expressions**(*exp1*, *exp2*)

Check is the two given expressions are equal

Parameters

- **exp1** (optlang.symbolics.Expr) –
- **exp2** (optlang.symbolics.Expr) –

Returns

etfl.debugging.**find_different_constraints**(*model1*, *model2*)

Given two models, find which expressions are different

Parameters

- **model1** –
- **model2** –

Returns

etfl.debugging.**find_translation_gaps**(*model*)

For each translation constraint in the model, finds the value of each variable, and then evaluates the LHS of the constraint

Constraints look like v_tsl - ktrans/L [RNAP_i] <= 0

Parameters
model –

Returns

etfl.debugging.**find_essentials_from**(*model*, *met_dict*)

Given a dictionnary of {met_id:uptake_reaction}, checks the value of the objective function at optimality when the given uptake is closed.

Uptake reactions are expected to be aligned according to the consensus directionality for systems : met_e <=> []

Parameters

- **model** –
- **met_dict** –

Returns

etfl.debugging.**get_model_argument**(*args*, *kwargs*, *arg_index=0*)

Utility function to get the model object from the arguments of a function

Parameters

- **args** –
- **kwargs** –
- **arg_index** –

Returns

etfl.debugging.**save_objective_function**(*fun*)

Decorator to restore the objective function after the execution of the decorated function.

Parameters
fun –

Returns

etfl.debugging.**save_growth_bounds**(*fun*)

Decorator to save the growth bound and restore them after the execution of the decorated function.

Parameters
fun –

Returns

etfl.debugging.**perform_iMM**(*model*, *uptake_dict*, *min_growth_coef=0.5*, *bigM=1000*)

An implementation of the *in silico Minimal Media* methods, which uses MILP to find a minimum set of uptakes necessary to meet growth requirements

See:

Bioenergetics-based modeling of Plasmodium falciparum metabolism reveals its essential genes, nutritional requirements, and thermodynamic bottlenecks Chiappino-Pepe A, Tymoshenko S, Ataman M, Soldati-Favre D, Hatzimanikatis V (2017) PLOS Computational Biology 13(3): e1005397. https://doi.org/10.1371/journal.pcbi.1005397

Parameters

- **model** (etfl.core.memodel.MEModel) –
- **uptake_dict** – {met_id : <reaction object>}
- **min_growth_coef** – minimum fraction of optimal growth to be met
- **bigM** – a big-M value for the optimization problem

Returns

etfl.debugging.**check_production_of_mets**(*model*, *met_ids*)

for each metabolite ID given, create a sink and maximize the production of the metabolite

Parameters

- **model** (etfl.core.memodel.MEModel) –
- **met_ids** –

Returns

etfl.debugging.**relax_catalytic_constraints**(*model*, *min_growth*)

Find a minimal set of catalytic constraints to relax to meet a minimum growth criterion

Parameters

- **model** (etfl.core.memodel.MEModel) –
- **min_growth** –

Returns

etfl.debugging.**relax_catalytic_constraints_bkwd**(*model*, *min_growth*)

Find a minimal set of catalytic constraints to relax to meet a minimum growth criterion

Parameters

- **model** (etfl.core.memodel.MEModel) –
- **min_growth** –

Returns

etfl.integration

Submodules

etfl.integration.transcriptomics

Module Contents

Classes

RelativeTranscriptomicsLB	Represents a lower bound on mRNA ratio in relative transcriptomics
RelativeTranscriptomicsUB	Represents an upper bound on mRNA ratio in relative transcriptomics
ReferenceLevel	Represents the reference level for relative transcriptomics

Functions

integrate_relative_transcriptomics(model, lower_bounds, upper_bounds, base=2)	Integrates log-ratio expression data to mRNA levels in ETFL

class etfl.integration.transcriptomics.**RelativeTranscriptomicsLB**

Bases: etfl.optim.constraints.GeneConstraint

Represents a lower bound on mRNA ratio in relative transcriptomics

prefix = RTL_

class etfl.integration.transcriptomics.**RelativeTranscriptomicsUB**

Bases: etfl.optim.constraints.GeneConstraint

Represents an upper bound on mRNA ratio in relative transcriptomics

prefix = RTU_

class etfl.integration.transcriptomics.**ReferenceLevel**

Bases: etfl.optim.variables.ModelVariable

Represents the reference level for relative transcriptomics

prefix = RL_

etfl.integration.transcriptomics.**integrate_relative_transcriptomics**(*model*, *lower_bounds*, *upper_bounds*, *base=2*)

Integrates log-ratio expression data to mRNA levels in ETFL

Parameters

- **model** (*etfl.core.memodel.MEModel*) – an ETFL model
- **lower_bounds** (*dict or pandas.Series*) –
- **upper_bounds** (*dict or pandas.Series*) –

Returns

etfl.io

Submodules

etfl.io.dict

Make the model serializable

Module Contents

Functions

metabolite_thermo_to_dict(metthermo)	
expressed_gene_to_dict(gene)	
coding_gene_to_dict(gene)	
enzyme_to_dict(enzyme)	
mrna_to_dict(mrna)	
ribosome_to_dict(ribosome)	
_single_ribosome_to_dict(ribosome)	
rnap_to_dict(rnap)	
_single_rnap_to_dict(rnap)	
dna_to_dict(dna)	
archive_variables(var_dict)	
archive_constraints(cons_dict)	
archive_compositions(compositions)	Turns a peptide compositions dict of the form:
_stoichiometry_to_dict(stoichiometric_dict)	Turns a stoichiometric compositions dict of the form:
archive_coupling_dict(coupling_dict)	Turns an enzyme coupling dict of the form:
archive_trna_dict(model)	Turns a tNA information dict of the form:

continues on next page

Table 2 – continued from previous page

get_solver_string(model)	
model_to_dict(model) **param model**	
_add_me_reaction_info(rxn, rxn_dict)	
_add_thermo_reaction_info(rxn, rxn_dict)	
_add_thermo_metabolite_info(met, met_dict)	
model_from_dict(obj, solver=None)	
prostprocess_me(new)	
init_me_model_from_dict(new, obj)	
init_thermo_model_from_dict(new, obj)	
init_thermo_me_model_from_dict(new, obj)	
rebuild_compositions(new, compositions_dict)	Performs the reverse operation of :func:archive_compositions
_rebuild_stoichiometry(new, stoich)	Performs the reverse operation of :func:_stoichiometry_to_dict
rebuild_coupling_dict(new, coupling_dict)	Performs the reverse operation of :func:archive_coupling_dict
enzyme_from_dict(obj)	
mrna_from_dict(obj)	
ribosome_from_dict(obj)	
_single_ribosome_from_dict(obj)	
rnap_from_dict(obj)	
_single_rnap_from_dict(obj)	
dna_from_dict(obj)	
find_enzymatic_reactions_from_dict(new, obj)	
find_translation_reactions_from_dict(new, obj)	
find_transcription_reactions_from_dict(new, obj)	
find_complexation_reactions_from_dict(new, obj)	
link_enzyme_complexation(new, obj)	

continues on next page

Table 2 – continued from previous page

find_degradation_reactions_from_dict(new, obj)
find_dna_formation_reaction_from_dict(new, obj)
find_peptides_from_dict(new, obj)
find_rrna_from_dict(new, obj)
rebuild_trna(new, obj)
find_genes_from_dict(new, obj)

Attributes

SOLVER_DICT
MW_OVERRIDE_KEY

etfl.**SOLVER_DICT**

etfl.**MW_OVERRIDE_KEY = molecular_weight_override**

etfl.**metabolite_thermo_to_dict**(*metthermo*)

etfl.**expressed_gene_to_dict**(*gene*)

etfl.**coding_gene_to_dict**(*gene*)

etfl.**enzyme_to_dict**(*enzyme*)

etfl.**mrna_to_dict**(*mrna*)

etfl.**ribosome_to_dict**(*ribosome*)

etfl.**_single_ribosome_to_dict**(*ribosome*)

etfl.**rnap_to_dict**(*rnap*)

etfl.**_single_rnap_to_dict**(*rnap*)

etfl.**dna_to_dict**(*dna*)

etfl.**archive_variables**(*var_dict*)

etfl.**archive_constraints**(*cons_dict*)

etfl.**archive_compositions**(*compositions*)

Turns a peptide compositions dict of the form:

```
{ 'b3991': defaultdict(int,
         {<Metabolite ala__L_c at 0x7f7d25504f28>: -42,
          <Metabolite arg__L_c at 0x7f7d2550bcf8>: -11,
          <Metabolite asn__L_c at 0x7f7d2550beb8>: -6,
          ...}),
...}
```

to:

```
{ 'b3991': defaultdict(int,,
        {'ala__L_c': -42,
         'arg__L_c': -11,
         'asn__L_c': -6,
          ...}),
...}
```

Parameters
compositions –

Returns

etfl.**_stoichiometry_to_dict**(*stoichiometric_dict*)

Turns a stoichiometric compositions dict of the form:

```
'b3991': defaultdict(int,
       {<Metabolite ala__L_c at 0x7f7d25504f28>: -42,
        <Metabolite arg__L_c at 0x7f7d2550bcf8>: -11,
        <Metabolite asn__L_c at 0x7f7d2550beb8>: -6,
        ...})
```

to:

```
'b3991': defaultdict(int,,
        {'ala__L_c': -42,
         'arg__L_c': -11,
         'asn__L_c': -6,
          ...})
```

etfl.**archive_coupling_dict**(*coupling_dict*)

Turns an enzyme coupling dict of the form:

```
{'AB6PGH': <Enzyme AB6PGH at 0x7f7d1371add8>,
 'ABTA': <Enzyme ABTA at 0x7f7d1371ae48>,
 'ACALD': <Enzyme ACALD at 0x7f7d1371aeb8>}
```

to:

```
{'AB6PGH': 'AB6PGH',
 'ABTA': 'ABTA',
 'ACALD': 'ACALD'
```

etfl.**archive_trna_dict**(*model*)

Turns a tNA information dict of the form:

```
{'ala__L_c': (<tRNA charged_tRNA_ala__L_c at 0x7f84c16d07b8>,
              <tRNA uncharged_tRNA_ala__L_c at 0x7f84c16d0be0>,
              <Reaction trna_ch_ala__L_c at 0x7f84c16d0978>),
 'arg__L_c': (<tRNA charged_tRNA_arg__L_c at 0x7f84c169b588>,
              <tRNA uncharged_tRNA_arg__L_c at 0x7f84c169b5f8>,
              <Reaction trna_ch_arg__L_c at 0x7f84c0563ef0>)}
```

to:

```
{'ala__L_c': ('charged_tRNA_ala__L_c',
              'uncharged_tRNA_ala__L_c',
              'trna_ch_ala__L_c'),
 'arg__L_c': ('charged_tRNA_arg__L_c',
              'uncharged_tRNA_arg__L_c',
              'trna_ch_arg__L_c')}
```

etfl.**get_solver_string**(*model*)

etfl.**model_to_dict**(*model*)

Parameters
model –

Returns

etfl.**_add_me_reaction_info**(*rxn*, *rxn_dict*)

etfl.**_add_thermo_reaction_info**(*rxn*, *rxn_dict*)

etfl.**_add_thermo_metabolite_info**(*met*, *met_dict*)

etfl.**model_from_dict**(*obj*, *solver=None*)

etfl.**prostprocess_me**(*new*)

etfl.**init_me_model_from_dict**(*new*, *obj*)

etfl.**init_thermo_model_from_dict**(*new*, *obj*)

etfl.**init_thermo_me_model_from_dict**(*new*, *obj*)

etfl.**rebuild_compositions**(*new*, *compositions_dict*)

Performs the reverse operation of :func:archive_compositions

Parameters

- **new** –
- **compositions_dict** –

Returns

etfl.**_rebuild_stoichiometry**(*new*, *stoich*)

Performs the reverse operation of :func:_stoichiometry_to_dict

Parameters

- **new** –
- **stoich** –

Returns

etfl.**rebuild_coupling_dict**(*new*, *coupling_dict*)

Performs the reverse operation of :func:archive_coupling_dict

Parameters

- **new** –
- **coupling_dict** –

Returns

etfl.**enzyme_from_dict**(*obj*)

etfl.**mrna_from_dict**(*obj*)

etfl.**ribosome_from_dict**(*obj*)

etfl.**_single_ribosome_from_dict**(*obj*)

etfl.**rnap_from_dict**(*obj*)

etfl.**_single_rnap_from_dict**(*obj*)

etfl.**dna_from_dict**(*obj*)

etfl.**find_enzymatic_reactions_from_dict**(*new*, *obj*)

etfl.**find_translation_reactions_from_dict**(*new*, *obj*)

etfl.**find_transcription_reactions_from_dict**(*new*, *obj*)

etfl.**find_complexation_reactions_from_dict**(*new*, *obj*)

etfl.**link_enzyme_complexation**(*new*, *obj*)

etfl.**find_degradation_reactions_from_dict**(*new*, *obj*)

etfl.**find_dna_formation_reaction_from_dict**(*new*, *obj*)

etfl.**find_peptides_from_dict**(*new*, *obj*)

etfl.**find_rrna_from_dict**(*new*, *obj*)

etfl.**rebuild_trna**(*new*, *obj*)

etfl.**find_genes_from_dict**(*new*, *obj*)

etfl.io.json

JSON serialization

Module Contents

Functions

save_json_model(model, filepath)	Saves the model as a JSON file
load_json_model(filepath, solver=None)	Loads a model from a JSON file
json_dumps_model(model)	Returns a JSON dump as a string
json_loads_model(s)	Loads a model from a string JSON dump

etfl.**save_json_model**(*model*, *filepath*)

Saves the model as a JSON file

Parameters

- **model** –
- **filepath** –

Returns

etfl.**load_json_model**(*filepath*, *solver=None*)

Loads a model from a JSON file

Parameters

- **filepath** –
- **solver** –

Returns

etfl.**json_dumps_model**(*model*)

Returns a JSON dump as a string

Parameters
model –

Returns

etfl.**json_loads_model**(*s*)

Loads a model from a string JSON dump

Parameters
s – JSON string

Returns

etfl.optim

Submodules

etfl.optim.config

Solver configuration helpers

Module Contents

Functions

standard_solver_config(model, verbose=True)	Basic solver settings for ETFL
gene_ko_config(model)	Solver settings for performing gene KO. Tuned using the grbtune tool on the
growth_uptake_config(model)	Solver settings for performing growth vs uptake studies. Tuned using the
redhuman_config(model)	Solver settings for optimizing growth on human cancer models. Tuned using the

ETFL.**standard_solver_config**(*model*, *verbose=True*)

Basic solver settings for ETFL :param model: :param verbose: :return:

ETFL.**gene_ko_config**(*model*)

Solver settings for performing gene KO. Tuned using the grbtune tool on the vETFL model iJO1366. The gene KO analysis is turned into a feasibility problem by putting a lower bound on growth.

Parameters
model –

Returns

ETFL.**growth_uptake_config**(*model*)

Solver settings for performing growth vs uptake studies. Tuned using the grbtune tool on the vETFL model iJO1366.

Parameters
model –

Returns

ETFL.**redhuman_config**(*model*)

Solver settings for optimizing growth on human cancer models. Tuned using the grbtune tool on the vETFL model from reduced RECON3.

Parameters
model –

Returns

etfl.optim.constraints

Constraints declarations

Module Contents

Classes

CatalyticConstraint	Class to represent a enzymatic constraint
ForwardCatalyticConstraint	Class to represent a enzymatic constraint
BackwardCatalyticConstraint	Class to represent a enzymatic constraint
EnzymeConstraint	Class to represent a variable attached to a enzyme
EnzymeMassBalance	Class to represent a enzymatic mass balance constraint
mRNAMassBalance	Class to represent a mRNA mass balance constraint
rRNAMassBalance	Class to represent a mRNA mass balance constraint
tRNAMassBalance	Class to represent a tRNA mass balance constraint
DNAMassBalance	Class to represent a DNA mass balance constraint
SynthesisConstraint	Class to represent a Translation constraint
GrowthCoupling	Class to represent a growth capacity constraint
TotalCapacity	Class to represent the total capacity of constraint of a species, e.g
TotalEnzyme	Class to represent the total amount of an enzyme species, forwards and backwards
ExpressionCoupling	Add the coupling between mRNA availability and ribosome charging
MinimalCoupling	Add the minimal activity of ribosome based on the availability of mRNA.
RNAPAllocation	Add the coupling between DNA availability and RNAP charging
MinimalAllocation	Add the minimal activity of RNAP based on the availability of gene.
EnzymeRatio	Represents the availability of free enzymes, e.g ribosomes (non bound)
RibosomeRatio	(Legacy) represents the availability of free ribosomes, e.g ribosomes (non bound)
EnzymeDegradation	v_deg = k_deg [E]
mRNADegradation	v_deg = k_deg [mRNA]
GrowthChoice	Class to represent a variable attached to a reaction
LinearizationConstraint	Class to represent a variable attached to a reaction
SOS1Constraint	Class to represent SOS 1 constraint
InterpolationConstraint	Class to represent an interpolation constraint
EnzymeDeltaPos	Represents a positive enzyme concentration variation for dETFL
EnzymeDeltaNeg	Represents a negative enzyme concentration variation for dETFL
mRNADeltaPos	Represents a positive mRNA concentration variation for dETFL
mRNADeltaNeg	Represents a negative mRNA concentration variation for dETFL
ConstantAllocation	Represents a similar share to FBA for RNA and protein
LipidMassBalance	Class to represent a lipid mass balance constraint
CarbohydrateMassBalance	Class to represent a carbohydrate mass balance constraint
IonMassBalance	Class to represent a ion mass balance constraint

class ETFL.**CatalyticConstraint**

Bases: pytfa.optim.ReactionConstraint

Class to represent a enzymatic constraint

prefix = CC_

class ETFL.**ForwardCatalyticConstraint**

Bases: pytfa.optim.ReactionConstraint

Class to represent a enzymatic constraint

prefix = FC_

class ETFL.**BackwardCatalyticConstraint**

Bases: pytfa.optim.ReactionConstraint

Class to represent a enzymatic constraint

prefix = BC_

class ETFL.**EnzymeConstraint**(*enzyme*, *expr*, ***kwargs*)

Bases: pytfa.optim.GenericConstraint

Class to represent a variable attached to a enzyme

prefix = EZ_

property enzyme(*self*)

property id(*self*)

property model(*self*)

class ETFL.**EnzymeMassBalance**(*enzyme*, *expr*, ***kwargs*)

Bases: *EnzymeConstraint*

Class to represent a enzymatic mass balance constraint

prefix = EB_

class ETFL.**mRNAMassBalance**

Bases: pytfa.optim.GeneConstraint

Class to represent a mRNA mass balance constraint

prefix = MB_

class ETFL.**rRNAMassBalance**

Bases: pytfa.optim.GeneConstraint

Class to represent a mRNA mass balance constraint

prefix = RB_

class ETFL.**tRNAMassBalance**

Bases: pytfa.optim.ModelConstraint

Class to represent a tRNA mass balance constraint

prefix = TB_

class ETFL.**DNAMassBalance**

Bases: pytfa.optim.ModelConstraint

Class to represent a DNA mass balance constraint

prefix = DB_

class ETFL.**SynthesisConstraint**

Bases: pytfa.optim.ReactionConstraint

Class to represent a Translation constraint

prefix = TR_

class ETFL.**GrowthCoupling**

Bases: pytfa.optim.ReactionConstraint

Class to represent a growth capacity constraint

prefix = GC_

class ETFL.**TotalCapacity**

Bases: pytfa.optim.ModelConstraint

Class to represent the total capacity of constraint of a species, e.g Ribosome or RNA

prefix = TC_

class ETFL.**TotalEnzyme**

Bases: *TotalCapacity*

Class to represent the total amount of an enzyme species, forwards and backwards

prefix = TE_

class ETFL.**ExpressionCoupling**

Bases: pytfa.optim.GeneConstraint

Add the coupling between mRNA availability and ribosome charging The number of ribosomes assigned to a mRNA species is lower than the number of such mRNA times the max number of ribosomes that can sit on the mRNA: [RPi] <= loadmax_i*[mRNAi]

prefix = EX_

class ETFL.**MinimalCoupling**

Bases: pytfa.optim.GeneConstraint

Add the minimal activity of ribosome based on the availability of mRNA. We modeled it as a fraction of the maximum loadmax and the fraction depends on the affinity of ribosome to the mRNA: [RPi] >= Fraction*loadmax_i*[mRNAi]

prefix = MC_

class ETFL.**RNAPAllocation**

Bases: pytfa.optim.GeneConstraint

Add the coupling between DNA availability and RNAP charging The number of RNAP assigned to a gene locus is lower than the number of such loci times the max number of RNAP that can sit on the locus: [RNAPi] <= loadmax_i*[# of loci]*[DNA]

prefix = RA_

class ETFL.**MinimalAllocation**

Bases: pytfa.optim.GeneConstraint

Add the minimal activity of RNAP based on the availability of gene. We modeled it as a fraction of the maximum loadmax and the fraction depends on the affinity of RNAP to the gene, i.e. the strength of the promoter: [RPi] >= Fraction*loadmax_i*[mRNAi]

prefix = MA_

class ETFL.**EnzymeRatio**(*enzyme*, *expr*, ***kwargs*)

Bases: *EnzymeConstraint*

Represents the availability of free enzymes, e.g ribosomes (non bound) R_free = 0.2*R_total

prefix = ER_

class ETFL.**RibosomeRatio**(*enzyme*, *expr*, ***kwargs*)

Bases: *EnzymeRatio*

(Legacy) represents the availability of free ribosomes, e.g ribosomes (non bound) R_free = 0.2*R_total

prefix = ER_

class ETFL.**EnzymeDegradation**(*enzyme*, *expr*, ***kwargs*)

Bases: *EnzymeConstraint*

v_deg = k_deg [E]

prefix = ED_

class ETFL.**mRNADegradation**

Bases: pytfa.optim.GeneConstraint

v_deg = k_deg [mRNA]

prefix = MD_

class ETFL.**GrowthChoice**

Bases: pytfa.optim.ModelConstraint

Class to represent a variable attached to a reaction

prefix = GR_

class ETFL.**LinearizationConstraint**

Bases: pytfa.optim.ModelConstraint

Class to represent a variable attached to a reaction

prefix = LC_

static from_constraints(*cons*, *model*)

class ETFL.**SOS1Constraint**

Bases: pytfa.optim.ModelConstraint

Class to represent SOS 1 constraint

prefix = S1_

class ETFL.**InterpolationConstraint**

Bases: pytfa.optim.ModelConstraint

Class to represent an interpolation constraint

prefix = IC_

class ETFL.**EnzymeDeltaPos**(*enzyme*, *expr*, ***kwargs*)

Bases: *EnzymeConstraint*

Represents a positive enzyme concentration variation for dETFL

prefix = dEP_

class ETFL.**EnzymeDeltaNeg**(*enzyme*, *expr*, ***kwargs*)

Bases: *EnzymeConstraint*

Represents a negative enzyme concentration variation for dETFL

prefix = dEN_

class ETFL.**mRNADeltaPos**

Bases: pytfa.optim.GeneConstraint

Represents a positive mRNA concentration variation for dETFL

prefix = dMP_

class ETFL.**mRNADeltaNeg**

Bases: pytfa.optim.GeneConstraint

Represents a negative mRNA concentration variation for dETFL

prefix = dMN_

class ETFL.**ConstantAllocation**

Bases: pytfa.optim.ModelConstraint

Represents a similar share to FBA for RNA and protein

prefix = CL_

class ETFL.**LipidMassBalance**

Bases: pytfa.optim.ModelConstraint

Class to represent a lipid mass balance constraint

prefix = LB_

class ETFL.**CarbohydrateMassBalance**

Bases: pytfa.optim.ModelConstraint

Class to represent a carbohydrate mass balance constraint

prefix = CB_

class ETFL.**IonMassBalance**

Bases: pytfa.optim.ModelConstraint

Class to represent a ion mass balance constraint

prefix = IB_

etfl.optim.utils

Optimisation utilities

Module Contents

Classes

SubclassIndexer	

Functions

make_subclasses_dict(cls)	Return a dictionary of the subclasses inheriting from the argument class.
fix_integers(model)	Fixes all integer and binary variables of a model, to make it sample-able
_gurobi_fix_integers(model)	If the solver of the model whose integers to fix has Gurobi as a solver,
_generic_fix_integers(model)	Fix the integers of a model to its solution, and removes the variables.
rebuild_variable(classname, model, this_id, lb, ub, scaling_factor, queue=True)	Rebuilds a variable from a classname and link it to the model
rebuild_constraint(classname, model, this_id, new_expr, lb, ub, queue=True)	Rebuilds a constraint from a classname and link it to the model
is_gurobi(model)	Check if the model uses Gurobi as a solver
fix_growth(model, solution=None)	Set the growth integers to their fixed values from a solution. If no
check_solution(model, solution)	Helper function. if solution is None, attempts to get it from the model.
release_growth(model)	After growth has been fixed by etfl.optim.utils.fix_growth(),
apply_warm_start(model, solution)	Gives a warm start to the model.
release_warm_start(model)	Releases the warm start provided by
get_active_growth_bounds(model, growth_rate=None)	Returns the growth bound closest to the growth flux calculated at the
safe_optim(model)	Catches *any* exception that can happen during solving, and logs it.
get_binding_constraints(model, epsilon)	

Attributes

INTEGER_VARIABLE_TYPES
DefaultSol

etfl.**make_subclasses_dict**(*cls*)

Return a dictionary of the subclasses inheriting from the argument class. Keys are String names of the classes, values the actual classes.

Parameters
cls –

Returns

class etfl.**SubclassIndexer**

__getitem__(*self*, *classtype*)

purge(*self*)

refresh(*self*)

etfl.**INTEGER_VARIABLE_TYPES = ['binary', 'integer']**

etfl.**fix_integers**(*model*)

Fixes all integer and binary variables of a model, to make it sample-able :param model: :return:

etfl.**_gurobi_fix_integers**(*model*)

If the solver of the model whose integers to fix has Gurobi as a solver, use the built-in method

Parameters
model – A model with a Gurobi backend

Returns

etfl.**_generic_fix_integers**(*model*)

Fix the integers of a model to its solution, and removes the variables.

Parameters
model –

Returns

etfl.**rebuild_variable**(*classname*, *model*, *this_id*, *lb*, *ub*, *scaling_factor*, *queue=True*)

Rebuilds a variable from a classname and link it to the model

Parameters

- **classname** –
- **model** –
- **this_id** –
- **lb** –
- **ub** –
- **queue** –

Returns

etfl.**rebuild_constraint**(*classname*, *model*, *this_id*, *new_expr*, *lb*, *ub*, *queue=True*)

Rebuilds a constraint from a classname and link it to the model

Parameters

- **classname** –
- **model** –
- **this_id** –
- **new_expr** –
- **lb** –
- **ub** –
- **queue** –

Returns

etfl.**DefaultSol**

etfl.**is_gurobi**(*model*)

Check if the model uses Gurobi as a solver

Parameters
model –

Returns

etfl.**fix_growth**(*model*, *solution=None*)

Set the growth integers to their fixed values from a solution. If no solution is provided, the model's latest solution is used. The growth can be released using the function `etfl.optim.utils.release_growth()`

Parameters

- **model** –
- **solution** –

Returns

etfl.**check_solution**(*model*, *solution*)

Helper function. if solution is None, attempts to get it from the model.

Parameters

- **model** –
- **solution** –

Returns

etfl.**release_growth**(*model*)

After growth has been fixed by `etfl.optim.utils.fix_growth()`, it can be released using this function.

Parameters
model –

Returns

etfl.**apply_warm_start**(*model*, *solution*)

Gives a warm start to the model. Release it with `etfl.optim.utils.release_warm_start()`.

Parameters

- **model** –
- **solution** –

Returns

etfl.**release_warm_start**(*model*)

Releases the warm start provided by `etfl.optim.utils.apply_warm_start()`.

Parameters
model –

Returns

etfl.**get_active_growth_bounds**(*model*, *growth_rate=None*)

Returns the growth bound closest to the growth flux calculated at the last solution.

Parameters
model –

Returns

etfl.**safe_optim**(*model*)

Catches *any* exception that can happen during solving, and logs it. Useful if you solve many problems in a sequence and some of then are infeasible. **Be careful** : This wil catch literally **any** Exception.

Parameters
model –

Returns

etfl.**get_binding_constraints**(*model*, *epsilon*)

etfl.optim.variables

Variables declarations

Module Contents

Classes

GrowthRate	Class to represent a growth rate
GrowthActivation	Class to represent a binary growth rate range activation in ME2 MILP
EnzymeVariable	Class to represent a enzyme variable
mRNAVariable	Class to represent a mRNA concentration
rRNAVariable	Class to represent a mRNA concentration
tRNAVariable	Class to represent a tRNA concentration
ForwardEnzyme	Represents assignment of an enzyme the a forward reaction flux
BackwardEnzyme	Represents assignment of an enzyme the a backward reaction flux
LinearizationVariable	Class to represent the product mu*[E] when performin linearization of the
DNAVariable	Class to represent DNA in the model
RibosomeUsage	Class to represent the ribosomes that are assigned to producing the enzyme
RNAPUsage	Class to represent the ribosomes that are assigned to producing the enzyme
FreeEnzyme	Class to represent the ribosomes that are affected to producing the enzyme
CatalyticActivator	Class to represent a binary variable that activates a catalytic constraint
BinaryActivator	Class to represent a binary variable that activates with growth levels
InterpolationVariable	Represents a variable that is interpolated
EnzymeRef	Represents a reference enzyme concentration - for example in dETFL
mRNARef	Represents a reference enzyme concentration - for example in dETFL
LipidVariable	Class to represent lipid in the model
CarbohydrateVariable	Class to represent carbohydrate in the model
IonVariable	Class to represent ion in the model

class ETFL.**GrowthRate**(*model*, ***kwargs*)

Bases: pytfa.optim.variables.ModelVariable

Class to represent a growth rate

prefix = MU_

class ETFL.**GrowthActivation**(*model*, *id_*, ***kwargs*)

Bases: pytfa.optim.variables.ModelVariable, pytfa.optim.variables.BinaryVariable

Class to represent a binary growth rate range activation in ME2 MILP

prefix = GA_

class ETFL.**EnzymeVariable**(*enzyme*, ***kwargs*)

Bases: pytfa.optim.variables.GenericVariable

Class to represent a enzyme variable

prefix = EZ_

property enzyme(*self*)

property id(*self*)

property model(*self*)

class ETFL.**mRNAVariable**

Bases: pytfa.optim.variables.GeneVariable

Class to represent a mRNA concentration

prefix = MR_

class ETFL.**rRNAVariable**

Bases: pytfa.optim.variables.GeneVariable

Class to represent a mRNA concentration

prefix = RR_

class ETFL.**tRNAVariable**

Bases: pytfa.optim.variables.ModelVariable

Class to represent a tRNA concentration

prefix = TR_

class ETFL.**ForwardEnzyme**(*enzyme*, ***kwargs*)

Bases: *EnzymeVariable*

Represents assignment of an enzyme the a forward reaction flux

prefix = FE_

class ETFL.**BackwardEnzyme**(*enzyme*, ***kwargs*)

Bases: *EnzymeVariable*

Represents assignment of an enzyme the a backward reaction flux

prefix = BE_

class ETFL.**LinearizationVariable**

Bases: pytfa.optim.variables.ModelVariable

Class to represent the product mu*[E] when performin linearization of the model

prefix = LZ_

class ETFL.**DNAVariable**

Bases: pytfa.optim.variables.ModelVariable

Class to represent DNA in the model

prefix = DN_

class ETFL.**RibosomeUsage**

Bases: pytfa.optim.variables.GeneVariable

Class to represent the ribosomes that are assigned to producing the enzyme for a reaction

prefix = RP_

class ETFL.**RNAPUsage**

Bases: pytfa.optim.variables.GeneVariable

Class to represent the ribosomes that are assigned to producing the enzyme for a reaction

prefix = RM_

class ETFL.**FreeEnzyme**(*enzyme*, ***kwargs*)

Bases: *EnzymeVariable*

Class to represent the ribosomes that are affected to producing the enzyme for a reaction

prefix = EF_

class ETFL.**CatalyticActivator**(*reaction*, ***kwargs*)

Bases: pytfa.optim.variables.ReactionVariable, pytfa.optim.variables.BinaryVariable

Class to represent a binary variable that activates a catalytic constraint or relaxes it

prefix = CA_

class ETFL.**BinaryActivator**(*model*, *id_*, ***kwargs*)

Bases: pytfa.optim.variables.ModelVariable, pytfa.optim.variables.BinaryVariable

Class to represent a binary variable that activates with growth levels

prefix = LA_

class ETFL.**InterpolationVariable**

Bases: pytfa.optim.variables.ModelVariable

Represents a variable that is interpolated

prefix = IV_

class ETFL.**EnzymeRef**(*enzyme*, ***kwargs*)

Bases: *EnzymeVariable*

Represents a reference enzyme concentration - for example in dETFL

prefix = EZ0_

class ETFL.**mRNARef**

Bases: *mRNAVariable*

Represents a reference enzyme concentration - for example in dETFL

prefix = MR0_

class ETFL.**LipidVariable**

Bases: pytfa.optim.variables.ModelVariable

Class to represent lipid in the model

prefix = LIP_

class ETFL.**CarbohydrateVariable**

Bases: pytfa.optim.variables.ModelVariable

Class to represent carbohydrate in the model

prefix = CAR_

class ETFL.IonVariable

Bases: pytfa.optim.variables.ModelVariable

Class to represent ion in the model

prefix = ION_

etfl.tests

Submodules

etfl.tests.small_model

Utilities to create a small model from a 1 reaction model in FBA

Module Contents

Functions

create_fba_model(solver=DEFAULT_SOLVER)	
add_e_metabolites(model)	Adds the metabolites necessary for the expression partof the problem:
create_etfl_model(has_thermo, has_neidhardt, n_mu_bins=64, mu_max=3, optimize=True, solver=DEFAULT_SOLVER)	
create_simple_dynamic_model()	

Attributes

CPLEX	
GUROBI	
GLPK	
DEFAULT_SOLVER	
essentials	
model	

ETFL.CPLEX = optlang-cplex

ETFL.**GUROBI = optlang-gurobi**

ETFL.**GLPK = optlang-glpk**

ETFL.**DEFAULT_SOLVER**

ETFL.**essentials**

ETFL.**create_fba_model**(*solver=DEFAULT_SOLVER*)

ETFL.**add_e_metabolites**(*model*)

Adds the metabolites necessary for the expression partof the problem: Amino acids, (d)N(M/T)Ps, PPi :param model: :return:

ETFL.**create_etfl_model**(*has_thermo*, *has_neidhardt*, *n_mu_bins=64*, *mu_max=3*, *optimize=True*, *solver=DEFAULT_SOLVER*)

ETFL.**create_simple_dynamic_model**()

ETFL.**model**

etfl.utils

Submodules

etfl.utils.parsing

Parsing utilities

Module Contents

Functions

isevaluable(s)	Test evaluability of a string for eval with sympy
parse_gpr(gpr)	Parses a string gpr into a sympy expression
multiple_replace(text, adict, ignore_case=False)	From https://www.oreilly.com/library/view/python-cookbook/0596001673/ch03s15.html
simplify_gpr(gpr)	
expand_gpr(gpr)	
expr2gpr(simplified_formatted_gpr)	
gpr2expr(gpr)	

Attributes

ESCAPE_CHARS
GPR2EXPR_SUBS_DICT
EXPR2GPR_SUBS_DICT

ETFL.**ESCAPE_CHARS = ['\\n', '\\t', '\\s', '"', "'"]**

ETFL.**GPR2EXPR_SUBS_DICT**

ETFL.**EXPR2GPR_SUBS_DICT**

ETFL.**isevaluable**(*s*)

Test evaluability of a string for eval with sympy

Parameters
s –

Returns

ETFL.**parse_gpr**(*gpr*)

Parses a string gpr into a sympy expression

Parameters
gpr –

Returns

ETFL.**multiple_replace**(*text*, *adict*, *ignore_case=False*)

From https://www.oreilly.com/library/view/python-cookbook/0596001673/ch03s15.html

ETFL.**simplify_gpr**(*gpr*)

ETFL.**expand_gpr**(*gpr*)

ETFL.**expr2gpr**(*simplified_formatted_gpr*)

ETFL.**gpr2expr**(*gpr*)

etfl.utils.utils

Module Contents

Functions

replace_by_enzymatic_reaction(model, reaction_id, enzymes, scaled)
replace_by_translation_reaction(model, reaction_id, gene_id, enzymes, trna_stoich, scaled)
replace_by_transcription_reaction(model, reaction_id, gene_id, enzymes, scaled)
replace_by_reaction_subclass(model, kind, reaction_id, **kwargs)
_replace_by_me_reaction(model, rxn, enz_rxn)
replace_by_me_gene(model, gene_id, sequence)
replace_by_coding_gene(model, gene_id)

etfl.utils.utils.**replace_by_enzymatic_reaction**(*model*, *reaction_id*, *enzymes*, *scaled*)

etfl.utils.utils.**replace_by_translation_reaction**(*model*, *reaction_id*, *gene_id*, *enzymes*, *trna_stoich*, *scaled*)

etfl.utils.utils.**replace_by_transcription_reaction**(*model*, *reaction_id*, *gene_id*, *enzymes*, *scaled*)

etfl.utils.utils.**replace_by_reaction_subclass**(*model*, *kind*, *reaction_id*, ***kwargs*)

etfl.utils.utils.**_replace_by_me_reaction**(*model*, *rxn*, *enz_rxn*)

etfl.utils.utils.**replace_by_me_gene**(*model*, *gene_id*, *sequence*)

etfl.utils.utils.**replace_by_coding_gene**(*model*, *gene_id*)

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

INDEX

Symbols

A

B

C

D

E

F

G

I

J

K

L

M

N

P

R

S

T

U

V

W

X