StrainDesign

Release 3.11

Philipp Schneider

Dec 02, 2023

CONTENTS:

CHAPTER

ONE

A COBRAPY-BASED PACKAGE FOR COMPUTATIONAL DESIGN OF METABOLIC NETWORKS

The comprehensive StrainDesign package for MILP-based strain design computation with the COBRApy toolbox supports MCS, MCS with nested optimization, OptKnock , RobustKnock and OptCouple , GPR-rule integration, gene and reaction knockouts and additions as well as regulatory interventions. The automatic lossless network and GPR compression allows strain design computations from genome-scale metabolic networks. Supported solvers are GLPK (available from COBRApy), CPLEX, Gurobi and SCIP .

Parts of the compression routine is done by efmtool's compression function (https://csb.ethz.ch/tools/software/efmtool.html).

CHAPTER

TWO

INSTALLATION:

The StrainDesign package is available on pip and Anaconda. To install the latest release, run:

```
pip install straindesign
```

or

```
conda install -c cnapy straindesign
```

2.1 Developer Installation:

Download the repository and run

```
pip install -e .
```

in the main folder. Through the installation with -e, updates from a 'git pull' are at once available in your Python envrionment without the need for a reinstallation.

2.2 JAVA_HOME path:

In some cases, installing the StrainDesign python package may fail with the error:

```
JVMNotFoundException: No JVM shared library file (libjli.dylib) found. Try setting up the
JAVA_HOME environment variable.
```

In this case, make sure Java is installed correctly and the JAVA_HOME varialbe is set. JAVA_HOME environment variable

If you're on OS X and get the error

```
OSError: [Errno 0] JVM DLL not found
```

check that your Java and the JPype library is set up correctly. The easiest way to avoid this error is to use conda to install StrainDesign.

CHAPTER

THREE

EXAMPLES:

Computation examples are provided in the different chapters of this documentation. The original Jupyer notebook files are located in the StrainDesign package at docs/source/examples.

CHAPTER

FOUR

HOW TO CITE:

Schneider P., Bekiaris P. S., von Kamp A., Klamt S. - StrainDesign: a comprehensive Python package for computational design of metabolic networks. Bioinformatics, btac632 (2022)

4.1 Solvers

4.1.1 3rd party solver installation

Through COBRApy, StrainDesign is already shipped with the free **GLPK** linear programming solver. Alternatively, the more powerful commercial solvers IBM CPLEX and Gurobi can be used by both COBRApy and StrainDesign, and the free solver SCIP can be used by StrainDesign. Using one of the GLPK alternatives is preferred, in particular, when using strain design algorithms like MCS, OptKnock etc. since their support of indicator contstraints renders computations significantly more stable.

In the following, you will find installation instructions for the individual solvers.

Warning:

The **free community versions** of CPLEX and Gurobi can be used. However, with larger problems (100+ reactions) their problem size limitations may result in uncaught errors.

4.1.1.1 CPLEX

Together with Gurobi, CPLEX is the perfect choice for computing strain designs. Its stability and support of advanced features like indicator constraints and populating solution pools make it indispensible for genome-scale computations.

Note:

You will need an academic or commercial licence to be able to use CPLEX.

Download and install the CPLEX suite to a location with non-root access since python will need to build some things to set up the CPLEX-API, later. Make sure that your CPLEX and Python versions are compatible. Currently (December 2023), CPLEX is not officially available for python>3.10. Once the installation is completed, you may use the installation to set up the CPLEX-API with your Python/conda environment.

This can be done either with pip

```
pip install yourCPLEXhome/python/VERSION/PLATFORM
```

or with conda. For an installation with conda, make sure to activate the same Python/conda environment where `cobra` and `straindesign` are installed. Then call

```
python yourCPLEXhome/python/VERSION/PLATFORM/setup.py install
```

Now CPLEX should be available for your computations. If you face difficulties with building CPLEX, consider downgrading the setuptools package to setuptools==58.2.0.

The official instructions can be found here: https://www.ibm.com/docs/en/icos/22.1.0?topic=cplex-setting-up-python-api

4.1.1.2 Gurobi

Similar to CPLEX, Gurobi offers a fast MILP solvers with the advanced features of indicator constraints and solution pooling. The installation steps are similar to the ones of CPLEX.

Note:

You will need an academic or commercial license and install the Gurobi solver software.

Ensure that the versions of Gurobi and Python versions are compatible, install Gurobi on your system and activate your license following the steps from the Gurobi manual. In the next step you will link your Gurobi installation to your Python/conda environment.

Using the command line, navigate to your CPLEX installation path and into the Python folder. The path should look similar to

```
C:/gurobi950/windows64
```

Make sure to activate the same Python/conda environment where `cobra` and `straindesign` are installed. Then call

```
python setup.py install
```

If your `gurobipy` package does not work right away, additionally install the gurobi package from conda or PyPi via

```
conda install -c gurobi gurobi
```

or

```
python -m pip install gurobipy
```

Now Gurobi is available for your computations.

The official instructions can be found here: https://support.gurobi.com/hc/en-us/articles/360044290292-How-do-I-install-Gurobi-for-Python-

4.1.1.3 SCIP

Less powerful than CPLEX and Gurobi, the open source solver SCIP still offers the solution of MILPs with indicator constraints, which gives it an edge above GLPK in terms of stability. If you want to use SCIP, you may install it via conda or pip:

```
conda install -c conda-forge pyscipopt
```

or

```
python -m pip install pyscipopt
```

Warning:

If you encounter program crashes with SCIP (a dependency of pyscipopt), make sure you use a SCIP version of 8.0.0 or older since newer versions are unreliable in solving MILPs and can produce errors (as of December 2023), this issue might get fixed in the future. You can, manually install pyscipopt 4.2.0 and scip version 8.0.0 through `conda install -c conda-forge pyscipopt=4.2.0 scip=8.0.0`. Keep in mind that SCIP 4.2.0 is currently (December 2023) not available for python>3.10.

Official website: https://github.com/scipopt/PySCIPOpt

4.1.2 Solver selection

For any type of LP or MILP-based analysis or design method, four different sovers are supported: **GLPK** (which is built into COBRApy/optlang), **IBM CPLEX**, **Gurobi** and **SCIP**. You can query the available solvers by accessing the set `straindesign.avail_solvers`. For the subsequent steps we also import the cobra module and load the *E. coli* core model.

[1]:
```
import cobra
import straindesign as sd
model = cobra.io.load_model('e_coli_core')

sd.avail_solvers
```

```
Set parameter Username
Academic license - for non-commercial use only - expires 2023-07-20
```

[1]:
```
{'cplex', 'glpk', 'gurobi', 'scip'}
```

You may enforce the use of a specific solver by specifying the "solver"-keyword. E.g., to enforce the use of GLPK, use:

[2]:
```
solution = sd.fba(model, solver='glpk')

print(f"Maximum growth: {solution.objective_value}.")
```

```
Maximum growth: 0.873921506969.
```

By default, the automatic solver selection uses COBRApy's selection. Therefore, StrainDesign will try to use the model's selected solver:

[3]:
```
print(f"When the model\'s solver is \'{model.solver.configuration}', StrainDesign␣
↪selects {sd.select_solver(None,model)}.")
```

```
When the model's solver is '<optlang.gurobi_interface.Configuration object at␣
↪0x0000014FE00DA400>', StrainDesign selects gurobi.
```

Otherwise COBRApy's global configuration is used.

[4]:
```
print(f"COBRApy\'s solver is \'{cobra.Configuration().solver.__name__}', StrainDesign␣
↪selects {sd.select_solver()}.")
```

```
COBRApy's solver is 'optlang.gurobi_interface', StrainDesign selects gurobi.
```

[5]:
```
model.solver = 'cplex'
sd.select_solver('glpk',model)
```

[5]:
```
'glpk'
```

4.2 Network Analysis

StrainDesign provides canonical functions for maximizing and minimizing metabolic fluxes in network. The output format is identical to the ones of COBRApy's functions.

4.2.1 Flux optimization (FBA/pFBA)

Flux Balance Analysis (FBA) is a linear program that optimizes a flux rate under the given steady-state network constraints. For the case of a growth-rate maximization, the problem is written as:

$$\begin{aligned} \text{maximize} \quad & v_{growth} \\ \text{subject to} \quad & \\ & \mathbf{S}\,\mathbf{v} = \mathbf{0} \\ & \mathbf{lb} \leq \mathbf{v} \leq \mathbf{ub} \end{aligned}$$

Where $\mathbf{S}$ is the stoichiometric matrix of the metabolic model, $\mathbf{v}$ is the vector of metabolic flux rates and $\mathbf{lb}$ and $\mathbf{ub}$ are the physiological lower and upper bounds of each flux rate which also define whether a reaction can run in the reverse direction ($\mathbf{lb} < 0$) or not ($\mathbf{lb} \geq 0$). $\mathbf{S}\,\mathbf{v} = \mathbf{0}$ represents all steady-state constraints, and $\mathbf{lb} \leq \mathbf{v} \leq \mathbf{ub}$ the allowed flux ranges.

Note:

All of the following computation examples will require the COBRApy and the StrainDesign package.

Here, we load both packages and the e_coli_core model from BiGG:

[1]:
```
import cobra
import straindesign as sd
model = cobra.io.load_model('e_coli_core')
```

```
Set parameter Username
Academic license - for non-commercial use only - expires 2023-07-20
```

An FBA is launched by a single function call. By default the model's objective function is optimized. The function returns a solution object, in which the objective value and the fluxes are stored in `solution.objective_value` and `solution.fluxes`.

[2]:
```
solution = sd.fba(model)

print(f"Maximum growth: {solution.objective_value}.")
```

```
Maximum growth: 0.8739215069684305.
```

You may also use a custom objective (in form of a linear expression) and change the optimization sense. Here we minimize the Glucose uptake rate through the PTS:

[3]:
```
solution = sd.fba(model,obj='GLCpts',obj_sense='minimize')

print(f"Minimum flux through GLCpts: {solution.objective_value}.")
```

```
Minimum flux through GLCpts: 0.4794285714285715.
```

We can also consider custom constraints (in this case limited oxygen uptake and an increased fixed ATP maintenance demand):

[4]:
```
solution = sd.fba(model,constraints=['-EX_o2_e <= 5', 'ATPM = 20'])

print(f"Maximum growth at limited oxygen uptake and high ATP maintenance: {solution.
↪objective_value}.")
```

```
Maximum growth at limited oxygen uptake and high ATP maintenance: 0.26305573292588313.
```

4.2.1.1 Parsimonious FBA (pFBA)

Parsimonious flux balance analysis optimizes a flux rate under the given steady-state network constraints, but also minimizes the sum of absolute fluxes to achieve this optimum. One can write:

$$\begin{aligned} &\text{minimize} && \Sigma|v_i| \\ &\text{subject to} \end{aligned}$$

$$\begin{aligned} &\text{maximize} && v_{growth} \\ &\text{subject to} \\ &&& \mathbf{S}\,\mathbf{v} = \mathbf{0} \\ &&& \mathbf{lb} \leq \mathbf{v} \leq \mathbf{ub} \end{aligned}$$

pFBA is the simpleset (although very rough) approach of emulating a cell's enzyme cost minimization (after an assumed growth maximization).

Note:

pFBA solutions are more often "unique" than pure FBA solutions, since the outer minimization leaves fewer degrees of freedom in the solution space.

StrainDesign computes pFBA solutions when you pass the 'pFBA'-argument with a value of 1.

[5]:
```
fba_sol = sd.fba(model)
pfba1_sol = sd.fba(model,pfba=1)
print(f"The sum of fluxes of the regular FBA: {sum([abs(v) for v in fba_sol.fluxes.
↪values()])} "+\
      f"is usually higher than of the parsimoniuos FBA: {sum([abs(v) for v in pfba1_sol.
↪fluxes.values()])}")
```

```
The sum of fluxes of the regular FBA: 2508.293334194643 is usually higher than of the␣
↪parsimoniuos FBA: 518.4220855176071
```

Likewise, it is possible to minimize the number of active reactions to attain an optimal flux distribution. We therefore use pFBA mode 2. Most of the times modes 1 and 2 yield the same results.

[6]:
```
pfba1_sol = sd.fba(model,pfba=1)
pfba2_sol = sd.fba(model,pfba=2)
print(f"The number of active reactions in pFBA1: {sum([v!=0 for v in pfba1_sol.fluxes.
↪values()])}, "+\
      f"and pFBA2: {sum([v!=0 for v in pfba2_sol.fluxes.values()])}, is often identical.
↪")
```

```
The number of active reactions in pFBA1: 48, and pFBA2: 48, is often identical.
```

4.2.2 Flux variability analysis (FVA)

The fva function determines the possible maximal and minimal flux ranges under the given model constraints:

[7]:
```
solution = sd.fva(model)
print(solution)
```

```
         minimum  maximum
PFK          0.0   176.61
PFL          0.0    40.00
PGI        -50.0    10.00
PGK        -20.0    -0.00
PGL          0.0    60.00
...          ...      ...
NADH16       0.0   120.00
NADTRHD      0.0   378.22
NH4t         0.0    10.00
O2t          0.0    60.00
PDH          0.0    40.00

[95 rows x 2 columns]
```

The parameters ‘solver’ and ‘constraints’ can also be used in the FVA function call. As an example, we determine the flux ranges for the case that the flux sum of PDH and PFL is smaller than 8.

[8]:
```
solution = sd.fva(model, constraints='PDH + PFL <= 8')
print(solution)
```

```
         minimum     maximum
PFK          0.0  147.610000
PFL          0.0    8.000000
PGI        -50.0   10.000000
PGK        -20.0   -0.000000
PGL          0.0   60.000000
...          ...         ...
NADH16       0.0  120.000000
NADTRHD      0.0  375.220000
NH4t         0.0    8.300263
```

(continues on next page)

(continued from previous page)

```
O2t          0.0   60.000000
PDH          0.0    8.000000

[95 rows x 2 columns]
```

4.2.3 Yield optimization

Yield optmization aims to maximize a given flux expression (e.g., a product's synthesis rate) divided by another such expression (e.g., a substrate's uptake rate).

Warning:

Yield optimization and Flux Balance Analysis are two different methods that produce distinct optimal values and also distinct optimal flux vectors.

Consider the following example of maximizing growth and biomass yield under limited oxygen uptake.

[9]:
```
numerator = 'BIOMASS_Ecoli_core_w_GAM'
denominator = '-EX_glc__D_e'
constraint = '-EX_o2_e <= 3'

sol_fba = sd.fba(model,obj='BIOMASS_Ecoli_core_w_GAM',obj_sense='maximize',
↪constraints=constraint)
fba_yield = sol_fba.fluxes[numerator] / -sol_fba.fluxes['EX_glc__D_e']

sol_yopt = sd.yopt(model,obj_num=numerator,obj_den=denominator,obj_sense='maximize',
↪constraints=constraint)
yopt_yield = sol_yopt.objective_value

print(f"Maximum yield (FBA): {fba_yield}.")
print(f"Maximum yield (yOpt): {yopt_yield}.")
```

```
Maximum yield (FBA): 0.031965425062067315.
Maximum yield (yOpt): 0.03629426243040193.
```

The best biomass yield is achieved when only respiration is used. The best growth rates use respiration and *additionally* overflow metabolism with poorer biomass yield.

The plotting chapter shows how relationships between yields and rates can be visualized.

4.2.3.1 Mathematical background

Constraint-based models ($\mathbf{S}\,\mathbf{v} = \mathbf{0}$, $\mathbf{lb} \leq \mathbf{v} \leq \mathbf{ub}$) can be rewritten in a single matrix-inequality term ($\mathbf{A}\,\mathbf{x} \leq \mathbf{b}$).

With this notation, the yield optimization is a linear fractional program (LFP):

$$\begin{array}{ll} \text{maximize} & \frac{\mathbf{c}^\intercal \mathbf{x}}{\mathbf{d}^\intercal \mathbf{x}} \\ \text{subject to} & \mathbf{A}\,\mathbf{x} \leq \mathbf{b}. \end{array}$$

Under the condition that the denominator term is strictly positive ($\mathbf{A}\,\mathbf{x} \leq \mathbf{b} \;\Rightarrow\; \mathbf{d}^\intercal \mathbf{x} > 0$), the LFP may be rewritten as an LP problem, using the Charnes-Cooper transformation. The formerly fixed boundaries $\mathbf{b}$ of the problem are then

scaled by the auxiliary variable e while the variable $y = \frac{\mathbf{c}^\intercal \mathbf{x}}{\mathbf{d}^\intercal \mathbf{x}}$ expresses the original objective function:

$$\begin{aligned} &\text{maximize} && y \\ &\text{subject to} && \begin{bmatrix} \mathbf{A} & -\mathbf{b} & \mathbf{0} \\ \mathbf{d}^\intercal & 0 & 0 \\ \mathbf{c}^\intercal & 0 & -1 \end{bmatrix} \begin{bmatrix} \tilde{\mathbf{x}} \\ e \\ y \end{bmatrix} \begin{matrix} \leq \\ = \\ = \end{matrix} \begin{bmatrix} \mathbf{0} \\ 1 \\ 0 \end{bmatrix} \\ & && e \geq 0. \end{aligned}$$

Solutions of $\mathbf{x}$ to the LFP (first problem) can be retrieved from a solution of the LP through $\mathbf{x} = \frac{\tilde{\mathbf{x}}}{e}$.

4.3 Plotting the flux space

There are different ways to visualize the space of feasible steady-state flux vectors. The two most popular plot types are production envelopes and yield spaces.

Note:

Production envelopes project the flux space of feasible steady-state flux vectors onto two dimensions whereas each dimension is a flux rate. Yield spaces map feasible yield combinations onto two dimensions whereas each dimension is a flux ratio.

StrainDesign provides functions for both of these plot types, but additionally support plotting of arbitrary other projections or mappings of rate and yield terms on two or three dimensions. Here, we use the e_coli_core example for demonstration purposes again.

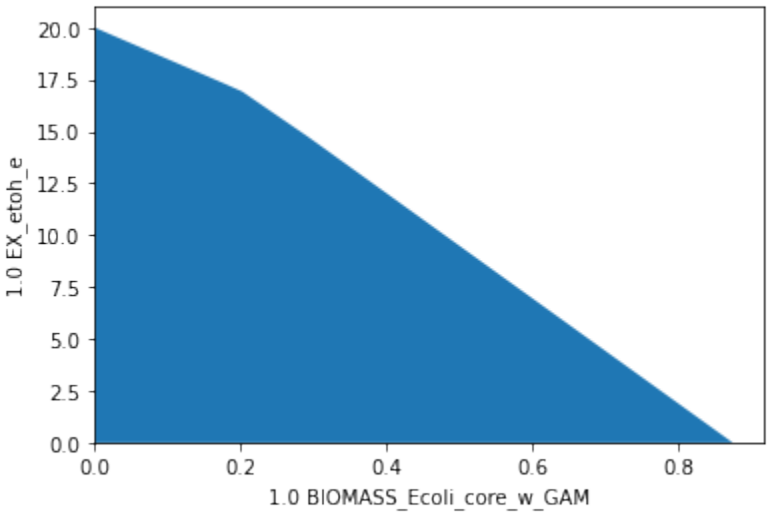
[1]:
```
import cobra
import straindesign as sd
model = cobra.io.load_model('e_coli_core')
```

```
Set parameter Username
Academic license - for non-commercial use only - expires 2023-07-20
```

4.3.1 Production Envelopes

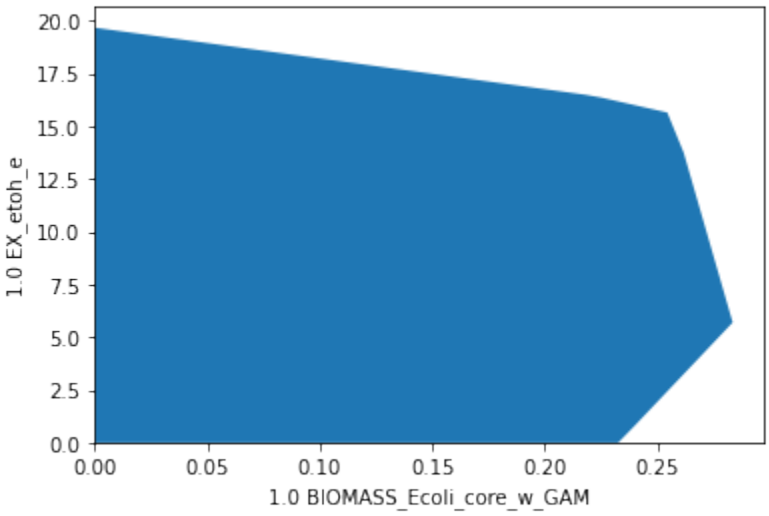
Production envelopes project the solution space of steady-state flux vectors onto the dimensions of growth rate and product synthesis rate. Such a plot can be generated by:

[2]:
```
sd.plot_flux_space(model,('BIOMASS_Ecoli_core_w_GAM','EX_etoh_e'));
```



Again, arbitrary constraints can be applied to the flux space to plot subspaces. Here, we plot the flux space within a small range of oxygen uptakes (1-2 $mmol_{0_2} g_{CDW}^{-1} h^{-1}$). This constraint reduces the maximum growth rate to a thrid, while maximum growth entails ethanol production.

[3]:
```
sd.plot_flux_space(model,('BIOMASS_Ecoli_core_w_GAM','EX_etoh_e'),
                   constraints=['-EX_o2_e >= 1', '-EX_o2_e <= 2']);
```



It is also possible to use arbitrary linear expressions for the axis. Here, for instance, we plot the carbon recovery in the oxidized products formate, acetate and CO2 versus the reduced product ethanol. The coefficients for each product are matched with their number of carbon atoms. Glucose uptake is set to 1 (so the input equals 6 carbon atoms). The plot now shows that, stoichiometrically, at most of 4 out of 6 atoms can be directed to either side, ethanol or oxidized products. Yet, to balance redox equivalents, it is necessary, to then direct the remaining 2 carbon atoms towars the other side.

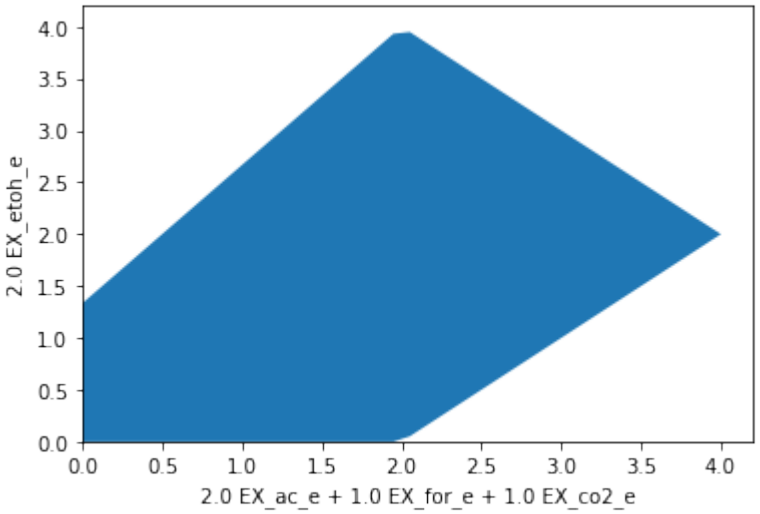
[4]:
```
constraints=[
             'EX_h_e >= 0',
             'EX_co2_e >= 0',
             'EX_o2_e = 0'
```

(continues on next page)

(continued from previous page)

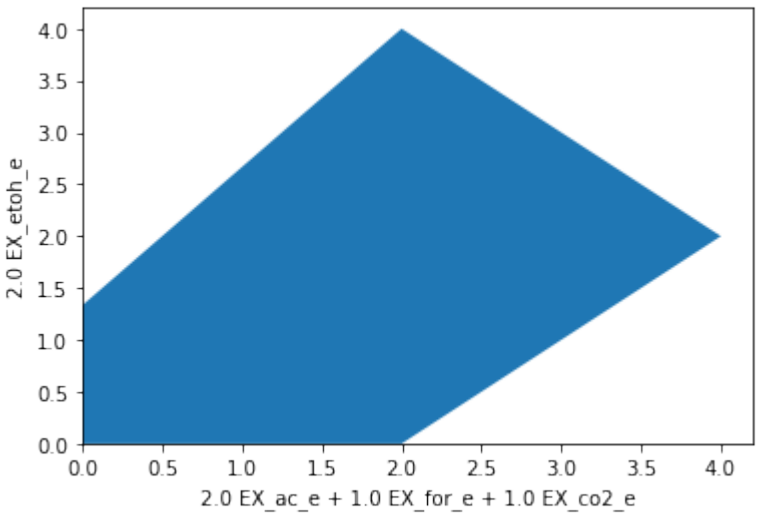
```
            ]
model1 = model.copy()
model1.reactions.ATPM.lower_bound = 0
model1.reactions.EX_glc__D_e.lower_bound = -1
model1.reactions.EX_glc__D_e.upper_bound = -1
datapoints, triang, plot1 = sd.plot_flux_space(model1,('2 EX_ac_e + 1 EX_for_e + 1 EX_
↪co2_e','2 EX_etoh_e'),
                                               constraints = constraints);
```

```
Read LP format model from file C:\Users\Philipp\AppData\Local\Temp\tmpmxmathi0.lp
Reading time = 0.00 seconds
: 72 rows, 190 columns, 720 nonzeros
```



Notice the bump in the top edge? We may use a finer sampling grid to even out such artifacts.

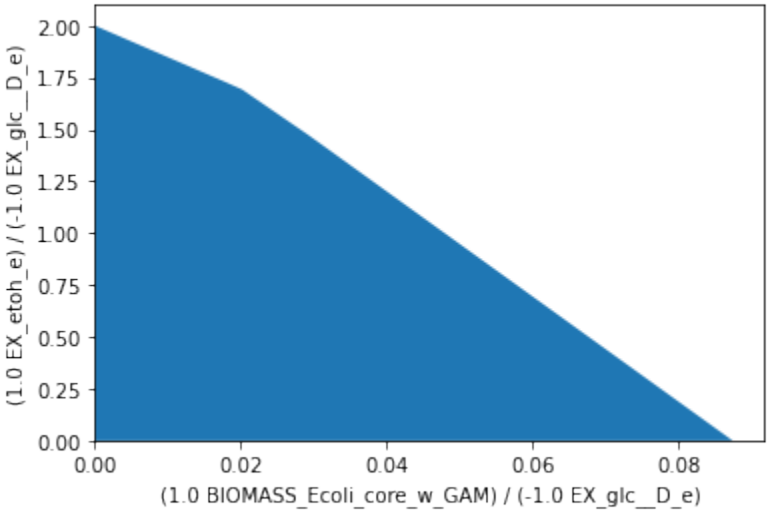
[5]:
```
datapoints, triang, plot1 = sd.plot_flux_space(model1,('2 EX_ac_e + 1 EX_for_e + 1 EX_
↪co2_e','2 EX_etoh_e'),
                                               constraints = constraints, points=55);
```



4.3.2 Yield Spaces

The yield space plot can be used to depict the relationship between two yield terms in a metabolic networks. Here, we plot the biomass yield vs the etahnol production yield.

[6]:
```
sd.plot_flux_space(model,(('BIOMASS_Ecoli_core_w_GAM','-EX_glc__D_e'),('EX_etoh_e','-EX_
↪glc__D_e')));
```



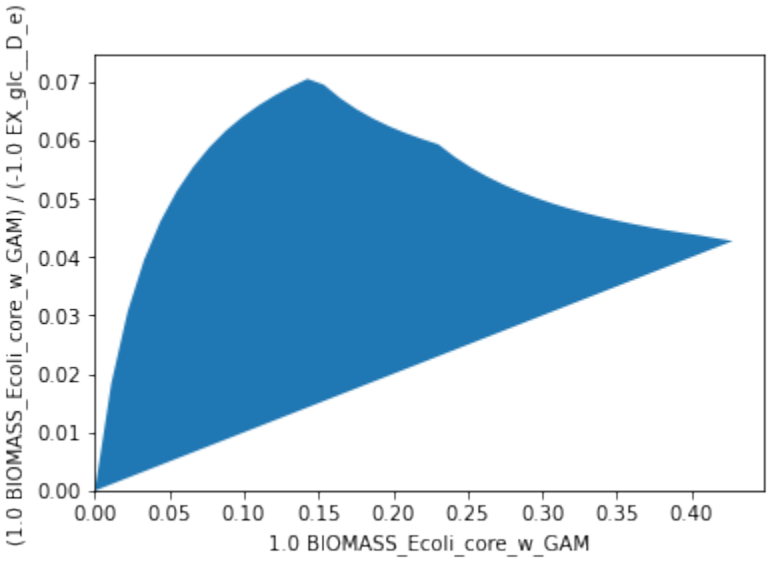
Yield space plots often look similar to their corresponding phase planes, as it is the case here.

> **Warning:**
>
> It should be noted that yield terms are generally non-linear and, adversely to pure flux-space projections, yield spaces may therfore be non-polyhedral or even non-convex.

4.3.3 Mixed Plots

The following plot uses a rate and a yield for the different axes. It shows the relationship between possible growth rates and biomass yields under limited oxygen uptake.

[7]:
```
sd.plot_flux_space(model,('BIOMASS_Ecoli_core_w_GAM',('BIOMASS_Ecoli_core_w_GAM','-EX_
↪glc__D_e')),
                   constraints='-EX_o2_e <= 6');
```



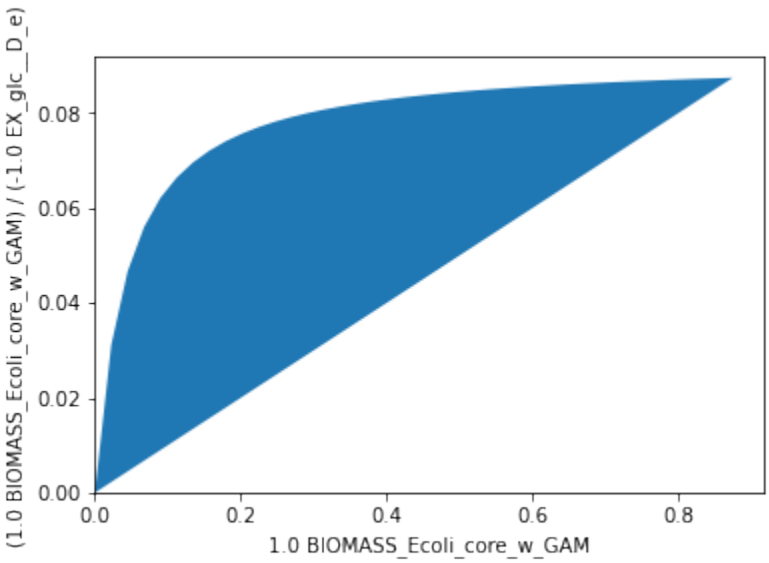
The graph shows that under limited oxygen uptake, maximum growth rate and maximum biomass yield do not coincide. As discussed in the chapter of yield optimization, overflow metabolism may be used to boost growth at the expense of biomass yield.

Note:

If growth yield and growth rate could be used interchangably, we would observe a straingt diagonal line here.

In the case of unlimited oxygen uptake, we observe that best yield and growth rate indeed coinside, since respiration can be fully exploited to attain the hightest possible biomass yield:

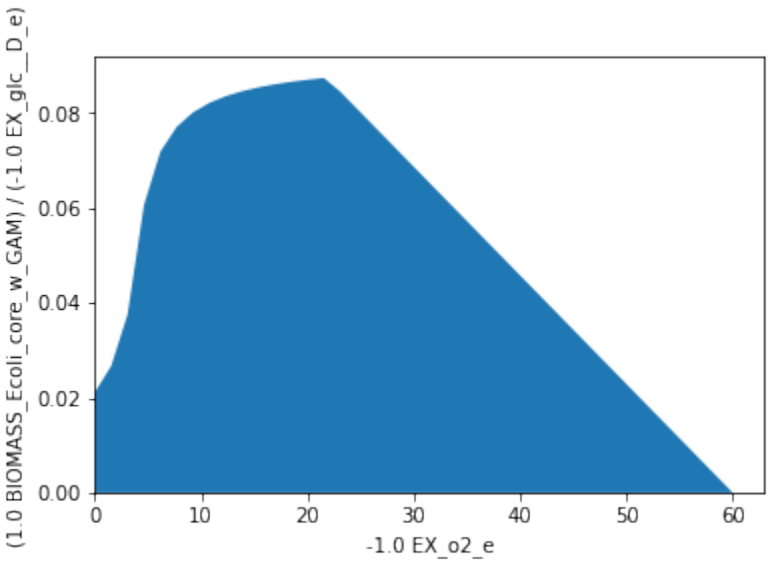
[8]:
```
sd.plot_flux_space(model,('BIOMASS_Ecoli_core_w_GAM',('BIOMASS_Ecoli_core_w_GAM','-EX_
↪glc__D_e')));
```



We may also use a mixed rate-yield (oxygen uptake vs biomass yield) plot to graphically determine the oxygen uptake

rate needed to attain the maximum yield. There is a sigmoidal increase of the biomass yield with increased oxygen availability:

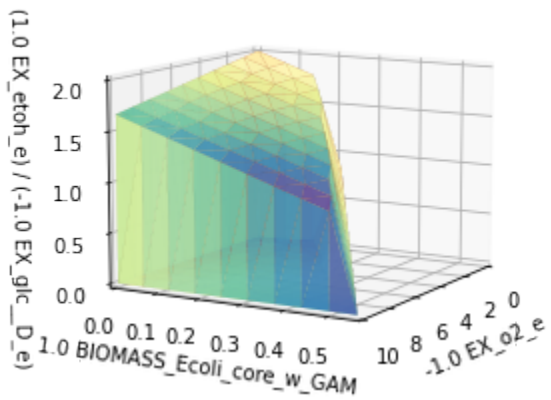
[9]:
```
sd.plot_flux_space(model,('-EX_o2_e',('BIOMASS_Ecoli_core_w_GAM','-EX_glc__D_e')));
```



4.3.4 3D plots

A 3-dimensional plot may help tp reveal more complex relationships between fluxes and yields. In the following plot, we analyze the relationship between oxygen uptake, growth rate and ethanol yield.

[10]:
```
sd.plot_flux_space(model,('-EX_o2_e','BIOMASS_Ecoli_core_w_GAM',('EX_etoh_e','-EX_glc__D_
↪e')),
                   constraints='-EX_o2_e <= 10',points=10);
```



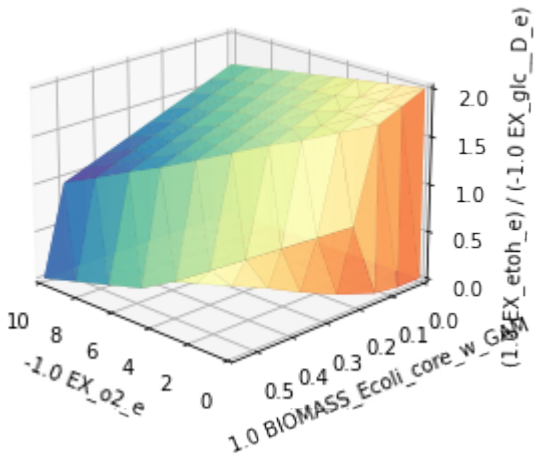
Since a 3D-plot does not show us the angle we are looking for, we may halt plotting and use matplotlib commands to rotate the plot before the actual plotting.

[11]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
_,_,plot1 = sd.plot_flux_space(model,('-EX_o2_e','BIOMASS_Ecoli_core_w_GAM',('EX_etoh_e',
↪'-EX_glc__D_e')),
                    constraints='-EX_o2_e <= 10',points=10,show=False);
plot1._axes.view_init(20, 135)
plt.show()
```



The plot now shows that at low or no oxygen uptake, a minimum ethanol yield is guaranteed at maximum growth rates, while at higher growh rates this is not the case. The critical point seems to be reached at an oxygen uptake rate of about $7\,mmol_{0_2} g_{CDW}^{-1} h^{-1}$.

4.3.4.1 Interactive and animated 3D plots

You may set a custom rendering option, to generate an interactive that can be rotated or a plot in a separate window. This can be done either through the IPython magic command, e.g., `%matplotlib tk`:

[12]:
```
%matplotlib tk
sd.plot_flux_space(model,('-EX_o2_e','BIOMASS_Ecoli_core_w_GAM',('EX_etoh_e','-EX_glc__D_
↪e')),
                    constraints='-EX_o2_e <= 10',points=10);
```

or by passing the parameter `plt_backend`:

[13]:
```
sd.plot_flux_space(model,('-EX_o2_e','BIOMASS_Ecoli_core_w_GAM',('EX_etoh_e','-EX_glc__D_
↪e')),
                    constraints='-EX_o2_e <= 10',points=10,
                    plt_backend='TkAgg');
```

You may also animate 3D figures and save these animations to GIFs or movie files. Please note, that the ffmpeg codec needs to be installed and available. For that matter, refer to the matplotlib reference.

[14]:
```
import matplotlib.animation as animation
from IPython import display

r1 = ('-EX_o2_e')
r2 = ('BIOMASS_Ecoli_core_w_GAM')
```

(continues on next page)

(continued from previous page)

```
y3 = ('EX_etoh_e', '-EX_glc__D_e')
constraints = ['NADTRHD = 0',
               'NADH16 = 0',
               'LDH_D = 0',
               'PPC = 0']
_,_,plot2 = sd.plot_flux_space(model, (r1, r2, y3),
                           constraints=constraints,
                           points=10,
                           show=False)
# specify animation
def animate(angle):
    plot2._axes.view_init(20, angle)
    return plot2

# generate animation
ani = animation.FuncAnimation(
    plot2.figure, animate, save_count=360)

# Save the animation in gif
writer = animation.FFMpegWriter(
    fps=25, bitrate=1000)
ani.save("movie.gif", writer=writer)

# Alternatively, display animation in Jupyter notebook
video = ani.to_html5_video()
html = display.HTML(video)
display.display(html)
```

```
<IPython.core.display.HTML object>
```

Plot generation:

For all plots, the term from the primary axis is minimized and maximized. The `points` parameter is used to generate a grid that is used in the maximizations and minimizations in the direction of the secondary axis. In case of a 3D plot, another grid is then generated on the first two axes and maximizations and minimizations are performed in the direction of the tertiary axis.

In 2D, the resulting polygon is then filled. In 3D, Delaunay triangles are generated to plot the surface of the shape.

4.4 Computational strain design: Growth-coupled production (GCP)

A main principle for the design of bioproduction hosts is the coupling of product synthesis to microbial growth, such that the product of interest becomes a byproduct of growth. Different notions of growth-coupling principle prevail in literature. With gradually increasing coupling strength, we can distinguish four different classes: - potentially growth-coupled production (pGCP): product synthesis is possible at maximum growth *(e.g., minimum demand in OptKnock computations)* - weakly growth-coupled production (wGCP): product synthesis is enforced at maximum growth *(e.g., demanded by RobustKnock or OptCouple)* - directionally growth-coupled production (dGCP): product synthesis is enforced at all growth rates greater than zero *(e.g., often demanded by MCS, often generated by OptCouple)* - substrate-uptake-coupled production (SUCP): product is synthesized whenever substrate is taken up, i.e., a minimum product yield is guaranteed *(e.g., often demanded by MCS)*

Before moving on the computation of strain designs, we will provide strain design examples for the different coupling types and show their main property. To keep it as simple as possible, we will use ethanol as the desired product, use the small "textbook model" e_coli_core, and introduce only reaction knockouts (and no knock-ins).

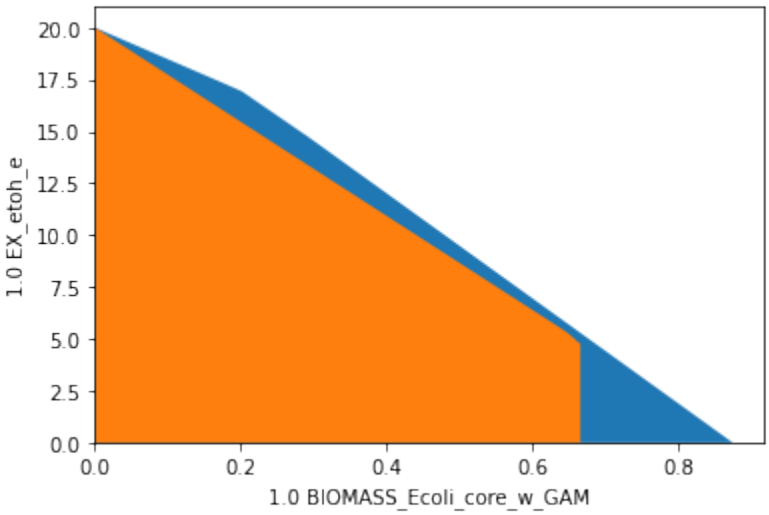
[1]:
```
import cobra
import straindesign as sd
model = cobra.io.load_model('e_coli_core')
```

```
Set parameter Username
Academic license - for non-commercial use only - expires 2023-07-20
```

4.4.1 pGCP: potentially growth-coupled production

When we knock out the reactions **PPS**, **THD2**, **TKT2** and **MDH** in e_coli_core, we obtain a potentially coupled strain design. Below, we plot the flux space of the wild type model (blue/background) on the dimensions of growth rate and the ethanol yield and compare it to the pGCP strain design (orange/foreground). In the wild type model, growth and ethanol production compete, whereas, in the strain designed for pGCP, product synthesis is also possible (yet *not required*) at growth-maximal flux states.

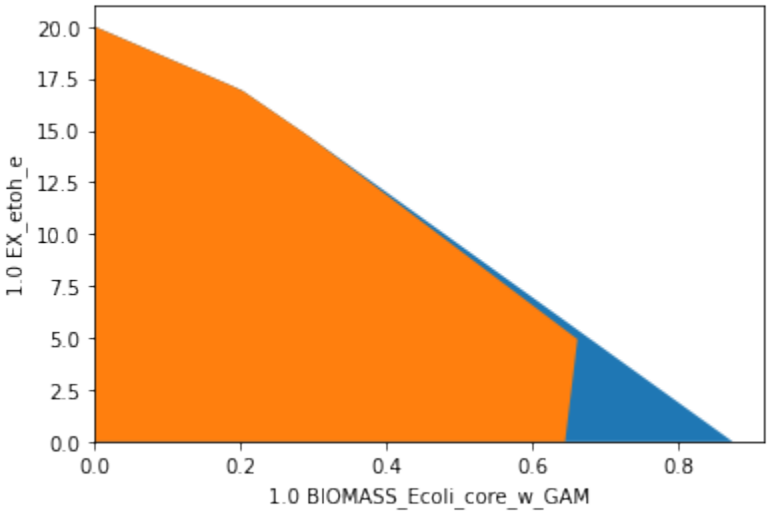
[2]:
```
import matplotlib.pyplot as plt
# Wild-type plot
datapoints, triang, plot1 = sd.plot_flux_space(model,
                                               ('BIOMASS_Ecoli_core_w_GAM','EX_etoh_e'),
                                               show=False);
# pGCP design plot
constraints = ['PPS=0', 'THD2=0', 'TKT2=0', 'MDH=0']
_,          _,      plot2 = sd.plot_flux_space(model,
                                               ('BIOMASS_Ecoli_core_w_GAM','EX_etoh_e'),
                                               constraints=constraints,
                                               show=False);
# adjust axes limits and show plot
plot2.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot2.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```



4.4.2 wGCP: weakly growth-coupled production

A weakly growth-coupled production strain design can be generated by knocking out the reactions **ACALDt**, **PTAr**, **PPS**, **PYRt2** and **MDH** in e_coli_core. Below, we plot the flux space of the wild type model (blue/background) on the dimensions of growth rate and the ethanol yield and compare it to the wGCP strain design (orange/foreground). In the wild type model, growth and ethanol production compete, whereas, in the strain designed for wGCP, growth-maximal flux states can only be reached by simultanously producing ethanol.

[3]:
```
import matplotlib.pyplot as plt
# Wild-type plot
datapoints, triang, plot1 = sd.plot_flux_space(model,
                                               ('BIOMASS_Ecoli_core_w_GAM','EX_etoh_e'),
                                               show=False);
# pGCP design plot
constraints = ['ACALDt=0', 'PTAr=0', 'PPS=0', 'PYRt2=0', 'MDH=0']
_,          _,      plot2 = sd.plot_flux_space(model,
                                               ('BIOMASS_Ecoli_core_w_GAM','EX_etoh_e'),
                                               constraints=constraints,
                                               show=False);
# adjust axes limits and show plot
plot2.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot2.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```



4.4.3 dGCP: directionally growth-coupled production

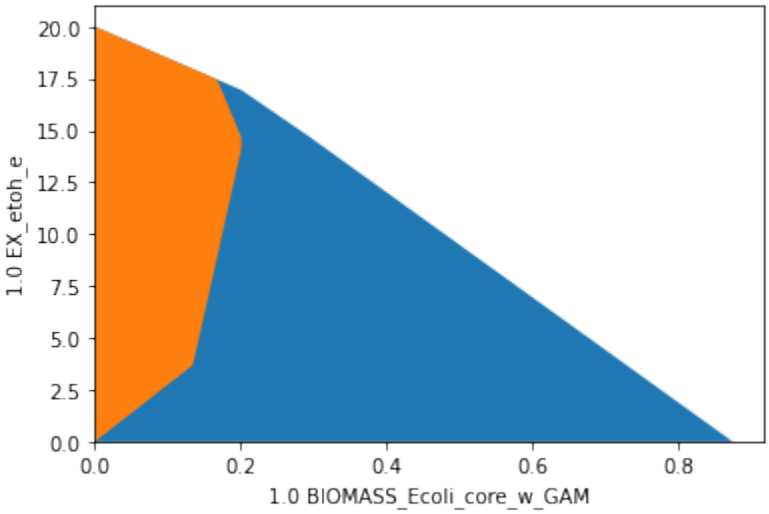
By knocking out the reactions **PGL**, **ATPS4r** and **NADH16** in e_coli_core, we obtain a directionally coupled strain design. Below, we plot the flux space of the the dGCP strain design (orange/foreground) on the dimensions of growth rate and the ethanol production. In the strain designed for dGCP, product synthesis is occurs at all growth-associated flux states. In this example significantly higher specific production rates are reached than in the pGCP and wGCP scenario, however at the detriment of high growth rates.

[4]:
```
import matplotlib.pyplot as plt
# Wild-type plot
datapoints, triang, plot1 = sd.plot_flux_space(model,
```

(continues on next page)

(continued from previous page)

```
                                                ('BIOMASS_Ecoli_core_w_GAM','EX_etoh_e'),
                                                show=False);
# pGCP design plot
constraints = ['PGL=0', 'ATPS4r=0', 'NADH16=0']
_,          _,      plot2 = sd.plot_flux_space(model,
                                                ('BIOMASS_Ecoli_core_w_GAM','EX_etoh_e'),
                                                constraints=constraints,
                                                show=False);
# adjust axes limits and show plot
plot2.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot2.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```



4.4.4 SUCP: substrate-uptake-coupled production

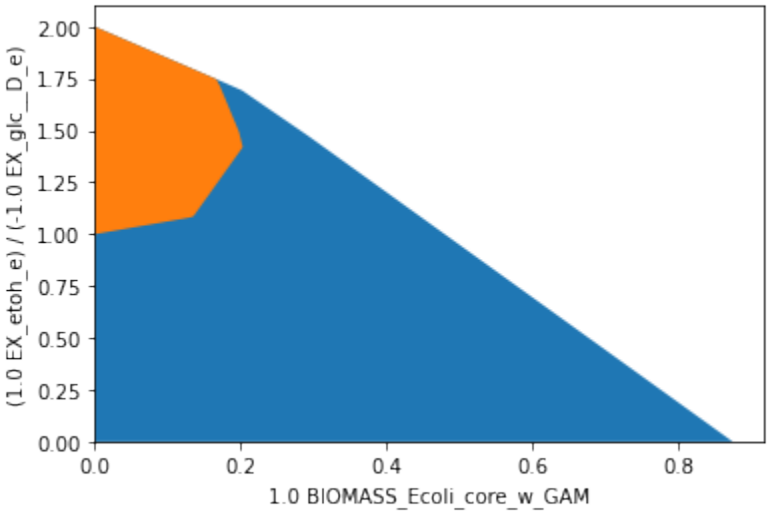
In strain designs with substrate-uptake-coupled production, all feasible flux states of substrate uptake also entail production. To obtain a such a strain design for the production of ethanol, the reactions knockouts **LDH_D**, **ATPS4r** and **NADH16** can be introduced to e_coli_core. Other than in the previous examples, we here plot the flux space of the SUCP strain design (orange/foreground) on the dimensions of growth rate and ethanol *yield*. In the SUCP strain design, a minimum product yield is ensured in all remaining feasible flux states. Again, the improved production capacities often come at the expense of viability.

[5]:
```
import matplotlib.pyplot as plt
# Wild-type plot
datapoints, triang, plot1 = sd.plot_flux_space(model,
                                                ('BIOMASS_Ecoli_core_w_GAM',('EX_etoh_e',
↪'-EX_glc__D_e')),
                                                show=False);
# pGCP design plot
constraints = ['LDH_D=0', 'ATPS4r=0', 'NADH16=0']
_,          _,      plot2 = sd.plot_flux_space(model,
                                                ('BIOMASS_Ecoli_core_w_GAM',('EX_etoh_e',
↪'-EX_glc__D_e')),
                                                constraints=constraints,
```

(continues on next page)

(continued from previous page)

```
                                                    show=False);
# adjust axes limits and show plot
plot2.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot2.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```



In the following chapter, we discuss how different algorithms can be used to generate growth-coupled strain designs.

4.5 Minimal Cut Sets (MCS)

The minimal cut set approach aims to find a minimum set of network intervetions that enforce some pre-definde behavior in a metabolic network. It may be used to suppress growth, suppress or enforce the synthesis of a certain product or to couple or decouple metabolic fluxes. For a better conceptional understanding of the MCS approach, please refer to examples 7 and 8. Examples 1-5 treat strain design applications in a small network, while example 6 presents a genome-scale strain design computation.

4.5.1 Prerequisites

For the shown examples, we again load the COBRApy and the StrainDesign packages as well as the small E. coli textbook-network "e_coli_core" and the example network "SmallExample". For CNApy users a special project (e_coli_core_14bdo) is available that contains the modified metabolic network together with strain design files for the various examples which can be loaded from the strain design dialog.

The following steps show how the function `compute_strain_designs` is employed to find MCS strain designs: 1) The production pathways, i.e., their metabolites and reactions are added to the model. 2) Analysis tools are used to identify adequate strain design goals. 3) Set up the according strain design problem by specifying two strain design *modules* 4) The strain design function is called, passing the model and the strain design module(s) as function arguments. 5) The results are analyzed.

Step (1) is required for all examples shown hereafter. Steps (2)-(4) are example-dependent. We do step (5) for all examples to verify the computed strain designs.

4.5.1.1 1) Add and verify production pathway

[1]:
```
import straindesign as sd
import cobra
cobra.Configuration().solver = 'cplex'

ecc = cobra.io.load_model('e_coli_core')
model = cobra.io.read_sbml_model('../../../tests/model_small_example.xml')

# Create copy of model to which pathway will be added
ecc_14bdo = ecc.copy()

# Add metabolites to model
ecc_14bdo.add_metabolites([ cobra.Metabolite('sucsal_c'),# Succinic semialdehyde
                            cobra.Metabolite('4hb_c'),   # 4-Hydroxybutanoate
                            cobra.Metabolite('4hbcoa_c'),# 4-Hydroxybutyryl-CoA
                            cobra.Metabolite('4hbal_c'), # 4-Hydroxybutanal
                            cobra.Metabolite('14bdo_c'), # Butane-1,4-diol (cytopl.)
                            cobra.Metabolite('14bdo_p'), # Butane-1,4-diol (peripl.)
                            cobra.Metabolite('14bdo_e')  # Butane-1,4-diol (extrac.)
                            ])

# Create reactions
SSCOARx    = cobra.Reaction('SSCOARx')
AKGDC      = cobra.Reaction('AKGDC')
HBD        = cobra.Reaction('4HBD')
HBCT       = cobra.Reaction('4HBCT')
HBDH       = cobra.Reaction('4HBDH')
HBDx       = cobra.Reaction('4HBDx')
BDOtpp     = cobra.Reaction('14BDOtpp')
BDOtex     = cobra.Reaction('14BDOtex')
EX_14bdo_e = cobra.Reaction('EX_14bdo_e')

# Add reactions to model
ecc_14bdo.add_reactions([SSCOARx,
                         AKGDC,
                         HBD,
                         HBCT,
                         HBDH,
                         HBDx,
                         BDOtpp,
                         BDOtex,
                         EX_14bdo_e])

# Define reaction equations
SSCOARx.reaction    = '1 h_c + 1 nadph_c + 1 succoa_c -> 1 coa_c + 1 nadp_c + 1 sucsal_c'
AKGDC.reaction      = '1 akg_c + 1 h_c -> 1 co2_c + 1 sucsal_c'
HBD.reaction        = '1 h_c + 1 nadh_c + 1 sucsal_c  -> 1 4hb_c + 1 nad_c'
HBCT.reaction       = '1 4hb_c + 1 accoa_c            -> 1 4hbcoa_c + 1 ac_c'
HBDH.reaction       = '1 4hbcoa_c + 1 h_c + 1 nadh_c  -> 1 4hbal_c + 1 coa_c + 1 nad_c'
HBDx.reaction       = '1 4hbal_c + 1 h_c + 1 nadh_c   -> 1 14bdo_c + 1 nad_c'
BDOtpp.reaction     = '1 14bdo_c                      -> 1 14bdo_p'
BDOtex.reaction     = '1 14bdo_p                      -> 1 14bdo_e'
```

(continues on next page)

(continued from previous page)

```
EX_14bdo_e.reaction = '1 14bdo_e                    ->'

# Verify that pathway is operational
sol = sd.fba(ecc_14bdo,obj='EX_14bdo_e',obj_sense='max')
print(f"Maximum possible 1,4-BDO synthesis rate: {sol.objective_value}.")
```

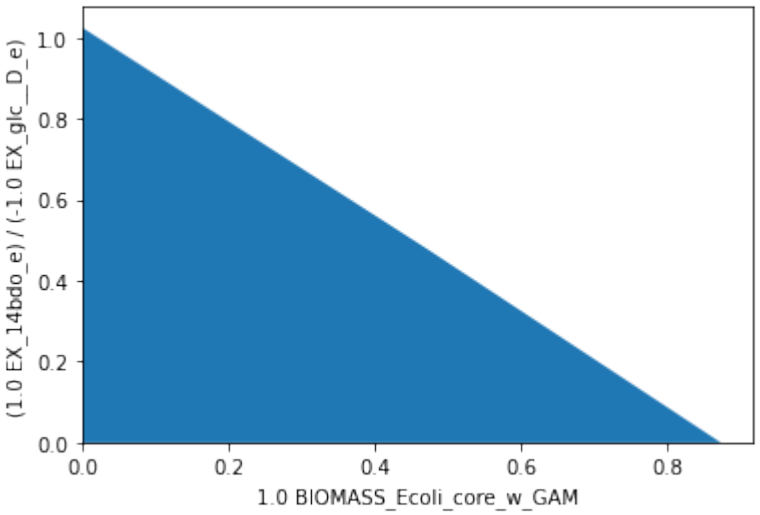
```
Maximum possible 1,4-BDO synthesis rate: 10.252923076923619.
```

4.5.2 Example 1: Strain designs with a minimum product (1,4-butanediol) yield (SUCP strain design)

We may use the MCS approach to generate strain designs with a guaranteed minimum yield of 1,4-butanediol (1,4-BDO) on glucose, i.e. *substrate-uptake-coupled production (SUCP)*.

We may plot the relationship between growth and product yield to get a feel for the production potential.

[2]:
```
sd.plot_flux_space(ecc_14bdo, ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e','-EX_glc__D_e
↪')));
```



Bacterial growth seems to compete with 1,4-BDO production. The maximum theoretical 1,4-BDO yield, is slightly above 1. We may now try to set realistic strain design goals. Assuming that knockout sets can be found the force a product yield above 0.3, while a growth rate of 0.2 is still attainable, we can specify according flux subspaces for protection or deletion through inequalities.

Since we decided to enforce a yield above 0.3, we aim to suppress all flux states with a yied inferior to this. Hence, we can describe the subspace of undesired fluxes with the inequality:

$$\frac{v_{1,4-BDO}}{v_{Glc,up}} \leq 0.3$$

which can be linearized unter the assumption that $v_{Glc_up} > 0$ to:

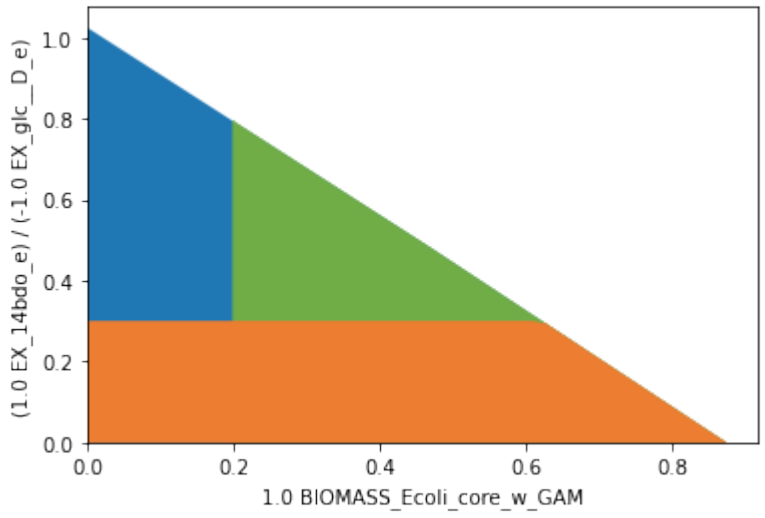
$$v_{1,4-BDO} - 0.3\, v_{Glc,up} \leq 0$$

The flux states, which we aim to protect (at least partically) can be described by:

$$v_{growth} \geq 0.2$$

We can use the plotting function to visualize the flux subspaces that we would like to suppress (orange) or protect (green).

[3]:
```
import matplotlib.pyplot as plt
# Wild-type plot
datapoints, triang, plot1 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e',
↪'-EX_glc__D_e')),
                                               show=False);
_,          _,      plot2 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e',
↪'-EX_glc__D_e')),
                                               constraints='BIOMASS_Ecoli_core_w_GAM>=0.2
↪',
                                               show=False);
plot2.set_facecolor('#70AD47')
plot2.set_edgecolor('#70AD47')
# pGCP design plot
_,          _,      plot3 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e',
↪'-EX_glc__D_e')),
                                               # The sign of the glucose exchange␣
↪reaction is flipped since
                                               # reaction is defined in the direction of␣
↪secretion.
                                               constraints='EX_14bdo_e + 0.3 EX_glc__D_e
↪<= 0',
                                               show=False);
plot3.set_facecolor('#ED7D31')
plot3.set_edgecolor('#ED7D31')
# adjust axes limits and show plot
plot3.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot3.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```



The StrainDesign package uses so-called “strain design modules” to specify strain design goals. In the case of MCS, the goal is to suppress and protect flux spaces denoted by (sets of) linear inequalities. In the following we specify these

modules.

[4]:
```
module_suppress = sd.SDModule(ecc_14bdo,sd.names.SUPPRESS,constraints='EX_14bdo_e + 0.3␣
↪EX_glc__D_e <= 0')
module_protect  = sd.SDModule(ecc_14bdo,sd.names.PROTECT, constraints='BIOMASS_Ecoli_
↪core_w_GAM>=0.2')
```

It must be noted that suppressed or protected flux spaces are not only denoted by single inequalities, but are always subspaces of the original model. For instance, the set of flux vectors in the "suppressed" flux space is constrained by the specified inequality

$$v_{1,4-BDO}^{supp} - 0.3\, v_{Glc,up}^{supp} \leq 0$$

but also by the model constraints

$$\mathbf{S} \cdot \mathbf{v}^{\mathbf{supp}} = \mathbf{0}$$

$$\mathbf{lb} \leq \mathbf{v}^{\mathbf{supp}} \leq \mathbf{ub}$$

We must pay close attention when specifying *protect* and *suppress' modules, as, in either case, we avoid to include the zero flux vector :math:`mathbf{v = 0}` in the according subspace. If the zero vector were contained in the*suppressed* flux space, the subspace could not be suppressed, since the zero vector can always be attained, even if all reactions were to be blocked. If the zero vector is part of the *protected* flux space, the module will be ineffective, since reactions knockouts can never disrupt this vector.

In the case of the e_coli_core, it is unnecessary to exclude the zero vector explicitly, since it is already excluded by default due to the minimum ATP maintenance demand. If this was not the case, one would need a auxiliary constraint, e.g., $ v_{Glc,up} :nbsphinx-math:`ge `0.1$.

We can now proceed with the strain design computation. Since we normally don't know if solutions to our strain design problems exist, we will start the computation with the most relaxed settings possible. This means, we compute only one single solution, within a given MILP timelimit of 5 minutes (for genome-scale setups, this should be increased to an hour), while omitting the minimality demand in the solutions and allow up to 30 knockouts. For this initial approach, we also activate logging to follow the progress of the computation.

[5]:
```
import logging
logging.basicConfig(level=logging.INFO)
# Compute strain designs
sols = sd.compute_strain_designs(ecc_14bdo,
                                 sd_modules = [module_suppress, module_protect],
                                 time_limit = 300,
                                 max_solutions = 1,
                                 max_cost = 10,
                                 solution_approach = sd.names.ANY)
# Print solutions
print(f"One compressed solution with cost {sols.sd_cost[0]} found and "+\
      f"expanded to {len(sols.reaction_sd)} solutions in the uncompressed netork.")
print(f"Example knockout set: {[s for s in sols.reaction_sd[0]]}")
```

```
INFO:root:Preparing strain design computation.
INFO:root:  Using cplex for solving LPs during preprocessing.
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:  FVA to identify blocked reactions and irreversibilities.
INFO:root:  FVA(s) to identify essential reactions.
INFO:root:Compressing Network (104 reactions).
INFO:root:  Removing blocked reactions.
```

(continues on next page)

(continued from previous page)

```
INFO:root:  Translating stoichiometric coefficients to rationals.
INFO:root:  Removing conservation relations.
INFO:root:  Compression 1: Applying compression from EFM-tool module.
INFO:root:  Reduced to 61 reactions.
INFO:root:  Compression 2: Lumping parallel reactions.
INFO:root:  Last step could not reduce size further (61 reactions).
INFO:root:  Network compression completed. (1 compression iterations)
INFO:root:  Translating stoichiometric coefficients back to float.
INFO:root:  FVA(s) in compressed model to identify essential reactions.
INFO:root:Finished preprocessing:
INFO:root:  Model size: 61 reactions, 36 metabolites
INFO:root:  50 targetable reactions
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:Constructing strain design MILP for solver: cplex.
INFO:root:  Bounding MILP.
INFO:root:Finding (also non-optimal) strain designs ...
INFO:root:Searching in full search space.
INFO:root:Minimizing number of interventions in subspace with 10 possible targets.
INFO:root:Strain design with cost 8.0: {'PGI': -1, 'ACALD': -1, 'ACKr*PTAr': -1,
↪'PPS*ADK1': -1, 'PYRt2*EX_pyr_e': -1, 'D_LACt2*EX_lac__D_e*LDH_D': -1, 'SUCOAS': -1,
↪'ICL*MALS': -1}
INFO:root:Finished solving strain design MILP.
INFO:root:1 solutions to MILP found.
INFO:root:  Decompressing.
INFO:root:48 solutions found.
```

```
One compressed solution with cost 8.0 found and expanded to 48 solutions in the␣
↪uncompressed netork.
Example knockout set: ['PGI', 'ACALD', 'SUCOAS', 'ACKr', 'PPS', 'PYRt2', 'D_LACt2', 'ICL
↪']
```

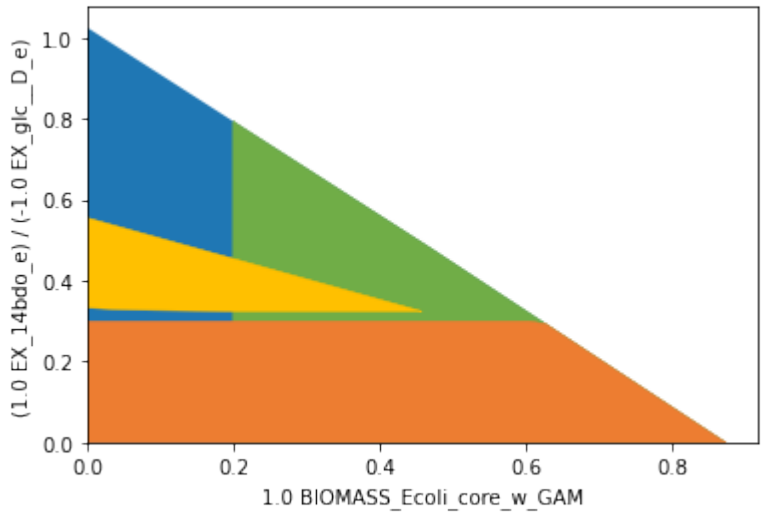
We may plot the computed strain design (yellow) on top of the wild type model (blue), the suppressed fluxes (orange) and the protected fluxes (green). The designed strain is forced to produce 1,4-butanediol but is still able to grow at a rate higher than 0.2 1/h.

[6]:
```
import matplotlib.pyplot as plt
# Wild-type plot
datapoints, triang, plot1 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e',
↪'-EX_glc__D_e')),
                                               show=False);
_,          _,      plot2 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e',
↪'-EX_glc__D_e')),
                                               constraints='BIOMASS_Ecoli_core_w_GAM>=0.2
↪',
                                               show=False);
plot2.set_facecolor('#70AD47')
plot2.set_edgecolor('#70AD47')
# pGCP design plot
_,          _,      plot3 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e',
↪'-EX_glc__D_e')),
```

(continues on next page)

(continued from previous page)

```
                                                # The sign of the glucose exchange␣
↪reaction is flipped since
                                                # reaction is defined in the direction of␣
↪secretion.
                                                constraints='EX_14bdo_e + 0.3 EX_glc__D_e
↪<= 0',
                                                show=False);
plot3.set_facecolor('#ED7D31')
plot3.set_edgecolor('#ED7D31')
# plotting designed strain
knockouts = [[{s:1.0},'=',0.0] for s in sols.reaction_sd[0]]
_,          _,      plot4 = sd.plot_flux_space(ecc_14bdo,
                                                ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e',
↪'-EX_glc__D_e')),
                                                # The sign of the glucose exchange␣
↪reaction is flipped since
                                                # reaction is defined in the direction of␣
↪secretion.
                                                constraints=knockouts,
                                                show=False);
plot4.set_facecolor('#FFC000')
plot4.set_edgecolor('#FFC000')
# adjust axes limits and show plot
plot4.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot4.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```



An easy way to compute gene-based MCS is to set the `gene_kos` parameter to True. All genes then are regarded as knockout candidates.

[7]:
```
import logging
logging.basicConfig(level=logging.INFO)
# Compute strain designs
sols = sd.compute_strain_designs(ecc_14bdo,
                                 sd_modules = [module_suppress, module_protect],
```

(continues on next page)

(continued from previous page)

```
                                time_limit = 300,
                                max_solutions = 1,
                                max_cost = 30,
                                solution_approach = sd.names.ANY,
                                gene_kos = True)
# Print solutions
print(f"One compressed solution with cost {sols.sd_cost[0]} found and "+\
      f"expanded to {len(sols.gene_sd)} solutions in the uncompressed netork.")
print(f"Example knockout set: {[s for s in sols.gene_sd[0]]}")
```

```
INFO:root:Preparing strain design computation.
INFO:root:  Using cplex for solving LPs during preprocessing.
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:  FVA to identify blocked reactions and irreversibilities.
INFO:root:  FVA(s) to identify essential reactions.
INFO:root:Preprocessing GPR rules (137 genes, 69 gpr rules).
INFO:root:  Simplifyied to 113 genes and 56 gpr rules.
INFO:root:  Extending metabolic network with gpr associations.
INFO:root:Compressing Network (307 reactions).
INFO:root:  Removing blocked reactions.
INFO:root:  Translating stoichiometric coefficients to rationals.
INFO:root:  Removing conservation relations.
INFO:root:  Compression 1: Applying compression from EFM-tool module.
INFO:root:  Reduced to 164 reactions.
INFO:root:  Compression 2: Lumping parallel reactions.
INFO:root:  Reduced to 142 reactions.
INFO:root:  Compression 3: Applying compression from EFM-tool module.
INFO:root:  Reduced to 135 reactions.
INFO:root:  Compression 4: Lumping parallel reactions.
INFO:root:  Reduced to 134 reactions.
INFO:root:  Compression 5: Applying compression from EFM-tool module.
INFO:root:  Reduced to 133 reactions.
INFO:root:  Compression 6: Lumping parallel reactions.
INFO:root:  Last step could not reduce size further (133 reactions).
INFO:root:  Network compression completed. (5 compression iterations)
INFO:root:  Translating stoichiometric coefficients back to float.
INFO:root:  FVA(s) in compressed model to identify essential reactions.
INFO:root:Finished preprocessing:
INFO:root:  Model size: 133 reactions, 78 metabolites
INFO:root:  47 targetable reactions
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:Constructing strain design MILP for solver: cplex.
INFO:root:  Bounding MILP.
INFO:root:Finding (also non-optimal) strain designs ...
INFO:root:Searching in full search space.
INFO:root:Minimizing number of interventions in subspace with 11 possible targets.
INFO:root:Strain design with cost 12.0: {'PYK*pykF*R0_g_b1676_or_g_b1854*pykA*R1_g_b1676_
↪or_g_b1854': -1, 'ME1*maeA': -1, 'ME2*maeB': -1, 'PDH*aceE*aceF*R_g_b0114_and_g_b0115_
↪and_g_b0116': -1, 'pgi': -1, 'kgtP': -1, 'sucC*sucD*R_g_b0728_and_g_b0729': -1,
↪'fumC*R0_g_b1611_or_g_b1612_or_g_b4122*fumA*R1_g_b1611_or_g_b1612_or_g_b4122*fumB*R2_g_
↪b1611_or_g_b1612_or_g_b4122': -1, 'gltP': -1}
INFO:root:Finished solving strain design MILP.
```

(continues on next page)

(continued from previous page)

```
INFO:root:1 solutions to MILP found.
INFO:root:  Decompressing.
INFO:root:  Preparing (reaction-)phenotype prediction of gene intervention strategies.
INFO:root:4 solutions found.
```

```
One compressed solution with cost 12.0 found and expanded to 4 solutions in the␣
↪uncompressed netork.
Example knockout set: ['pgi', 'kgtP', 'gltP', 'maeA', 'maeB', 'aceE', 'pykF', 'pykA',
↪'sucC', 'fumC', 'fumA', 'fumB']
```

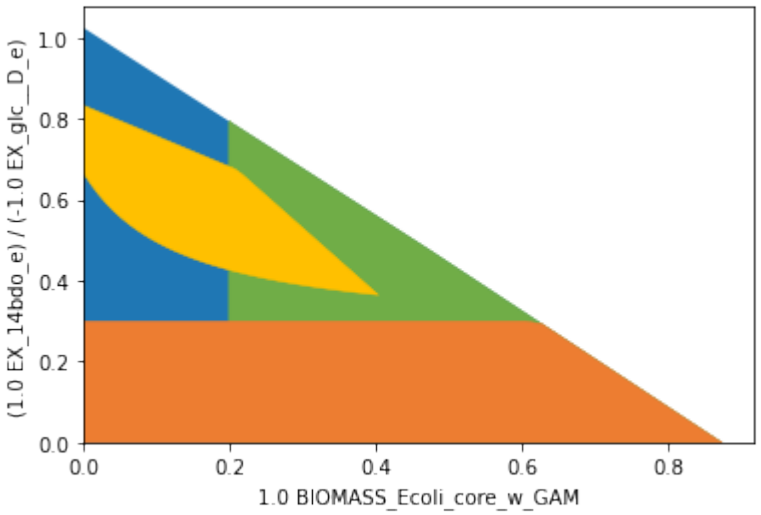
We may now analyze the resulting strain designs stored in `sols.gene_sd`. When strain designs were computed with genetic interventions, such interventions need to be translated back to the reaction level in order to enable network analysis tools. This is done automatically at the end of each computation. The reaction-intervention equivalent can be accessed through the field `sols.reaction_sd`.

[8]:
```
import matplotlib.pyplot as plt
# Wild-type plot
datapoints, triang, plot1 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e',
↪'-EX_glc__D_e')),
                                               show=False);
_,          _,      plot2 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e',
↪'-EX_glc__D_e')),
                                               constraints='BIOMASS_Ecoli_core_w_GAM>=0.2
↪',
                                               show=False);
plot2.set_facecolor('#70AD47')
plot2.set_edgecolor('#70AD47')
# pGCP design plot
_,          _,      plot3 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e',
↪'-EX_glc__D_e')),
                                               # The sign of the glucose exchange␣
↪reaction is flipped since
                                               # reaction is defined in the direction of␣
↪secretion.
                                               constraints='EX_14bdo_e + 0.3 EX_glc__D_e
↪<= 0',
                                               show=False);
plot3.set_facecolor('#ED7D31')
plot3.set_edgecolor('#ED7D31')
# plotting designed strain
knockouts = [[{s:1.0},'=',0.0] for s in sols.reaction_sd[0]]
_,          _,      plot4 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e',
↪'-EX_glc__D_e')),
                                               # The sign of the glucose exchange␣
↪reaction is flipped since
                                               # reaction is defined in the direction of␣
↪secretion.
                                               constraints=knockouts,
                                               show=False);
```

(continues on next page)

(continued from previous page)

```
plot4.set_facecolor('#FFC000')
plot4.set_edgecolor('#FFC000')
# adjust axes limits and show plot
plot4.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot4.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```



As before, the designed strain complies with the specified MCS strain design goals.

In step (1), we added two pathways that can be used to produce the intermediate succinic semialdehyde branching from alpha-ketoglutarate. In fact, only one reaction, AKGDC or SSCOARx, is needed for producing 1,4-BDO. If we want to save experimental effort, we may want to consider the addition of only one of the two respective enzymes.

For the MCS computation, we will therefore mark the reactions AKGDC and SSCOARx as "addition candidates". The algorithm will now combine a knockout strategy with the addition of one of the two reactions. For this computation example we will: - relax the strain design demands. - allow the knockout of all genes (apart from the pseudogene for spontanous reactions) - allow the knockout of O2, thereby simulating anaerobic conditions - allow the limitation of O2 uptake down to a rate of below 1 - treat AKGDC and SSCOARx as addition candidates (Instead of reaction additions, one could also use gene-additions. Since the gene-reaction-association is 1:1, the computation results would be identical.). We associate different intervention costs to the addition of these reactions. AKGDC addition has the cost 1, SSCOARx addition has the cost 5.

[9]:
```
module_suppress = sd.SDModule(ecc_14bdo,sd.names.SUPPRESS,constraints='EX_14bdo_e + 0.05␣
↪EX_glc__D_e <= 0')
module_protect  = sd.SDModule(ecc_14bdo,sd.names.PROTECT, constraints='BIOMASS_Ecoli_
↪core_w_GAM>=0.02')

# allow all gene knockouts except for spontanuos
gko_cost = {g.name:1 for g in ecc_14bdo.genes}
gko_cost.pop('s0001')
# possible knockout of O2
ko_cost = {'EX_o2_e': 1}
# possible throttling of O2 uptake
reg_cost = {'-EX_o2_e <= 1' : 1}
# addition candidates
ki_cost = {'AKGDC':1, 'SSCOARx':5}
```

(continues on next page)

(continued from previous page)

```
# compute strain designs
import logging
logging.basicConfig(level=logging.INFO)
sols = sd.compute_strain_designs(ecc_14bdo,
                                 sd_modules = [module_suppress, module_protect],
                                 max_cost = 9,
                                 solution_approach = sd.names.POPULATE,
                                 ko_cost = ko_cost,
                                 gko_cost = gko_cost,
                                 reg_cost = reg_cost,
                                 ki_cost = ki_cost)
```

```
INFO:root:Preparing strain design computation.
INFO:root:  Using cplex for solving LPs during preprocessing.
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:  FVA to identify blocked reactions and irreversibilities.
INFO:root:  FVA(s) to identify essential reactions.
INFO:root:Preprocessing GPR rules (137 genes, 69 gpr rules).
INFO:root:  Simplifyied to 111 genes and 52 gpr rules.
INFO:root:  Extending metabolic network with gpr associations.
INFO:root:Compressing Network (301 reactions).
INFO:root:  Removing blocked reactions.
INFO:root:  Translating stoichiometric coefficients to rationals.
INFO:root:  Removing conservation relations.
INFO:root:  Compression 1: Applying compression from EFM-tool module.
INFO:root:  Reduced to 155 reactions.
INFO:root:  Compression 2: Lumping parallel reactions.
INFO:root:  Reduced to 133 reactions.
INFO:root:  Compression 3: Applying compression from EFM-tool module.
INFO:root:  Reduced to 126 reactions.
INFO:root:  Compression 4: Lumping parallel reactions.
INFO:root:  Reduced to 125 reactions.
INFO:root:  Compression 5: Applying compression from EFM-tool module.
INFO:root:  Reduced to 124 reactions.
INFO:root:  Compression 6: Lumping parallel reactions.
INFO:root:  Last step could not reduce size further (124 reactions).
INFO:root:  Network compression completed. (5 compression iterations)
INFO:root:  Translating stoichiometric coefficients back to float.
INFO:root:  FVA(s) in compressed model to identify essential reactions.
INFO:root:Finished preprocessing:
INFO:root:  Model size: 124 reactions, 73 metabolites
INFO:root:  48 targetable reactions
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:Constructing strain design MILP for solver: cplex.
INFO:root:  Bounding MILP.
INFO:root:Enumerating strain designs ...
INFO:root:Strain designs with cost 7.0: [{'CYTBD*EX_o2_e*O2t*cydA*cydB*R_g_b0733_and_g_
↪b0734*R0_g_b0733_and_g_b0734_or_g_b0978_and_g_b0979*cbdA*cbdB*R_g_b0978_and_g_b0979*R1_
↪g_b0733_and_g_b0734_or_g_b0978_and_g_b0979': -1.0, 'SSCOARx': 1.0, 'pgi': -1.0}]
INFO:root:Strain designs with cost 9.0: [{'SSCOARx': 1.0, 'pgi': -1.0, 'focA*R0_g_b0904_
↪or_g_b2492*focB*R1_g_b0904_or_g_b2492': -1.0, 'n-1.0_EX_o2_e_le_1p0': -1.0}]
```

(continues on next page)

(continued from previous page)

```
INFO:root:Strain designs with cost 9.0: [{'PFL*EX_for_e*pflA*pflB*R_g_b0902_and_g_
↪b0903*R0_g_b0902_and_g_b0903_or_g_b0902_and_g_b3114_or_g_b3951_and_g_b3952*tdcE*R_g_
↪b0902_and_g_b3114*R1_g_b0902_and_g_b0903_or_g_b0902_and_g_b3114_or_g_b3951_and_g_
↪b3952...': -1.0, 'SSCOARx': 1.0, 'pgi': -1.0, 'n-1.0_EX_o2_e_le_1p0': -1.0}]
INFO:root:Strain designs with cost 9.0: [{'PYK*pykF*R0_g_b1676_or_g_b1854*pykA*R1_g_
↪b1676_or_g_b1854': -1.0, 'ICL*MALS*aceA*glcB*R0_g_b2976_or_g_b4014*aceB*R1_g_b2976_or_
↪g_b4014': -1.0, 'AKGDC': 1.0, 'pgi': -1.0, 'kgtP': -1.0, 'lpd': -1.0, 'gltP': -1.0,
↪'mdh': -1.0}]
INFO:root:Strain designs with cost 9.0: [{'AKGDH*sucA*sucB*R_g_b0116_and_g_b0726_and_g_
↪b0727': -1.0, 'PYK*pykF*R0_g_b1676_or_g_b1854*pykA*R1_g_b1676_or_g_b1854': -1.0,
↪'ICL*MALS*aceA*glcB*R0_g_b2976_or_g_b4014*aceB*R1_g_b2976_or_g_b4014': -1.0, 'AKGDC':␣
↪1.0, 'pgi': -1.0, 'kgtP': -1.0, 'gltP': -1.0, 'mdh': -1.0}]
INFO:root:Strain designs with cost 9.0: [{'PYK*pykF*R0_g_b1676_or_g_b1854*pykA*R1_g_
↪b1676_or_g_b1854': -1.0, 'ICL*MALS*aceA*glcB*R0_g_b2976_or_g_b4014*aceB*R1_g_b2976_or_
↪g_b4014': -1.0, 'AKGDC': 1.0, 'pgi': -1.0, 'kgtP': -1.0, 'sucC*sucD*R_g_b0728_and_g_
↪b0729': -1.0, 'gltP': -1.0, 'mdh': -1.0}]
INFO:root:Strain designs with cost 9.0: [{'SSCOARx': 1.0, 'pgi': -1.0, 'mhpF*R0_g_b0351_
↪or_g_b1241': -1.0, 'adhE': -1.0, 'n-1.0_EX_o2_e_le_1p0': -1.0}]
INFO:root:Finished solving strain design MILP.
INFO:root:7 solutions to MILP found.
INFO:root:  Decompressing.
INFO:root:  Preparing (reaction-)phenotype prediction of gene intervention strategies.
INFO:root:14 solutions found.
```

[10]:
```
for i,s in enumerate(sols.gene_sd):
    st = [['+'+t if v>0 else '-'+t][0] for t,v in s.items() if v != 0]
    print('solution '+str(i+1)+': '+', '.join(st))
```

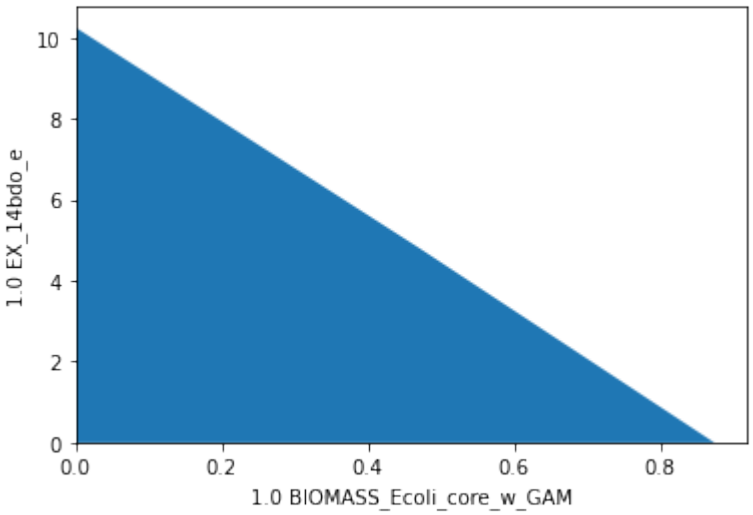
```
solution 1: +SSCOARx, -pgi, -EX_o2_e
solution 2: +SSCOARx, -pgi, -cydB, -cbdA
solution 3: +SSCOARx, -pgi, -cydB, -cbdB
solution 4: +AKGDC, -pgi, -kgtP, -gltP, -mdh, -aceA, -pykF, -pykA, -sucC
solution 5: +AKGDC, -pgi, -kgtP, -gltP, -mdh, -aceA, -pykF, -pykA, -sucD
solution 6: +SSCOARx, -pgi, -pflA, -pflD, +-EX_o2_e <= 1
solution 7: +SSCOARx, -pgi, -pflA, -pflC, +-EX_o2_e <= 1
solution 8: +AKGDC, -pgi, -kgtP, -gltP, -mdh, -sucA, -aceA, -pykF, -pykA
solution 9: +AKGDC, -pgi, -kgtP, -gltP, -mdh, -sucB, -aceA, -pykF, -pykA
solution 10: +SSCOARx, -pgi, -cydA, -cbdA
solution 11: +SSCOARx, -pgi, -cydA, -cbdB
solution 12: +SSCOARx, -pgi, -focA, -focB, +-EX_o2_e <= 1
solution 13: +SSCOARx, -pgi, -adhE, -mhpF, +-EX_o2_e <= 1
solution 14: +AKGDC, -pgi, -kgtP, -lpd, -gltP, -mdh, -aceA, -pykF, -pykA
```

4.5.3 Example 2: Enforce product (1,4-BDO) synthesis at all growth states (dGCP strain design)

For computing dGCP strain designs with MCS, we follow the identical steps from Example 1:

1) The production pathways, i.e., their metabolites and reactions are added to the model.
2) Analysis tools are used to identify adequate strain design goals.
3) Set up the according strain design problem by specifying two strain design *modules*, one that demand the suppression of flux states with low product yields, the other one protecting the functions that are essential for bacterial growth.
4) The strain design function is called, passing the model and the strain design module(s) as function arguments.
5) The results are analyzed.

[11]:
```
# Plot production envelope
sd.plot_flux_space(ecc_14bdo, ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'));
```



For dGCP, we demand that all flux states of microbial growth carry production. Ideally, we want to (a) ensure a minimum ratio of product synthesis rate and growth rate $\frac{v_{product}}{v_{growth}} > Y^{min}_{P/BM}$ (2) ensure that growth is still possible. Hence, we define one suppression and one protection module.

Suppress module (removing flux states, where $\frac{v_{product}}{v_{growth}} \leq Y^{min}_{P/BM}$):

$$v_{product} - Y^{min}_{P/BM} v_{growth} \leq 0$$

$$v_{growth} \geq 0.01\ h^{-1}$$

The latter constraint is used to explicitly exclude the $\mathbf{v} = \mathbf{0}$ vector from the suppressed flux states. Here, we use an arbitrary threshold of $Y^{min}_{P/BM} = 5$ that we inferred from the production envelope.

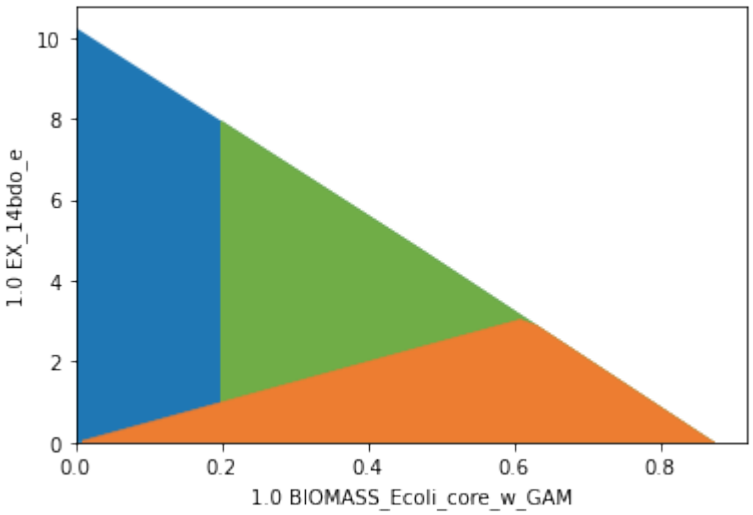
Protect module:

$$v_{growth} \geq 0.2\ h^{-1}$$

[12]:
```
module_suppress = sd.SDModule(ecc_14bdo,sd.names.SUPPRESS,constraints=['EX_14bdo_e - 5␣
↪BIOMASS_Ecoli_core_w_GAM <= 0',
                                                                        'BIOMASS_Ecoli_
↪core_w_GAM >= 0.01'])
module_protect  = sd.SDModule(ecc_14bdo,sd.names.PROTECT, constraints='BIOMASS_Ecoli_
↪core_w_GAM>=0.2')
```

Plotting the flux spaces in the production envelope returns:

[13]:
```
import matplotlib.pyplot as plt
# Wild-type plot
datapoints, triang, plot1 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'),
                                               show=False);
_,          _,      plot2 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'),
                                               constraints='BIOMASS_Ecoli_core_w_GAM>=0.2
↪',
                                               show=False);
plot2.set_facecolor('#70AD47')
plot2.set_edgecolor('#70AD47')
# pGCP design plot
_,          _,      plot3 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'),
                                               # The sign of the glucose exchange␣
↪reaction is flipped since
                                               # reaction is defined in the direction of␣
↪secretion.
                                               constraints=['EX_14bdo_e - 5 BIOMASS_
↪Ecoli_core_w_GAM <= 0',
                                                            'BIOMASS_Ecoli_core_w_GAM >=␣
↪0.01'],
                                               show=False);
plot3.set_facecolor('#ED7D31')
plot3.set_edgecolor('#ED7D31')
# adjust axes limits and show plot
plot3.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot3.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```



[14]:
```
import logging
logging.basicConfig(level=logging.ERROR)
# allow all gene knockouts except for spontanuos
gko_cost = {g.name:1 for g in ecc_14bdo.genes}
gko_cost.pop('s0001')
# possible knockout of O2
ko_cost = {'EX_o2_e': 1}
ki_cost = {'AKGDC':1, 'SSCOARx':1} # AKGDC was added in example 1.c)

# compute strain designs
sols = sd.compute_strain_designs(ecc_14bdo,
                                 sd_modules = [module_suppress, module_protect],
                                 max_solutions = 1,
                                 max_cost = 30,
                                 solution_approach = sd.names.BEST,
                                 ko_cost = ko_cost,
                                 gko_cost = gko_cost,
                                 ki_cost = ki_cost)
# Print solutions
print(f"One compressed solution with cost {sols.sd_cost[0]} found and "+\
      f"expanded to {len(sols.reaction_sd)} solutions in the uncompressed netork.")
print(f"Example intervention set: {['+'+s if v>0 else '-'+s for s,v in sols.gene_sd[0].
↪items() if v!=0]}")
print(f"Knockout set on the reaction level: {[s for s in sols.reaction_sd[0]]}")
```

```
INFO:root:Preparing strain design computation.
INFO:root:  Using cplex for solving LPs during preprocessing.
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:  FVA to identify blocked reactions and irreversibilities.
INFO:root:  FVA(s) to identify essential reactions.
INFO:root:Preprocessing GPR rules (137 genes, 69 gpr rules).
INFO:root:  Simplifyied to 111 genes and 52 gpr rules.
INFO:root:  Extending metabolic network with gpr associations.
INFO:root:Compressing Network (299 reactions).
INFO:root:  Removing blocked reactions.
```

(continues on next page)

(continued from previous page)

```
INFO:root:  Translating stoichiometric coefficients to rationals.
INFO:root:  Removing conservation relations.
INFO:root:  Compression 1: Applying compression from EFM-tool module.
INFO:root:  Reduced to 153 reactions.
INFO:root:  Compression 2: Lumping parallel reactions.
INFO:root:  Reduced to 131 reactions.
INFO:root:  Compression 3: Applying compression from EFM-tool module.
INFO:root:  Reduced to 124 reactions.
INFO:root:  Compression 4: Lumping parallel reactions.
INFO:root:  Reduced to 123 reactions.
INFO:root:  Compression 5: Applying compression from EFM-tool module.
INFO:root:  Reduced to 122 reactions.
INFO:root:  Compression 6: Lumping parallel reactions.
INFO:root:  Last step could not reduce size further (122 reactions).
INFO:root:  Network compression completed. (5 compression iterations)
INFO:root:  Translating stoichiometric coefficients back to float.
INFO:root:  FVA(s) in compressed model to identify essential reactions.
INFO:root:Finished preprocessing:
INFO:root:  Model size: 122 reactions, 72 metabolites
INFO:root:  47 targetable reactions
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:Constructing strain design MILP for solver: cplex.
INFO:root:  Bounding MILP.
INFO:root:Finding optimal strain designs ...
INFO:root:Strain design with cost 6.0: {'ME2*maeB': -1, 'AKGDC': 1, 'kgtP': -1,
↪'sucC*sucD*R_g_b0728_and_g_b0729': -1, 'pntB*pntA*R_g_b1602_and_g_b1603': -1, 'zwf': -
↪1}
INFO:root:Finished solving strain design MILP.
INFO:root:1 solutions to MILP found.
INFO:root:  Decompressing.
INFO:root:  Preparing (reaction-)phenotype prediction of gene intervention strategies.
INFO:root:4 solutions found.
```

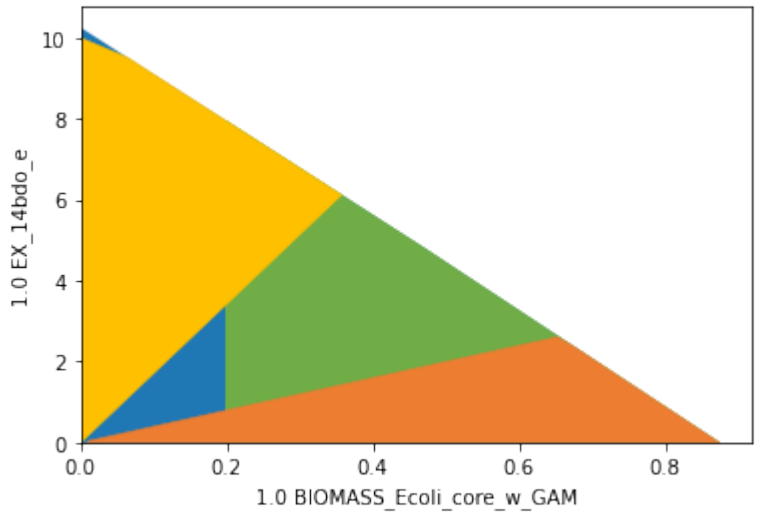
```
One compressed solution with cost 6.0 found and expanded to 4 solutions in the␣
↪uncompressed netork.
Example intervention set: ['+AKGDC', '-kgtP', '-zwf', '-maeB', '-sucC', '-pntB']
Knockout set on the reaction level: ['SUCOAS', 'G6PDH2r', 'ME2', 'THD2', 'AKGt2r', 'AKGDC
↪', 'SSCOARx']
```

[15]:
```
import matplotlib.pyplot as plt
# Wild-type plot
datapoints, triang, plot1 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'),
                                               show=False);
_,          _,      plot2 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'),
                                               constraints='BIOMASS_Ecoli_core_w_GAM>=0.2
↪',
                                               show=False);
plot2.set_facecolor('#70AD47')
plot2.set_edgecolor('#70AD47')
# pGCP design plot
```

(continues on next page)

(continued from previous page)

```
_,          _,      plot3 = sd.plot_flux_space(ecc_14bdo,
                                                ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'),
                                                # The sign of the glucose exchange␣
↪reaction is flipped since
                                                # reaction is defined in the direction of␣
↪secretion.
                                                constraints=['EX_14bdo_e - 4 BIOMASS_
↪Ecoli_core_w_GAM <= 0',
                                                             'BIOMASS_Ecoli_core_w_GAM >=␣
↪0.01'],
                                                show=False);
plot3.set_facecolor('#ED7D31')
plot3.set_edgecolor('#ED7D31')
# plotting designed strain
interventions = [[{s:1.0},'=',0.0] for s,v in sols.reaction_sd[0].items() if v < 1]
_,          _,      plot4 = sd.plot_flux_space(ecc_14bdo,
                                                ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'),
                                                # The sign of the glucose exchange␣
↪reaction is flipped since
                                                # reaction is defined in the direction of␣
↪secretion.
                                                constraints=interventions,
                                                show=False);
plot4.set_facecolor('#FFC000')
plot4.set_edgecolor('#FFC000')
# adjust axes limits and show plot
plot4.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot4.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```



The resulting strain design shows a linear increase of ensured production with increasing growth.

4.5.4 Example 3: Suppress flux states that are optimal with respect to a pre-defined objective function (wGCP strain design)

The MCS approach also allows the suppression (or protection) of flux vectors that are optimal regarding a pre-defined objective function. With an inner objective function, the SUPPRESS module can be used to disrupt any growth-maximal flux states that don't carry production and thereby enforce - *at least* - weakly growth-coupled production. To avoid lethal knockouts and ensure that growth is still possible at reasonable growth rates, we additionally use a PROTECT region.

[16]:
```
# Wild-type plot
print('Preparing plot of wild type and strain design setup.')
datapoints, triang, plot1 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e',
↪'-EX_glc__D_e')),
                                               show=False);
# wGCP protect plot
_,          _,      plot2 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e',
↪'-EX_glc__D_e')),
                                               constraints='BIOMASS_Ecoli_core_w_GAM>=0.
↪15',
                                               show=False);
plot2.set_facecolor('#70AD47')
plot2.set_edgecolor('#70AD47')
# wGCP suppress plot
_,          _,      plot3 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e',
↪'-EX_glc__D_e')),
                                               # The sign of the glucose exchange␣
↪reaction is flipped since
                                               # reaction is defined in the direction of␣
↪secretion.
                                               constraints='EX_14bdo_e + 0.25 EX_glc__D_
↪e <= 0',
                                               show=False);
plot3.set_facecolor('#ED7D31')
plot3.set_edgecolor('#ED7D31')


print('Computing MCS.')
module_wgcp = sd.SDModule(  ecc_14bdo,sd.names.SUPPRESS,
                            inner_objective='BIOMASS_Ecoli_core_w_GAM',
                            constraints='EX_14bdo_e + 0.25 EX_glc__D_e <= 0')
module_protect = sd.SDModule(  ecc_14bdo,sd.names.PROTECT,
                            constraints='BIOMASS_Ecoli_core_w_GAM >= 0.15')

# allow all gene knockouts except for spontanuos
gko_cost = {g.name:1 for g in ecc_14bdo.genes}
gko_cost.pop('s0001')
# possible knockout of O2
ko_cost = {'EX_o2_e': 1}
# addition candidates
ki_cost = {'AKGDC':1, 'SSCOARx':1} # AKGDC was added in example 1.c)
```

(continues on next page)

(continued from previous page)

```
sols = sd.compute_strain_designs(ecc_14bdo,
                                 sd_modules = [module_wgcp,module_protect],
                                 time_limit = 300,
                                 max_solutions = 1,
                                 max_cost = 30,
                                 gko_cost = gko_cost,
                                 ko_cost = ko_cost,
                                 ki_cost = ki_cost,
                                 solution_approach = sd.names.BEST)
# Print solutions
print(f"One compressed solution with cost {sols.sd_cost[0]} found and "+\
      f"expanded to {len(sols.gene_sd)} solutions in the uncompressed netork.")
print(f"Example intervention set: {['+'+s if v>0 else '-'+s for s,v in sols.gene_sd[0].
↪items() if v!=0]}")

# wGCP design plot
interventions = [[{s:1.0},'=',0.0] for s,v in sols.reaction_sd[0].items() if v < 1]
_,          _,      plot4 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e',
↪'-EX_glc__D_e')),
                                               # The sign of the glucose exchange␣
↪reaction is flipped since
                                               # reaction is defined in the direction of␣
↪secretion.
                                               constraints=interventions,
                                               show=False);
plot4.set_facecolor('#FFC000')
plot4.set_edgecolor('#FFC000')
# adjust axes limits and show plot
plot4.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot4.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```

```
Preparing plot of wild type and strain design setup.
Computing MCS.
```

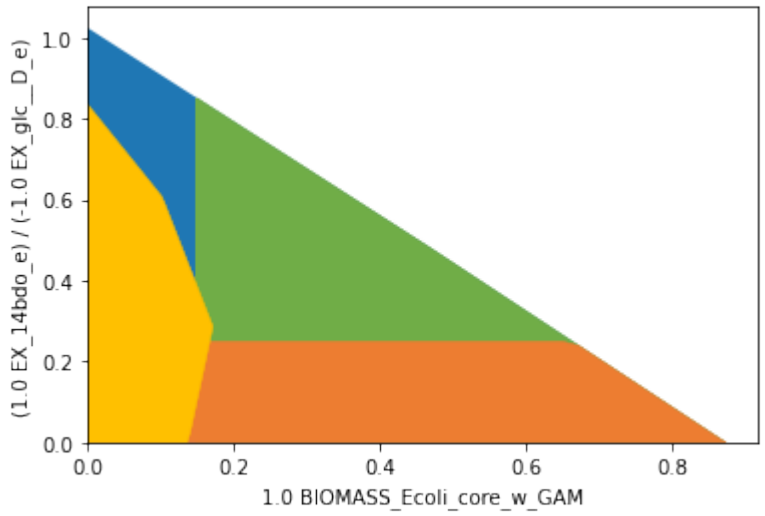
```
INFO:root:Preparing strain design computation.
INFO:root:  Using cplex for solving LPs during preprocessing.
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:  FVA to identify blocked reactions and irreversibilities.
INFO:root:  FVA(s) to identify essential reactions.
INFO:root:Preprocessing GPR rules (137 genes, 69 gpr rules).
INFO:root:  Simplifyied to 111 genes and 52 gpr rules.
INFO:root:  Extending metabolic network with gpr associations.
INFO:root:Compressing Network (299 reactions).
INFO:root:  Removing blocked reactions.
INFO:root:  Translating stoichiometric coefficients to rationals.
INFO:root:  Removing conservation relations.
INFO:root:  Compression 1: Applying compression from EFM-tool module.
INFO:root:  Reduced to 153 reactions.
INFO:root:  Compression 2: Lumping parallel reactions.
INFO:root:  Reduced to 131 reactions.
```

(continues on next page)

(continued from previous page)

```
INFO:root:  Compression 3: Applying compression from EFM-tool module.
INFO:root:  Reduced to 124 reactions.
INFO:root:  Compression 4: Lumping parallel reactions.
INFO:root:  Reduced to 123 reactions.
INFO:root:  Compression 5: Applying compression from EFM-tool module.
INFO:root:  Reduced to 122 reactions.
INFO:root:  Compression 6: Lumping parallel reactions.
INFO:root:  Last step could not reduce size further (122 reactions).
INFO:root:  Network compression completed. (5 compression iterations)
INFO:root:  Translating stoichiometric coefficients back to float.
INFO:root:  FVA(s) in compressed model to identify essential reactions.
INFO:root:Finished preprocessing:
INFO:root:  Model size: 122 reactions, 72 metabolites
INFO:root:  47 targetable reactions
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:Constructing strain design MILP for solver: cplex.
INFO:root:  Bounding MILP.
INFO:root:Finding optimal strain designs ...
INFO:root:Strain design with cost 4.0: {'CYTBD*EX_o2_e*O2t*cydA*cydB*R_g_b0733_and_g_
↪b0734*R0_g_b0733_and_g_b0734_or_g_b0978_and_g_b0979*cbdA*cbdB*R_g_b0978_and_g_b0979*R1_
↪g_b0733_and_g_b0734_or_g_b0978_and_g_b0979': -1, 'SSCOARx': 1, 'mhpF*R0_g_b0351_or_g_
↪b1241': -1, 'adhE': -1}
INFO:root:Finished solving strain design MILP.
INFO:root:1 solutions to MILP found.
INFO:root:  Decompressing.
INFO:root:  Preparing (reaction-)phenotype prediction of gene intervention strategies.
INFO:root:5 solutions found.
```

```
One compressed solution with cost 4.0 found and expanded to 5 solutions in the␣
↪uncompressed netork.
Example intervention set: ['+SSCOARx', '-adhE', '-EX_o2_e', '-mhpF']
```



As specified by the additional *inner objective function*, product yields inferior to 0.25 (orange) are supressed at growth-maximal flux states, only. At lower growth rates, the strain design *is allowed to have lower product yields*. The protected region, marking healthy growth rates (green), has to be intersected.

4.5.5 Example 4: Protect flux states that are optimal with respect to a pre-defined objective function (pGCP strain design)

The following setup shows how this function can be used to generate strain designs with at least potentially growth-coupled production. We specify a PROTECT module to ensure that there are growth-maximal flux states with a growth rate of at least 0.4 and a product yield of at least $0.4\ \frac{mol_{1,4\text{-}BDO}}{mol_{D\text{-}Glucose}}$.

[17]:
```
# Wild-type plot
print('Preparing plot of wild type and strain design setup.')
datapoints, triang, plot1 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e',
↪'-EX_glc__D_e')),
                                               show=False);
# wGCP protect plot
_,          _,      plot2 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e',
↪'-EX_glc__D_e')),
                                               constraints=['BIOMASS_Ecoli_core_w_GAM>=0.
↪4','EX_14bdo_e + 0.4 EX_glc__D_e >= 0'],
                                               show=False);
plot2.set_facecolor('#70AD47')
plot2.set_edgecolor('#70AD47')

print('Computing MCS.')
module_pgcp = sd.SDModule(  ecc_14bdo,sd.names.PROTECT,
                            inner_objective='BIOMASS_Ecoli_core_w_GAM',
                            constraints=[   'BIOMASS_Ecoli_core_w_GAM >= 0.4',
                                            'EX_14bdo_e + 0.4 EX_glc__D_e >= 0'])

# allow all gene knockouts except for spontanuos
gko_cost = {g.name:1 for g in ecc_14bdo.genes}
gko_cost.pop('s0001')
# possible knockout of O2
ko_cost = {'EX_o2_e': 1}
# addition candidates
ki_cost = {'AKGDC':1, 'SSCOARx':1} # AKGDC was added in example 1.c)

sols = sd.compute_strain_designs(ecc_14bdo,
                                 sd_modules = module_pgcp,
                                 time_limit = 300,
                                 max_solutions = 1,
                                 max_cost = 30,
                                 gko_cost = gko_cost,
                                 ko_cost = ko_cost,
                                 ki_cost = ki_cost,
                                 solution_approach = sd.names.BEST)
# Print solutions
print(f"One compressed solution with cost {sols.sd_cost[0]} found and "+\
      f"expanded to {len(sols.gene_sd)} solutions in the uncompressed netork.")
print(f"Example intervention set: {['+'+s if v>0 else '-'+s for s,v in sols.gene_sd[0].
↪items() if v!=0]}")

# wGCP design plot
```

(continues on next page)

(continued from previous page)

```
interventions = [[{s:1.0},'=',0.0] for s,v in sols.reaction_sd[0].items() if v < 1]
_,          _,      plot4 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM',('EX_14bdo_e',
↪'-EX_glc__D_e')),
                                               # The sign of the glucose exchange␣
↪reaction is flipped since
                                               # reaction is defined in the direction of␣
↪secretion.
                                               constraints=interventions,
                                               show=False);
plot4.set_facecolor('#FFC000')
plot4.set_edgecolor('#FFC000')
# adjust axes limits and show plot
plot4.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot4.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```

```
Preparing plot of wild type and strain design setup.
Computing MCS.
```

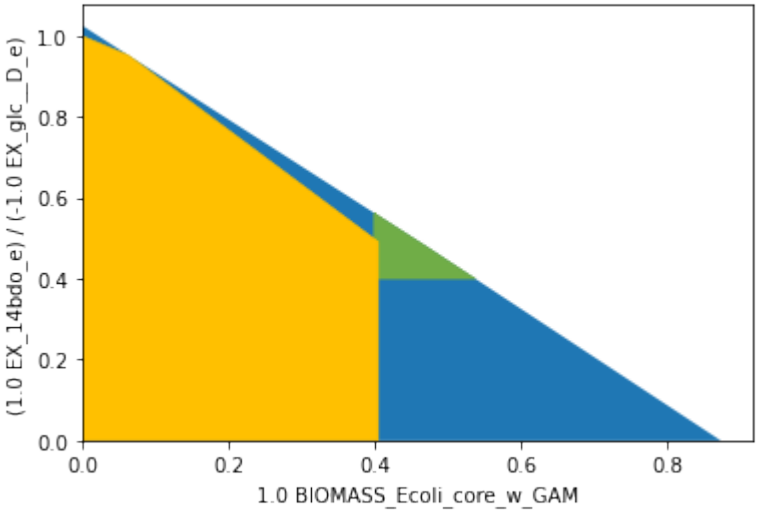
```
INFO:root:Preparing strain design computation.
INFO:root:  Using cplex for solving LPs during preprocessing.
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:  FVA to identify blocked reactions and irreversibilities.
INFO:root:  FVA(s) to identify essential reactions.
INFO:root:Preprocessing GPR rules (137 genes, 69 gpr rules).
INFO:root:  Simplifyied to 86 genes and 49 gpr rules.
INFO:root:  Extending metabolic network with gpr associations.
INFO:root:Compressing Network (267 reactions).
INFO:root:  Removing blocked reactions.
INFO:root:  Translating stoichiometric coefficients to rationals.
INFO:root:  Removing conservation relations.
INFO:root:  Compression 1: Applying compression from EFM-tool module.
INFO:root:  Reduced to 149 reactions.
INFO:root:  Compression 2: Lumping parallel reactions.
INFO:root:  Reduced to 128 reactions.
INFO:root:  Compression 3: Applying compression from EFM-tool module.
INFO:root:  Reduced to 122 reactions.
INFO:root:  Compression 4: Lumping parallel reactions.
INFO:root:  Reduced to 121 reactions.
INFO:root:  Compression 5: Applying compression from EFM-tool module.
INFO:root:  Reduced to 120 reactions.
INFO:root:  Compression 6: Lumping parallel reactions.
INFO:root:  Last step could not reduce size further (120 reactions).
INFO:root:  Network compression completed. (5 compression iterations)
INFO:root:  Translating stoichiometric coefficients back to float.
INFO:root:  FVA(s) in compressed model to identify essential reactions.
INFO:root:Finished preprocessing:
INFO:root:  Model size: 120 reactions, 71 metabolites
INFO:root:  44 targetable reactions
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:Constructing strain design MILP for solver: cplex.
INFO:root:  Bounding MILP.
```

(continues on next page)

(continued from previous page)

```
INFO:root:Finding optimal strain designs ...
INFO:root:Strain design with cost 6.0: {'PGL*GND*pgl*gnd': -1, 'ME2*maeB': -1, 'AKGDC':␣
↪1, 'adk': -1, 'pntB*pntA*R_g_b1602_and_g_b1603': -1, 'mdh': -1}
INFO:root:Finished solving strain design MILP.
INFO:root:1 solutions to MILP found.
INFO:root:  Decompressing.
INFO:root:  Preparing (reaction-)phenotype prediction of gene intervention strategies.
INFO:root:4 solutions found.
```

```
One compressed solution with cost 6.0 found and expanded to 4 solutions in the␣
↪uncompressed netork.
Example intervention set: ['+AKGDC', '-adk', '-mdh', '-pgl', '-maeB', '-pntB']
```



The designed strain has potentially growth coupled production. It must be noted that *only one PROTECT region* (green) was used in this computation. Without the *inner objective function*, the wild type strain already fulfills the modules demands, because flux states with simultanous growth and production already exist in the wildtype. In this strain design computation, we enforce that *growth-maximal* flux states exist with growth and product synthesis.

4.5.6 Example 5: All single gene knockouts that prohibit growth (synthetic lethals).

The MCS approach can be used to identify essential genes. For this type of computation we may use the MCS approach with a single suppress-module that targets flux states of microbial growth, i.e. flux states in which the inequality

$$v_{Biomass} > 0$$

approximated as

$$v_{Biomass} \geq 0.001$$

holds. We construct the corresponding module and launch an exhaustive computation of all solutions of the size 1.

[18]:
```
# Construct module
module_suppress  = sd.SDModule(ecc,sd.names.SUPPRESS, constraints='BIOMASS_Ecoli_core_w_
↪GAM>=0.001')
# Compute strain designs
```

(continues on next page)

(continued from previous page)

```
sols = sd.compute_strain_designs(ecc,
                                 sd_modules = module_suppress,
                                 max_cost = 1,
                                 solution_approach = sd.names.POPULATE,
                                 gene_kos = True)
# Print solutions
print(f"{len(sols.gene_sd)} lethal single gene knockouts were found.")
for i,sol in enumerate(sols.gene_sd):
    print(f"Solution {i+1}: {[s for s in sol][0]}")
```

```
INFO:root:Preparing strain design computation.
INFO:root:  Using cplex for solving LPs during preprocessing.
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:  FVA to identify blocked reactions and irreversibilities.
INFO:root:  FVA(s) to identify essential reactions.
INFO:root:Preprocessing GPR rules (137 genes, 69 gpr rules).
INFO:root:  Simplifyied to 135 genes and 69 gpr rules.
INFO:root:  Extending metabolic network with gpr associations.
INFO:root:Compressing Network (348 reactions).
INFO:root:  Removing blocked reactions.
INFO:root:  Translating stoichiometric coefficients to rationals.
INFO:root:  Removing conservation relations.
INFO:root:  Compression 1: Applying compression from EFM-tool module.
INFO:root:  Reduced to 198 reactions.
INFO:root:  Compression 2: Lumping parallel reactions.
INFO:root:  Reduced to 169 reactions.
INFO:root:  Compression 3: Applying compression from EFM-tool module.
INFO:root:  Reduced to 160 reactions.
INFO:root:  Compression 4: Lumping parallel reactions.
INFO:root:  Reduced to 158 reactions.
INFO:root:  Compression 5: Applying compression from EFM-tool module.
INFO:root:  Reduced to 156 reactions.
INFO:root:  Compression 6: Lumping parallel reactions.
INFO:root:  Last step could not reduce size further (156 reactions).
INFO:root:  Network compression completed. (5 compression iterations)
INFO:root:  Translating stoichiometric coefficients back to float.
INFO:root:  FVA(s) in compressed model to identify essential reactions.
INFO:root:Finished preprocessing:
INFO:root:  Model size: 156 reactions, 91 metabolites
INFO:root:  60 targetable reactions
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:Constructing strain design MILP for solver: cplex.
INFO:root:  Bounding MILP.
INFO:root:Enumerating strain designs ...
INFO:root:Strain designs with cost 1.0: [{'eno': -1.0}]
INFO:root:Strain designs with cost 1.0: [{'gapA': -1.0}]
INFO:root:Strain designs with cost 1.0: [{'CS*gltA': -1.0}]
INFO:root:Strain designs with cost 1.0: [{'icd': -1.0}]
INFO:root:Strain designs with cost 1.0: [{'EX_glc__D_e*GLCpts*ptsH*ptsI*manX*manY*manZ*R_
↪g_b1817_and_g_b1818_and_g_b1819_and_g_b2415_and_g_b2416...': -1.0}]
INFO:root:Strain designs with cost 1.0: [{'pgk': -1.0}]
INFO:root:Finished solving strain design MILP.
```

(continues on next page)

(continued from previous page)

```
INFO:root:6 solutions to MILP found.
INFO:root:  Decompressing.
INFO:root:  Preparing (reaction-)phenotype prediction of gene intervention strategies.
INFO:root:7 solutions found.
```

```
7 lethal single gene knockouts were found.
Solution 1: eno
Solution 2: gapA
Solution 3: icd
Solution 4: pgk
Solution 5: gltA
Solution 6: ptsH
Solution 7: ptsI
```

4.5.7 Example 6: Genome-scale strain designs with a minimum product (1,4-butanediol) yield (SUCP strain design)

Supplementary preparation steps are necessary for computing strain designs in a genome-scale setup. Here, we again use the example of enforcing substrat-uptake-coupled production of 1,4-butanediol. 1) Add production pathways, i.e., metabolites and reactions, to the model. 2) Ensure that only relevant exchange reactions are kept open in the model. 3) Analyze production capacities. 3) Set up the strain design problem through modules. 4) Launch the strain design computation. 5) Analyze the results.

[19]:
```
import straindesign as sd
import cobra

cobra.Configuration().solver = 'cplex'
iml = cobra.io.load_model('iML1515')

# Create copy of model to which pathway will be added
iml_14bdo = iml.copy()
# If available, set the solver to cplex or gurobi. This will increase the chances
# of sucess enormously
iml_14bdo.solver = 'cplex'

# Add metabolites to model
iml_14bdo.add_metabolites([ cobra.Metabolite('4hb_c'),    # 4-Hydroxybutanoate
                            cobra.Metabolite('4hbcoa_c'),# 4-Hydroxybutyryl-CoA
                            cobra.Metabolite('4hbal_c'), # 4-Hydroxybutanal
                            cobra.Metabolite('14bdo_c'), # Butane-1,4-diol (cytopl.)
                            cobra.Metabolite('14bdo_p'), # Butane-1,4-diol (peripl.)
                            cobra.Metabolite('14bdo_e')  # Butane-1,4-diol (extrac.)
                            ])

# Create reactions
AKGDC      = cobra.Reaction('AKGDC')
SSCOARx    = cobra.Reaction('SSCOARx')
HBD        = cobra.Reaction('4HBD')
HBCT       = cobra.Reaction('4HBCT')
HBDH       = cobra.Reaction('4HBDH')
HBDx       = cobra.Reaction('4HBDx')
```

(continues on next page)

(continued from previous page)

```
BDOtpp     = cobra.Reaction('14BDOtpp')
BDOtex     = cobra.Reaction('14BDOtex')
EX_14bdo_e = cobra.Reaction('EX_14bdo_e')

# Add reactions to model
iml_14bdo.add_reactions([SSCOARx,
                         AKGDC,
                         HBD,
                         HBCT,
                         HBDH,
                         HBDx,
                         BDOtpp,
                         BDOtex,
                         EX_14bdo_e])

# Define reaction equations
SSCOARx.reaction    = '1 h_c + 1 nadph_c + 1 succoa_c -> 1 coa_c + 1 nadp_c + 1 sucsal_c'
AKGDC.reaction      = '1 akg_c + 1 h_c -> 1 co2_c + 1 sucsal_c'
HBD.reaction        = '1 h_c + 1 nadh_c + 1 sucsal_c  -> 1 4hb_c + 1 nad_c'
HBCT.reaction       = '1 4hb_c + 1 accoa_c            -> 1 4hbcoa_c + 1 ac_c'
HBDH.reaction       = '1 4hbcoa_c + 1 h_c + 1 nadh_c  -> 1 4hbal_c + 1 coa_c + 1 nad_c'
HBDx.reaction       = '1 4hbal_c + 1 h_c + 1 nadh_c   -> 1 14bdo_c + 1 nad_c'
BDOtpp.reaction     = '1 14bdo_c                      -> 1 14bdo_p'
BDOtex.reaction     = '1 14bdo_p                      -> 1 14bdo_e'
EX_14bdo_e.reaction = '1 14bdo_e                      ->'

# define GPR rules for knock-inable genes
SSCOARx.gene_reaction_rule  = 'gsscoar'
AKGDC.gene_reaction_rule    = 'gakgdc'
iml_14bdo.genes.gsscoar.name = 'sscoar'
iml_14bdo.genes.gakgdc.name = 'akgdc'

# Verify that pathway is operational
sol = sd.fba(iml_14bdo,obj='EX_14bdo_e',obj_sense='max')
print(f"Maximum possible 1,4-BDO synthesis rate: {sol.objective_value}.")
```

```
Maximum possible 1,4-BDO synthesis rate: 10.489195402311951.
```

The iML1515 model contains metabolite exchange capacities that are usually not observed in experiments such as exchanges of CoA-associated or phosphorylated metabolites. While unrealistic, these exchanges are still represented in the model and stoichiometrically possible, hence the MCS approach would have to target many of them in order to enforce SUCP of 1,4-BDO. Closing unrealistic exchanges ahead of the computation reduces the computation effort and aviods the excessive introduction of interventions.

We block the import of all carbon-containing metabolites apart from D-glucose.

We here opt to block the export of all metabolites apart from CO2, Ethanol, Acetate, Formate, D-Lactate, Succinate, Methanol, O2, H+, H2, H2O2 and tungsten. We furthermore allow a number of exchange reactions with the prefix DM_ that are required for the iML biomass synthesis pseudoreaction.

We replace long and complex GPR rules with shorter ones. The integrated compression routine would compresses GPR rules, which would allow a fast computation. However decompression of computed strain designs can result in countless equivalent strain designs. For instance, the simultanous knockouts of CYTBO3_4pp (cyoA & cyoB & cyoC & cyoD) and FRD2 (frdA & frdB & frdC & frdD) can be achieved by any KO combination of the subunits and hence by permutations (cyoA, frdA) or (cyoA, frdB) or (cyoB, frdA) and so on … This is an effect we try to minimize by

substituting GPR rules.

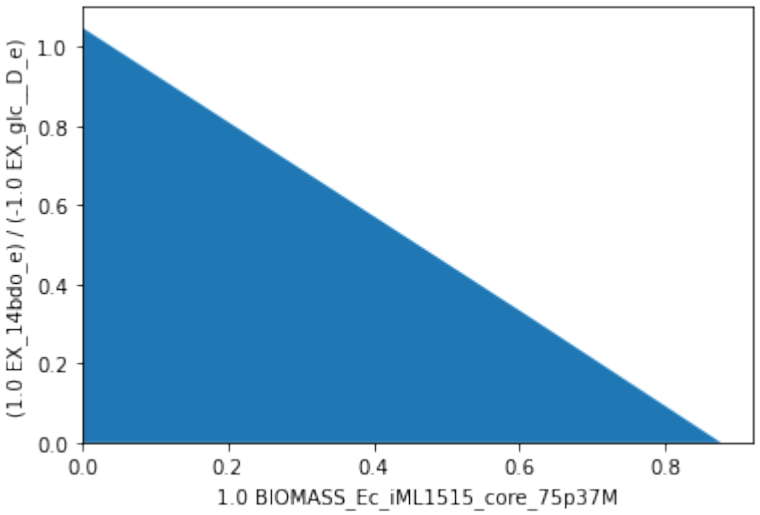
[20]:
```
exchange_reacs = [r for r in iml_14bdo.reactions if all(s<0 for s in r.metabolites.
↪values())]
# shut all exchange fluxes
for r in exchange_reacs:
    r.upper_bound = 0.0

# shut CO2 uptake
iml_14bdo.reactions.EX_co2_e.lower_bound    = 0.0

# keep main fermentation products open
iml_14bdo.reactions.EX_14bdo_e.upper_bound  = 1000.0
iml_14bdo.reactions.EX_ac_e.upper_bound     = 1000.0
iml_14bdo.reactions.EX_co2_e.upper_bound    = 1000.0
iml_14bdo.reactions.EX_etoh_e.upper_bound   = 1000.0
iml_14bdo.reactions.EX_for_e.upper_bound    = 1000.0
iml_14bdo.reactions.EX_h2_e.upper_bound     = 1000.0
iml_14bdo.reactions.EX_h2o2_e.upper_bound   = 1000.0
iml_14bdo.reactions.EX_h2o_e.upper_bound    = 1000.0
iml_14bdo.reactions.EX_h_e.upper_bound      = 1000.0
iml_14bdo.reactions.EX_lac__D_e.upper_bound = 1000.0
iml_14bdo.reactions.EX_meoh_e.upper_bound   = 1000.0
iml_14bdo.reactions.EX_o2_e.upper_bound     = 1000.0
iml_14bdo.reactions.EX_succ_e.upper_bound   = 1000.0
iml_14bdo.reactions.EX_tungs_e.upper_bound  = 1000.0
iml_14bdo.reactions.DM_4crsol_c.upper_bound = 1000.0
iml_14bdo.reactions.DM_5drib_c.upper_bound  = 1000.0
iml_14bdo.reactions.DM_aacald_c.upper_bound = 1000.0
iml_14bdo.reactions.DM_amob_c.upper_bound   = 1000.0
iml_14bdo.reactions.DM_mththf_c.upper_bound = 1000.0
iml_14bdo.reactions.DM_oxam_c.upper_bound   = 1000.0

# substitute GPR rule in ATPS4rpp (all genes occur only in ATPS)
atps_genes = [g for g in iml_14bdo.reactions.ATPS4rpp.genes]
iml_14bdo.reactions.ATPS4rpp.gene_reaction_rule = str(atps_genes[0])
cobra.manipulation.remove_genes(iml_14bdo,atps_genes[1:])
atps_genes[0].name = 'atpX'

# substitute GPR rules in NADH16pp, NADH17pp, NADH18pp (all genes occur only in NADHxxpp␣
↪and GPR rules are identical)
nuo_genes = [g for g in iml_14bdo.reactions.NADH16pp.genes]
iml_14bdo.reactions.NADH16pp.gene_reaction_rule = str(nuo_genes[0])
iml_14bdo.reactions.NADH17pp.gene_reaction_rule = str(nuo_genes[0])
iml_14bdo.reactions.NADH18pp.gene_reaction_rule = str(nuo_genes[0])
cobra.manipulation.remove_genes(iml_14bdo,nuo_genes[1:])
nuo_genes[0].name = 'nuoX'

# substitute GPR rules in FRD2, FRD3 (genes occur only in FRD2 and FRD3 and GPR rules␣
↪are identical)
frd_genes = [g for g in iml_14bdo.reactions.FRD2.genes]
iml_14bdo.reactions.FRD2.gene_reaction_rule = str(frd_genes[0])
iml_14bdo.reactions.FRD3.gene_reaction_rule = str(frd_genes[0])
```

(continues on next page)

(continued from previous page)

```
cobra.manipulation.remove_genes(iml_14bdo,frd_genes[1:])
frd_genes[0].name = 'frdX'

# substitute GPR rule in CYTBO3_4pp
cyo_genes = [g for g in iml_14bdo.reactions.CYTBO3_4pp.genes]
iml_14bdo.reactions.CYTBO3_4pp.gene_reaction_rule = str(cyo_genes[0])
cobra.manipulation.remove_genes(iml_14bdo,cyo_genes[1:])
cyo_genes[0].name = 'cyoX'

# substitute GPR rule in THD2pp
pnt_genes = [g for g in iml_14bdo.reactions.THD2pp.genes]
iml_14bdo.reactions.THD2pp.gene_reaction_rule = str(pnt_genes[0])
cobra.manipulation.remove_genes(iml_14bdo,pnt_genes[1:])
pnt_genes[0].name = 'pntX'

# substitute GPR rule in PDH and AKGDH
ace_genes = [g for g in iml_14bdo.genes if g.name in ['aceE', 'aceF']]
sucAB_genes = [g for g in iml_14bdo.genes if g.name in ['sucA', 'sucB']]
lpd = [g for g in iml_14bdo.genes if g.name == 'lpd'][0]
iml_14bdo.reactions.PDH.gene_reaction_rule = str(ace_genes[0])+' and '+str(lpd)
iml_14bdo.reactions.AKGDH.gene_reaction_rule = str(sucAB_genes[0])+' and '+str(lpd)
cobra.manipulation.remove_genes(iml_14bdo,ace_genes[1:])
cobra.manipulation.remove_genes(iml_14bdo,sucAB_genes[1:])
ace_genes[0].name = 'aceEF'
sucAB_genes[0].name = 'sucAB'

# substitute GPR rule in SUCOAS
sucCD_genes = [g for g in iml_14bdo.reactions.SUCOAS.genes]
iml_14bdo.reactions.SUCOAS.gene_reaction_rule = str(sucCD_genes[0])
cobra.manipulation.remove_genes(iml_14bdo,sucCD_genes[1:])
sucCD_genes[0].name = 'sucCD'

# substitute GPR rule in SUCDi
sdh_genes = [g for g in iml_14bdo.reactions.SUCDi.genes]
iml_14bdo.reactions.SUCDi.gene_reaction_rule = str(sdh_genes[0])
cobra.manipulation.remove_genes(iml_14bdo,sdh_genes[1:])
sdh_genes[0].name = 'sdhX'
```

[21]:
```
sd.plot_flux_space(iml_14bdo, ('BIOMASS_Ec_iML1515_core_75p37M',('EX_14bdo_e','-EX_glc__
↪D_e')));
```



As in the small-scale setup, we set up again the SUPRESS and PROTECT modules to enforce substrate-uptake-coupled production of 1,4-BDO. The genome-scale model contains many more secondary pathways that can be used to disrupt SUCP. We therefore start with a conservative setup of demanding a minimum product yield of 0.1 $\frac{mol_{1,4\text{-}BDO}}{mol_{D\text{-}Glucose}}$. At the same time we ensure that growth is still possible at rates of 0.05 h^{-1} and above.

[22]:
```
min_14bdo_yield = 0.25
min_growth = 0.1
module_suppress = sd.SDModule(iml_14bdo,sd.names.SUPPRESS,constraints=f'EX_14bdo_e +
↪{min_14bdo_yield} EX_glc__D_e <= 0')
module_protect  = sd.SDModule(iml_14bdo,sd.names.PROTECT, constraints=f'BIOMASS_Ec_
↪iML1515_core_75p37M>={min_growth}')
```

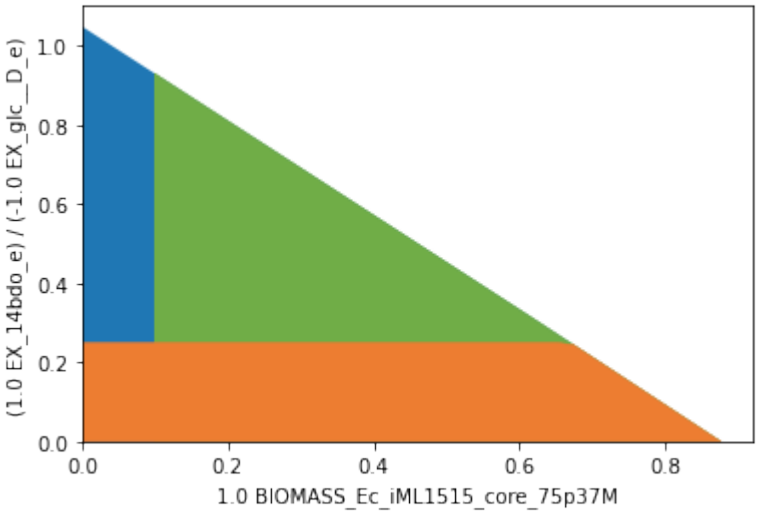
Plotted:

[23]:
```
import matplotlib.pyplot as plt
# Wild-type plot
datapoints, triang, plot1 = sd.plot_flux_space(iml_14bdo,
                                               ('BIOMASS_Ec_iML1515_core_75p37M',('EX_
↪14bdo_e','-EX_glc__D_e')),
                                               show=False);
_,          _,      plot2 = sd.plot_flux_space(iml_14bdo,
                                               (f'BIOMASS_Ec_iML1515_core_75p37M',('EX_
↪14bdo_e','-EX_glc__D_e')),
                                               constraints=f'BIOMASS_Ec_iML1515_core_
↪75p37M >= {min_growth}',
                                               show=False);
plot2.set_facecolor('#70AD47')
plot2.set_edgecolor('#70AD47')
# pGCP design plot
_,          _,      plot3 = sd.plot_flux_space(iml_14bdo,
                                               ('BIOMASS_Ec_iML1515_core_75p37M',('EX_
↪14bdo_e','-EX_glc__D_e')),
                                               # The sign of the glucose exchange␣
↪reaction is flipped since
                                               # reaction is defined in the direction of␣
```

(continues on next page)

(continued from previous page)

```
↪secretion.
                                                    constraints=f'EX_14bdo_e + {min_14bdo_
↪yield} EX_glc__D_e <= 0',
                                                    show=False);
plot3.set_facecolor('#ED7D31')
plot3.set_edgecolor('#ED7D31')
# adjust axes limits and show plot
plot3.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot3.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```



In the genome-case setup using iML1515, it is still unnecessary to exclude the 0-vector explicitly, since it is already excluded by default due to the minimum ATP maintenance demand.

We can now proceed with the strain design computation. Since we normally don't know if solutions to our strain design problems exist, and even more so in genome-scale setups, we will start the computation with the most relaxed settings possible. This means, we compute only one single solution, without a time limit, while omitting the minimality demand in the solutions and allow up to 25 knockouts. We also activate logging to follow the progress of the computation. If this computation is not successful on the first run, we should retry. The success of genome-scale computations often depends on the MILP search tree, whose construction varies with different computation seeds. A repeated computation may therefore conclude in a different (but also valid) strain design solution. Generally, it is possible that a computation completes in a couple of minutes, while the repetitions runs into timeout.

[24]:
```
import logging
logging.basicConfig(level=logging.INFO)

# allow all gene knockouts except for spontanuos
gko_cost = {g.name:1 for g in iml_14bdo.genes}
gko_cost.pop('s0001')
gko_cost.pop('akgdc')
gko_cost.pop('sscoar')
# allow knock-in of akgdc and sscoar
gki_cost = {'akgdc':1, 'sscoar':1}
# possible knockout of O2
ko_cost = {'EX_o2_e': 1}
# Compute strain designs
```

(continues on next page)

(continued from previous page)

```
sols = sd.compute_strain_designs(iml_14bdo,
                                 sd_modules = [module_suppress, module_protect],
                                 max_solutions = 1,
                                 max_cost = 40,
                                 ko_cost = ko_cost,
                                 gko_cost = gko_cost,
                                 gki_cost = gki_cost,
                                 solution_approach = sd.names.ANY)
# Print solutions
print(f"One compressed solution with cost {sols.sd_cost[0]} found and "+\
      f"expanded to {len(sols.reaction_sd)} solutions in the uncompressed netork.")
print(f"Example intervention set: {['+'+s if v>0 else '-'+s for s,v in sols.gene_sd[0].
↪items() if v!=0]}")
```

```
INFO:root:Preparing strain design computation.
INFO:root:  Using cplex for solving LPs during preprocessing.
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:  FVA to identify blocked reactions and irreversibilities.
INFO:root:  FVA(s) to identify essential reactions.
INFO:root:Preprocessing GPR rules (1485 genes, 2268 gpr rules).
INFO:root:  Simplifyied to 1229 genes and 1853 gpr rules.
INFO:root:  Extending metabolic network with gpr associations.
INFO:root:Compressing Network (5252 reactions).
INFO:root:  Removing blocked reactions.
INFO:root:  Translating stoichiometric coefficients to rationals.
INFO:root:  Removing conservation relations.
INFO:root:  Compression 1: Applying compression from EFM-tool module.
INFO:root:  Reduced to 2450 reactions.
INFO:root:  Compression 2: Lumping parallel reactions.
INFO:root:  Reduced to 2341 reactions.
INFO:root:  Compression 3: Applying compression from EFM-tool module.
INFO:root:  Reduced to 2298 reactions.
INFO:root:  Compression 4: Lumping parallel reactions.
INFO:root:  Reduced to 2292 reactions.
INFO:root:  Compression 5: Applying compression from EFM-tool module.
INFO:root:  Reduced to 2288 reactions.
INFO:root:  Compression 6: Lumping parallel reactions.
INFO:root:  Last step could not reduce size further (2288 reactions).
INFO:root:  Network compression completed. (5 compression iterations)
INFO:root:  Translating stoichiometric coefficients back to float.
INFO:root:  FVA(s) in compressed model to identify essential reactions.
INFO:root:Finished preprocessing:
INFO:root:  Model size: 2288 reactions, 968 metabolites
INFO:root:  569 targetable reactions
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:Constructing strain design MILP for solver: cplex.
INFO:root:  Bounding MILP.
INFO:root:Finding (also non-optimal) strain designs ...
INFO:root:Searching in full search space.
INFO:root:Minimizing number of interventions in subspace with 35 possible targets.
INFO:root:Strain design with cost 19.0: {'PGL*pgl': -1, 'GLXCL*gcl': -1,
↪'URIC*ALLTN*XAND*ALLTAMH2*UGCIAMH*allB*allC*allE': -1, 'MALS*glcB*R0_g_b2976_or_g_
```

(continues on next page)

(continued from previous page)

```
↪b4014*aceB*R1_g_b2976_or_g_b4014': -1, 'CITt7pp*citT': -1, 'SUCDi*sdhX': -1,
↪'AKGDC*akgdc': 1, 'adhE': -1, 'adhP*R1_g_b1241_or_g_b1478': -1, 'satP': -1, 'dcuC*R0_g_
↪b0621_or_g_b4123_or_g_b4138': -1, 'dcuB': -1, 'dcuA': -1, 'focA*R0_g_b0904_or_g_
↪b2492*focB*R1_g_b0904_or_g_b2492': -1, 'glcA*R0_g_b2975_or_g_b3603*lldP*R1_g_b2975_or_
↪g_b3603': -1, 'dauA': -1}
INFO:root:Finished solving strain design MILP.
INFO:root:1 solutions to MILP found.
INFO:root:  Decompressing.
INFO:root:  Preparing (reaction-)phenotype prediction of gene intervention strategies.
INFO:root:3 solutions found.
```

```
One compressed solution with cost 19.0 found and expanded to 3 solutions in the␣
↪uncompressed netork.
Example intervention set: ['-adhE', '-satP', '-dcuB', '-dcuA', '-dauA', '-pgl', '-gcl',
↪'-allB', '-citT', '-sdhX', '+akgdc', '-adhP', '-glcB', '-aceB', '-dcuC', '-focA', '-
↪focB', '-glcA', '-lldP']
```

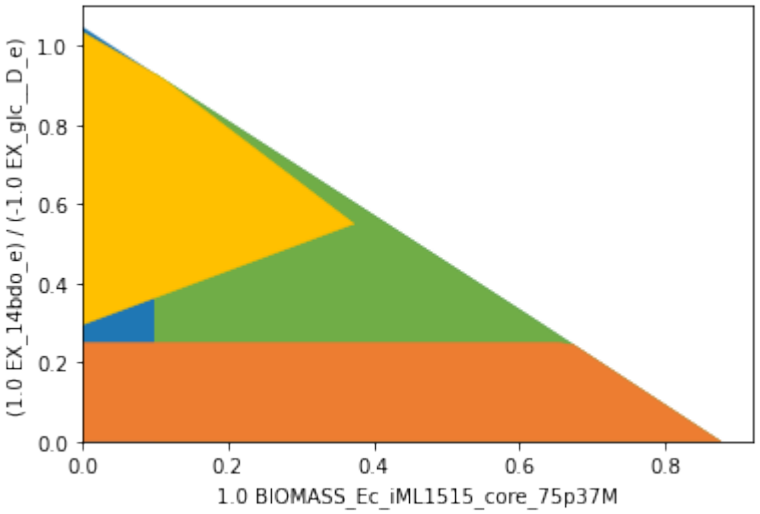
We may plot the computed strain design (yellow) on top of the wild type model (blue), the suppressed fluxes (orange) and the protected fluxes (green). The designed strain is forced to produce 1,4-butanediol but is still able to grow at a relatively high rate.

[25]:
```
import matplotlib.pyplot as plt
# Wild-type plot
datapoints, triang, plot1 = sd.plot_flux_space(iml_14bdo,
                                               ('BIOMASS_Ec_iML1515_core_75p37M',('EX_
↪14bdo_e','-EX_glc__D_e')),
                                               show=False);
_,          _,      plot2 = sd.plot_flux_space(iml_14bdo,
                                               ('BIOMASS_Ec_iML1515_core_75p37M',('EX_
↪14bdo_e','-EX_glc__D_e')),
                                               constraints=f'BIOMASS_Ec_iML1515_core_
↪75p37M>={min_growth}',
                                               show=False);
plot2.set_facecolor('#70AD47')
plot2.set_edgecolor('#70AD47')
# pGCP design plot
_,          _,      plot3 = sd.plot_flux_space(iml_14bdo,
                                               ('BIOMASS_Ec_iML1515_core_75p37M',('EX_
↪14bdo_e','-EX_glc__D_e')),
                                               # The sign of the glucose exchange␣
↪reaction is flipped since
                                               # reaction is defined in the direction of␣
↪secretion.
                                               constraints=f'EX_14bdo_e + {min_14bdo_
↪yield} EX_glc__D_e <= 0',
                                               show=False);
plot3.set_facecolor('#ED7D31')
plot3.set_edgecolor('#ED7D31')
# plotting designed strain
interventions = [[{s:1.0},'=',0.0] for s,v in sols.reaction_sd[0].items() if v < 1]
_,          _,      plot4 = sd.plot_flux_space(iml_14bdo,
                                               ('BIOMASS_Ec_iML1515_core_75p37M',('EX_
↪14bdo_e','-EX_glc__D_e')),
```

(continues on next page)

(continued from previous page)

```
                                                # The sign of the glucose exchange␣
↪reaction is flipped since
                                                # reaction is defined in the direction of␣
↪secretion.
                                                constraints=interventions,
                                                show=False);
plot4.set_facecolor('#FFC000')
plot4.set_edgecolor('#FFC000')
# adjust axes limits and show plot
plot4.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot4.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```



MCS suggests the addition of only one of the two enzymes SSCOAR or AKGDC and suggests the knockout of plenty of other genes. The knockout suggestions proposed by MCS are often overly conservative. In practice, a subset of knockouts is sufficient, in particular, knockouts may be unnecessary when the affected pathway does not carry a large flux in the first place.

4.5.8 Example 7: Suppress flux states in a toy network

The MCS concept can be showcased in a small example network:

Suppose, we want block all metabolic flux through **R4** to avoid the production of metabolite **E**. One can now identify irreducible sets of reaction knockouts to achieve this. Each of these set is called a minimal cut set, short MCS. This figure shows all possible MCS for blocking reaction R4.

Set up the according strain design problem by specifying a *module* that demands the suppression of flux states with $R4 > 0$.

Since strict inequalities are not allowed in mixed integer linear programming (MILP), we need to approximate it by an inclusive inequality and a sufficiently small value $\varepsilon > 0$. Here we pick $\varepsilon = 1$, such that the flux states that we aim to delete are those that fulfill the inequality:

$$R4 >= 1$$

[26]:
```
modules  =  [sd.SDModule(model,sd.names.SUPPRESS,constraints='R4 >= 1')]
# modules +=  [sd.SDModule(model,sd.names.PROTECT, constraints='R3 >= 1')]


sols = sd.compute_strain_designs(model,sd_modules = modules)
for s in sols.reaction_sd:
    print(s)
```

```
INFO:root:Preparing strain design computation.
INFO:root:  Using cplex for solving LPs during preprocessing.
INFO:root:  FVA to identify blocked reactions and irreversibilities.
INFO:root:  FVA(s) to identify essential reactions.
INFO:root:Compressing Network (10 reactions).
INFO:root:  Removing blocked reactions.
INFO:root:  Translating stoichiometric coefficients to rationals.
INFO:root:  Removing conservation relations.
INFO:root:  Compression 1: Applying compression from EFM-tool module.
INFO:root:  Reduced to 8 reactions.
INFO:root:  Compression 2: Lumping parallel reactions.
INFO:root:  Last step could not reduce size further (8 reactions).
INFO:root:  Network compression completed. (1 compression iterations)
INFO:root:  Translating stoichiometric coefficients back to float.
INFO:root:  FVA(s) in compressed model to identify essential reactions.
INFO:root:Finished preprocessing:
INFO:root:  Model size: 8 reactions, 4 metabolites
INFO:root:  8 targetable reactions
INFO:root:Constructing strain design MILP for solver: cplex.
INFO:root:  Bounding MILP.
INFO:root:Finding optimal strain designs ...
INFO:root:Strain design with cost 1.0: {'R4*R7*R10': -1}
INFO:root:Strain design with cost 1.0: {'R1': -1}
INFO:root:Strain design with cost 1.0: {'R3': -1}
INFO:root:Strain design with cost 2.0: {'R6': -1, 'R8': -1}
INFO:root:Strain design with cost 3.0: {'R2': -1, 'R5': -1, 'R6': -1}
INFO:root:Finished solving strain design MILP.
INFO:root:5 solutions to MILP found.
INFO:root:  Decompressing.
INFO:root:7 solutions found.
```

```
{'R1': -1.0}
{'R3': -1.0}
{'R4': -1.0}
{'R7': -1.0}
{'R10': -1.0}
{'R6': -1.0, 'R8': -1.0}
{'R2': -1.0, 'R5': -1.0, 'R6': -1.0}
```

An adequate value for ε can inferred from the model, i.e., the coefficients of the stoichiometric matrix and the flux boundaries. In the shown example values of 1e-7 up to 50 will yield the same results. However, too small values may result in longer runtimes or numerical issues. While large values may not approximate the strict inequality well enough.

4.5.9 Example 8: Suppress and protect flux states in a toy network

It may sometimes be required to protect certain flux states, for instance, to guarantee that the model stays feasible despite the deletion, or to guarantee that microbial growth is still possible despite the introduced where **R4** must be deleted and additionally demand that **R3** must still be able to carry flux.

[27]:
```
import straindesign as sd
import cobra

model = cobra.io.read_sbml_model('../../../tests/model_small_example.xml')

modules =   [sd.SDModule(model,sd.names.SUPPRESS,constraints='R4 >= 1')]
modules +=  [sd.SDModule(model,sd.names.PROTECT, constraints='R3 >= 1')]

sols = sd.compute_strain_designs(model,sd_modules = modules)
for s in sols.reaction_sd:
    print(s)
```

```
INFO:root:Preparing strain design computation.
INFO:root:  Using cplex for solving LPs during preprocessing.
INFO:root:  FVA to identify blocked reactions and irreversibilities.
INFO:root:  FVA(s) to identify essential reactions.
INFO:root:Compressing Network (10 reactions).
INFO:root:  Removing blocked reactions.
INFO:root:  Translating stoichiometric coefficients to rationals.
INFO:root:  Removing conservation relations.
INFO:root:  Compression 1: Applying compression from EFM-tool module.
INFO:root:  Reduced to 8 reactions.
INFO:root:  Compression 2: Lumping parallel reactions.
INFO:root:  Last step could not reduce size further (8 reactions).
INFO:root:  Network compression completed. (1 compression iterations)
INFO:root:  Translating stoichiometric coefficients back to float.
INFO:root:  FVA(s) in compressed model to identify essential reactions.
INFO:root:Finished preprocessing:
INFO:root:  Model size: 8 reactions, 4 metabolites
INFO:root:  7 targetable reactions
INFO:root:Constructing strain design MILP for solver: cplex.
INFO:root:  Bounding MILP.
INFO:root:Finding optimal strain designs ...
INFO:root:Strain design with cost 1.0: {'R4*R7*R10': -1}
INFO:root:Strain design with cost 1.0: {'R1': -1}
INFO:root:Strain design with cost 2.0: {'R6': -1, 'R8': -1}
INFO:root:Finished solving strain design MILP.
INFO:root:3 solutions to MILP found.
INFO:root:  Decompressing.
INFO:root:5 solutions found.
```

```
{'R1': -1.0}
{'R4': -1.0}
{'R7': -1.0}
{'R10': -1.0}
{'R6': -1.0, 'R8': -1.0}
```

As can be seen, the computation returns the 5 out of 7 MCS that block R4 but not R3. The set of theses *constrained* MCS (cMCS) is a subset of the former MCS solution pool.

4.5.10 Theoretical background

In the simplest case, the Minimal Cut Set approach seeks to identify the smallest set of knockouts that render undesired flux states, such as flux states with poor productivity or yield, unattainable. In reference to the space of steady-state flux vectors: one introduces *knockouts that exclude a certain flux-subspace from the space of feasible flux states*.

(Mixed integer) linear programming operates on *feasible* problem spaces and, *per se*, does not allow sub-problems to be infeasible. To express the infeasibility of a certain sub-problem, one needs to formulate an equivalent but "mirrored" problem whose feasibility under any conditions coincides with the infeasibility of the original problem. Farkas' Lemma is a theorem that will serve for exactly that purpose. It states that, to a system of linear inequalities, a second system of linear inequalities can be defined such that *one and only one* of the two problems is feasible. We call the original system the primary and secondary system the Farkas-dual. The feasibility of one problem is a *certificate for the infeasibility* of the other one.

Exactly one of the following is true:

$$\mathbf{A}\,\mathbf{x} \leq \mathbf{b} \qquad \text{or} \qquad \mathbf{A}^\intercal\mathbf{y} = \mathbf{0}, \quad \mathbf{y} \geq \mathbf{0}, \quad \mathbf{y}^\intercal\mathbf{b} < 0.$$

Farkas-duality is closely related to Lagrange-duality in Linear Progamming and variables in one problem correspond to constraints in the other. When we introduce interventions (i.e., we force certain flux variables to take the value 0), we need to adapt the Farkas-dual that we use in the MCS-MILP accordingly. When a variable is more constrained in one problem (or fixed to zero), the other problem relaxes in the corresponding constraint (up to the complete lifting of the constraint).

In the following we show how Farkas' Lemma can be used to construct the MCS-MILP. We start with the original MCS-system simplified as $\mathbf{A_T} \cdot \mathbf{x} = \mathbf{0} \leq \mathbf{b_T}$ that is rendered infeasible through the introduction of knockouts as additional constraints.

$$\begin{bmatrix} \mathbf{A_T} \\ \mathbf{I_{KO}} \end{bmatrix} \mathbf{x} \begin{matrix} \leq \\ = \end{matrix} \begin{bmatrix} \mathbf{b_T} \\ \mathbf{0} \end{bmatrix}.$$

The introduced permanent knockouts will later be controlled through binary decision variables. It must be noted that the constraints only render the primal system infeasible when they *contradict* the primal system. As was mentioned above, it is therefore not possible to use undesired systems that have the zero vector $\mathbf{x} = \mathbf{0}$ as a feasible solution because then even the knockout of all reactions would not make the primal system infeasible.

Using Farkas-dualization, the dual system is given by:

$$\begin{bmatrix} \mathbf{A_T^\intercal} & \mathbf{I_{KO}} \\ \mathbf{b_T^\intercal} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{y} \\ \mathbf{v} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ -1 \end{bmatrix}$$
$$\mathbf{y} \geq \mathbf{0},$$

which is, per definition, feasible. The single inequality in the last row corresponds to $\mathbf{b}^\intercal\mathbf{y} < 0$ of the Farkas-dual system. The latter needs to be replaced with $\mathbf{b_T^\intercal y} \leq -1$ because the system will later be used in a MILP that cannot handle strict inequalities. The replacement is allowed because any found solution of the Farkas-dual can be scaled to also fulfill $\mathbf{b_T^\intercal y} \leq -1$ without affecting the support of the solution (which makes up the MCS).

As can be seen, the variable knockouts in the primal ($\mathbf{I_{KO}}\,\mathbf{x} = \mathbf{0}$) translate to the variables $\mathbf{v}$ in the Farkas-dual system that mime the knockout of all dual constraints by allowing for arbitrary large slack. This system can now be used to identify MCS, since a minimal subset of constraint relaxations (indicated by $v_i \neq 0$) that solves the Farkas-dual system corresponds directly to a minimal subset of primal knockout-constraints within $\mathbf{I_{KO}}\,\mathbf{x} = \mathbf{0}$ that keeps the primal system infeasible. Hence, every solution of the problem with a support-minimal vector $\mathbf{v}$ represents one MCS.

A constraint can be switched on or off by controlling its slack variable v_i by a corresponding binary variable z_i either via indicator constraints,

$$z_i = 0 \rightarrow v_i = 0,$$

or with the big-M method (with M being a sufficiently large number)

$$-M \cdot z_i \leq v_i \leq M \cdot z_i.$$

In this case, there is a 1:1 association of metabolic knockouts, indicated by z_i and slack variables v_i. With the binary variables z_i at hand, we may now finally pose a MILP problem with an objective function that minimizes the number of interventions to block the target system (we use here the version with indicator constraints):

$$\begin{aligned} \text{minimize} \quad & \sum z_i \\ \text{subject to} \quad & \begin{bmatrix} \mathbf{A_T^\intercal} & \mathbf{I_{KO}} \\ \mathbf{b_T^\intercal} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{y} \\ \mathbf{v} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ -1 \end{bmatrix} \\ & \forall i : z_i = 0 \rightarrow v_i = 0 \\ & \mathbf{y} \geq \mathbf{0}, \quad z_i \in \{0,1\}. \end{aligned}$$

This MILP finds the smallest irreducible set of interventions (support-minimal in $\mathbf{v}$) that blocks the target system, hence an MCS with the smallest possible cardinality.

We note that an even more concise formulation can be constructed by omitting the slack-variables v_i and linking the removal of constraints directly to z_i:

$$\begin{aligned} \text{minimize} \quad & \sum z_i \\ \text{subject to} \quad & \begin{bmatrix} \mathbf{b_T^\intercal} & \mathbf{0} \\ \mathbf{0} & \mathbf{A_D} \end{bmatrix} \begin{bmatrix} \mathbf{y} \\ \mathbf{x} \end{bmatrix} \leq \begin{bmatrix} -1 \\ \mathbf{b_D} \end{bmatrix} \\ & \forall i : z_i = 0 \rightarrow \mathbf{A_{T,}}_i^\intercal \mathbf{y} = 0 \\ & \forall i : z_i = 1 \rightarrow x_i = 0 \\ & \mathbf{y} \geq \mathbf{0}, \quad z_i \in \{0,1\}. \end{aligned}$$

The MCS S (containing the indices of the knocked-out reactions), computed by the MILP, is given by $S = \{i | z_i = 1\}$. Multiple MCS solutions (with increasing cardinality) can be found by excluding previously found solutions and their supersets through integer cut constraints and solving the MILP repeatedly.

4.6 Multi-level strain optimization approaches

Several strain design approaches use MILPs with nested optimization to enforce growth-coupled production. As the first nested optimization algorithm, OptKnock [2] aimed to resolve the conflict between the microbial objective of fast growth with the engineering goal of fast production. It therefore constructs a max-max problem for the maximization of product synthesis under the assumption that the cell itself, will maximize its growth rate. One problem is that this formulation leads to overly optimistic strain designs, since it assumes that a cell would maximize production when attaining its maximal growth rate. In the worst case, however (potentially growth coupled prodcution), a cell might turn off production completely.

Successors of OptKnock, such as RobustKnock [3] and OptCouple [4] have overcome this problem since they guarantee production at maximum growth (weakly growth-coupled production). In the following sections we will explain how one can compute strain designs with OptKnock, RobustKnock and OptCouple, and how these methods can be combined with the minimial cut set approach.

[1]:
```
import straindesign as sd
import cobra

ecc = cobra.io.load_model('e_coli_core')
```

```
Set parameter Username
Academic license - for non-commercial use only - expires 2023-07-20
```

For the following examples we again look at 1,4-BDO production. Hence, we first need to introduce the 1,4-BDO pathway again into the e_coli_core model and ensure that it is operational.

[2]:
```
# Create copy of model to which pathway will be added
ecc_14bdo = ecc.copy()

# Add metabolites to model
ecc_14bdo.add_metabolites([ cobra.Metabolite('sucsal_c'),# Succinic semialdehyde
                            cobra.Metabolite('4hb_c'),   # 4-Hydroxybutanoate
                            cobra.Metabolite('4hbcoa_c'),# 4-Hydroxybutyryl-CoA
                            cobra.Metabolite('4hbal_c'), # 4-Hydroxybutanal
                            cobra.Metabolite('14bdo_c'), # Butane-1,4-diol (cytopl.)
                            cobra.Metabolite('14bdo_p'), # Butane-1,4-diol (peripl.)
                            cobra.Metabolite('14bdo_e')  # Butane-1,4-diol (extrac.)
                            ])

# Create reactions
SSCOARx    = cobra.Reaction('SSCOARx')
AKGDC      = cobra.Reaction('AKGDC')
HBD        = cobra.Reaction('4HBD')
HBCT       = cobra.Reaction('4HBCT')
HBDH       = cobra.Reaction('4HBDH')
HBDx       = cobra.Reaction('4HBDx')
BDOtpp     = cobra.Reaction('14BDOtpp')
BDOtex     = cobra.Reaction('14BDOtex')
EX_14bdo_e = cobra.Reaction('EX_14bdo_e')

# Add reactions to model
ecc_14bdo.add_reactions([SSCOARx,
                         AKGDC,
                         HBD,
                         HBCT,
                         HBDH,
                         HBDx,
                         BDOtpp,
                         BDOtex,
                         EX_14bdo_e])

# Define reaction equations
SSCOARx.reaction    = '1 h_c + 1 nadph_c + 1 succoa_c -> 1 coa_c + 1 nadp_c + 1 sucsal_c'
AKGDC.reaction      = '1 akg_c + 1 h_c -> 1 co2_c + 1 sucsal_c'
HBD.reaction        = '1 h_c + 1 nadh_c + 1 sucsal_c  -> 1 4hb_c + 1 nad_c'
HBCT.reaction       = '1 4hb_c + 1 accoa_c            -> 1 4hbcoa_c + 1 ac_c'
HBDH.reaction       = '1 4hbcoa_c + 1 h_c + 1 nadh_c  -> 1 4hbal_c + 1 coa_c + 1 nad_c'
HBDx.reaction       = '1 4hbal_c + 1 h_c + 1 nadh_c   -> 1 14bdo_c + 1 nad_c'
BDOtpp.reaction     = '1 14bdo_c                      -> 1 14bdo_p'
BDOtex.reaction     = '1 14bdo_p                      -> 1 14bdo_e'
EX_14bdo_e.reaction = '1 14bdo_e                      ->'

# Verify that pathway is operational
sol = sd.fba(ecc_14bdo,obj='EX_14bdo_e',obj_sense='max')
print(f"Maximum possible 1,4-BDO synthesis rate: {sol.objective_value}.")
```

```
Read LP format model from file C:\Users\Philipp\AppData\Local\Temp\tmp_htaelki.lp
Reading time = 0.00 seconds
: 72 rows, 190 columns, 720 nonzeros
```

(continues on next page)

(continued from previous page)

```
Maximum possible 1,4-BDO synthesis rate: 10.252923076923079.
```

4.6.1 OptKnock

4.6.1.1 Example 9: OptKnock strain design

Optknock [2] is based on a bi-level optimization problem:

$$
\begin{array}{lll}
\text{maximize} & v_{production} & \\
 & \text{maximize} & v_{biomass} \\
 & \text{subject to} & \mathbf{S}\,\mathbf{v} = \mathbf{0} \\
\text{subject to} & & v_{BM} \geq v_{BM}^{min} \\
 & & (1 - z_i) \cdot lb_i \leq v_i \leq (1 - z_i) \cdot ub_i,\ \forall i \in \{1, \ldots, n\} \\
 & & \sum z_i \leq \text{maxKOs} \\
 & & z_i \in \{0, 1\}
\end{array}
$$

The nested optimization is translated into a single-layer problem and can then be solved as a mixed-integer linear problem (MILP).

Translating the nested optimization into a single level optimization yields:

$$
\begin{gathered}
\text{maximize } v_{production} \\
\text{subject to} \\
\begin{bmatrix} \mathbf{G} & \mathbf{0} & \mathbf{0} \\ \mathbf{D} & \mathbf{0} & \mathbf{0} \\ -\mathbf{c}^\intercal & \mathbf{g}^\intercal & \mathbf{0} \\ \mathbf{0} & \mathbf{G}^\intercal & \mathbf{I_{KO}} \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ \mathbf{y} \\ \mathbf{s} \end{bmatrix} \begin{matrix} \leq \\ \leq \\ \leq \\ = \end{matrix} \begin{bmatrix} \mathbf{g} \\ \mathbf{d} \\ 0 \\ \mathbf{c} \end{bmatrix} \\
\forall i : z_i = 1 \rightarrow v_i = 0 \\
\forall i : z_i = 0 \rightarrow s_i = 0 \\
\sum z_i \leq MaxNoKO \\
\mathbf{y} \geq \mathbf{0}, \quad z \in \{0, 1\}
\end{gathered}
$$

The steps of translating a bi-level problem to a MILP are automated in StrainDesign. In the following, we will compute OptKnock strain designs using `e_coli_core`.

An advantage of OptKnock strain designs is that they often allow for a higher maximal growth rate. However, the predicted strain designs are often overly optimistic since they imply that the cells are able to actually reach their stoichiometrically highest possible growth rate and then also tune teir metabolism towards production. OptKnock strain designs might be preferred if the production pathway is known to be well expressed and active, even without introducing knockouts. OptKnock may then assure that laboratory evolution selects against production. We set up the strain design module for computing OptKnock strain desings. The OptKnock module consists of an outer and inner objective and additional constraints. The additional constraints are used to enforce the minimal desired growth rate.

[3]:
```
module_optknock = sd.SDModule(ecc_14bdo,sd.names.OPTKNOCK,
                              inner_objective='BIOMASS_Ecoli_core_w_GAM',
                              outer_objective='EX_14bdo_e',
                              constraints='BIOMASS_Ecoli_core_w_GAM >= 0.5')
```

We then call the strain design function with the constructed module. Since OptKnock uses an outer objective, we should use the solution approach 'BEST' to enforce optimality. By default, inner and outer objective functions are defined in the sense of *maximization* for minimization, the `inner_opt_sense` and `outer_opt_sense` can be set to `'maximize'` or one can simply use negative coefficients in the objective functions.

[4]:
```
import logging
logging.basicConfig(level=logging.INFO)
## Compute strain designs
# allow all gene knockouts except for spontanuos
gko_cost = {g.name:1 for g in ecc_14bdo.genes}
gko_cost.pop('s0001')
# possible knockout of O2
ko_cost = {'EX_o2_e': 1}
# addition candidates
ki_cost = {'AKGDC':1, 'SSCOARx':1} # AKGDC was added in example 1.c)

sols = sd.compute_strain_designs(ecc_14bdo,
                                 sd_modules = module_optknock,
                                 max_solutions = 1,
                                 max_cost = 30,
                                 gko_cost = gko_cost,
                                 ko_cost = ko_cost,
                                 ki_cost = ki_cost,
                                 solution_approach = sd.names.BEST)
# Print solution
print(f"One compressed solution with cost {sols.sd_cost[0]} found and "+\
      f"expanded to {len(sols.gene_sd)} solutions in the uncompressed netork.")
print(f"Example intervention set: {['+'+s if v>0 else '-'+s for s,v in sols.gene_sd[0].
↪items() if v!=0]}")
```

```
INFO:root:Preparing strain design computation.
INFO:root:  Using gurobi for solving LPs during preprocessing.
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:  FVA to identify blocked reactions and irreversibilities.
INFO:root:  FVA(s) to identify essential reactions.
INFO:root:Preprocessing GPR rules (137 genes, 69 gpr rules).
INFO:root:  Simplifyied to 86 genes and 49 gpr rules.
INFO:root:  Extending metabolic network with gpr associations.
INFO:root:Compressing Network (267 reactions).
INFO:root:  Removing blocked reactions.
INFO:root:  Translating stoichiometric coefficients to rationals.
INFO:root:  Removing conservation relations.
INFO:root:  Compression 1: Applying compression from EFM-tool module.
INFO:root:  Reduced to 149 reactions.
INFO:root:  Compression 2: Lumping parallel reactions.
INFO:root:  Reduced to 128 reactions.
INFO:root:  Compression 3: Applying compression from EFM-tool module.
INFO:root:  Reduced to 122 reactions.
INFO:root:  Compression 4: Lumping parallel reactions.
INFO:root:  Reduced to 121 reactions.
INFO:root:  Compression 5: Applying compression from EFM-tool module.
INFO:root:  Reduced to 120 reactions.
INFO:root:  Compression 6: Lumping parallel reactions.
INFO:root:  Last step could not reduce size further (120 reactions).
INFO:root:  Network compression completed. (5 compression iterations)
INFO:root:  Translating stoichiometric coefficients back to float.
INFO:root:  FVA(s) in compressed model to identify essential reactions.
INFO:root:Finished preprocessing:
```

(continues on next page)

(continued from previous page)

```
INFO:root:  Model size: 120 reactions, 71 metabolites
INFO:root:  44 targetable reactions
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:Constructing strain design MILP for solver: gurobi.
INFO:root:  Bounding MILP.
INFO:root:Finding optimal strain designs ...
INFO:root:Found solution with objective value 2.6195399380012496
INFO:root:Minimizing number of interventions in subspace with 16 possible targets.
INFO:root:Strain design with cost 16.0: {'PFL*EX_for_e*pflA*pflB*R_g_b0902_and_g_
↪b0903*R0_g_b0902_and_g_b0903_or_g_b0902_and_g_b3114_or_g_b3951_and_g_b3952*tdcE*R_g_
↪b0902_and_g_b3114*R1_g_b0902_and_g_b0903_or_g_b0902_and_g_b3114_or_g_b3951_and_g_
↪b3952...': -1, 'PYK*pykF*R0_g_b1676_or_g_b1854*pykA*R1_g_b1676_or_g_b1854': -1,
↪'FRD7*frdD*frdC*frdB*frdA*R_g_b4151_and_g_b4152_and_g_b4153_and_g_b4154': -1, 'ME1*maeA
↪': -1, 'ME2*maeB': -1, 'AKGDC': 1, 'pgi': -1, 'kgtP': -1, 'purT*R0_g_b1849_or_g_b2296_
↪or_g_b3115*ackA*R1_g_b1849_or_g_b2296_or_g_b3115*tdcD*R2_g_b1849_or_g_b2296_or_g_b3115
↪': -1, 'sucC*sucD*R_g_b0728_and_g_b0729': -1, 'gdhA': -1, 'mdh': -1}
INFO:root:Finished solving strain design MILP.
INFO:root:1 solutions to MILP found.
INFO:root:  Decompressing.
INFO:root:  Preparing (reaction-)phenotype prediction of gene intervention strategies.
INFO:root:32 solutions found.
```

```
One compressed solution with cost 16.0 found and expanded to 32 solutions in the␣
↪uncompressed netork.
Example intervention set: ['+AKGDC', '-pgi', '-kgtP', '-gdhA', '-mdh', '-pflA', '-frdD',
↪'-maeA', '-maeB', '-pflD', '-purT', '-ackA', '-tdcD', '-pykF', '-pykA', '-sucC']
```

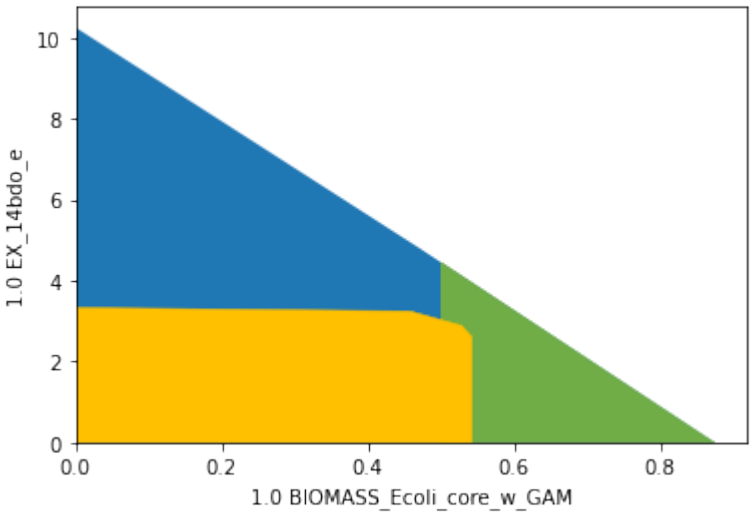
We can now compare the computed strain design to the wild type and the minimal enforced growth rate. As OptKnock is rate instead of yield based, we plot our strain designs in the production envelope.

[5]:
```
import matplotlib.pyplot as plt
# Wild-type plot
datapoints, triang, plot1 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'),
                                               show=False);
# Plot minimal enforced growth rate
_,          _,      plot2 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'),
                                               constraints='BIOMASS_Ecoli_core_w_GAM>=0.5
↪',
                                               show=False);
plot2.set_facecolor('#70AD47')
plot2.set_edgecolor('#70AD47')

# OptKnock design plot
interventions = [[{s:1.0},'=',0.0] for s,v in sols.reaction_sd[0].items() if v < 1]
_,          _,      plot3 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'),
                                               # The sign of the glucose exchange␣
↪reaction is flipped since
                                               # reaction is defined in the direction of␣
↪secretion.
                                               constraints=interventions,
```

(continues on next page)

(continued from previous page)

```
                                                show=False);
plot3.set_facecolor('#FFC000')
plot3.set_edgecolor('#FFC000')
# adjust axes limits and show plot
plot3.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot3.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```



The computed strain design generates a pGCP strain design with a relatively high maximal growth rate, however, at the cost of relatively many knockouts. The plot shows that OptKnock has exploited all means to improve maximal production at maximum growth. The maximum growth rate is only slightly above the demanded minimum.

4.6.1.2 Example 10: OptKnock strain design with a tilted objective function

The last example showed that predicted strain designs not always *guarantee* product synthesis at maximum growth. A tilted objective function can be used to simulate that the cell not only maximizes growth but also minimizes product synthesis and thus opposes the engineering goal. Factoring in this secondary goal by the cell with a sufficiently small objective coefficient will lead to more agressive stain designs that counteract the cells possible minimization of production.

[6]:
```
import matplotlib.pyplot as plt
# Wild-type plot
datapoints, triang, plot1 = sd.plot_flux_space(ecc_14bdo,
                                                ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'),
                                                show=False);
# Plot minimal enforced growth rate
_,          _,      plot2 = sd.plot_flux_space(ecc_14bdo,
                                                ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'),
                                                constraints='BIOMASS_Ecoli_core_w_GAM>=0.5
↪',
                                                show=False);
plot2.set_facecolor('#70AD47')
plot2.set_edgecolor('#70AD47')
```

(continues on next page)

(continued from previous page)

```
module_tilted_optknock = sd.SDModule(ecc_14bdo,sd.names.OPTKNOCK,
                              inner_objective='BIOMASS_Ecoli_core_w_GAM - 0.001 EX_14bdo_
↪e',
                              outer_objective='EX_14bdo_e',
                              constraints='BIOMASS_Ecoli_core_w_GAM >= 0.5')

# allow all gene knockouts except for spontanuos
gko_cost = {g.name:1 for g in ecc_14bdo.genes}
gko_cost.pop('s0001')
# possible knockout of O2
ko_cost = {'EX_o2_e': 1}
# addition candidates
ki_cost = {'AKGDC':1, 'SSCOARx':1} # AKGDC was added in example 1.c)

sols = sd.compute_strain_designs(ecc_14bdo,
                                 sd_modules = module_tilted_optknock,
                                 time_limit = 300,
                                 max_solutions = 1,
                                 max_cost = 30,
                                 gko_cost = gko_cost,
                                 ko_cost = ko_cost,
                                 ki_cost = ki_cost,
                                 solution_approach = sd.names.BEST)

# Print solutions
print(f"One compressed solution with cost {sols.sd_cost[0]} found and "+\
      f"expanded to {len(sols.gene_sd)} solutions in the uncompressed netork.")
print(f"Example intervention set: {['+'+s if v>0 else '-'+s for s,v in sols.gene_sd[0].
↪items() if v!=0]}")

# OptKnock design plot
interventions = [[{s:1.0},'=',0.0] for s,v in sols.reaction_sd[0].items() if v < 1]
_,          _,      plot3 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'),
                                               # The sign of the glucose exchange␣
↪reaction is flipped since
                                               # reaction is defined in the direction of␣
↪secretion.
                                               constraints=interventions,
                                               show=False);
plot3.set_facecolor('#FFC000')
plot3.set_edgecolor('#FFC000')
# adjust axes limits and show plot
plot3.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot3.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```

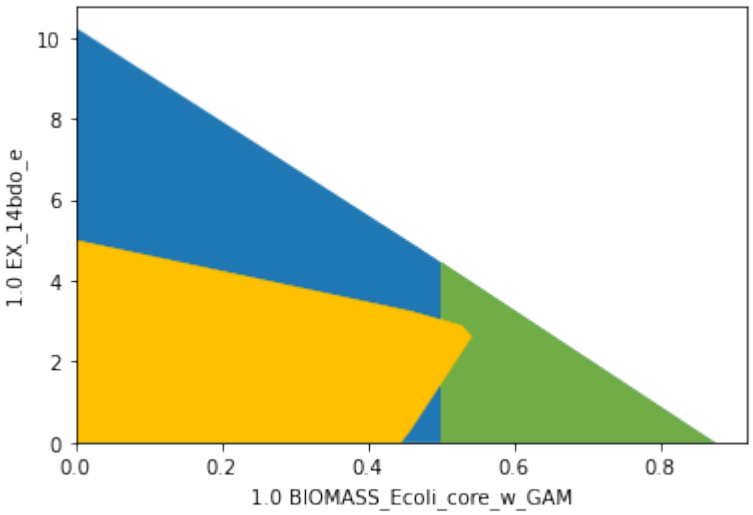
```
INFO:root:Preparing strain design computation.
INFO:root:  Using gurobi for solving LPs during preprocessing.
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:  FVA to identify blocked reactions and irreversibilities.
INFO:root:  FVA(s) to identify essential reactions.
```

(continues on next page)

(continued from previous page)

```
INFO:root:Preprocessing GPR rules (137 genes, 69 gpr rules).
INFO:root:  Simplifyied to 86 genes and 49 gpr rules.
INFO:root:  Extending metabolic network with gpr associations.
INFO:root:Compressing Network (267 reactions).
INFO:root:  Removing blocked reactions.
INFO:root:  Translating stoichiometric coefficients to rationals.
INFO:root:  Removing conservation relations.
INFO:root:  Compression 1: Applying compression from EFM-tool module.
INFO:root:  Reduced to 149 reactions.
INFO:root:  Compression 2: Lumping parallel reactions.
INFO:root:  Reduced to 128 reactions.
INFO:root:  Compression 3: Applying compression from EFM-tool module.
INFO:root:  Reduced to 122 reactions.
INFO:root:  Compression 4: Lumping parallel reactions.
INFO:root:  Reduced to 121 reactions.
INFO:root:  Compression 5: Applying compression from EFM-tool module.
INFO:root:  Reduced to 120 reactions.
INFO:root:  Compression 6: Lumping parallel reactions.
INFO:root:  Last step could not reduce size further (120 reactions).
INFO:root:  Network compression completed. (5 compression iterations)
INFO:root:  Translating stoichiometric coefficients back to float.
INFO:root:  FVA(s) in compressed model to identify essential reactions.
INFO:root:Finished preprocessing:
INFO:root:  Model size: 120 reactions, 71 metabolites
INFO:root:  44 targetable reactions
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:Constructing strain design MILP for solver: gurobi.
INFO:root:  Bounding MILP.
INFO:root:Finding optimal strain designs ...
INFO:root:Found solution with objective value 2.6195399376393156
INFO:root:Minimizing number of interventions in subspace with 18 possible targets.
INFO:root:Strain design with cost 13.0: {'PYK*pykF*R0_g_b1676_or_g_b1854*pykA*R1_g_b1676_
↪or_g_b1854': -1, 'ICL*MALS*aceA*glcB*R0_g_b2976_or_g_b4014*aceB*R1_g_b2976_or_g_b4014':
↪ -1, 'ME1*maeA': -1, 'ME2*maeB': -1, 'AKGDC': 1, 'pgi': -1, 'kgtP': -1, 'pta*R0_g_
↪b2297_or_g_b2458*eutD*R1_g_b2297_or_g_b2458': -1, 'sucC*sucD*R_g_b0728_and_g_b0729': -
↪1, 'gdhA': -1, 'gltP': -1}
INFO:root:Finished solving strain design MILP.
INFO:root:1 solutions to MILP found.
INFO:root:  Decompressing.
INFO:root:  Preparing (reaction-)phenotype prediction of gene intervention strategies.
INFO:root:4 solutions found.
```

```
One compressed solution with cost 13.0 found and expanded to 4 solutions in the␣
↪uncompressed netork.
Example intervention set: ['+AKGDC', '-pgi', '-kgtP', '-gdhA', '-gltP', '-aceA', '-maeA',
↪ '-maeB', '-pta', '-eutD', '-pykF', '-pykA', '-sucC']
```



As a result, we obtain an at least weakly growth coupled strain design.

4.6.1.3 Example 11: Genome-scale OptKnock strain design

Other than RobustKnock and OptCouple, OptKnock's computational effort is moderate so that the approach may be used in genome scale metabolic models. The first steps of model manipulation are identical to the ones in example 6.

[7]:
```
import straindesign as sd
import cobra

cobra.Configuration().solver = 'cplex'
ijo = cobra.io.load_model('iJO1366')

# Create copy of model to which pathway will be added
ijo_14bdo = ijo.copy()
# If available, set the solver to cplex or gurobi. This will increase the chances
# of sucess enormously
ijo_14bdo.solver = 'cplex'

# Add metabolites to model
ijo_14bdo.add_metabolites([ cobra.Metabolite('4hb_c'),    # 4-Hydroxybutanoate
                            cobra.Metabolite('4hbcoa_c'),# 4-Hydroxybutyryl-CoA
                            cobra.Metabolite('4hbal_c'), # 4-Hydroxybutanal
                            cobra.Metabolite('14bdo_c'), # Butane-1,4-diol (cytopl.)
                            cobra.Metabolite('14bdo_p'), # Butane-1,4-diol (peripl.)
                            cobra.Metabolite('14bdo_e')  # Butane-1,4-diol (extrac.)
                            ])

# Create reactions
AKGDC      = cobra.Reaction('AKGDC')
SSCOARx    = cobra.Reaction('SSCOARx')
HBD        = cobra.Reaction('4HBD')
HBCT       = cobra.Reaction('4HBCT')
HBDH       = cobra.Reaction('4HBDH')
HBDx       = cobra.Reaction('4HBDx')
```

(continues on next page)

(continued from previous page)

```
BDOtpp     = cobra.Reaction('14BDOtpp')
BDOtex     = cobra.Reaction('14BDOtex')
EX_14bdo_e = cobra.Reaction('EX_14bdo_e')

# Add reactions to model
ijo_14bdo.add_reactions([SSCOARx,
                         AKGDC,
                         HBD,
                         HBCT,
                         HBDH,
                         HBDx,
                         BDOtpp,
                         BDOtex,
                         EX_14bdo_e])

# Define reaction equations
SSCOARx.reaction    = '1 h_c + 1 nadph_c + 1 succoa_c -> 1 coa_c + 1 nadp_c + 1 sucsal_c'
AKGDC.reaction      = '1 akg_c + 1 h_c -> 1 co2_c + 1 sucsal_c'
HBD.reaction        = '1 h_c + 1 nadh_c + 1 sucsal_c  -> 1 4hb_c + 1 nad_c'
HBCT.reaction       = '1 4hb_c + 1 accoa_c            -> 1 4hbcoa_c + 1 ac_c'
HBDH.reaction       = '1 4hbcoa_c + 1 h_c + 1 nadh_c  -> 1 4hbal_c + 1 coa_c + 1 nad_c'
HBDx.reaction       = '1 4hbal_c + 1 h_c + 1 nadh_c   -> 1 14bdo_c + 1 nad_c'
BDOtpp.reaction     = '1 14bdo_c                      -> 1 14bdo_p'
BDOtex.reaction     = '1 14bdo_p                      -> 1 14bdo_e'
EX_14bdo_e.reaction = '1 14bdo_e                      ->'

# define gene rules for knock-inable genes
SSCOARx.gene_reaction_rule  = 'gsscoar'
AKGDC.gene_reaction_rule    = 'gakgdc'

exchange_reacs = [r for r in ijo_14bdo.reactions if all(s<0 for s in r.metabolites.
↪values())]
# shut all exchange fluxes
for r in exchange_reacs:
    r.upper_bound = 0.0

# shut CO2 uptake
ijo_14bdo.reactions.EX_co2_e.lower_bound    = 0.0

# keep main fermentation products open
ijo_14bdo.reactions.EX_14bdo_e.upper_bound  = 1000.0
ijo_14bdo.reactions.EX_ac_e.upper_bound     = 1000.0
ijo_14bdo.reactions.EX_co2_e.upper_bound    = 1000.0
ijo_14bdo.reactions.EX_etoh_e.upper_bound   = 1000.0
ijo_14bdo.reactions.EX_for_e.upper_bound    = 1000.0
ijo_14bdo.reactions.EX_h2_e.upper_bound     = 1000.0
ijo_14bdo.reactions.EX_h2o2_e.upper_bound   = 1000.0
ijo_14bdo.reactions.EX_h2o_e.upper_bound    = 1000.0
ijo_14bdo.reactions.EX_h_e.upper_bound      = 1000.0
ijo_14bdo.reactions.EX_lac__D_e.upper_bound = 1000.0
ijo_14bdo.reactions.EX_meoh_e.upper_bound   = 1000.0
ijo_14bdo.reactions.EX_o2_e.upper_bound     = 1000.0
```

(continues on next page)

(continued from previous page)

```
ijo_14bdo.reactions.EX_succ_e.upper_bound   = 1000.0
ijo_14bdo.reactions.EX_tungs_e.upper_bound  = 1000.0
ijo_14bdo.reactions.DM_4crsol_c.upper_bound = 1000.0
ijo_14bdo.reactions.DM_5drib_c.upper_bound  = 1000.0
ijo_14bdo.reactions.DM_aacald_c.upper_bound = 1000.0
ijo_14bdo.reactions.DM_amob_c.upper_bound   = 1000.0
ijo_14bdo.reactions.DM_mththf_c.upper_bound = 1000.0
ijo_14bdo.reactions.DM_oxam_c.upper_bound   = 1000.0

# Verify that pathway is operational
sol = sd.fba(ijo_14bdo,obj='EX_14bdo_e',obj_sense='max')
print(f"Maximum possible 1,4-BDO synthesis rate: {sol.objective_value}.")
```

```
Maximum possible 1,4-BDO synthesis rate: 10.659770114941434.
```

Now that we introduced the 1,4-BDO pathway into the model and prepared it for strain design computation, we can continue with the problem formulation. We reduce the enforced attainable growth rate to relax the strain design problem. By introducing a minimum production threshold, we are able to still use the 'ANY' approach. This guarantees at least potentially growth-coupled production, although it is unlikely to identify the strategy with the highetst production potential this way. If the computation is successful, we can repeat the computation with the 'BEST' approach. If the computation is unsuccessful, one should consider using smaller models, such as iJO1366 or compute reaction instead of gene interventions.

[8]:
```
import matplotlib.pyplot as plt
import logging
logging.basicConfig(level=logging.INFO)

# Wild-type plot
datapoints, triang, plot1 = sd.plot_flux_space(ijo_14bdo,
                                               ('BIOMASS_Ec_iJO1366_core_53p95M ','EX_
↪14bdo_e'),
                                               show=False);
# Plot minimal enforced growth rate
_,          _,      plot2 = sd.plot_flux_space(ijo_14bdo,
                                               ('BIOMASS_Ec_iJO1366_core_53p95M ','EX_
↪14bdo_e'),
                                               constraints=['BIOMASS_Ec_iJO1366_core_
↪53p95M >=0.2',
                                                            'EX_14bdo_e >=3'],
                                               show=False);
plot2.set_facecolor('#70AD47')
plot2.set_edgecolor('#70AD47')

module_tilted_optknock = sd.SDModule(ijo_14bdo,sd.names.OPTKNOCK,
                             inner_objective='BIOMASS_Ec_iJO1366_core_53p95M ', #  - 0.
↪001 EX_14bdo_e
                             outer_objective='EX_14bdo_e',
                             constraints=['BIOMASS_Ec_iJO1366_core_53p95M  >= 0.2','EX_
↪14bdo_e >=3'])

# add AKGDC and SSCOARx as adition candidates
ki_cost = {'AKGDC': 1.0, 'SSCOARx':1}
# possible knockout of O2
```

(continues on next page)

(continued from previous page)

```
ko_cost = {r.id : 1.0 for r in ijo_14bdo.reactions if r.genes and ijo_14bdo.genes.s0001␣
↪not in r.genes}
ko_cost.update({'EX_o2_e': 1.0})
# remove AKGDC and SSCOARx from knockout candidates
ko_cost.pop('AKGDC')
ko_cost.pop('SSCOARx')
# Compute strain designs
sols = sd.compute_strain_designs(ijo_14bdo,
                                 sd_modules = module_tilted_optknock,
                                 max_solutions = 1,
                                 max_cost = 3,
                                 ki_cost = ki_cost,
                                 ko_cost = ko_cost,
                                 solution_approach = sd.names.BEST)

# Print solutions
print(f"One compressed solution with cost {sols.sd_cost[0]} found and "+\
      f"expanded to {len(sols.reaction_sd)} solutions in the uncompressed netork.")
print(f"Example intervention set: {['+'+s if v>0 else '-'+s for s,v in sols.reaction_
↪sd[0].items() if v!=0]}")

# OptKnock design plot
interventions = [[{s:1.0},'=',0.0] for s,v in sols.reaction_sd[0].items() if v < 1]
_,          _,      plot3 = sd.plot_flux_space(ijo_14bdo,
                                               ('BIOMASS_Ec_iJO1366_core_53p95M ','EX_
↪14bdo_e'),
                                               # The sign of the glucose exchange␣
↪reaction is flipped since
                                               # reaction is defined in the direction of␣
↪secretion.
                                               constraints=interventions,
                                               show=False);
plot3.set_facecolor('#FFC000')
plot3.set_edgecolor('#FFC000')
# adjust axes limits and show plot
plot3.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot3.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```

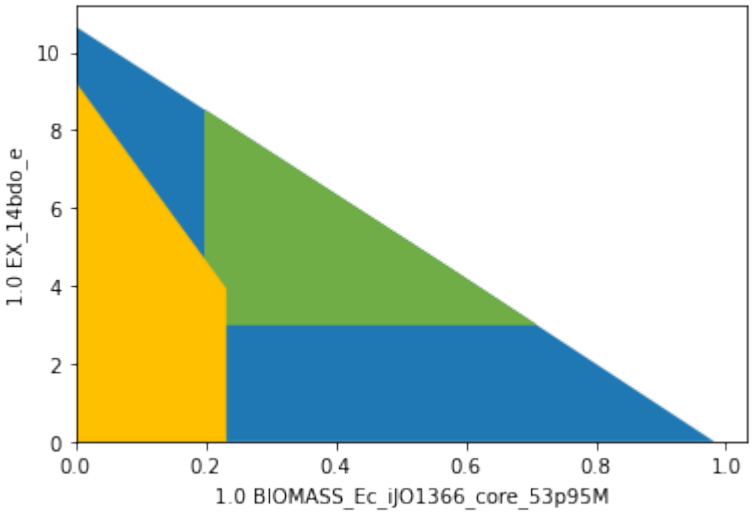
```
INFO:root:Preparing strain design computation.
INFO:root:  Using cplex for solving LPs during preprocessing.
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:  FVA to identify blocked reactions and irreversibilities.
INFO:root:  FVA(s) to identify essential reactions.
INFO:root:Compressing Network (2592 reactions).
INFO:root:  Removing blocked reactions.
c:\Users\Philipp\anaconda3\envs\cnapy-dev\lib\site-packages\cobra\core\group.py:148:␣
↪UserWarning: need to pass in a list
  warn("need to pass in a list")
INFO:root:  Translating stoichiometric coefficients to rationals.
INFO:root:  Removing conservation relations.
INFO:root:  Compression 1: Applying compression from EFM-tool module.
```

(continues on next page)

(continued from previous page)

```
INFO:root:  Reduced to 931 reactions.
INFO:root:  Compression 2: Lumping parallel reactions.
INFO:root:  Last step could not reduce size further (931 reactions).
INFO:root:  Network compression completed. (1 compression iterations)
INFO:root:  Translating stoichiometric coefficients back to float.
INFO:root:  FVA(s) in compressed model to identify essential reactions.
INFO:root:Finished preprocessing:
INFO:root:  Model size: 931 reactions, 421 metabolites
INFO:root:  811 targetable reactions
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:Constructing strain design MILP for solver: cplex.
INFO:root:  Bounding MILP.
INFO:root:Finding optimal strain designs ...
INFO:root:Found solution with objective value 3.9226348048271436
INFO:root:Minimizing number of interventions in subspace with 3 possible targets.
INFO:root:Strain design with cost 3.0: {'ACKr*PTAr': -1, 'AKGDH': -1, 'ATPS4rpp': -1}
INFO:root:Finished solving strain design MILP.
INFO:root:1 solutions to MILP found.
INFO:root:  Decompressing.
INFO:root:2 solutions found.
```

```
One compressed solution with cost 3.0 found and expanded to 2 solutions in the␣
↪uncompressed netork.
Example intervention set: ['-AKGDH', '-ATPS4rpp', '-ACKr']
```



4.6.2 RobustKnock

The RobustKnock [3] approach usues multi-level optimization in a similar way to OptKnock. Whereas Optknock guarantees potentially growth-coupled production, RobusKnock guarantees at least weakly growth coupled production. This is achieved by introducing a third optimization layer. In order to avoid growth-maximal flux states with no production, RobustKnock *maximizes the production rate at maximum growth under the premise that is previously minimized*. The problem can be formalized as follows:

$$
\begin{array}{lll}
\text{maximize} & v_{production} & \\
 & \text{minimize} & v_{production} \\
 & & \text{maximize} \quad v_{biomass} \\
 & & \text{subject to} \quad \mathbf{S}\,\mathbf{v} = \mathbf{0} \\
\text{subject to} & \text{subject to} & v_{BM} \geq v_{BM}^{min} \\
 & & (1 - z_i) \cdot lb_i \leq v_i \leq (1 - z_i) \cdot ub_i,\ \forall i \in \{1, \ldots, n\} \\
 & & \sum z_i \leq \text{maxKOs} \\
 & & z_i \in \{0, 1\}
\end{array}
$$

To translate the presented problem into a single-layer probem that can be used in a Mixed-Integer Linear Problem, one applies the duality principle twice, once on the original problem, and then once again to the linearized nested problem. This yields the MILP:

$$
\begin{gathered}
\text{maximize } v_{production} \\
\text{subject to} \\
\begin{bmatrix}
\mathbf{G} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{D} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
-\mathbf{c}_{\mathbf{prod,min}}^{\intercal} & \mathbf{g}^{\intercal} & \mathbf{0} & \mathbf{c}_{\mathbf{BM}}^{\intercal} & \mathbf{0} \\
\mathbf{0} & \mathbf{G}^{\intercal} & -\mathbf{c}_{\mathbf{BM}}^{\intercal} & \mathbf{0} & \mathbf{I}_{\mathbf{KO}} \\
\mathbf{0} & \mathbf{0} & \mathbf{g} & \mathbf{G} & \mathbf{0}
\end{bmatrix}
\begin{bmatrix} \mathbf{v} \\ \mathbf{u} \\ \mathbf{t} \\ \mathbf{r} \\ \mathbf{s} \end{bmatrix}
\begin{matrix} \leq \\ \leq \\ = \\ = \\ \geq \end{matrix}
\begin{bmatrix} \mathbf{g} \\ \mathbf{d} \\ 0 \\ \mathbf{c}_{\mathbf{BM}}^{\intercal} \\ \mathbf{0} \end{bmatrix} \\
\forall i : z_i = 1 \rightarrow v_i = 0 \\
\forall i : z_i = 1 \rightarrow r_i = 0 \\
\forall i : z_i = 0 \rightarrow s_i = 0 \\
\sum z_i \leq MaxNoKO \\
\mathbf{u} \geq \mathbf{0}, \quad z \in \{0, 1\}
\end{gathered}
$$

4.6.2.1 Example 12: RobustKnock strain design

In StrainDesign, computing OptKnock strain designs is not any different from computing OptKnock strategies. One simply declares a RobustKnock instead of an Optknock module.

[9]:
```
module_robustknock = sd.SDModule(ecc_14bdo,sd.names.ROBUSTKNOCK,
                                 inner_objective='BIOMASS_Ecoli_core_w_GAM',
                                 outer_objective='EX_14bdo_e',
                                 constraints='BIOMASS_Ecoli_core_w_GAM >= 0.5')
```

Launching the strain design computation:

[10]:
```
import logging
logging.basicConfig(level=logging.INFO)
## Compute strain designs
# allow all gene knockouts except for spontanuos
gko_cost = {g.name:1 for g in ecc_14bdo.genes}
```

(continues on next page)

(continued from previous page)

```
gko_cost.pop('s0001')
# possible knockout of O2
ko_cost = {'EX_o2_e': 1}
# addition candidates
ki_cost = {'AKGDC':1, 'SSCOARx':1} # AKGDC was added in example 1.c)

sols = sd.compute_strain_designs(ecc_14bdo,
                                 sd_modules = module_robustknock,
                                 max_solutions = 1,
                                 max_cost = 30,
                                 gko_cost = gko_cost,
                                 ko_cost = ko_cost,
                                 ki_cost = ki_cost,
                                 solution_approach = sd.names.BEST)
# Print solution
print(f"One compressed solution with cost {sols.sd_cost[0]} found and "+\
      f"expanded to {len(sols.gene_sd)} solutions in the uncompressed netork.")
print(f"Example intervention set: {['+'+s if v>0 else '-'+s for s,v in sols.gene_sd[0].
↪items() if v!=0]}")
```

```
INFO:root:Preparing strain design computation.
INFO:root:  Using gurobi for solving LPs during preprocessing.
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:  FVA to identify blocked reactions and irreversibilities.
INFO:root:  FVA(s) to identify essential reactions.
INFO:root:Preprocessing GPR rules (137 genes, 69 gpr rules).
INFO:root:  Simplifyied to 86 genes and 49 gpr rules.
INFO:root:  Extending metabolic network with gpr associations.
INFO:root:Compressing Network (267 reactions).
INFO:root:  Removing blocked reactions.
INFO:root:  Translating stoichiometric coefficients to rationals.
INFO:root:  Removing conservation relations.
INFO:root:  Compression 1: Applying compression from EFM-tool module.
INFO:root:  Reduced to 149 reactions.
INFO:root:  Compression 2: Lumping parallel reactions.
INFO:root:  Reduced to 128 reactions.
INFO:root:  Compression 3: Applying compression from EFM-tool module.
INFO:root:  Reduced to 122 reactions.
INFO:root:  Compression 4: Lumping parallel reactions.
INFO:root:  Reduced to 121 reactions.
INFO:root:  Compression 5: Applying compression from EFM-tool module.
INFO:root:  Reduced to 120 reactions.
INFO:root:  Compression 6: Lumping parallel reactions.
INFO:root:  Last step could not reduce size further (120 reactions).
INFO:root:  Network compression completed. (5 compression iterations)
INFO:root:  Translating stoichiometric coefficients back to float.
INFO:root:  FVA(s) in compressed model to identify essential reactions.
INFO:root:Finished preprocessing:
INFO:root:  Model size: 120 reactions, 71 metabolites
INFO:root:  44 targetable reactions
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:Constructing strain design MILP for solver: gurobi.
```

(continues on next page)

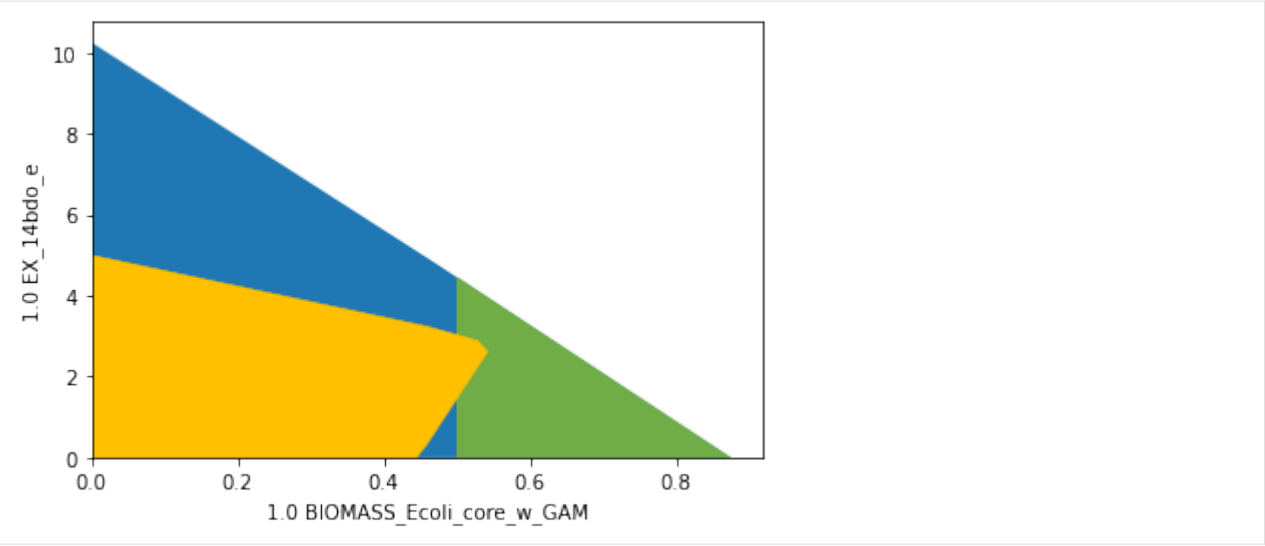
(continued from previous page)

```
INFO:root:  Bounding MILP.
INFO:root:Finding optimal strain designs ...
INFO:root:Found solution with objective value 2.6195399380001269
INFO:root:Minimizing number of interventions in subspace with 14 possible targets.
INFO:root:Strain design with cost 13.0: {'PYK*pykF*R0_g_b1676_or_g_b1854*pykA*R1_g_b1676_
↪or_g_b1854': -1, 'ICL*MALS*aceA*glcB*R0_g_b2976_or_g_b4014*aceB*R1_g_b2976_or_g_b4014':
↪ -1, 'ME1*maeA': -1, 'ME2*maeB': -1, 'AKGDC': 1, 'pgi': -1, 'kgtP': -1, 'pta*R0_g_
↪b2297_or_g_b2458*eutD*R1_g_b2297_or_g_b2458': -1, 'sucC*sucD*R_g_b0728_and_g_b0729': -
↪1, 'gdhA': -1, 'gltP': -1}
INFO:root:Finished solving strain design MILP.
INFO:root:1 solutions to MILP found.
INFO:root:  Decompressing.
INFO:root:  Preparing (reaction-)phenotype prediction of gene intervention strategies.
INFO:root:4 solutions found.
```

```
One compressed solution with cost 13.0 found and expanded to 4 solutions in the␣
↪uncompressed netork.
Example intervention set: ['+AKGDC', '-pgi', '-kgtP', '-gdhA', '-gltP', '-aceA', '-maeA',
↪ '-maeB', '-pta', '-eutD', '-pykF', '-pykA', '-sucC']
```

Plotting results:

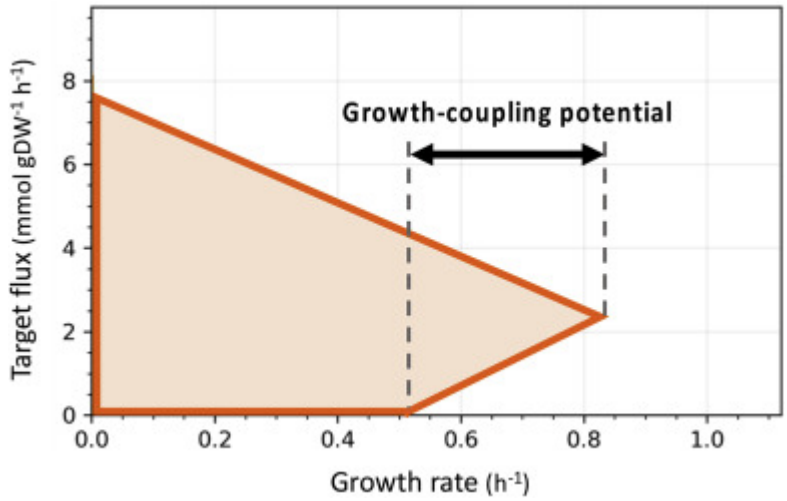
[11]:
```
import matplotlib.pyplot as plt
# Wild-type plot
datapoints, triang, plot1 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'),
                                               show=False);
# Plot minimal enforced growth rate
_,          _,      plot2 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'),
                                               constraints='BIOMASS_Ecoli_core_w_GAM>=0.5
↪',
                                               show=False);
plot2.set_facecolor('#70AD47')
plot2.set_edgecolor('#70AD47')

# OptKnock design plot
interventions = [[{s:1.0},'=',0.0] for s,v in sols.reaction_sd[0].items() if v < 1]
_,          _,      plot3 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'),
                                               # The sign of the glucose exchange␣
↪reaction is flipped since
                                               # reaction is defined in the direction of␣
↪secretion.
                                               constraints=interventions,
                                               show=False);
plot3.set_facecolor('#FFC000')
plot3.set_edgecolor('#FFC000')
# adjust axes limits and show plot
plot3.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot3.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```



The RobustKnock problem is significantly larger than the OptKnock problem. Genome-scale strain design problems are often computationally too expensive to solve. To assess whether genome-scale computation is possible, one should first define a very relaxed problem, use reaction KOs and solve with the 'ANY' approach before attempting more complex setups.

4.6.3 OptCouple

Optcouple [4] pursues a slightly different approach from OptKnock and RobustKnock as it aims to maximize the "growth-coupling potential" (GCP), the span of growth rates ranging from (left) the maximum growth rate without product synthesis to (right) the global maximum growth rate (Figure below taken from the original publication).



Mathematically, the OptCouple appraoch has many parallels to other nested multi-level approaches. In particular OptCouple demands the solution of two distinct flux states. The first subproblem determines the globally maximum flux state. Herefore, LP-duality may be employed to construct a subproblem that is identical to an MCS PROTECT module with an implicit optimization of growth (and optionally a minimum threshold for productivity). The second flux state that needs to be referenced has maximum growth under the condition that there is no product synthesis. This is again similar to the MCS PROTECT module, however the constraint $v_{production} = 0$ must be taken into account in the primal and its dual system. A weakness of OptCouple is that it does not return strain designs with substrate-uptake-coupled production, since a flux state with no production *must* exist as a lower reference point.

4.6.3.1 Example 13: OptCouple strain design

In StrainDesign, the computation of OptCouple designs is as straightforward as for the other approaches.

[12]:
```
module_optcouple = sd.SDModule(ecc_14bdo,sd.names.OPTCOUPLE,
                               inner_objective='BIOMASS_Ecoli_core_w_GAM',
                               prod_id='EX_14bdo_e',
                               min_gcp=0.3)

import logging
logging.basicConfig(level=logging.INFO)
## Compute strain designs
# allow all gene knockouts except for spontanuos
gko_cost = {g.name:1 for g in ecc_14bdo.genes}
gko_cost.pop('s0001')
# possible knockout of O2
ko_cost = {'EX_o2_e': 1}
# addition candidates
ki_cost = {'AKGDC':1, 'SSCOARx':1} # AKGDC was added in example 1.c)

sols = sd.compute_strain_designs(ecc_14bdo,
                                 sd_modules = module_optcouple,
                                 max_solutions = 1,
                                 max_cost = 15,
                                 gko_cost = gko_cost,
                                 ko_cost = ko_cost,
                                 ki_cost = ki_cost,
                                 solution_approach = sd.names.BEST)
# Print solution
print(f"One compressed solution with cost {sols.sd_cost[0]} found and "+\
      f"expanded to {len(sols.gene_sd)} solutions in the uncompressed netork.")
print(f"Example intervention set: {['+'+s if v>0 else '-'+s for s,v in sols.gene_sd[0].
↪items() if v!=0]}")

import matplotlib.pyplot as plt
# Wild-type plot
datapoints, triang, plot1 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'),
                                               show=False);

# OptKnock design plot
interventions = [[{s:1.0},'=',0.0] for s,v in sols.reaction_sd[0].items() if v < 1]
_,          _,      plot3 = sd.plot_flux_space(ecc_14bdo,
                                               ('BIOMASS_Ecoli_core_w_GAM','EX_14bdo_e'),
                                               # The sign of the glucose exchange␣
↪reaction is flipped since
                                               # reaction is defined in the direction of␣
↪secretion.
                                               constraints=interventions,
                                               show=False);
plot3.set_facecolor('#FFC000')
plot3.set_edgecolor('#FFC000')
# adjust axes limits and show plot
```

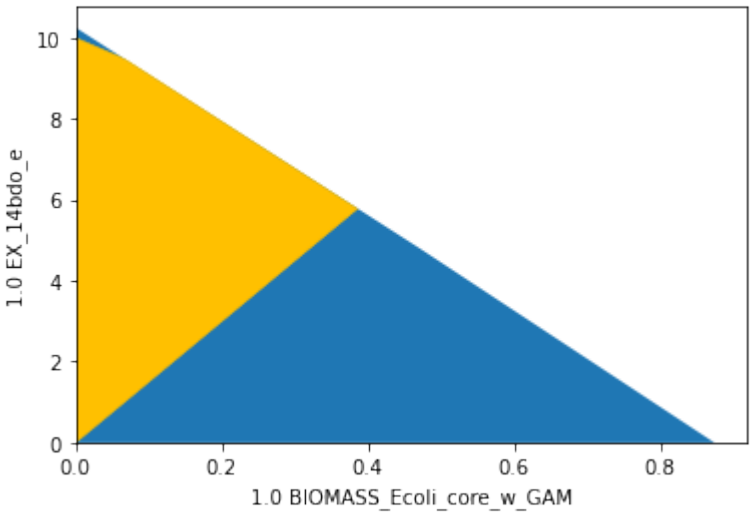
(continues on next page)

(continued from previous page)

```
plot3.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot3.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```

```
INFO:root:Preparing strain design computation.
INFO:root:  Using gurobi for solving LPs during preprocessing.
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:  FVA to identify blocked reactions and irreversibilities.
INFO:root:  FVA(s) to identify essential reactions.
INFO:root:Preprocessing GPR rules (137 genes, 69 gpr rules).
INFO:root:  Simplifyied to 127 genes and 63 gpr rules.
INFO:root:  Extending metabolic network with gpr associations.
INFO:root:Compressing Network (335 reactions).
INFO:root:  Removing blocked reactions.
INFO:root:  Translating stoichiometric coefficients to rationals.
INFO:root:  Removing conservation relations.
INFO:root:  Compression 1: Applying compression from EFM-tool module.
INFO:root:  Reduced to 182 reactions.
INFO:root:  Compression 2: Lumping parallel reactions.
INFO:root:  Reduced to 154 reactions.
INFO:root:  Compression 3: Applying compression from EFM-tool module.
INFO:root:  Reduced to 146 reactions.
INFO:root:  Compression 4: Lumping parallel reactions.
INFO:root:  Reduced to 145 reactions.
INFO:root:  Compression 5: Applying compression from EFM-tool module.
INFO:root:  Reduced to 144 reactions.
INFO:root:  Compression 6: Lumping parallel reactions.
INFO:root:  Last step could not reduce size further (144 reactions).
INFO:root:  Network compression completed. (5 compression iterations)
INFO:root:  Translating stoichiometric coefficients back to float.
INFO:root:  FVA(s) in compressed model to identify essential reactions.
INFO:root:Finished preprocessing:
INFO:root:  Model size: 144 reactions, 84 metabolites
INFO:root:  58 targetable reactions
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:Constructing strain design MILP for solver: gurobi.
INFO:root:  Bounding MILP.
INFO:root:Finding optimal strain designs ...
INFO:root:Found solution with objective value 0.38539164202549997
INFO:root:Minimizing number of interventions in subspace with 10 possible targets.
INFO:root:Strain design with cost 7.0: {'ME2*maeB': -1, 'AKGDC': 1, 'kgtP': -1, 'rpe*R0_
↪g_b3386_or_g_b4301*sgcE*R1_g_b3386_or_g_b4301': -1, 'sucC*sucD*R_g_b0728_and_g_b0729':␣
↪-1, 'pntB*pntA*R_g_b1602_and_g_b1603': -1}
INFO:root:Finished solving strain design MILP.
INFO:root:1 solutions to MILP found.
INFO:root:  Decompressing.
INFO:root:  Preparing (reaction-)phenotype prediction of gene intervention strategies.
INFO:root:4 solutions found.
```

```
One compressed solution with cost 7.0 found and expanded to 4 solutions in the␣
↪uncompressed netork.
Example intervention set: ['+AKGDC', '-kgtP', '-maeB', '-rpe', '-sgcE', '-sucC', '-pntB']
```



As in this case, OptCouple computations often result in directionally growth-coupled strain designs, as the maximum growth coupling potential is often obtained when the maximal growth rate without production is zero. In our exampe, we observe a growth-coupling potential of 0.4.

4.6.4 Combining nested optimization strain design with MCS

Nested strain design methods can be combined with the MCS approach to enforce additional properties. This allows flux space tailoring with unprecedented possibilities and precision.

> **Warning:**
>
> In genome-scale setups the MILP of combined approaches is usually too large for efficient solving.

4.6.4.1 Example 14: Combining OptKnock with a tilted objective function and the MCS approach

Let's construct a strain design with at least weakly growth-coupled production, while we avoid ethanol nor succinate as by-products.

We construct the strain design problems with an OptKnock with a tilted objective function and an MCS module as follows:

[13]:
```
import logging
import matplotlib.pyplot as plt
logging.basicConfig(level=logging.INFO)

# Enforce 1,4-BDO production at maximal growth
module_optknock = sd.SDModule(ecc_14bdo,sd.names.OPTKNOCK,
                              inner_objective='BIOMASS_Ecoli_core_w_GAM - 0.01 EX_14bdo_e
↪',
                              outer_objective='EX_14bdo_e',
                              constraints=['BIOMASS_Ecoli_core_w_GAM>=0.5'])
# Suppress Ethanol and Succinate production at maximal growth
module_mcs_suppress_etoh_succ = sd.SDModule(ecc_14bdo,sd.names.SUPPRESS,
```

(continues on next page)

(continued from previous page)

```
                                inner_objective='BIOMASS_Ecoli_core_w_GAM',
                                constraints=['EX_etoh_e + EX_succ_e >= 0.1'])

## Compute strain designs
# allow all gene knockouts except for spontanuos
gko_cost = {g.name:1 for g in ecc_14bdo.genes}
gko_cost.pop('s0001')
# possible knockout of O2
ko_cost = {'EX_o2_e': 1}
# addition candidates
ki_cost = {'AKGDC':1, 'SSCOARx':1} # AKGDC was added in example 1.c)

sols = sd.compute_strain_designs(ecc_14bdo,
                                 sd_modules = [module_optknock,module_mcs_suppress_etoh_
↪succ],
                                 max_solutions = 1,
                                 max_cost = 15,
                                 gko_cost = gko_cost,
                                 ko_cost = ko_cost,
                                 ki_cost = ki_cost,
                                 solution_approach = sd.names.BEST)
# Print solution
print(f"One compressed solution with cost {sols.sd_cost[0]} found and "+\
      f"expanded to {len(sols.gene_sd)} solutions in the uncompressed netork.")
print(f"Example intervention set: {['+'+s if v>0 else '-'+s for s,v in sols.gene_sd[0].
↪items() if v!=0]}")


%matplotlib inline
# Aerobic design plot
interventions = [[{s:1.0},'=',0.0] for s,v in sols.reaction_sd[0].items() if v < 1]
_,_,plot1 = sd.plot_flux_space(ecc_14bdo,('BIOMASS_Ecoli_core_w_GAM','EX_etoh_e + EX_
↪succ_e','EX_14bdo_e'),
                   constraints=interventions,points=50,show=False);
plot1._axes.view_init(20, 60)
plt.show()
```

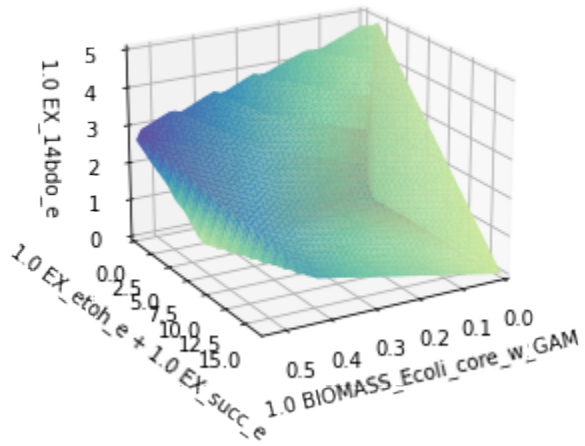
```
INFO:root:Preparing strain design computation.
INFO:root:  Using gurobi for solving LPs during preprocessing.
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:  FVA to identify blocked reactions and irreversibilities.
INFO:root:  FVA(s) to identify essential reactions.
INFO:root:Preprocessing GPR rules (137 genes, 69 gpr rules).
INFO:root:  Simplifyied to 86 genes and 49 gpr rules.
INFO:root:  Extending metabolic network with gpr associations.
INFO:root:Compressing Network (267 reactions).
INFO:root:  Removing blocked reactions.
INFO:root:  Translating stoichiometric coefficients to rationals.
INFO:root:  Removing conservation relations.
INFO:root:  Compression 1: Applying compression from EFM-tool module.
INFO:root:  Reduced to 149 reactions.
INFO:root:  Compression 2: Lumping parallel reactions.
```

(continues on next page)

(continued from previous page)

```
INFO:root:  Reduced to 128 reactions.
INFO:root:  Compression 3: Applying compression from EFM-tool module.
INFO:root:  Reduced to 122 reactions.
INFO:root:  Compression 4: Lumping parallel reactions.
INFO:root:  Reduced to 121 reactions.
INFO:root:  Compression 5: Applying compression from EFM-tool module.
INFO:root:  Reduced to 120 reactions.
INFO:root:  Compression 6: Lumping parallel reactions.
INFO:root:  Last step could not reduce size further (120 reactions).
INFO:root:  Network compression completed. (5 compression iterations)
INFO:root:  Translating stoichiometric coefficients back to float.
INFO:root:  FVA(s) in compressed model to identify essential reactions.
INFO:root:Finished preprocessing:
INFO:root:  Model size: 120 reactions, 71 metabolites
INFO:root:  44 targetable reactions
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:Constructing strain design MILP for solver: gurobi.
INFO:root:  Bounding MILP.
INFO:root:Finding optimal strain designs ...
INFO:root:Found solution with objective value 2.6195399376393134
INFO:root:Minimizing number of interventions in subspace with 13 possible targets.
INFO:root:Strain design with cost 13.0: {'PYK*pykF*R0_g_b1676_or_g_b1854*pykA*R1_g_b1676_
↪or_g_b1854': -1, 'ICL*MALS*aceA*glcB*R0_g_b2976_or_g_b4014*aceB*R1_g_b2976_or_g_b4014':
↪ -1, 'ME1*maeA': -1, 'ME2*maeB': -1, 'SSCOARx': 1, 'pgi': -1, 'kgtP': -1, 'pta*R0_g_
↪b2297_or_g_b2458*eutD*R1_g_b2297_or_g_b2458': -1, 'sucC*sucD*R_g_b0728_and_g_b0729': -
↪1, 'gdhA': -1, 'gltP': -1}
INFO:root:Finished solving strain design MILP.
INFO:root:1 solutions to MILP found.
INFO:root:  Decompressing.
INFO:root:  Preparing (reaction-)phenotype prediction of gene intervention strategies.
INFO:root:4 solutions found.
```

```
One compressed solution with cost 13.0 found and expanded to 4 solutions in the␣
↪uncompressed netork.
Example intervention set: ['+SSCOARx', '-pgi', '-kgtP', '-gdhA', '-gltP', '-aceA', '-maeA
↪', '-maeB', '-pta', '-eutD', '-pykF', '-pykA', '-sucC']
```



The plot shows the flux space of the designed strain. While at lower growth rates ethanol and succinate production is still possible, there is a growth-based selection against ethanol and succinate producers and in favor of 1,4-BDO producing phenotypes. The "waves" on top of the shape are artifacts of the plotting grid and would disappear with infinte resolution.

4.7 Standalone network compression

An effective network compression is essential to any strain design computation. Since it may also be of interest outside the context of strain design, this example may help you using the network compression routine independently. Likewise, StrainDesign also offers the integration of GPR rules into the metabolic networks as a separate function.

The network compression routine removes blocked reactions, removes conservation relations and then performs alternatingly lumps dependent (`compress_model_efmtool`) and parallel (`compress_model_parallel`) reactions. The compression returns a compressed network and a list of so-called "compression maps". Each map consists of a dictionary that contains complete information for reversing the compression steps successively and expand information obtained from the compressed model to the full model. Each entry of each map contains the id of a compressed reaction, associated with the original reaction names and their factor (provided as a rational number) with which they were lumped.

Furthermore, the user can select reactions that should be exempt from the parallel compression. In the following, we provide the code snippet that can be used to call the compression.

[1]:
```
from straindesign import compress_model, remove_dummy_bounds
import cobra
import logging
logging.basicConfig(level=logging.INFO)

# load model
iml1515 = cobra.io.load_model('iML1515')
# replace dummy bounds with infinite
logging.info(f"Replacing dummy bounds of +/- 1000 with infinite.")
remove_dummy_bounds(iml1515)
logging.info(f"The original model contains {len(iml1515.reactions)} reactions.")
cmp_map = compress_model(iml1515)
```

```
Set parameter Username
```

```
INFO:gurobipy.gurobipy:Set parameter Username
```

```
Academic license - for non-commercial use only - expires 2023-07-20
```

```
INFO:gurobipy.gurobipy:Academic license - for non-commercial use only - expires 2023-07-
↪20
INFO:root:Replacing dummy bounds of +/- 1000 with infinite.
WARNING:root:  Removing reaction bounds when larger than the cobra-threshold of 1000.
INFO:root:The original model contains 2712 reactions.
INFO:root:  Removing blocked reactions.
INFO:root:  Translating stoichiometric coefficients to rationals.
INFO:root:  Removing conservation relations.
INFO:root:  Compression 1: Applying compression from EFM-tool module.
INFO:root:  Reduced to 1244 reactions.
INFO:root:  Compression 2: Lumping parallel reactions.
INFO:root:  Reduced to 1225 reactions.
INFO:root:  Compression 3: Applying compression from EFM-tool module.
INFO:root:  Reduced to 1216 reactions.
```

(continues on next page)

(continued from previous page)

```
INFO:root:  Compression 4: Lumping parallel reactions.
INFO:root:  Reduced to 1213 reactions.
INFO:root:  Compression 5: Applying compression from EFM-tool module.
INFO:root:  Last step could not reduce size further (1213 reactions).
INFO:root:  Network compression completed. (4 compression iterations)
INFO:root:  Translating stoichiometric coefficients back to float.
```

Here, the original model of iML1515 was compressed from 2712 to 1213 reactions in 4 steps. The greatest reduction in size is achieved in the first step. Still, the benefit of the parallel compression should not be underestimated since it removes superfluous degrees of freedom in the model that don't add significant information. The resulting compression maps can be used to translate data between the compressed and the original network.

[2]:
```
# Let us track the compression of the ADK4 reaction
orig_reac = 'ADK4'

for i,c in enumerate(cmp_map):
    logging.info(f"Compression step {i+1} was {['parallel' if c['parallel'] else 'linear
↪'][0]}.")
    lumped_reaction = [k for k,v in c['reac_map_exp'].items() if orig_reac in v][0]
    if len(c['reac_map_exp'][lumped_reaction]) == 1:
        logging.info(f"Reaction '{orig_reac}' was not affected by this reaction")
    else:
        logging.info(f"Reaction '{orig_reac}' was lumped to {lumped_reaction} with the␣
↪coefficients {c['reac_map_exp'][lumped_reaction]}.")
    orig_reac = lumped_reaction
```

```
INFO:root:Compression step 1 was linear.
INFO:root:Reaction 'ADK4' was lumped to ADK4*NTP10 with the coefficients {'ADK4': -1,
↪'NTP10': 1}.
INFO:root:Compression step 2 was parallel.
INFO:root:Reaction 'ADK4*NTP10' was not affected by this reaction
INFO:root:Compression step 3 was linear.
INFO:root:Reaction 'ADK4*NTP10' was not affected by this reaction
INFO:root:Compression step 4 was parallel.
INFO:root:Reaction 'ADK4*NTP10' was lumped to␣
↪ADK4*NTP10*NADPHXD*NADPHXE*NADPHHR*NADPHHS*NADHHR*NADHXE*NADHHS*NADHXD with the␣
↪coefficients {'ADK4*NTP10': 1, 'NADPHXD*NADPHXE*NADPHHR*NADPHHS': 1,
↪'NADHHR*NADHXE*NADHHS*NADHXD': 1}.
```

4.7.1 Standalone GPR-integraton

GPR rules can be introduced to the metabolic model in a way that the logical GPR-terms are reflected and the gene presence or absence can be simulated by setting flux bounds. This can be useful to investigate the space of feasible steady-state flux vectors after gene knockouts or be used to investigate the role of genes in different elementary flux modes or vectors.

[3]:
```
from straindesign import extend_model_gpr
import cobra
import logging
logging.basicConfig(level=logging.INFO)
# load model
```

(continues on next page)

(continued from previous page)

```
e_coli_core = cobra.io.load_model('e_coli_core')
# extend model with GPR rules using gene-IDs
extend_model_gpr(e_coli_core)
# printing the last 10 reactions (corresponding to genes and GPR rules) of the GPR-
↪extend network
logging.info('The first 95 reactions are original reactions from e_coli_core. All other␣
↪reactions result from the integration of GPR rules.')
logging.info('Here, we only print the last 10 reactions of the model for showcasing.')
[print(r) for r in e_coli_core.reactions[-10:]];
```

```
INFO:root:The first 95 reactions are original reactions from e_coli_core. All other␣
↪reactions result from the integration of GPR rules.
INFO:root:Here, we only print the last 10 reactions of the model for showcasing.
```

```
R_g_b2276_and_g_b2277_and_g_b2278_and_g_b2279_and_g_b2280_and_g_b2281_and_g_b2282_and_g_
↪b2283_and_g_b2284_and_g_b2285_and_g_b2286_and_g_b2287_and_g_b2288: g_b2276 + g_b2277 +␣
↪g_b2278 + g_b2279 + g_b2280 + g_b2281 + g_b2282 + g_b2283 + g_b2284 + g_b2285 + g_
↪b2286 + g_b2287 + g_b2288 --> g_b2276_and_g_b2277_and_g_b2278_and_g_b2279_and_g_b2280_
↪and_g_b2281_and_g_b2282_and_g_b2283_and_g_b2284_and_g_b2285_and_g_b2286_and_g_b2287_
↪and_g_b2288
b3962:  --> g_b3962
R0_g_b3962_or_g_b1602_and_g_b1603: g_b3962 --> g_b3962_or_g_b1602_and_g_b1603
R1_g_b3962_or_g_b1602_and_g_b1603: g_b1602_and_g_b1603 --> g_b3962_or_g_b1602_and_g_b1603
b0451:  --> g_b0451
R0_g_s0001_or_g_b0451: g_s0001 --> g_s0001_or_g_b0451
R1_g_s0001_or_g_b0451: g_b0451 --> g_s0001_or_g_b0451
b0114:  --> g_b0114
b0115:  --> g_b0115
R_g_b0114_and_g_b0115_and_g_b0116: g_b0114 + g_b0115 + g_b0116 --> g_b0114_and_g_b0115_
↪and_g_b0116
```

4.7.1.1 Gene perturbation studies

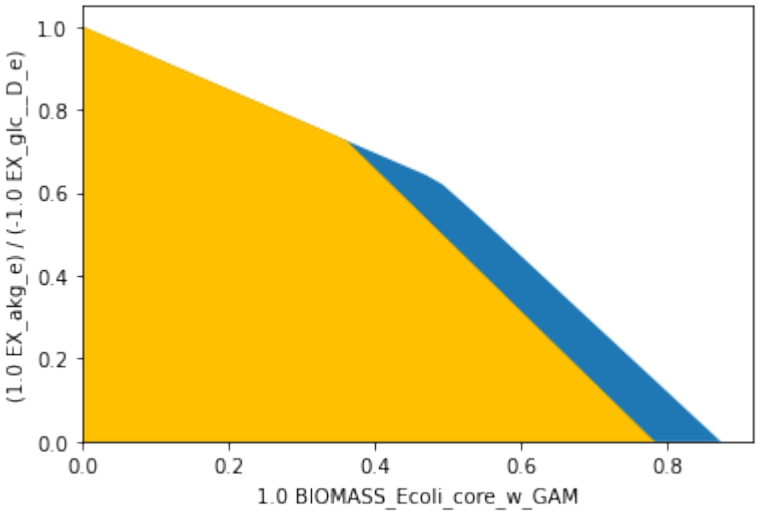
GPR-extended models can be used to study how single or multiple gene-KOs affect the steady-state flux space. We can therefore integrate the GPR-rules in the model and then plot flux spaces that take into account knockouts. In the plots below, we show how knocking out the gene *lpd* affects bacterial growth and AKG yields.

[4]:
```
import straindesign as sd
import cobra
import logging
import matplotlib.pyplot as plt
logging.basicConfig(level=logging.INFO)
# load model
e_coli_core = cobra.io.load_model('e_coli_core')
# extend model with GPR rules using gene-names instead of IDs.
sd.extend_model_gpr(e_coli_core, use_names=True)

# Wild-type plot
datapoints, triang, plot1 = sd.plot_flux_space(e_coli_core,
                                               ('BIOMASS_Ecoli_core_w_GAM',('EX_akg_e','-
↪EX_glc__D_e')),
                                               show=False);
```

(continues on next page)

(continued from previous page)

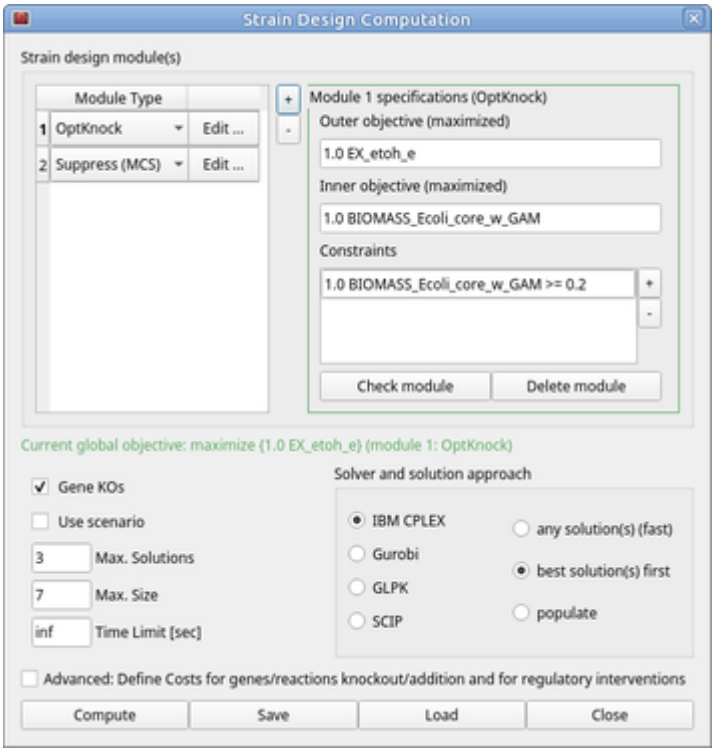
```
# plotting designed strain
_,          _,      plot2 = sd.plot_flux_space(e_coli_core,
                                                ('BIOMASS_Ecoli_core_w_GAM',('EX_akg_e','-
↪EX_glc__D_e')),
                                                constraints=['lpd = 0'],
                                                show=False);
plot2.set_facecolor('#FFC000')
plot2.set_edgecolor('#FFC000')
# adjust axes limits and show plot
plot2.axes.set_xlim(0, 1.05*max([a[0] for a in datapoints]))
plot2.axes.set_ylim(0, 1.05*max([a[1] for a in datapoints]))
plt.show()
```



The blue shape represents the wild type and the yellow shape the *lpd* knockout strain flux space. The model predicts that the knockout will not affect the globally attainable -ketoglutarate yield, but reduces the maximal possible growth rate. At higher growth rates, -ketoglutarate yield is reduced.

4.8 CNApy interface

CNApy, a GUI-featured toolbox for metabolic modeling offers a graphical user interface for the modeling and design of metabolic networks. The software provides a dialog box for specifying strain design problems. A full documentation can be found here.



4.9 StrainDesign API

4.9.1 straindesign

StrainDesign package for computational metabolic engineering

4.9.1.1 Submodules

straindesign.compute_strain_designs

Function: computing metabolic strain designs (compute_strain_designs)

Module Contents

straindesign.compute_strain_designs.**compute_strain_designs**(*model: cobra.Model*, ***kwargs: dict*) → straindesign.SDSolutions

Computes strain designs for a user-defined strain design problem

A number of arguments can be specified to detail the problem and influence the solution process. This function supports the computation of Minimal Cut Sets (MCS), OptKock, RobustKnock and OptCouple strain designs. It is possible to combine any of the latter ones with the MCS approach, e.g., to engineer growth coupled production, but also suppress the production of an undesired by-product. The computation can be started in two different ways. Either by specifying the computation parameters indivdually or reuse a parameters dictionary from a previous computation. CNApy stores strain design setup dics as JSON ".sd"-files that can be loaded in python and used as an input for this function.

Example

sols = compute_strain_designs(model, sd_modules=[sd_module1, sd_module2], solution_approach = 'any')

Parameters

- **model** (*cobra.Model*) – A metabolic model that is an instance of the cobra.Model class. The model may or may not contain genes/GPR-rules.
- **sd_setup** (*dict*) – sd_setup should be a dictionary containing a set of parameters for strain design computation. The allowed keywords are the same listed hereafter. Therefore, *sd_setup and other arguments (except for model) must not be used together*.
- **sd_modules** (*[straindesign.SDModule]*) – List of strain design modules that describe the sub-problems, such as the MCS-like protection or suppression of flux subspaces or the OptKnock, RobustKnock or OptCouple objective and constraints. The list of modules determines the global objective function of the strain design computation. If only SUPPRESS and PROTECT modules are used, the strain design computation is MCS-like, such that the number of interventions is minimized. If a module for one of the nested optimization approaches is used, the global objective function is retrieved from this module. The number of SUPPRESS and PROTECT modules is unrestricted and can be combined with the other modules, however only one of the modules OPTKNOCK, ROBUSKNOCK and OPTCOUPLE may be used at a time. For details, see SDModule.
- **solver** (*optional (str)*) – (Default: same as defined in model / COBRApy) The solver that should be used for preparing and carrying out the strain design computation. Allowed values are 'cplex', 'gurobi', 'scip' and 'glpk'.
- **max_cost** (*optional (int)*) – (Default: inf): The maximum cost threshold for interventions. Every possible intervention is associated with a cost value (1, by default). Strain designs cannot exceed the max_cost threshold. Individual intervention cost factors may be defined through ki_cost, ko_cost, gki_cost, gko_cost and reg_cost.
- **max_solutions** (*optional (int)*) – (Default: inf) The maximum number of MILP solutions that are generated for a strain design problem. The number of returned strain designs is usually larger than the number of max_solutions, since a MILP solution is decompressed to multiple strain designs. When the compress-flag is set to 'False' the number of returned solutions is equal to max_solutions.
- **M** (*optional (int)*) – (Default: None) If this value is specified (and non-zero, not None), the computation uses the big-M method instead of indicator constraints. Since GLPK does not support indicator constraints it uses the big-M method by default (with M=1000). M should be chosen 'sufficiently large' to avoid computational artifacts and 'sufficiently small' to avoid numerical issues.
- **compress** (*optional (bool)*) – (Default: True) If 'True', the interative network compressor is used.
- **gene_kos** (*optional (bool)*) – (Default: False) If 'True', strain designs are computed based on gene-knockouts instead of reaction knockouts. This parameter needs not be defined if any of ki_cost, ko_cost, gki_cost, gko_cost and reg_cost is used. By default, reactions are considered as knockout targets.
- **ko_cost** (*optional (dict)*) – (Default: None) A dictionary of reaction identifiers and their associated knockout costs. If not specified, all reactions are treated as knockout candidates, equivalent to ko_cost = {'r1':1, 'r2':1, ...}. If a subset of reactions is listed in the dict, all other are not considered as knockout candidates.
- **ki_cost** (*optional (dict)*) – (Default: None) A dictionary of reaction identifiers and their associated costs for addition. If not specified, all reactions are treated as knockout can-

didates. Reaction addition candidates must be present in the original model with the intended flux boundaries **after** insertion. Additions are treated adversely to knockouts, meaning that their exclusion from the network is not associated with any cost while their presence entails intervention costs.

- **gko_cost** (*optional (dict)*) – (Default: None) A dictionary of gene identifiers and their associated knockout costs. To reference genes, gene IDs can be used,as well as gene names. If not specified, genes are not treated as knockout candidates. An exception is the 'gene_kos' argument. If 'gene_kos' is used, all genes are treated as knockout candidates with intervention costs of 1. This is equivalent to gko_cost = {'g1':1, 'g2':1, ... }.
- **gki_cost** (*optional (dict)*) – (Default: None) A dictionary of gene identifiers and their associated addition costs. To reference genes, gene IDs can be used, as well as gene names. If not specified, none of the genes are treated as addition candidates.
- **reg_cost** (*optional [dict]*) – (Default: None) Regulatory interventions candidates can be optionally specified as a list. Thereby, the constraint marking the regulatory intervention is put as key and the associated intervention cost is used as the corresponding value. E.g., reg_cost = {'1 EX_o2_e = -1': 1, ... <other regulatory interventions>}. Instead of strings, constraints can also be passed as lists. reg_cost = {[{'EX_o2_e':1}, '=', -1]: 1, ... }
- **solution_approach** (*optional (str)*) – (Default: 'best') The approach used to find strain designs. Possible values are 'any', 'best' or 'populate'. 'any' is usually the fastest option, since optimality is not enforced. Hereby computed MCS are still irreducible intervention sets, however, not MCS with the fewest possible number of interventions. 'best' computes globally optimal strain designs, that is, MCS with the fewest number of interventions, OptKnock strain designs with the highest possible production rate, OptCouple strain designs with the hightest growth coupling potential etc.. 'populate' does the same as 'best', but makes use of CPLEX' and Gurobi's populate function to generate multiple strain designs. It is identical to 'best' when used with SCIP or GLPK. Attention: If 'any' used with OptKnock, for instance, the MILP may return the wild type as a possible immediately. Technically, the wiltype fulfills the criterion of maximal growth (inner objective) and maximality of the global objective is omitted by using 'any', so that carrying no product synthesis is permitted. Additional constraints can be used in the OptKnock problem to circumvent this. However, Optknock should generally be used with the 'best' option.
- **time_limit** (*optional (int)*) – (Default: inf) The time limit in seconds for the MILP-solver.
- **advanced** (*optional (bool)*) – Dummy parameters used for the CNApy interface.
- **use_scenario** (*optional (bool)*) – Dummy parameters used for the CNApy interface.

Returns

An object that contains all computed strain designs. If strain designs were computed as gene-interventions, the solution object contains a set of corresponding reaction-interventions that facilitate the analysis of the computed strain designs with COBRA methods.

Return type

(*SDSolutions*)

straindesign.compute_strain_designs.**postprocess_reg_sd**(*reg_cost*, *sd*)

Postprocess regulatory interventions

Mark regulatory interventions with true or false

straindesign.cplex_interface

CPLEX solver interface for LP and MILP

Module Contents

class straindesign.cplex_interface.**Cplex_MILP_LP**(*c=None*, *A_ineq=None*, *b_ineq=None*, *A_eq=None*, *b_eq=None*, *lb=None*, *ub=None*, *vtype=None*, *indic_constr=None*)

Bases: cplex.Cplex

CPLEX interface for MILP and LP

This class is a wrapper for the CPLEX-Python API to offer bindings and namings for functions for the construction and manipulation of MILPs and LPs in an vector-matrix-based manner that are consistent with those of the other solver interfaces in the StrainDesign package. The purpose is to unify the instructions for operating with MILPs and LPs throughout StrainDesign.

The CPLEX interface provides support for indicator constraints as well as for the populate function.

Accepts a (mixed integer) linear problem in the form:
minimize(c), subject to: A_ineq * x <= b_ineq, A_eq * x = b_eq, lb <= x <= ub, forall(i) type(x_i) = vtype(i) (continous, binary, integer), indicator constraints: x(j) = [0|1] -> a_indic * x [<=|=|>=] b_indic

Please ensure that the number of variables and (in)equalities is consistent

Example

cplex = Cplex_MILP_LP(c, A_ineq, b_ineq, A_eq, b_eq, lb, ub, vtype, indic_constr)

Parameters

- **c** (*list of float*) – (Default: None) The objective vector (Objective sense: minimization).
- **A_ineq** (*sparse.csr_matrix*) – (Default: None) A coefficient matrix of the static inequalities.
- **b_ineq** (*list of float*) – (Default: None) The right hand side of the static inequalities.
- **A_eq** (*sparse.csr_matrix*) – (Default: None) A coefficient matrix of the static equalities.
- **b_eq** (*list of float*) – (Default: None) The right hand side of the static equalities.
- **lb** (*list of float*) – (Default: None) The lower variable bounds.
- **ub** (*list of float*) – (Default: None) The upper variable bounds.
- **vtype** (*str*) – (Default: None) A character string that specifies the type of each variable: ‘c’ontinous, ‘b’inary or ‘i’nteger
- **indic_constr** (IndicatorConstraints) – (Default: None) A set of indicator constraints stored in an object of IndicatorConstraints (see reference manual or docstring).
- **Returns** – (Cplex_MILP_LP):

  A CPLEX MILP/LP interface class.

add_eq_constraints(*A_eq*, *b_eq*)

Add equality constraints to the model

Additional equality constraints have the form A_eq * x = b_eq. The number of columns in A_eq must match with the number of variables x in the problem.

Parameters

- **A_eq** (*sparse.csr_matrix*) – The coefficient matrix
- **b_eq** (*list of float*) – The right hand side vector

add_ineq_constraints(*A_ineq*, *b_ineq*)

Add inequality constraints to the model

Additional inequality constraints have the form A_ineq * x <= b_ineq. The number of columns in A_ineq must match with the number of variables x in the problem.

Parameters

- **A_ineq** (*sparse.csr_matrix*) – The coefficient matrix
- **b_ineq** (*list of float*) – The right hand side vector

populate(*n*) → Tuple[List, float, float]

Generate a solution pool for MILPs

Example

sols_x, optim, status = cplex.populate()

Returns

(Tuple[List of lists, float, float])

solution_vectors, optimal_value, optimization_status

set_ineq_constraint(*idx*, *a_ineq*, *b_ineq*)

Replace a specific inequality constraint

Replace the constraint with the index idx with the constraint a_ineq*x ~ b_ineq

Parameters

- **idx** (*int*) – Index of the constraint
- **a_ineq** (*list of float*) – The coefficient vector
- **b_ineq** (*float*) – The right hand side value

set_objective(*c*)

Set the objective function with a vector

set_objective_idx(*C*)

Set the objective function with index-value pairs

e.g.: C=[[1, 1.0], [4,-0.2]]

set_time_limit(*t*)

Set the computation time limit (in seconds)

set_ub(*ub*)

Set the upper bounds to a given vector

slim_solve() → float

Solve the MILP or LP, but return only the optimal value

Example

optim = cplex.slim_solve()

Returns

(float)

Optimum value of the objective function.

solve() → Tuple[List, float, float]

Solve the MILP or LP

Example

sol_x, optim, status = cplex.solve()

Returns

(Tuple[List, float, float])

solution_vector, optimal_value, optimization_status

straindesign.efmtool

Functions for the compression of metabolic networks, taken from the efmtool_link package

Functions in this module not meant to be used outside the compression of networks. For a the documentation of the efmtool compression provided by StrainDesign, refer to the networktools module.

Module Contents

straindesign.efmtool.**basic_columns_rat**(*mx*, *tolerance=0*)

efmtool: Translate matrix coefficients to rational numbers

straindesign.efmtool.**jBigFraction2sympyRat**(*val*)

efmtool: Translate rational numbers to sympy rational numbers

straindesign.efmtool.**jBigIntegerPair2sympyRat**(*numer*, *denom*)

efmtool: Translate big integer pair to sympy rational numbers

straindesign.efmtool.**jpypeArrayOfArrays2numpy_mat**(*jmat*)

efmtool: Translate array of arrays to numpy matrix

straindesign.efmtool.**numpy_mat2jpypeArrayOfArrays**(*npmat*)

efmtool: Translate matrix to array of arrays

straindesign.efmtool.**sympyRat2jBigIntegerPair**(*val*)

efmtool: Translate rational numbers to big integer pair

straindesign.glpk_interface

GLPK solver interface for LP and MILP

Module Contents

class straindesign.glpk_interface.**GLPK_MILP_LP**(*c*, *A_ineq*, *b_ineq*, *A_eq*, *b_eq*, *lb*, *ub*, *vtype*, *indic_constr*, *M=None*)

GLPK interface for MILP and LP

This class is a wrapper for the GLPK-Python API to offer bindings and namings for functions for the construction and manipulation of MILPs and LPs in an vector-matrix-based manner that are consistent with those of the other solver interfaces in the StrainDesign package. The purpose is to unify the instructions for operating with MILPs and LPs throughout StrainDesign.

The GLPK interface does not natively support indicator constraints. They are hence translated to bigM-constraints when passed to the GLPK constructor (see docstring of IndicatorConstraints). The GLPK interface does not natively support the populate function. A high level implementation emulates the behavior of populate.

Accepts a (mixed integer) linear problem in the form:

minimize(c), subject to: A_ineq * x <= b_ineq, A_eq * x = b_eq, lb <= x <= ub, forall(i) type(x_i) = vtype(i) (continous, binary, integer), indicator constraints: x(j) = [0|1] -> a_indic * x [<=|=|>=] b_indic

Please ensure that the number of variables and (in)equalities is consistent

Example

glpk = GLPK_MILP_LP(c, A_ineq, b_ineq, A_eq, b_eq, lb, ub, vtype, indic_constr, M)

Parameters

- **c** (*list of float*) – (Default: None) The objective vector (Objective sense: minimization).
- **A_ineq** (*sparse.csr_matrix*) – (Default: None) A coefficient matrix of the static inequalities.
- **b_ineq** (*list of float*) – (Default: None) The right hand side of the static inequalities.
- **A_eq** (*sparse.csr_matrix*) – (Default: None) A coefficient matrix of the static equalities.
- **b_eq** (*list of float*) – (Default: None) The right hand side of the static equalities.
- **lb** (*list of float*) – (Default: None) The lower variable bounds.
- **ub** (*list of float*) – (Default: None) The upper variable bounds.
- **vtype** (*str*) – (Default: None) A character string that specifies the type of each variable: ‘c’ontinous, ‘b’inary or ‘i’nteger
- **indic_constr** (IndicatorConstraints) – (Default: None) A set of indicator constraints stored in an object of IndicatorConstraints. To make GLPK compatible with indicator constraints, they are translated into bigM-constraints (see reference manual or docstring of IndicatorConstraints).
- **M** (*int*) – (Default: None) A large value that is used in the translation of indicator constraints to bigM-constraints. If no value is provided, 1000 is used.

- **Returns** – (GLPK_MILP_LP):

  A GLPK MILP/LP interface class.

addExclusionConstraintsIneq(*x*)

Function to add exclusion constraint (GLPK compatibility function)

add_eq_constraints(*A_eq*, *b_eq*)

Add equality constraints to the model

Additional equality constraints have the form A_eq * x = b_eq. The number of columns in A_eq must match with the number of variables x in the problem.

Parameters

- **A_eq** (*sparse.csr_matrix*) – The coefficient matrix
- **b_eq** (*list of float*) – The right hand side vector

add_ineq_constraints(*A_ineq*, *b_ineq*)

Add inequality constraints to the model

Additional inequality constraints have the form A_ineq * x <= b_ineq. The number of columns in A_ineq must match with the number of variables x in the problem.

Parameters

- **A_ineq** (*sparse.csr_matrix*) – The coefficient matrix
- **b_ineq** (*list of float*) – The right hand side vector

getSolution(*status*) → list

Retrieve solution from GLPK backend

populate(*pool_limit*) → Tuple[List, float, float]

Generate a solution pool for MILPs

This is only a high-level implementation of the populate function. There is no native support in GLPK.

Example

sols_x, optim, status = glpk.populate()

Returns

(Tuple[List of lists, float, float])

solution_vectors, optimal_value, optimization_status

set_ineq_constraint(*idx*, *a_ineq*, *b_ineq*)

Replace a specific inequality constraint

Replace the constraint with the index idx with the constraint a_ineq*x ~ b_ineq

Parameters

- **idx** (*int*) – Index of the constraint
- **a_ineq** (*list of float*) – The coefficient vector
- **b_ineq** (*float*) – The right hand side value

set_objective(*c*)

Set the objective function with a vector

set_objective_idx(*C*)

Set the objective function with index-value pairs

e.g.: C=[[1, 1.0], [4,-0.2]]

set_time_limit(*t*)

Set the computation time limit (in seconds)

set_ub(*ub*)

Set the upper bounds to a given vector

slim_solve() → float

Solve the MILP or LP, but return only the optimal value

Example

optim = glpk.slim_solve()

Returns

(float)

Optimum value of the objective function.

solve() → Tuple[List, float, float]

Solve the MILP or LP

Example

sol_x, optim, status = glpk.solve()

Returns

(Tuple[List, float, float])

solution_vector, optimal_value, optimization_status

solve_MILP_LP() → Tuple[float, int, bool]

Trigger GLPK solution through backend

straindesign.gurobi_interface

Gurobi solver interface for LP and MILP

Module Contents

class straindesign.gurobi_interface.**Gurobi_MILP_LP**(*c*, *A_ineq*, *b_ineq*, *A_eq*, *b_eq*, *lb*, *ub*, *vtype*, *indic_constr*)

Bases: gurobipy.Model

Gurobi interface for MILP and LP

This class is a wrapper for the Gurobi-Python API to offer bindings and namings for functions for the construction and manipulation of MILPs and LPs in an vector-matrix-based manner that are consistent with those of the other solver interfaces in the StrainDesign package. The purpose is to unify the instructions for operating with MILPs and LPs throughout StrainDesign.

The Gurobi interface provides support for indicator constraints as well as for the populate function.

Accepts a (mixed integer) linear problem in the form:
minimize(c), subject to: A_ineq * x <= b_ineq, A_eq * x = b_eq, lb <= x <= ub, forall(i) type(x_i) = vtype(i) (continous, binary, integer), indicator constraints: x(j) = [0|1] -> a_indic * x [<=|=|>=] b_indic

Please ensure that the number of variables and (in)equalities is consistent

Example

gurobi = Gurobi_MILP_LP(c, A_ineq, b_ineq, A_eq, b_eq, lb, ub, vtype, indic_constr)

Parameters

- **c** (*list of float*) – (Default: None) The objective vector (Objective sense: minimization).
- **A_ineq** (*sparse.csr_matrix*) – (Default: None) A coefficient matrix of the static inequalities.
- **b_ineq** (*list of float*) – (Default: None) The right hand side of the static inequalities.
- **A_eq** (*sparse.csr_matrix*) – (Default: None) A coefficient matrix of the static equalities.
- **b_eq** (*list of float*) – (Default: None) The right hand side of the static equalities.
- **lb** (*list of float*) – (Default: None) The lower variable bounds.
- **ub** (*list of float*) – (Default: None) The upper variable bounds.
- **vtype** (*str*) – (Default: None) A character string that specifies the type of each variable: 'c'ontinous, 'b'inary or 'i'nteger
- **indic_constr** (IndicatorConstraints) – (Default: None) A set of indicator constraints stored in an object of IndicatorConstraints (see reference manual or docstring).
- **Returns** – (Gurobi_MILP_LP):

  A Gurobi MILP/LP interface class.

add_eq_constraints(*A_eq*, *b_eq*)

Add equality constraints to the model

Additional equality constraints have the form A_eq * x = b_eq. The number of columns in A_eq must match with the number of variables x in the problem.

Parameters

- **A_eq** (*sparse.csr_matrix*) – The coefficient matrix
- **b_eq** (*list of float*) – The right hand side vector

add_ineq_constraints(*A_ineq*, *b_ineq*)

Add inequality constraints to the model

Additional inequality constraints have the form A_ineq * x <= b_ineq. The number of columns in A_ineq must match with the number of variables x in the problem.

Parameters

- **A_ineq** (*sparse.csr_matrix*) – The coefficient matrix
- **b_ineq** (*list of float*) – The right hand side vector

getSolution() → list

Retrieve solution from Gurobi backend

getSolutions() → list

Retrieve solution pool from Gurobi backend

populate(*n*) → Tuple[List, float, float]

Generate a solution pool for MILPs

Example

sols_x, optim, status = cplex.populate()

Returns

(Tuple[List of lists, float, float])

solution_vectors, optimal_value, optimization_status

set_ineq_constraint(*idx*, *a_ineq*, *b_ineq*)

Replace a specific inequality constraint

Replace the constraint with the index idx with the constraint a_ineq*x ~ b_ineq

Parameters

- **idx** (*int*) – Index of the constraint
- **a_ineq** (*list of float*) – The coefficient vector
- **b_ineq** (*float*) – The right hand side value

set_objective(*c*)

Set the objective function with a vector

set_objective_idx(*C*)

Set the objective function with index-value pairs

e.g.: C=[[1, 1.0], [4,-0.2]]

set_time_limit(*t*)

Set the computation time limit (in seconds)

set_ub(*ub*)

Set the upper bounds to a given vector

slim_solve() → float

Solve the MILP or LP, but return only the optimal value

Example

optim = gurobi.slim_solve()

Returns

(float)

Optimum value of the objective function.

solve() → Tuple[List, float, float]

Solve the MILP or LP

Example

sol_x, optim, status = gurobi.solve()

Returns

(Tuple[List, float, float])

solution_vector, optimal_value, optimization_status

straindesign.indicatorConstraints

Class for indicator contraints (IndicatorConstraints)

Module Contents

class straindesign.indicatorConstraints.**IndicatorConstraints**(*binv*, *A*, *b*, *sense*, *indicval*)

A class for storing indicator contraints

This class is a container for indicator constraints. Indicator constraints are used to link the fulfillment of a constraint to an indicating variable. For instance the indicator constraint: z = 1 -> 2*d - 1*e <= 3 can be described as: If z=1, then 2*d - 1*e <= 3 An alternative formulation of this association is possible with a bigM constraint: 2*d - 1*e - z*M <= 3, with M = very large The constraint z = 0 -> 2*d - 1*e <= 3 would translate to 2*d - 1*e + z*M <= 3 + M Generally, indicator constraints are preferred over bigM, because they provide better numerical stability. However, not all solvers support indicator constraints

Indicator constraints have the form: x_binv = indicval -> a * x <sense> b e.g.,: x_35 = 1 -> 2 * x_2 + 3 *x_3 'L' 6

(<=)

This class contains a set of indicator constraints: x_binv_1 = indicval_1 -> A_1 * x <sense_1> b_1 x_binv_2 = indicval_2 -> A_2 * x <sense_2> b_2 ...

Example

ic = IndicatorConstraints(binv, A, b, sense, indicval)

Parameters

- **binv** (*list of int*) – (e.g.: [25, 27, 30]) The index of the binary, indicating variables (indicators) for all indicator constraints. Integers are allowed ot occur more than once.
- **A** (*sparse.csr_matrix*) – Coefficient vectors for all indicator constraints, stored in one matrix, whereas each row is used in one indicator constraint. (num_columns = number of variables, num_rows = number of indicator constraints)
- **b** (*list of float*) – Right hand sides of all indicator constraints. (e.g.: [0.1, 2.0, 3])
- **sense** (*str*) – (In)equality signs for all indicator constraints. 'L'ess or equal, 'E'qual or, 'G'reater or equal (e.g.: 'EEGLGLGE')
- **indicval** (*list of int*) – Indicator values for all indicator constraints. Which value of the indicator enforces the constraint, 0 or 1? (e.g., 0001010110)

Returns

An object of the IndicatorConstraints class to pass indicator constraints.

Return type
(*IndicatorConstraints*)

straindesign.lptools

A collection of functions for the LP-based analysis of metabolic networks

Module Contents

straindesign.lptools.**ceil_dec**(*v*, *n*)

Round up v to n decimals

straindesign.lptools.**fba**(*model*, ***kwargs*) → cobra.core.Solution

Flux Balance Analysis (FBA), parsimonius Flux Balance Analysis (pFBA),

Flux Balance Analysis optimizes a *linear objective function* in a space of steady-state flux vectors given by a constraint-based metabolic model. FVA is often used to determine the (stoichiometrically) maximal possible growth rate, or flux rate towards a particular product. This FBA function allows to us a custom objective function and sense and allows the user to narrow down the flux states with additional constraints. In addition, one may use different types of parsimonious FBAs to either reduce the total sum of fluxes or the total number of active reactions after the primary objective is optimized.

Example

optim = fba(model, constraints='EX_o2_e=0', solver='gurobi', pfba=1)

Parameters

- **model** (*cobra.Model*) – A metabolic model that is an instance of the cobra.Model class. If no custom objective function is provided, the model's objective function is retrieved from the fields model.reactions[i].objective_coefficient.
- **solver** (*optional (str)*) – The solver that should be used for FBA.
- **constraints** (*optional (str) or (list of str) or (list of [dict,str, float])*) – (Default: '') List of *linear* constraints to be applied on top of the model: signs + or -, scalar factors for reaction rates, inclusive (in)equalities and a float value on the right hand side. The parsing of the constraints input allows for some flexibility. Correct (and identical) inputs are, for instance: constraints='-EX_o2_e <= 5, ATPM = 20' or constraints=['-EX_o2_e <= 5', 'ATPM = 20'] or constraints=[[{'EX_o2_e':-1},'<=',5], [{'ATPM':1},'=',20]]
- **obj** (*optional (str) or (dict)*) – As a custom objective function, any linear expression can be used, either provided as a single string or as a dict. Correct (and identical) inputs are, for instance: inner_objective='BIOMASS_Ecoli_core_w_GAM' inner_objective={'BIOMASS_Ecoli_core_w_GAM': 1}
- **obj_sense** (*optional (str)*) – (Default: 'maximize') The optimization direction can be set either to 'maximize' (or 'max') or 'minimize' (or 'min').
- **pfba** (*optional (int)*) – (Default: 0) The level of parsimonious FBA that should be applied. 0: no pFBA, only optmize the primary objective, 1: minimize sum of fluxes after the primary objective is optimized, 2: minimize the number of active reactions after the primary objective is optimized.

Returns
A solution object that contains the objective value, an optimal flux vector and the optmization status.

Return type
(cobra.core.Solution)

straindesign.lptools.**floor_dec**(*v*, *n*)

Round down v to n decimals

straindesign.lptools.**fva**(*model*, ***kwargs*) → pandas.DataFrame

Flux Variability Analysis (FVA)

Flux Variability Analysis determines the global flux ranges of reactions by minimizing and maximizing the flux through all reactions of a given metabolic network. This FVA function additionally allows the user to narrow down the flux states with additional constraints.

Example

flux_ranges = fva(model, constraints='EX_o2_e=0', solver='gurobi')

Parameters

- **model** (*cobra.Model*) – A metabolic model that is an instance of the cobra.Model class.
- **solver** (*optional (str)*) – The solver that should be used for FVA.
- **constraints** (*optional (str) or (list of str) or (list of [dict,str, float])*) – (Default: '') List of *linear* constraints to be applied on top of the model: signs + or -, scalar factors for reaction rates, inclusive (in)equalities and a float value on the right hand side. The parsing of the constraints input allows for some flexibility. Correct (and identical) inputs are, for instance: constraints='-EX_o2_e <= 5, ATPM = 20' or constraints=['-EX_o2_e <= 5', 'ATPM = 20'] or constraints=[[{'EX_o2_e':-1},'<=',5], [{'ATPM':1},'=',20]]

Returns
A data frame containing the minimum and maximum attainable flux rates for all reactions.

Return type
(pandas.DataFrame)

straindesign.lptools.**fva_worker_compute**(*i*) → Tuple[int, float]

Helper function for parallel FVA

Run a single LP as a step of FVA. Is executed on workers, not on main thread.

Parameters
i (*int*) – Index of the computation step.

straindesign.lptools.**fva_worker_compute_glpk**(*i*) → Tuple[int, float]

Helper function for parallel FVA

Run a single LP for GLPK as a step of FVA. Is executed on workers, not on main thread.

Parameters
i (*int*) – Index of the computation step.

straindesign.lptools.**fva_worker_init**(*A_ineq*, *b_ineq*, *A_eq*, *b_eq*, *lb*, *ub*, *solver*)

Helper function for parallel FVA

Initialize the LP that will be solved iteratively. Is executed on workers, not on main thread.

Parameters

- **A_ineq** – The LP.
- **b_ineq** – The LP.
- **A_eq** – The LP.
- **b_eq** – The LP.
- **lb** – The LP.
- **ub** – The LP.
- **solver** (*str*) – Solver to be used.

straindesign.lptools.**fva_worker_init_glpk**(*A_ineq*, *b_ineq*, *A_eq*, *b_eq*, *lb*, *ub*)

Helper function for parallel FVA

Initialize the LP for GLPK that will be solved iteratively. Is executed on workers, not on main thread.

Parameters

- **A_ineq** – The LP.
- **b_ineq** – The LP.
- **A_eq** – The LP.
- **b_eq** – The LP.
- **lb** – The LP.
- **ub** – The LP.

straindesign.lptools.**idx2c**(*i*, *prev*) → list

Helper function for parallel FVA

Builds the objective function for minimizing or maximizing the flux through the reaction with the index floor(i / 2). If i is even, there is a maximization.

Parameters

- **i** (*float*) – An index between 0 and 2*num_reacs.
- **prev** (*optional (str)*) – Index of the previously optimized reaction.

Returns
An optimization vector.

Return type
(list)

straindesign.lptools.**plot_flux_space**(*model*, *axes*, ***kwargs*) → Tuple[list, list, list]

Plot projections of the space of steady-state flux vectors onto two or three dimensions.

This function uses LP and matplotlib to generate lower dimensional representations of the flux space. Custom *linear* or *fractional-linear* expressions can be used for the plot axis. The most commonly used flux space reprentations are the *production envelope* that plots the growth rate (x) vs the product synthesis rate (y) and the *yield space plot* that plots the biomass yield (x) vs the product yiel (y). One may specify additional constraints to investigate subspaces of the metabolic steady-state flux space.

Example

plot_flux_space(model,('BIOMASS_Ecoli_core_w_GAM','EX_etoh_e'))

Parameters

- **model** (*cobra.Model*) – A metabolic model that is an instance of the cobra.Model class.
- **axes** (*(list of lists) or (list of str)*) – A set of linear expressions that specify which reactions/expressions/dimensions should be used on the axis. Examples: axes=['BIOMASS_Ecoli_core_w_GAM','EX_etoh_e'] or axes=[['BIOMASS_Ecoli_core_w_GAM','-EX_glc_e'],['EX_etoh_e','-EX_glc_e']] or axes=[['BIOMASS_Ecoli_core_w_GAM'],['EX_etoh_e','-EX_glc_e']]
- **solver** (*optional (str)*) – The solver that should be used for scanning the flux space.
- **constraints** (*optional (str) or (list of str) or (list of [dict,str, float])*) – (Default: '') List of *linear* constraints to be applied on top of the model: signs + or -, scalar factors for reaction rates, inclusive (in)equalities and a float value on the right hand side. The parsing of the constraints input allows for some flexibility. Correct (and identical) inputs are, for instance: constraints='-EX_o2_e <= 5, ATPM = 20' or constraints=['-EX_o2_e <= 5', 'ATPM = 20'] or constraints=[[{'EX_o2_e':-1},'<=',5], [{'ATPM':1},'=',20]]
- **plt_backend** (*optional (str)*) – The matplotlib backend that should be used for plotting: interactive backends: GTK3Agg, GTK3Cairo, GTK4Agg, GTK4Cairo, MacOSX, nbAgg, QtAgg, QtCairo,

  TkAgg, TkCairo, WebAgg, WX, WXAgg, WXCairo, Qt5Agg, Qt5Cairo

  non-interactive backends: agg, cairo, pdf, pgf, ps, svg, template
- **show** (*optional (bool)*) – (Default: True) Should matplotlib show the plot or should it stop after plot generation. show=False can be useful if multiple flux spaces should be plotted at once or the plot should be modified before been shown.
- **points** (*optional (int)*) – (Default: 25 (3D) or 40 (2D)) The number of intervals in which the flux space should be sampled along each axis. A higher number will increase resoltion but also computation time.

Returns

(datapoints, triang, plot1). The array of datapoints from which the plot was generated. These datapoints are optimal values for different optimizations within the flux space. The triang variable contains information about which datapoints need to be connected in triangles to render a consistend 3-D surface with Delaunay triangles (Delaunay triangulation). The last variable contains the matplotlib object.

Return type

(Tuple)

straindesign.lptools.**select_solver**(*solver=None*, *model=None*) → str

Select a solver for subsequent MILP/LP computations

This function will determine the solver to be used for subsequend MILP/LP computations. If no argument is provided, this function will try to determine the currently selected solver from the COBRA configuration. If unavailable, the solver will be inferred from the packages available at package initialization and one of the solvers will be picked and retured in the prioritized order: 'glpk', 'cplex', 'gurobi', 'scip' One may provide a solver or a model manually. This function then checks if the selected solver is available, or else, if the solver indicated in the model is available. If yes, this function returns the name of the solver as a str. If both arguments are specified, the function prefers 'solver' over 'model'.

Example

solver = select_solver('cplex')

Parameters

- **solver** (*optional (str)*) – A user preferred solver, that should be checked for availability: 'glpk', 'cplex', 'gurobi' or 'scip'.
- **model** (*optional (cobra.Model)*) – A metabolic model that is an instance of the cobra.Model class. The function will try to dertermine the selected solver by accessing the field model.solver.

Returns
The selected solver name as a str (one of the following: 'glpk', 'cplex', 'gurobi', 'scip').

Return type
(str)

straindesign.lptools.**yopt**(*model*, ***kwargs*) → cobra.core.Solution

Yield optmization (YOpt)

Yield optimization optimizes a *fractional objective function* in a space of steady-state flux vectors given by a constraint-based metabolic model. Yield optimization employs linear fractional programming, and is often utilized to determine the (stoichiometrically) maximal possible product yield, that is, the fraction between the product exchange and the substrate uptake flux. This function requires a custom fractional objective function specified by a *linear* numerator and denominator terms. Coefficients in the linear numerator or denominator expression can be used to optimize for carbon recovery, for instance: objective: (3*pyruvate_ex)/(2*ac_up+6*glc_up) The user may also specify the optimization sense. In addition, additional constraints can be specified to narrow down the flux space.

Yield optimization can fail because of several reasons. Here is how the function reacts:

1. **The model is infeasible:**
   The function returns infeasible with no flux vector
2. **The denominator is fixed to zero:**
   The function returns infeasible with no flux vector
3. **The numerator is unbounded:**
   The function returns unbounded with no flux vector
4. **The denominator can become zero:**
   The function returns unbounded, and a flux vector is computed by fixing the the denominator

Example

optim = yopt(model, obj_num='2 EX_etoh_e', obj_den='-6 EX_glc__D_e', constraints='EX_o2_e=0')

Parameters

- **model** (*cobra.Model*) – A metabolic model that is an instance of the cobra.Model class.
- **obj_num** ((*str*) *or* (*dict*)) – The numerator of the fractional objective function, provided as a linear expression, either as a character string or as a dict. E.g.: obj_num='EX_prod_e' or obj_num={'EX_prod_e': 1}
- **obj_den** ((*str*) *or* (*dict*)) – The denominator of the fractional objective function, provided as a linear expression, either as a character string or as a dict. E.g.: obj_num='1.0 EX_subst_e' or obj_num={'EX_subst_e': 1}

- **obj_sense** (*optional (str)*) – (Default: 'maximize') The optimization direction can be set either to 'maximize' (or 'max') or 'minimize' (or 'min').
- **solver** (*optional (str)*) – The solver that should be used for YOpt.
- **constraints** (*optional (str) or (list of str) or (list of [dict,str, float])*) – (Default: '') List of *linear* constraints to be applied on top of the model: signs + or -, scalar factors for reaction rates, inclusive (in)equalities and a float value on the right hand side. The parsing of the constraints input allows for some flexibility. Correct (and identical) inputs are, for instance: constraints='-EX_o2_e <= 5, ATPM = 20' or constraints=['-EX_o2_e <= 5', 'ATPM = 20'] or constraints=[[{'EX_o2_e':-1},'<=',5], [{'ATPM':1},'=',20]]

Returns
A solution object that contains the objective value, an optimal flux vector and the optmization status.

Return type
(cobra.core.Solution)

straindesign.names

Static strings used in the StrainDesign package

Model and module

MODEL_ID = 'model_id'

PROTECT = 'mcs_lin'

SUPPRESS = 'mcs_bilvl'

SUPPRESS = 'suppress'

PROTECT = 'protect'

OPTKNOCK = 'optknock'

ROBUSTKNOCK = 'robustknock'

OPTCOUPLE = 'optcouple'

MODULE_TYPE = 'module_type'

CONSTRAINTS = 'constraints'

INNER_OBJECTIVE = 'inner_objective'

INNER_OPT_SENSE = 'inner_opt_sense'

OUTER_OBJECTIVE = 'outer_objective'

OUTER_OPT_SENSE = 'outer_opt_sense'

PROD_ID = 'prod_id'

MIN_GCP = 'min_gcp'

Solvers and status codes

SOLVER = 'solver'

CPLEX = 'cplex'

GUROBI = 'gurobi'

SCIP = ‘scip’

GLPK = ‘glpk’

OPTIMAL = ‘optimal’ # from optlang interface

INFEASIBLE =’infeasible’ # from optlang interface

TIME_LIMIT = ‘time_limit’ # from optlang interface

UNBOUNDED = ‘unbounded’ # from optlang interface

TIME_LIMIT_W_SOL = ‘time_limit_w_sols’

ERROR = ‘error’

Strain design setup

KOCOST = ‘ko_cost’

KICOST = ‘ki_cost’

GKOCOST = ‘gko_cost’

GKICOST = ‘gki_cost’

REGCOST = ‘reg_cost’

MODULES = ‘sd_modules’

SETUP = ‘sd_setup’

MAX_SOLUTIONS = ‘max_solutions’

MAX_COST = ‘max_cost’

T_LIMIT = ‘time_limit’

SOLUTION_APPROACH = ‘solution_approach’

ANY = ‘any’

BEST = ‘best’

POPULATE = ‘populate’

Analysis

MAXIMIZE = ‘maximize’

MINIMIZE = ‘minimize’

straindesign.networktools

Functions for metabolic network compression and extension with GPR rules

Module Contents

straindesign.networktools.**bound_blocked_or_irrevers_fva**(*model*, *solver=None*)

Use FVA to determine the flux ranges. Use this information to update the model bounds

If flux ranges for a reaction are narrower than its bounds in the mode, these bounds can be omitted, since other reactions must constrain the reaction flux. If (upper or lower) flux bounds are found to be zero, the model bounds are updated to reduce the model complexity.

straindesign.networktools.**compress_ki_ko_cost**(*kocost*, *kicost*, *cmp_mapReac*)

Compress knockout/addition cost vectors to match with a compressed model

When knockout/addition cost vectors have been specified (as dicts) and the original metabolic model was compressed, one needs to update the knockout/addition cost vectors. This function takes care of this. In particular it makes sure that the resulting costs are calculated correctly.

E.g.: r_ko_a (cost 1) and r_ko_b (cost 2) are lumped parallel: The resulting cost of r_ko_ab is 3 If they are lumped as dependent reactions the resulting cost is 1. If one of the two reactions is an addition candidate, the resulting reaction will be an addition candidate when lumped as dependent reactions and a knockout candidate when lumped in parallel. There are various possible cases that are treated by this function.

Example

kocost, kicost, cmp_mapReac = compress_ki_ko_cost(kocost, kicost, cmp_mapReac)

Parameters

- **kocost** (*dict*) – Knockout and addition cost vectors
- **kicost** (*dict*) – Knockout and addition cost vectors
- **cmp_mapReac** (*list of dicts*) – Compression map obtained from cmp_mapReac = compress_model(model)

Returns

Updated vectors of KO costs and KI costs and an updated compression map that contains information on how to expand strain designs and correctly distinguish between knockouts and additions.

Return type

(Tuple)

straindesign.networktools.**compress_model**(*model*, *no_par_compress_reacs=set()*)

Compress a metabolic model with a number of different techniques

The network compression routine removes blocked reactions, removes conservation relations and then performs alternatingly lumps dependent (compress_model_efmtool) and parallel (compress_model_parallel) reactions. The compression returns a compressed network and a list of compression maps. Each map consists of a dictionary that contains complete information for reversing the compression steps successively and expand information obtained from the compressed model to the full model. Each entry of each map contains the id of a compressed reaction, associated with the original reaction names and their factor (provided as a rational number) with which they were lumped.

Furthermore, the user can select reactions that should be exempt from the parallel compression. This is a critical feature for strain design computations. There is currently no way to exempt reactions from the efmtool/dependency compression.

Example

comression_map = compress_model(model,set('EX_etoh_e','PFL'))

Parameters

- **model** (*cobra.Model*) – A metabolic model that is an instance of the cobra.Model class
- **no_par_compress_reacs** (*set or list of str*) – (Default: set()) A set of reaction identifiers whose reactions should not be lumped with other parallel reactions.

Returns

A list of compression maps. Each map is a dict that contains information for reversing the compression steps successively and expand information obtained from the compressed model to the full model. Each entry of each map contains the id of a compressed reaction, associated with the original reaction identifiers and their factor with which they are represented in the lumped reaction (provided as a rational number) with which they were lumped.

Return type

(list of dict)

straindesign.networktools.**compress_model_efmtool**(*model*)

Compress model by lumping dependent reactions using the efmtool compression approach

Example

cmp_mapReac = compress_model_efmtool(model)

Parameters

model (*cobra.Model*) – A metabolic model that is an instance of the cobra.Model class

Returns

A dict that contains information about the lumping done in the compression process. process. E.g.: {'reaction_lumped1' : {'reaction_orig1' : 2/3 'reaction_orig2' : 1/2}, ... }

Return type

(dict)

straindesign.networktools.**compress_model_parallel**(*model*, *protected_rxns=set()*)

Compress model by lumping parallel reactions

Example

cmp_mapReac = compress_model_parallel(model)

Parameters

model (*cobra.Model*) – A metabolic model that is an instance of the cobra.Model class

Returns

A dict that contains information about the lumping done in the compression process. E.g.: {'reaction_lumped1' : {'reaction_orig1' : 1 'reaction_orig2' : 1}, ... }

Return type

(dict)

straindesign.networktools.**compress_modules**(*sd_modules*, *cmp_mapReac*)

Compress strain design modules to match with a compressed model

When a strain design task has been specified with modules and the original metabolic model was compressed, one needs to refit the strain design modules (objects of the SDModule class) to the new compressed model. This function takes a list of modules and a compression map and returns the strain design modules for a compressed network.

Example

comression_map = compress_modules(sd_modules, cmp_mapReac)

Parameters

- **model** (*list of* SDModule) – A list of strain design modules
- **cmp_mapReac** (*list of dicts*) – Compression map obtained from cmp_mapReac = compress_model(model)

Returns
A list of strain design modules for the compressed network

Return type
(list of *SDModule*)

straindesign.networktools.**expand_sd**(*sd*, *cmp_mapReac*)

Expand computed strain designs from a compressed to a full model

Needed after computing strain designs in a compressed model

Example

expanded_sd = expand_sd(compressed_sds, cmp_mapReac)

Parameters

- **sd** (SDSolutions) – Solutions of a strain design computation that refer to a compressed model
- **cmp_mapReac** (*list of dicts*) – Compression map obtained from cmp_mapReac = compress_model(model) and updated with kocost, kicost, cmp_mapReac = compress_ki_ko_cost(kocost, kicost, cmp_mapReac)

Returns
Strain design solutions that refer to the uncompressed model

Return type
(*SDSolutions*)

straindesign.networktools.**extend_model_gpr**(*model*, *use_names=False*)

Integrate GPR-rules into a metabolic model as pseudo metabolites and reactions

COBRA modules often have gene-protein-reaction (GPR) rules associated with each reaction. These can be integrated into the metabolic network structure through pseudo reactions and variables. As GPR rules are integrated into the metabolic network, the metabolic flux space does not change. After integration, the gene-pseudoreactions can be fixed to a flux of zero to simulate gene knockouts. Gene pseudoreactions are referenced either by the gene name or the gene identifier (user selected).

GPR-rule integration enables the computation of strain designs based on genetic interventions.

This function requires all GPR-rules to be provided in DNF (disjunctive normal form). (e.g. g1 and g2 or g1 and g3, NOT g1 and (g2 or g3)). Brackets are allowed but not required. If reversible reactions are associated with GPR-rules, these reactions are spit during GPR-integration. The function returns a mapping of old and new reaction identifiers.

Example

reac_map = extend_model_gpr(model):

Parameters

- **model** (*cobra.Model*) – A metabolic model that is an instance of the cobra.Model class containing GPR rules in DNF
- **use_names** (*bool*) – (Default: False) If set to True, the gene pseudoreactions will carry the gene name as reaction identifier. If False, the gene identifier will be used. By default this option is turned off because many models do not provide gene names.

Returns

A dictionary to reference old and new reaction identifiers, for reversible reactions that were split (when they are associated with GPR rules). Entries have the form: {'Reaction1' : {'Reaction1' : 1, 'Reaction1_reverse_a59c' : -1}}

Return type

(dict)

straindesign.networktools.**extend_model_regulatory**(*model*, *reg_itv*)

Extend a metabolic model to account for regulatory constraints

This function emulates regulatory interventions in a network. These can either be added permanently or linked to a pseudoreation whose boundaries can be fixed to zero used to activate the regulatory constraint.

Accounting for regulatory interventions, such as applying an upper or lower bound to a reaction or gene pseudoreaction, can be achieved by combining different pseudometabolites and reactions. For instance, to introduce the regulatory constraint:

2*r_1 + 3*r_2 <= 4

and make it 'toggleable', one adds 1 metabolite 'm' and 2 reactions, 'r_bnd' to account for the bound/rhs and r_ctl to control whether the regulatory intervention is active or not:

dm/dt = 2*r_1 + 3*r_2 - r_bnd + r_ctl = 0, -inf <= r_bnd <= 4, -inf <= r_ctl <= inf

When r_ctl is fixed to zero, the constraint 2*r_1 + 3*r_2 <= 4 is enforced, otherwise, the constraint is non binding, thus virtually non-existant. To use this mechanism for strain design, we add the metabolite and reactions as described above and tag r_ctl as knockout candidate. If the algorithm decides to knockout r_ctl, this means, it choses to add the regulatory intervention 2*r_1 + 3*r_2 <= 4.

If the constraint is be added permanently, this function completely omits the r_ctl reaction.

Example

reg_itv_costs = extend_model_regulatory(model, {'1 PDH + 1 PFL <= 5' : 1, '-EX_o2_e <= 2' : 1.5})

Parameters

- **model** (*cobra.Model*) – A metabolic model that is an instance of the cobra.Model class
- **reg_itv** (*dict or list of str or str*) – A set of regulatory constraints that should be added to the model. If reg_itv is a string or a list of strings, regulatory constraints are added permanently. If reg_itv is a dict, regulatory interventions are added in a controllable manner. The id of the reaction that controls the constraint is contained in the return variable. The constraint to be added will be parsed from strings, so ensure that you use the correct reaction identifiers. Valid inputs are: reg_itv = '-EX_o2_e <= 2' # A single permanent regulatory constraint reg_itv = ['1 PDH + 1 PFL <= 5', '-EX_o2_e <= 2'] # Two permanent constraints reg_itv = {'1 PDH + 1 PFL <= 5' : 1, '-EX_o2_e <= 2' : 1.5} # Two controllable constraints # one costs '1' and the other one '1.5' to be added. The function returns a dict with # {'p1_PDH_p1_PFK_le_5' : 1 'nEX_o2_e_le_2' : 1.5}. Fixing the reaction # p1_PDH_p1_PFK_le_5 to zero will activate the constraint in the model.

Returns
A dictionary that contains the cost of adding each constraint; e.g., {'p1_PDH_p1_PFK_le_5' : 1 'n1EX_o2_e_le_2' : 1.5}

Return type
(dict)

straindesign.networktools.**filter_sd_maxcost**(*sd*, *max_cost*, *kocost*, *kicost*)

Filter out strain designs that exceed the maximum allowed intervention costs

Returns
Strain design solutions complying with the intervention costs limit

Return type
(*SDSolutions*)

straindesign.networktools.**modules_coeff2float**(*sd_modules*)

Convert coefficients occurring in SDModule objects to floats

straindesign.networktools.**modules_coeff2rational**(*sd_modules*)

Convert coefficients occurring in SDModule objects to rational numbers

straindesign.networktools.**remove_blocked_reactions**(*model*) → List

Remove blocked reactions from a network

straindesign.networktools.**remove_conservation_relations**(*model*)

Remove conservation relations in a model

This reduces the number of metabolites in a model while maintaining the original flux space. This is a compression technique.

straindesign.networktools.**remove_dummy_bounds**(*model*)

Replace COBRA standard bounds with +/-inf

Retrieve the standard bounds from the COBRApy Configuration and replace model bounds of the same value with +/-inf.

straindesign.networktools.**remove_ext_mets**(*model*)

Remove (unbalanced) external metabolites from the compartment External_Species

straindesign.networktools.**remove_irrelevant_genes**(*model*, *essential_reacs*, *gkis*, *gkos*)

Remove genes whose that do not affect the flux space of the model

This function is used in preprocessing of computational strain design computations. Often, certain reactions, for instance, reactions essential for microbial growth can/must not be targeted by interventions. That can be exploited to reduce the set of genes in which interventions need to be considered.

Given a set of essential reactions that is to be maintained operational, some genes can be removed from a metabolic model, either because they only affect only blocked reactions or essential reactions, or because they are essential reactions and must not be removed. As a consequence, the GPR rules of a model can be simplified.

Example

remove_irrelevant_genes(model, essential_reacs, gkis, gkos):

Parameters

- **model** (*cobra.Model*) – A metabolic model that is an instance of the cobra.Model class containing GPR rules
- **essential_reacs** (*list of str*) – A list of identifiers of essential reactions.
- **gkis** (*dict*) – Dictionaries that contain the costs for gene knockouts and additions. E.g., gkos={'adhE': 1.0, 'ldhA' : 1.0 . . . }
- **gkos** (*dict*) – Dictionaries that contain the costs for gene knockouts and additions. E.g., gkos={'adhE': 1.0, 'ldhA' : 1.0 . . . }

Returns
An updated dictionary of the knockout costs in which irrelevant genes are removed.

Return type
(dict)

straindesign.networktools.**stoichmat_coeff2float**(*model*)

Convert coefficients from to stoichiometric matrix to floats

straindesign.networktools.**stoichmat_coeff2rational**(*model*)

Convert coefficients from to stoichiometric matrix to rational numbers

straindesign.parse_constr

Functions for parsing and converting constraints and linear expressions

Module Contents

straindesign.parse_constr.**get_rids**(*expr*, *reaction_ids*)

Get reaction identifiers that are present in string

E.g.: input: D={'R1':-1.0, 'R3': 2.0}, translates to the string: '- 1.0 R1 + 2.0 R3'

Parameters

- **expr** (*str*) – A character string
- **reaction_ids** (*list of str*) – List of reaction identifiers or variable names

Returns
A list of strings containing the reaction/variable strings present in the input string

Return type
(list of str)

straindesign.parse_constr.**lineq2list**(*equations*, *reaction_ids*) → List

Translates *linear* (in)equalities to list format: [lhs,sign,rhs]

Input inequalities in the form of strings are translated into a specific list format that facilitates the readout of left-hand-side, equality sign and right-hand-side of the inequality.

equations = ['2*c - b +3*a <= 2','c - b = 0','2*b -5...], reaction_ids = ['a','b','c']

This will be translated to the [[{'a':3.0,'b':-1.0,'c':2.0},'<=',2.0],[{'b':-1.0,'c':1.0},'=',0.0], ...]

Parameters

- **equations** (*list of str*) – (List of) (in)equalities in string form equations=['r1 + 3*r2 = 0.3', '-5*r3 -r4 <= -0.5']
- **reaction_ids** (*list of str*) – List of reaction identifiers or variable names that are used to recognize variables in the provided (in)equalities

Returns

(In)equalities presented in the form: [[{'a':3.0,'b':-1.0,'c':2.0},'<=',2.0], # e1

[{'b':-1.0,'c':1.0},'=',0.0], # e2 ...] # ...

Return type
(list of lists)

straindesign.parse_constr.**lineq2mat**(*equations*, *reaction_ids*) → Tuple[scipy.sparse.csr_matrix, Tuple, scipy.sparse.csr_matrix, Tuple]

Translates *linear* (in)equalities to matrices

Input inequalities in the form of strings is translated into matrices and vectors. The reaction list defines the order of variables and thus the columns of the resulting matrices, the order of (in)equalities will be preserved in the output matrices. As an example, take the input:

equations = ['2*c - b +3*a <= 2','c - b = 0','2*b -a >=-2'], reaction_ids = ['a','b','c']

This will be translated to the form A_ineq * x <= b_ineq, A_eq * x = b_eq and hence to

A_ineq = sparse.csr_matrix([[3,-1,2],[1,-2,0]]), b_ineq = [2,2], A_eq = sparse.csr_matrix([[1,-2,0]]), b_eq = [0]

Parameters

- **equations** (*list of str*) – (List of) (in)equalities in string form equations=['r1 + 3*r2 = 0.3', '-5*r3 -r4 <= -0.5']
- **reaction_ids** (*list of str*) – List of reaction identifiers or variable names that are used to recognize variables in the provided (in)equalities

Returns
A_ineq, b_ineq, A_eq, b_eq. Coefficient matrices and right hand sides that represent the input (in)equalities as matrix-vector multiplications

Return type
(Tuple)

straindesign.parse_constr.**lineqlist2mat**(*D*, *reaction_ids*) → Tuple[scipy.sparse.csr_matrix, Tuple, scipy.sparse.csr_matrix, Tuple]

Translates *linear* (in)equalities presented in the list of lists format to matrices

Input inequalities in the list of lists form is translated into matrices and vectors. The reaction list defines the order of variables and thus the columns of the resulting matrices, the order of (in)equalities will be preserved in the output matrices. As an example, take the input:

D = [[{'a':3.0,'b':-1.0,'c':2.0},'<=',2.0],[{'b':-1.0,'c':1.0},'=',0.0], [{'a':-1,'b':2.0},'>=',-2.0]]

This will be translated to the form A_ineq * x <= b_ineq, A_eq * x = b_eq and hence to

A_ineq = sparse.csr_matrix([[3,-1,2],[1,-2,0]]), b_ineq = [2,2], A_eq = sparse.csr_matrix([[1,-2,0]]), b_eq = [0]

Parameters

- **D** (*list of dict*) – (List of) (in)equalities in the list of list form: [[{'a':3.0,'b':-1.0,'c':2.0},'<=',2.0],[{'b':-1.0,'c':1.0},'=',0.0], ...]
- **reaction_ids** (*list of str*) – List of reaction identifiers or variable names that are used to recognize variables in the provided (in)equalities

Returns

A_ineq, b_ineq, A_eq, b_eq. Coefficient matrices and right hand sides that represent the input (in)equalities as matrix-vector multiplications

Return type

(Tuple)

straindesign.parse_constr.**lineqlist2str**(*D*)

Translates a *linear* (in)equality from the list format [lhs,sign,rhs] to a string

E.g. input: D=[{'a':3.0,'b':-1.0,'c':2.0},'<=',2.0]] is translated to: out='3.0 a - 1.0 b + 2.0 c <= 2'

Parameters

D (*list*) – (In)equality in list form, e.g.: D=[{'a':3.0,'b':-1.0,'c':2.0},'<=',2.0]]

Returns

A list of (in)equalities in string form

Return type

(str)

straindesign.parse_constr.**linexpr2dict**(*expr*, *reaction_ids*) → dict

Translates a linear expression into a dictionary

E.g.: input: expr='2 R3 - R1', reaction_ids=['R1', 'R2', 'R3', 'R4'] translates to a dict D={'R1':-1.0, 'R3': 2.0}

Parameters

- **expr** (*str*) – (In)equalities as a character string, e.g.: expr='2 R3 - R1'
- **reaction_ids** (*list of str*) – List of reaction identifiers or variable names that are used to recognize variables in the input

Returns

A dictionary that contains the variable names and the variable coefficients in the linear expression

Return type

(dict)

straindesign.parse_constr.**linexpr2mat**(*expr*, *reaction_ids*) → scipy.sparse.csr_matrix

Translates a linear expression into a vector

E.g.: input: expr='2 R3 - R1', reaction_ids=['R1', 'R2', 'R3', 'R4'] translates into sparse matrix: A = [-1 0 2 0]

Parameters

- **expr** (*str*) – (In)equality as a character string: e.g., expr='2 R3 - R1'
- **reaction_ids** (*list of str*) – List of reaction identifiers or variable names that are used to recognize variables in the input

Returns
A single-row coefficient matrix that represents the input expression when multiplied with the variable vector

Return type
(sparse.csr_matrix)

straindesign.parse_constr.**linexprdict2mat**(*D*, *reaction_ids*) → scipy.sparse.csr_matrix

Translates a linear expression from dict into a matrix

E.g.: input: D={'R1':-1.0, 'R3': 2.0}, reaction_ids=['R1', 'R2', 'R3', 'R4'] translates into sparse matrix: A = [-1 0 2 0]

Parameters

- **D** (*dict*) – Linear expression as a dictionary
- **reaction_ids** (*list of str*) – List of reaction identifiers or variable names

Returns
A single-row coefficient matrix that represents the input expression when multiplied with the variable vector

Return type
(sparse.csr_matrix)

straindesign.parse_constr.**linexprdict2str**(*D*)

Translates a linear expression from dict into a caracter string

E.g.: input: D={'R1':-1.0, 'R3': 2.0}, translates to the string: '- 1.0 R1 + 2.0 R3'

Parameters
D (*dict*) – Linear expression as a dictionary

Returns
The input linear expression as a character string

Return type
(str)

straindesign.parse_constr.**parse_constraints**(*constr*, *reaction_ids*) → list

Parses linear constraints written as strings

Parses one or more *linear* constraints written as strings.

Parameters

- **constr** (*str or list of str*) – (List of) constraints in string form. E.g.: ['r1 + 3*r2 = 0.3', '-5*r3 -r4 <= -0.5'] or '1.0 r1 + 3.0*r2 =0.3,-r4-5*r3<=-0.5' or ...
- **reaction_ids** (*list of str*) – List of reaction identifiers.

Returns
List of constraints. Each constraint is a list of three elements. E.g.: [[{'r1':1.0,'r2':3.0},'=',0.3],[{'r3':-5.0,'r4':-1.0},'<=',-0.5],...]

Return type
(List of dicts)

straindesign.parse_constr.**parse_linexpr**(*expr*, *reaction_ids*) → List

Parses linear expressions written as strings

Parses one or more *linear* expressions written as strings.

Parameters

- **expr** (*str or list of str*) – (List of) expressions in string form. E.g.: ['r1 + 3*r2', '-5*r3 -r4'] or '1.0 r1 + 3.0*r2,-r4-5*r3' or ...
- **reaction_ids** (*list of str*) – List of reaction identifiers.

Returns
List of expressions. Each expression is a dictionary. E.g.: [{'r1':1.0,'r2':3.0},{'r3':-5.0,'r4':-1.0},...]

Return type
(List of dicts)

straindesign.pool

Provide a process pool with enhanced performance on Windows, copied and slightly adapted from cobra.

Module Contents

class straindesign.pool.**SDPool**(*processes: int | None = None*, *initializer: Callable | None = None*, *initargs: Tuple = ()*, *maxtasksperchild: int | None = None*, *context=None*)

Bases: multiprocessing.pool.Pool

Multiprocessing process pool with enhanced Windows compatibility

Initialize a process pool.

Add a thin layer on top of the *multiprocessing.Pool* that, on Windows, passes initialization code to workers via a pickle file rather than directly. This is done to avoid a performance issue that exists on Windows. Please, also see the discussion [*1*].

References

__exit__(**args*, ***kwargs*)

Clean up resources when leaving a context

close()

Call cleanup function and close

straindesign.scip_interface

SCIP and SoPlex solver interface for LP and MILP

Module Contents

class straindesign.scip_interface.**SCIP_LP**(*c*, *A_ineq*, *b_ineq*, *A_eq*, *b_eq*, *lb*, *ub*)

Bases: pyscipopt.LP

SoPlex interface for LP

This class is a wrapper for the SoPlex-Python API to offer bindings and namings for functions for the construction and manipulation of LPs in an vector-matrix-based manner that are consistent with those of the other solver interfaces in the StrainDesign package. The purpose is to unify the instructions for operating with MILPs and LPs throughout StrainDesign.

Constructor of the SCIP (SoPlex) LP interface class

Accepts a (mixed integer) linear problem in the form:

minimize(c) subject to: A_ineq * x <= b_ineq

A_eq * x = b_eq lb <= x <= ub forall(i) type(x_i) = vtype(i) (continous, binary, integer) indicator constraints: x(j) = [0|1] -> a_indic * x [<=|=|>=] b_indic

Please ensure that the number of variables and (in)equalities is consistent

Example

scip = SCIP_LP(c, A_ineq, b_ineq, A_eq, b_eq, lb, ub)

Parameters

- **c** (*list of float*) – (Default: None) The objective vector (Objective sense: minimization).
- **A_ineq** (*sparse.csr_matrix*) – (Default: None) A coefficient matrix of the static inequalities.
- **b_ineq** (*list of float*) – (Default: None) The right hand side of the static inequalities.
- **A_eq** (*sparse.csr_matrix*) – (Default: None) A coefficient matrix of the static equalities.
- **b_eq** (*list of float*) – (Default: None) The right hand side of the static equalities.
- **lb** (*list of float*) – (Default: None) The lower variable bounds.
- **ub** (*list of float*) – (Default: None) The upper variable bounds.
- **Returns** – (SCIP_LP):

  A SCIP LP interface class.

add_eq_constraints(*A_eq*, *b_eq*)

Add equality constraints to the model

Additional equality constraints have the form A_eq * x = b_eq. The number of columns in A_eq must match with the number of variables x in the problem.

Parameters

- **A_eq** (*sparse.csr_matrix*) – The coefficient matrix

- **b_eq** (*list of float*) – The right hand side vector

add_ineq_constraints(*A_ineq*, *b_ineq*)

Add inequality constraints to the model

Additional inequality constraints have the form A_ineq * x <= b_ineq. The number of columns in A_ineq must match with the number of variables x in the problem.

Parameters

- **A_ineq** (*sparse.csr_matrix*) – The coefficient matrix
- **b_ineq** (*list of float*) – The right hand side vector

set_objective(*c*)

Set the objective function with a vector

set_objective_idx(*C*)

Set the objective function with index-value pairs

e.g.: C=[[1, 1.0], [4,-0.2]]

slim_solve() → float

Solve the LP, but return only the optimal value

Example

optim = scip.slim_solve()

Returns

(float)

Optimum value of the objective function.

solve() → Tuple[List, float, float]

Solve the LP

Example

sol_x, optim, status = scip.solve()

Returns

(Tuple[List, float, float])

solution_vector, optimal_value, optimization_status

class straindesign.scip_interface.**SCIP_MILP**(*c*, *A_ineq*, *b_ineq*, *A_eq*, *b_eq*, *lb*, *ub*, *vtype*, *indic_constr*)

Bases: pyscipopt.Model

SCIP interface for MILP

This class is a wrapper for the SCIP-Python API to offer bindings and namings for functions for the construction and manipulation of MILPs in an vector-matrix-based manner that are consistent with those of the other solver interfaces in the StrainDesign package. The purpose is to unify the instructions for operating with MILPs and LPs throughout StrainDesign.

The SCIP interface provides support for indicator constraints as well as for the populate function. The SCIP interface does not natively support the populate function. A high level implementation emulates the behavior of populate.

Accepts a (mixed integer) linear problem in the form:
minimize(c), subject to: A_ineq * x <= b_ineq, A_eq * x = b_eq, lb <= x <= ub, forall(i) type(x_i) = vtype(i) (continous, binary, integer), indicator constraints: x(j) = [0|1] -> a_indic * x [<=|=|>=] b_indic

Please ensure that the number of variables and (in)equalities is consistent

Example

scip = SCIP_MILP(c, A_ineq, b_ineq, A_eq, b_eq, lb, ub, vtype, indic_constr)

Parameters

- **c** (*list of float*) – (Default: None) The objective vector (Objective sense: minimization).
- **A_ineq** (*sparse.csr_matrix*) – (Default: None) A coefficient matrix of the static inequalities.
- **b_ineq** (*list of float*) – (Default: None) The right hand side of the static inequalities.
- **A_eq** (*sparse.csr_matrix*) – (Default: None) A coefficient matrix of the static equalities.
- **b_eq** (*list of float*) – (Default: None) The right hand side of the static equalities.
- **lb** (*list of float*) – (Default: None) The lower variable bounds.
- **ub** (*list of float*) – (Default: None) The upper variable bounds.
- **vtype** (*str*) – (Default: None) A character string that specifies the type of each variable: 'c'ontinous, 'b'inary or 'i'nteger
- **indic_constr** (IndicatorConstraints) – (Default: None) A set of indicator constraints stored in an object of IndicatorConstraints (see reference manual or docstring).
- **Returns** – (SCIP_MILP):

  A SCIP MILP interface class.

addExclusionConstraintIneq(*x*)

Function to add exclusion constraint (SCIP compatibility function)

add_eq_constraints(*A_eq*, *b_eq*)

Add equality constraints to the model

Additional equality constraints have the form A_eq * x = b_eq. The number of columns in A_eq must match with the number of variables x in the problem.

Parameters

- **A_eq** (*sparse.csr_matrix*) – The coefficient matrix
- **b_eq** (*list of float*) – The right hand side vector

add_ineq_constraints(*A_ineq*, *b_ineq*)

Add inequality constraints to the model

Additional inequality constraints have the form A_ineq * x <= b_ineq. The number of columns in A_ineq must match with the number of variables x in the problem.

Parameters

- **A_ineq** (*sparse.csr_matrix*) – The coefficient matrix
- **b_ineq** (*list of float*) – The right hand side vector

getSolution() → list

Retrieve solution from SCIP backend

set_ineq_constraint(*idx*, *a_ineq*, *b_ineq*)

Replace a specific inequality constraint

Replace the constraint with the index idx with the constraint a_ineq*x ~ b_ineq

Parameters

- **idx** (*int*) – Index of the constraint
- **a_ineq** (*list of float*) – The coefficient vector
- **b_ineq** (*float*) – The right hand side value

set_objective(*c*)

Set the objective function with a vector

set_objective_idx(*C*)

Set the objective function with index-value pairs

e.g.: C=[[1, 1.0], [4,-0.2]]

set_time_limit(*t*)

Set the computation time limit (in seconds)

set_ub(*ub*)

Set the upper bounds to a given vector

slim_solve() → float

Solve the MILP, but return only the optimal value

Example

optim = scip.slim_solve()

Returns

(float)

Optimum value of the objective function.

solve() → Tuple[List, float, float]

Solve the MILP

Example

sol_x, optim, status = scip.solve()

Returns

(Tuple[List, float, float])

solution_vector, optimal_value, optimization_status

straindesign.solver_interface

Unified solver interface for LPs and MILPs (MILP_LP)

Module Contents

class straindesign.solver_interface.**MILP_LP**(***kwargs*)

Bases: object

Unified MILP and LP interface

This class is a wrapper for several solver interfaces to offer unique and consistent bindings for the construction and manipulation of MILPs and LPs in an vector-matrix-based manner and their solution.

Accepts a (mixed integer) linear problem in the form:

minimize(c), subject to: A_ineq * x <= b_ineq, A_eq * x = b_eq, lb <= x <= ub, forall(i) type(x_i) = vtype(i) (continous, binary, integer), indicator constraints: x(j) = [0|1] -> a_indic * x [<=|=|>=] b_indic

Please ensure that the number of variables and (in)equalities is consistent

Example

milp = MILP_LP(c, A_ineq, b_ineq, A_eq, b_eq, lb, ub, vtype, indic_constr)

Parameters

- **c** (*list of float*) – (Default: None) The objective vector (Objective sense: minimization).
- **A_ineq** (*sparse.csr_matrix*) – (Default: None) A coefficient matrix of the static inequalities.
- **b_ineq** (*list of float*) – (Default: None) The right hand side of the static inequalities.
- **A_eq** (*sparse.csr_matrix*) – (Default: None) A coefficient matrix of the static equalities.
- **b_eq** (*list of float*) – (Default: None) The right hand side of the static equalities.
- **lb** (*list of float*) – (Default: None) The lower variable bounds.
- **ub** (*list of float*) – (Default: None) The upper variable bounds.
- **vtype** (*str*) – (Default: None) A character string that specifies the type of each variable: ‘c’ontinous, ‘b’inary or ‘i’nteger
- **indic_constr** (IndicatorConstraints) – (Default: None) A set of indicator constraints stored in an object of IndicatorConstraints (see reference manual or docstring).
- **M** (*int*) – (Default: None) A large value that is used in the translation of indicator constraints to bigM-constraints for solvers that do not natively support them. If no value is provided, 1000 is used.
- **solver** (*str*) – (Default: taken from avail_solvers) Solver backend that should be used: ‘cplex’, ‘gurobi’, ‘glpk’ or ‘scip’
- **skip_checks** (*bool*) – (Default: False) Upon MILP construction, the dimensions of all provided vectors and matrices are checked to verify their consistency. If skip_checks=True is set, these checks are skipped.
- **tlim** (*float*) – Solution time limit in seconds.

- **Returns** – (MILP_LP):

  A MILP/LP solver interface class.

add_eq_constraints(*A_eq*, *b_eq*)

Add equality constraints to the model

Additional equality constraints have the form A_eq * x = b_eq. The number of columns in A_eq must match with the number of variables x in the problem.

Parameters

- **A_eq** (*sparse.csr_matrix*) – The coefficient matrix
- **b_eq** (*list of float*) – The right hand side vector

add_ineq_constraints(*A_ineq*, *b_ineq*)

Add inequality constraints to the model

Additional inequality constraints have the form A_ineq * x <= b_ineq. The number of columns in A_ineq must match with the number of variables x in the problem.

Parameters

- **A_ineq** (*sparse.csr_matrix*) – The coefficient matrix
- **b_ineq** (*list of float*) – The right hand side vector

clear_objective()

Clear objective

Set all coefficients in the objective vector to 0.

populate(*n*) → Tuple[List, float, float]

Generate a solution pool for MILPs

Example

sols_x, optim, status = cplex.populate()

Returns

(Tuple[List of lists, float, float])

solution_vectors, optimal_value, optimization_status

set_ineq_constraint(*idx*, *a_ineq*, *b_ineq*)

Replace a specific inequality constraint

Replace the constraint with the index idx with the constraint a_ineq*x ~ b_ineq

Parameters

- **idx** (*int*) – Index of the constraint
- **a_ineq** (*list of float*) – The coefficient vector
- **b_ineq** (*float*) – The right hand side value

set_objective(*c*)

Set the objective function with a vector

set_objective_idx(*C*)

Set the objective function with index-value pairs

e.g.: C=[[1, 1.0], [4,-0.2]]

set_time_limit(*t*)

Set the computation time limit (in seconds)

set_ub(*ub*)

Set the upper bounds to a given vector

slim_solve() → float

Solve the MILP or LP, but return only the optimal value

Example

optim = cplex.slim_solve()

Returns

(float)

Optimum value of the objective function.

solve() → Tuple[List, float, float]

Solve the MILP or LP

Example

sol_x, optim, status = milp.solve()

Returns

(Tuple[List, float, float])

solution_vector, optimal_value, optimization_status

straindesign.strainDesignMILP

Classes and function for the solution of strain design MILPs

Module Contents

class straindesign.strainDesignMILP.**SDMILP**(*model: straindesign.Model*, *sd_modules: List[straindesign.SDModule]*, ***kwargs*)

Bases: straindesign.SDProblem, straindesign.MILP_LP

Class that contains functions for the solution of the strain design MILP

This class is a wrapper and inherited from the casses SDProblem, MILP_LP. The constructor of SDProblem (see strainDesignProblem.py) translates a given problem into a MILP. The constructor of MILP_LP (see solver_interface.py) then sets up the solver interface for the selected solver. In addition to the functions from SDProblem and MILP_LP, SDMILP provides functions for the solution of the strain design MILP, such as verification of strain design solutions or introduction of exclusion constraints for computing multiple solutions.

Parameters

- **model** (*cobra.Model*) – A metabolic model that is an instance of the cobra.Model class.
- **sd_modules** (*(list of) straindesign.SDModule*) – Modules that specify the strain design problem, e.g., protected or suppressed flux states for MCS strain design or inner and outer objective functions for OptKnock. See description of SDModule for more information on how to set up modules.
- **ko_cost** (*optional (dict)*) – (Default: None) A dictionary of reaction identifiers and their associated knockout costs. If not specified, all reactions are treated as knockout candidates, equivalent to ko_cost = {'r1':1, 'r2':1, ...}. If a subset of reactions is listed in the dict, all other are not considered as knockout candidates.
- **ki_cost** (*optional (dict)*) – (Default: None) A dictionary of reaction identifiers and their associated costs for addition. If not specified, all reactions are treated as knockout candidates. Reaction addition candidates must be present in the original model with the intended flux boundaries **after** insertion. Additions are treated adversely to knockouts, meaning that their exclusion from the network is not associated with any cost while their presence entails intervention costs.
- **max_cost** (*optional (int)*) – (Default: inf): The maximum cost threshold for interventions. Every possible intervention is associated with a cost value (1, by default). Strain designs cannot exceed the max_cost threshold. Individual intervention cost factors may be defined through ki_cost and ko_cost.
- **solver** (*optional (str)*) – (Default: same as defined in model / COBRApy) The solver that should be used for preparing and carrying out the strain design computation. Allowed values are 'cplex', 'gurobi', 'scip' and 'glpk'.
- **M** (*optional (int)*) – (Default: None) If this value is specified (and non-zero, not None), the computation uses the big-M method instead of indicator constraints. Since GLPK does not support indicator constraints it uses the big-M method by default (with COBRA standard M=1000). M should be chosen 'sufficiently large' to avoid computational artifacts and 'sufficiently small' to avoid numerical issues.
- **essential_kis** (*optional (set)*) – A set of reactions that are marked as addable and that are essential for at least one of the strain design modules. Providing such "essential knock-ins" may speed up the strain design computation.

Returns
An instance of SDProblem containing the strain design MILP and providing several functions for its solution

Return type
(*SDMILP*)

add_exclusion_constraints(*z*)

Exclude binary solution in z and all supersets from MILP

add_exclusion_constraints_ineq(*z*)

Exclude binary solution in z (but not its supersets) from MILP

build_sd_solution(*sd_dict*, *status*, *solution_approach*)

Build the strain design solution object

compute(***kwargs*)

Compute arbitrary solutions of the strain design MILP and iteratively find further solutions

Parameters

- **max_solutions** (*optional (int)*) – (Default: inf) The maximum number of MILP solutions that are generated for a strain design problem.

- **time_limit** (*optional (int)*) – (Default: inf) The time limit in seconds for the MILP-solver.
- **show_no_ki** (*optional (bool)*) – (Default: True) Indicate non-added addition candidates in a solution specifically with a value of 0

Returns
Strain design solutions provided as an SDSolutions object

Return type
(*SDSolutions*)

compute_optimal(***kwargs*)

Compute the global optimum of the strain design MILP and iteratively find the next best solution

Parameters

- **max_solutions** (*optional (int)*) – (Default: inf) The maximum number of MILP solutions that are generated for a strain design problem.
- **time_limit** (*optional (int)*) – (Default: inf) The time limit in seconds for the MILP-solver.
- **show_no_ki** (*optional (bool)*) – (Default: True) Indicate non-added addition candidates in a solution specifically with a value of 0

Returns
Strain design solutions provided as an SDSolutions object

Return type
(*SDSolutions*)

enumerate(***kwargs*)

Find all globally optimal solutions to the strain design MILP and iteratively construct pools for the suboptimal values

Parameters

- **max_solutions** (*optional (int)*) – (Default: inf) The maximum number of MILP solutions that are generated for a strain design problem.
- **time_limit** (*optional (int)*) – (Default: inf) The time limit in seconds for the MILP-solver.
- **show_no_ki** (*optional (bool)*) – (Default: True) Indicate non-added addition candidates in a solution specifically with a value of 0

Returns
Strain design solutions provided as an SDSolutions object

Return type
(*SDSolutions*)

fixObjective(*c*, *cx*)

Enforce a certain objective function and value (or any other constraint of the form c*x <= cx)

populateZ(*n*) → Tuple[List, int]

Populate MILP, and return only binary variables rounded to 5 decimals (should return ints)

resetObjective()

Reset objective to the one set upon MILP construction

resetTargetableZ()

Reset targetable/switchable intervention indicators / allow all intervention candidates

sd2dict(*sol*, **args*) → Dict

Translate binary solution vector to dictionary for human-readable output

setMinIntvCostObjective()

Reset minimization of intervention costs as global objective

setTargetableZ(*sol*)

Only allow a subset of intervention candidates

solveZ() → Tuple[List, int]

Solve MILP, and return only binary variables rounded to 5 decimals (should return ints)

verify_sd(*sols*) → List

Verify computed strain design

straindesign.strainDesignModule

Class: strain design module (SDModule)

Module Contents

class straindesign.strainDesignModule.**SDModule**(*model*, *module_type*, **args*, ***kwargs*)

Bases: straindesign.parse_constr.Dict

Strain design modules are used to specify the goal of a strain design computation

(Lists of) SDModule objects are passed to compute_strain_design to specify the goal strain design computation. Strain design modules indicate the appraoch that should be used (OptKnock, RobustKnock, OptCouple or MCS) and the parameters for each approach. In each strain design computation, one of the following modules can be used at most once: OptKnock, RobustKnock and OptCouple. Additionally an arbitrary number of MCS modules (PROTECT or SUPPRESS) may me used. The global objective of a strain design computation depends on the specified modules. If an OptKnock or RobustKnock module is used, the global objective function will be defined by 'outer_objective' and 'outer_opt_sense'. If OptCouple is used, the global objective is derived from by 'inner_objective' and 'inner_opt_sense'. If only MCS-like modules (suppress, protect) are used in a computation, the number of interventions is globally minimized.

In the following, the modules and their mandatory/optional arguments are presented in detail.

OptKnock:

Globally maximize an *outer objective* subjected to the maximization/optimization of an *inner objective*. For instance, maximize product synthesis, assuming that the strain maximizes its growth rate. When used with the 'best' or 'populate' approach (see compute_strain_designs), this module will guarantee that the found intervention set allows for the highest possible outer objective (e.g., production) under the premise that the inner objective (growth) is forced to be maximal. This means that the *production potential* is maximal. However it does not necessarily mean that production is enforced at high growth rates. For enforced growth coupling, refer to the other module types. Additional constraints can be used to impose certain properties on the designed strains: constraints='growth >= 0.5' will guarantee that the designed strain can reach growth rates above 0.5. constraints=['growth >= 0.5', 'EX_byprod_e <= 2'] additionally guarantees that the synthesis rate of a by-product stays below 2 at growth maximal flux states. Alternative inner or outer objective functions (e.g., ATP maintenance) can be used for diffent types of strain design.

mandatory arguments: model, module_type='optknock', inner_objective, outer_objective optional arguments: constraints, inner_opt_sense, outer_opt_sense, skip_checks (Detailed description of the arguments follow below)

RobustKnock:
Globally minimize the maximum of an *outer objective* subjected to the maximization/optimization of an *inner objective*. For instance, maximize the minimal product synthesis rate, assuming that the strain maximizes its growth rate. When used with the 'best' or 'populate' approach (see compute_strain_designs), this module will guarantee that the found intervention set enforces that the minimum of the outer objective (e.g., maximal production) is maximal (max-min), given that the inner objective (growth) is forced to be maximal. This means that the *minimal guaranteed production* is maximal and production is guaranteed at growth-maximal flux states. Additional constraints can be used to impose certain properties on the designed strains: constraints='growth >= 0.5' will guarantee that the designed strain can reach growth rates above 0.5/h. constraints=['growth >= 0.5', 'EX_byprod_e<=2'] additionally guarantees that the synthesis rate of a by-product stays below 2 mmol/gCDW/h at growth maximal flux states.

mandatory arguments: model, module_type='robustknock', inner_objective, outer_objective optional arguments: constraints, inner_opt_sense, outer_opt_sense, skip_checks, reac_ids (Detailed description of the arguments follow below)

OptCouple:
Globally maximize the *growth-coupling potential*, that is, the difference between the maximal growth rate without product synthesis and the maximum overall growth rate (with product synthesis). This strain design approach often leads to directionally growth-coupled strain designs. Again, alternative definitions of this objective are possible to, for instance, couple production to the synthesis of ATP. To specify the product for which coupling should be engineered, the reaction identifier of the product exchange (pseudo)reaction is passed through the *prod_id* parameter (see below). One may addidionally define a minimum growth-coupling potential through the parameter *min_gcp*.

mandatory arguments: model, module_type='optcouple', inner_objective, prod_id optional arguments: constraints, inner_opt_sense, min_gcp, skip_checks, reac_ids (Detailed description of the arguments follow below)

Suppress:
MCS-like suppression of a subspace of all steady-state flux vectors. The 'constraints' parameter is used to descirbe the flux states that should not be eliminated from the flux space. Depending on the goal of the strain design computation, undesired flux states may be those with production of an undesired by-product, low product synthesis rates, low product yields or even flux states with microbial growth. In addition to constraints, an inner objective function can be defined that is enforced. In that case flux states are suppressed that are optimal with respect to the specified inner objective function and additionally fulfil the specified constraints. If the goal is to find minimal sets of knockouts that are lethal for an organism, one can use a suppress module with the constraint parameter: constraints='growth >= 0.01'. The algorithm will then find thes smallest sets of knockouts that render flux states with growth >= 0.01 infeasible. If we take the example of production strain design, one could use the suppress module to enforce a certain product yield (prod/subst > min_yield) growth-maximal flux states. This can be done by defining an inner objective function for optmizing growth and selecting a minimum production threshold to be attained at maximum growth: inner_objective='1.0 growth', constraints='EX_prod_e - min_yield*UP_subst_e <= 0'. If used without any constraints, the suppress module ensures that the model is infeasible.

mandatory arguments: model, module_type='suppress' optional arguments: constraints, inner_objective, inner_opt_sense, skip_checks, reac_ids (Detailed description of the arguments follow below)

Protect:
MCS-like protection of a subspace of all steady-state flux vectors. The 'constraints' parameter is used to descirbe flux states that should become/stay feasible when/by introducing interventions. This can be used to maintain or protect certain metabolic functions, like microbial growth, despite engineering a strain for bioroduction. When provided with an inner objective, the protect module will ensure that flux vectors

optimal with respect to the inner objective function will be able to fulfil the constraints set in the 'constraints' parameter. This can be used to design potentially growth-coupled strains with a minimum set of metabolic interventions. As an example, if one uses the protect module to ensure that growth with rates above 0.1/h remains feasible, one sets constraints='growth >= -0.1'. If the goal is to ensure that product synthesis is possible at rates of more than 5 mmol/gCDW/h at maximal growth, one sets: inner_objective='1.0 growth' and constraints='EX_prod_e >= 5'. If used without any constraints, the protect module just ensures that the model is feasible.

mandatory arguments: model, module_type='protect' optional arguments: constraints,inner_objective, inner_opt_sense, skip_checks, reac_ids (Detailed description of the arguments follow below)

Example

m = SDModule(model,'optknock',outer_objective='growth', inner_objective='EX_etoh_e', constraints='growth >= 0.2')

Parameters

- **model** (*cobra.Model*) – A metabolic model that is an instance of the cobra.Model class. Instead of a model, a dummy object can be used as long as it has the field 'id'. If a dummy is used, skip_checks=True must be used and a list of reaction ids must be provided through reac_ids=*list_of_strings*.
- **module_type** (*str*) – A string that specifies the module type. Allowed values are 'optknock', 'robustknock', 'optcouple', 'protect', 'suppress'. Depending on the specified module type, other parameters must be set accordingly (see description above).
- **constraints** (*optional (str) or (list of str) or (list of [dict,str, float])*) – (Default: '')

  List of *linear* constraints to be used in the module, e.g., to be enforced, suppressed or taken into account: signs + or -, scalar factors for reaction rates, inclusive (in)equalities and a float value on the right hand side. The parsing of the constraints input allows for some flexibility. Correct (and identical) inputs are, for instance: constraints='-EX_o2_e <= 5, ATPM = 20' or constraints=['-EX_o2_e <= 5', 'ATPM = 20'] or constraints=[[{'EX_o2_e':-1},'<=',5], [{'ATPM':1},'=',20]]
- **inner_objective** (*optional (str) or (dict)*) – The *linear* inner objective function for any module type. This parameter is mandatory for OptKnock, RobustKnock and OptCouple modules and optional for suppress and protect modules. If used, an optimization is nested into the global problem. It can be used to account for the biological objective of growth (inner_objective='BIOMASS_Ecoli_core_w_GAM'). Any linear expression can be used as input, either as a single string or as a dict. Correct (and identical) inputs are, for instance: inner_objective='BIOMASS_Ecoli_core_w_GAM - 0.05 EX_etoh_e' inner_objective={'BIOMASS_Ecoli_core_w_GAM': 1, 'EX_etoh_e': -0.05}
- **inner_opt_sense** (*optional (str)*) – (Default: 'maximize')

  Sense of the inner optimization (maximization or minimization). Allowed values are 'minimize' and 'maximize'.
- **outer_objective** (*optional (str) or (dict)*) – The *linear* outer objective function for any module type. This parameter is mandatory for OptKnock and RobustKnock and cannot be used with OptCouple, suppress and protect modules. If applied, this objective function is used as the global objective function. In case of OptCouple, the global objective is the maximization of the growth-coupling potential and the outer objective is not specified manually. Typical outer objectives are the optimization

of product synthesis (outer_objective='BIOMASS_Ecoli_core_w_GAM'), but also combinations of product synthesis and growth are possible. Any linear expression can be used as input, either as a single string or as a dict. Correct (and identical) inputs are, for instance: outer_objective='BIOMASS_Ecoli_core_w_GAM + 0.1 EX_prod_e' outer_objective={'BIOMASS_Ecoli_core_w_GAM': 1, 'EX_prod_e': 0.1}

- **outer_opt_sense** (*optional (str)*) – (Default: 'maximize')

  Sense of the outer optimization (maximization or minimization). Allowed values are 'minimize' and 'maximize'.

- **prod_id** (*optional (str) or (dict)*) – The reaction id of the product of interest. This parameter is *only* used in OptCouple strain design and will have no effect as part of any other module. Permitted is any linear expression either in the form of a string or as a dict: prod_id='EX_etoh' prod_id={'EX_etoh': 1}

- **min_gcp** (*optional (float)*) – (Default: 0.0)

  Minimial growth-coupling potential (GCP). I.e., the minimum difference between maximum growth with and without production. In practice there are two nested optimizations, one that optimizes inner_objective and one that optimizes inner_objective and additionally demands that the constraint: prod_id=0 holds. Therefore, min_gcp presents the minimum enforced growth-coupling potential, so, a minimum objective value. This parameter is supposed to be used with the 'any' approach and has (virtually) no effect when used with 'best' or 'populate', since GCP is maximized anyway.

- **skip_checks** (*optional (bool)*) – (Default: False)

  Skip the module verification. If checks are not skipped, the constructor will verify, if the module is feasible with the original model, that is, if all entered parameters parse correctly and if the given sets of constraints can be applied on the model without rendering it infeasible. Finally, it throws an error if the trivial 0-vector is feasible in the subspaces defined by a suppress or protect module, since the user should ensure that the 0-vector is excluded from these subspaces (see online documentation for detail).

Returns
A strain design module object that can be used with the function compute_strain_design. Multiple modules can be used to specify a strain design problem.

Return type
(*SDModule*)

Initialize self. See help(type(self)) for accurate signature.

copy()
Create a deep copy of a strain design module.

straindesign.strainDesignProblem

Classes and functions for the construction of strain design MILPs

This module contains functions that help construct mixed-integer linear problems, mainly functions that facilitate the construction of LP and Farkas dual problems from linear problems of the type A_ineq*x<=b_ineq, A_eq*x<=b_eq, lb<=x<=ub. The functions also help keeping track of the relationship of constraints and variables and their individual counterparts in dual problems, which is essential when simulating knockouts in dual problems. Most of the time, the sparse datatype is used to store and edit matrices for improved speed and memory.

Module Contents

class straindesign.strainDesignProblem.**ContMILP**(*A_ineq*, *b_ineq*, *A_eq*, *b_eq*, *lb*, *ub*, *c*, *z_map_constr_ineq*, *z_map_constr_eq*, *z_map_vars*)

Continuous representation of the strain design MILP.

This MILP can be used to verify computation results. Since this class also stores the relationship between intervention variables z and corresponding (in)equality constraints and variables in the problem, it can be used to verify computed designs quickly and in a numerically stable manner.

class straindesign.strainDesignProblem.**SDProblem**(*model: cobra.Model*, *sd_modules: List[straindesign.SDModule]*, **args*, ***kwargs*)

Strain design MILP

The construcor of this class translates a model and strain design modules into a mixed integer linear problem. This class, however, is the backbone of the strain design computation. Preprocessing steps that enable gene, reaction and regulatory interventions, or network compression usually preceed the construction of an SDProblem-object and are integrated in the function *compute_strain_designs*.

Parameters

- **model** (*cobra.Model*) – A metabolic model that is an instance of the cobra.Model class.
- **sd_modules** (*(list of) straindesign.SDModule*) – Modules that specify the strain design problem, e.g., protected or suppressed flux states for MCS strain design or inner and outer objective functions for OptKnock. See description of SDModule for more information on how to set up modules.
- **ko_cost** (*optional (dict)*) – (Default: None) A dictionary of reaction identifiers and their associated knockout costs. If not specified, all reactions are treated as knockout candidates, equivalent to ko_cost = {'r1':1, 'r2':1, ... }. If a subset of reactions is listed in the dict, all other are not considered as knockout candidates.
- **ki_cost** (*optional (dict)*) – (Default: None) A dictionary of reaction identifiers and their associated costs for addition. If not specified, all reactions are treated as knockout candidates. Reaction addition candidates must be present in the original model with the intended flux boundaries **after** insertion. Additions are treated adversely to knockouts, meaning that their exclusion from the network is not associated with any cost while their presence entails intervention costs.
- **max_cost** (*optional (int)*) – (Default: inf): The maximum cost threshold for interventions. Every possible intervention is associated with a cost value (1, by default). Strain designs cannot exceed the max_cost threshold. Individual intervention cost factors may be defined through ki_cost and ko_cost.
- **solver** (*optional (str)*) – (Default: same as defined in model / COBRApy) The solver that should be used for preparing and carrying out the strain design computation. Allowed values are 'cplex', 'gurobi', 'scip' and 'glpk'.
- **M** (*optional (int)*) – (Default: None) If this value is specified (and non-zero, not None), the computation uses the big-M method instead of indicator constraints. Since GLPK does not support indicator constraints it uses the big-M method by default (with COBRA standard M=1000). M should be chosen 'sufficiently large' to avoid computational artifacts and 'sufficiently small' to avoid numerical issues.
- **essential_kis** (*optional (set)*) – A set of reactions that are marked as addable and that are essential for at least one of the strain design modules. Providing such "essential knock-ins" may speed up the strain design computation.

Returns
An instance of SDProblem containing the strain design MILP

Return type
(*SDProblem*)

`addModule`(*sd_module*)

Generate module LP and z-linking-matrix for each module and add them to the strain design MILP

Parameters
sd_module (*straindesign.SDModule*) – Modules to describe strain design problems like protected or suppressed flux states for MCS strain design or inner and outer objective functions for OptKnock. See description of SDModule for more information on how to set up modules.

`link_z`()

Connect binary intervention variables to variables and constraints of the strain design problem

Function that uses the maps between intervention indicators z and variables and constraints of the linear strain design (in)equality system (self.z_map_constr_ineq, self.z_map_constr_eq and self.z_map_vars) to set up the strain design MILP.

MILP construction uses the following steps:

(1) Translate equality-KOs/KIs to two inequality-KOs/KIs

(2) Translate variable-KOs/KIs to inequality-KIs/KOs

(3) Try to bound the problem with LPs

(4) Use LP-determined bounds to link z-variables, where such bounds were found

(5) Translate remaining inequalities back to equalities when possible and link z via indicator constraints. If necessary, the solver interface will translate them to big-M constraints. (6) Remove redundant equalities from static problem

straindesign.strainDesignProblem.**LP_dualize**(*A_ineq_p*, *b_ineq_p*, *A_eq_p*, *b_eq_p*, *lb_p*, *ub_p*, *c_p*, *z_map_constr_ineq_p=None*, *z_map_constr_eq_p=None*, *z_map_vars_p=None*) → Tuple[scipy.sparse.csr_matrix, Tuple, scipy.sparse.csr_matrix, Tuple, Tuple, Tuple, scipy.sparse.csr_matrix, scipy.sparse.csr_matrix, scipy.sparse.csr_matrix]

Translate a primal system to its LP dual system

The primal system must be given in the standard form: A_ineq x <= b_ineq, A_eq x = b_eq, lb <= x < ub, min{c'x}. The LP duality theorem defines a set of two problems. If one of the LPs is a maximization and and optimum exists, the optimal value of this LP is identical to the minimal optimum of its LP dual problem. LP duality can be used for nested optimization, since solving the primal and the LP dual problem, while enfocing equality of the objective value, guarantees optimality.

Construction of the LP dual:

Variables translate to constraints:
x={R} -> = x>=0 -> >= (new constraint is multiplied with -1 to translate to <= e.g. -A_i' y <= -c_i)
x<=0 -> <=

Constraints translate to variables:
= -> y={R} <= -> y>=0

Parameters

- **A_ineq_p** (*sparse.csr_matrix and list of float*) – A coefficient matrix and a vector that describe the linear inequalities of the primal LP A_ineq_p*x <= b_ineq_p
- **b_ineq_p** (*sparse.csr_matrix and list of float*) – A coefficient matrix and a vector that describe the linear inequalities of the primal LP A_ineq_p*x <= b_ineq_p
- **A_eq_p** (*sparse.csr_matrix and list of float*) – A coefficient matrix and a vector that describe the linear equalities of the primal LP A_eq_p*x <= b_eq_p
- **b_eq_p** (*sparse.csr_matrix and list of float*) – A coefficient matrix and a vector that describe the linear equalities of the primal LP A_eq_p*x <= b_eq_p
- **lb_p** (*list of float*) – Upper and lower variable bounds in vector form.
- **ub_p** (*list of float*) – Upper and lower variable bounds in vector form.
- **c_p** (*list of float*) – The objective coefficient vector of the primal minimization-LP. z_map_constr_ineq_p, z_map_constr_eq_p, z_map_vars_p
- **z_map_constr_ineq** (*optional (sparse.csr_matrix)*) – Matrices that contain the relationship between metabolic reactions and different parts of the LP, such as reactions, metabolites or other (in)equalities. These matrices help keeping track of the parts of the LP that are affected by reaction knockouts and additions. When a reaction (i) knockout removes the variable or constraint (j), the respective matrix contains a coefficient 1 at this position. -1 marks additions. E.g.: If the knockout of reaction i corresponds to the removal of inequality constraint j, there is a matrix entry z_map_constr_ineq_(i,j) = 1. If these matrices are provided, they are updated for the dualized LP, if not, the dual problem is constructed without returning information about these relationships.
- **z_map_constr_eq** (*optional (sparse.csr_matrix)*) – Matrices that contain the relationship between metabolic reactions and different parts of the LP, such as reactions, metabolites or other (in)equalities. These matrices help keeping track of the parts of the LP that are affected by reaction knockouts and additions. When a reaction (i) knockout removes the variable or constraint (j), the respective matrix contains a coefficient 1 at this position. -1 marks additions. E.g.: If the knockout of reaction i corresponds to the removal of inequality constraint j, there is a matrix entry z_map_constr_ineq_(i,j) = 1. If these matrices are provided, they are updated for the dualized LP, if not, the dual problem is constructed without returning information about these relationships.
- **z_map_vars** (*optional (sparse.csr_matrix)*) – Matrices that contain the relationship between metabolic reactions and different parts of the LP, such as reactions, metabolites or other (in)equalities. These matrices help keeping track of the parts of the LP that are affected by reaction knockouts and additions. When a reaction (i) knockout removes the variable or constraint (j), the respective matrix contains a coefficient 1 at this position. -1 marks additions. E.g.: If the knockout of reaction i corresponds to the removal of inequality constraint j, there is a matrix entry z_map_constr_ineq_(i,j) = 1. If these matrices are provided, they are updated for the dualized LP, if not, the dual problem is constructed without returning information about these relationships.

Returns

(Tuple): The LP dual of the problem in the format: A_ineq, b_ineq, A_eq, b_eq, c, lb, ub and optionally also z_map_constr_ineq, z_map_constr_eq, z_map_vars

straindesign.strainDesignProblem.**build_primal_from_cbm**(*model*, *V_ineq=None*, *v_ineq=None*, *V_eq=None*, *v_eq=None*, *c=None*) → Tuple[scipy.sparse.csr_matrix, Tuple, scipy.sparse.csr_matrix, Tuple, Tuple, Tuple, scipy.sparse.csr_matrix, scipy.sparse.csr_matrix, scipy.sparse.csr_matrix]

Builds primal LP from constraint-based model and (optionally) additional constraints.

standard form: A_ineq x <= b_ineq, A_eq x = b_eq, lb <= x <= ub, min{c'x}. Additionally, this function also returns a set of matrices that associate each variable (and constraint) with reactions. In the primal problems all variables correspond to reactions (z), therefore, the z_map_vars matrix is an identity matrix. The constraints correspond to metabolites, thus z_map_constr_ineq, z_map_constr_eq are all-zero.

Parameters

- **model** (*cobra.Model*) – A metabolic model that is an instance of the cobra.Model class
- **V_ineq** (*sparse.csr_matrix, list of float*) – Linear inequality constraints of the form V_ineq*x <= v_ineq. Ensure that the number of columns in V_ineq is identical to the number of reactions in the model.
- **v_ineq** (*sparse.csr_matrix, list of float*) – Linear inequality constraints of the form V_ineq*x <= v_ineq. Ensure that the number of columns in V_ineq is identical to the number of reactions in the model.
- **V_eq** (*sparse.csr_matrix, list of float*) – Linear equality constraints of the form V_eq*x = v_eq. Ensure that the number of columns in V_eq is identical to the number of reactions in the model.
- **v_eq** (*sparse.csr_matrix, list of float*) – Linear equality constraints of the form V_eq*x = v_eq. Ensure that the number of columns in V_eq is identical to the number of reactions in the model.
- **c** (*list of float*) – Object coefficient vector (same lenght as variable vector).

Returns

A_ineq, b_ineq, A_eq, b_eq, lb, ub, c, z_map_constr_ineq, z_map_constr_eq, z_map_vars. A constraint-based steady-state model in the form of a linear (in)equality system. The matrices z_map_constr_ineq, z_map_constr_eq, z_map_vars contain the association between reactions and different parts of the LP, such as reactions, metabolites or other (in)equalities.

Return type

(Tuple)

straindesign.strainDesignProblem.**farkas_dualize**(*A_ineq_p*, *b_ineq_p*, *A_eq_p*, *b_eq_p*, *lb_p*, *ub_p*, *z_map_constr_ineq_p=None*, *z_map_constr_eq_p=None*, *z_map_vars_p=None*) → Tuple[scipy.sparse.csr_matrix, Tuple, scipy.sparse.csr_matrix, Tuple, Tuple, scipy.sparse.csr_matrix, scipy.sparse.csr_matrix, scipy.sparse.csr_matrix]

Translate a primal system of linear (in)equality to its Farkas dual

The primal system must be given in the standard form: A_ineq x <= b_ineq, A_eq x = b_eq, lb <= x < ub. Farkas' lemma defines a set of two systems of linear (in)equalities of which exactly one is feasible. Since the feasibility of one is a certificate for the infeasibility of the other one, this theorem can be used to set up problems that imply the infeasibility and thus exclusion of a certain subspace. This priciple is used for MCS calculation (the SUPPRESS module).

Consider that the following is not implemented: In the case of (1) A x = b, (2) x={R}, (3) b~=0, Farkas' lemma is special, because b'y ~= 0 is required to make the primal infeasible instead of b'y < 0. 1. This does not occur very often. 2. Splitting the equality into two inequalities that translate to y>=0 would be posible, and yield b'y < 0 in the farkas' lemma. Maybe splitting is required, but I actually don't think so. Using the special case of b'y < 0 for b'y ~= 0 should be enough.

Parameters

- **A_ineq_p** (*sparse.csr_matrix and list of float*) – A coefficient matrix and a vector that describe the linear inequalities of the primal LP A_ineq_p*x <= b_ineq_p
- **b_ineq_p** (*sparse.csr_matrix and list of float*) – A coefficient matrix and a vector that describe the linear inequalities of the primal LP A_ineq_p*x <= b_ineq_p
- **A_eq_p** (*sparse.csr_matrix and list of float*) – A coefficient matrix and a vector that describe the linear equalities of the primal LP A_eq_p*x <= b_eq_p
- **b_eq_p** (*sparse.csr_matrix and list of float*) – A coefficient matrix and a vector that describe the linear equalities of the primal LP A_eq_p*x <= b_eq_p
- **lb_p** (*list of float*) – Upper and lower variable bounds in vector form.
- **ub_p** (*list of float*) – Upper and lower variable bounds in vector form.
- **z_map_constr_ineq** (*optional (sparse.csr_matrix)*) – Matrices that contain the relationship between metabolic reactions and different parts of the LP, such as reactions, metabolites or other (in)equalities. These matrices help keeping track of the parts of the LP that are affected by reaction knockouts and additions. When a reaction (i) knockout removes the variable or constraint (j), the respective matrix contains a coefficient 1 at this position. -1 marks additions. E.g.: If the knockout of reaction i corresponds to the removal of inequality constraint j, there is a matrix entry z_map_constr_ineq_(i,j) = 1. If these matrices are provided, they are updated for the dualized LP, if not, the dual problem is constructed without returning information about these relationships.
- **z_map_constr_eq** (*optional (sparse.csr_matrix)*) – Matrices that contain the relationship between metabolic reactions and different parts of the LP, such as reactions, metabolites or other (in)equalities. These matrices help keeping track of the parts of the LP that are affected by reaction knockouts and additions. When a reaction (i) knockout removes the variable or constraint (j), the respective matrix contains a coefficient 1 at this position. -1 marks additions. E.g.: If the knockout of reaction i corresponds to the removal of inequality constraint j, there is a matrix entry z_map_constr_ineq_(i,j) = 1. If these matrices are provided, they are updated for the dualized LP, if not, the dual problem is constructed without returning information about these relationships.
- **z_map_vars** (*optional (sparse.csr_matrix)*) – Matrices that contain the relationship between metabolic reactions and different parts of the LP, such as reactions, metabolites or other (in)equalities. These matrices help keeping track of the parts of the LP that are affected by reaction knockouts and additions. When a reaction (i) knockout removes the variable or constraint (j), the respective matrix contains a coefficient 1 at this position. -1 marks additions. E.g.: If the knockout of reaction i corresponds to the removal of inequality constraint j, there is a matrix entry z_map_constr_ineq_(i,j) = 1. If these matrices are provided, they are updated for the dualized LP, if not, the dual problem is constructed without returning information about these relationships.

Returns

(Tuple): The Farkas dual of the linear (in)equality system in the format: A_ineq, b_ineq, A_eq, b_eq, lb, ub and optionally also z_map_constr_ineq, z_map_constr_eq, z_map_vars

straindesign.strainDesignProblem.**prevent_boundary_knockouts**(*A_ineq*, *b_ineq*, *lb*, *ub*, *z_map_constr_ineq*, *z_map_vars*) → Tuple[scipy.sparse.csr_matrix, Tuple, Tuple, Tuple, scipy.sparse.csr_matrix]

Put negative lower bounds and positive upper bounds into (notknockable) inequalities

This is a helper function that puts negative lower bounds and positive upper bounds into (not-knockable) inequalities. Later on, one may simulate the knockouts by multiplying the upper and lower bounds with a binary variable z. This functions prevents that

Parameters

- **A_ineq_p** (*sparse.csr_matrix and list of float*) – A coefficient matrix and a vector that describe the linear inequalities of the primal LP A_ineq_p*x <= b_ineq_p
- **b_ineq_p** (*sparse.csr_matrix and list of float*) – A coefficient matrix and a vector that describe the linear inequalities of the primal LP A_ineq_p*x <= b_ineq_p
- **lb_p** (*list of float*) – Upper and lower variable bounds in vector form.
- **ub_p** (*list of float*) – Upper and lower variable bounds in vector form.
- **z_map_constr_ineq** (*optional (sparse.csr_matrix)*) – Matrices that contain the relationship between metabolic reactions and different parts of the LP, such as reactions, metabolites or other (in)equalities. These matrices help keeping track of the parts of the LP that are affected by reaction knockouts and additions. When a reaction (i) knockout removes the variable or constraint (j), the respective matrix contains a coefficient 1 at this position. -1 marks additions. E.g.: If the knockout of reaction i corresponds to the removal of inequality constraint j, there is a matrix entry z_map_constr_ineq_(i,j) = 1. If these matrices are provided, they are updated. Otherwise, all reactions are assumed to be knockable and thus all negative upper and positive lower bounds are translated into constraints.
- **z_map_vars** (*optional (sparse.csr_matrix)*) – Matrices that contain the relationship between metabolic reactions and different parts of the LP, such as reactions, metabolites or other (in)equalities. These matrices help keeping track of the parts of the LP that are affected by reaction knockouts and additions. When a reaction (i) knockout removes the variable or constraint (j), the respective matrix contains a coefficient 1 at this position. -1 marks additions. E.g.: If the knockout of reaction i corresponds to the removal of inequality constraint j, there is a matrix entry z_map_constr_ineq_(i,j) = 1. If these matrices are provided, they are updated. Otherwise, all reactions are assumed to be knockable and thus all negative upper and positive lower bounds are translated into constraints.

Returns

(Tuple): A linear (in)equality system in the format: A_ineq, b_ineq, A_eq, b_eq, lb, ub and optionally also updated z_map_constr_ineq, z_map_constr_eq

straindesign.strainDesignProblem.**reassign_lb_ub_from_ineq**(*A_ineq*, *b_ineq*, *A_eq*, *b_eq*, *lb*, *ub*, *z_map_constr_ineq=None*, *z_map_constr_eq=None*, *z_map_vars=None*) → Tuple[scipy.sparse.csr_matrix, Tuple, scipy.sparse.csr_matrix, Tuple, Tuple, Tuple, scipy.sparse.csr_matrix, scipy.sparse.csr_matrix]

Remove single constraints in A_ineq or A_eq in favor of lower and upper bounds on variables

Constraints on single variables instead translated into lower and upper bounds (lb, ub). This is useful to filter out redundant bounds on variables and keep the (in)equality system concise. To avoid interference with the knock-

out logic, negative upper bounds and positive lower bounds are not put into lb and ub, when reactions are flagged knockable with z_map_vars.

Parameters

- **A_ineq_p** (*sparse.csr_matrix and list of float*) – A coefficient matrix and a vector that describe the linear inequalities of the primal LP A_ineq_p*x <= b_ineq_p
- **b_ineq_p** (*sparse.csr_matrix and list of float*) – A coefficient matrix and a vector that describe the linear inequalities of the primal LP A_ineq_p*x <= b_ineq_p
- **A_eq_p** (*sparse.csr_matrix and list of float*) – A coefficient matrix and a vector that describe the linear equalities of the primal LP A_eq_p*x <= b_eq_p
- **b_eq_p** (*sparse.csr_matrix and list of float*) – A coefficient matrix and a vector that describe the linear equalities of the primal LP A_eq_p*x <= b_eq_p
- **lb_p** (*list of float*) – Upper and lower variable bounds in vector form.
- **ub_p** (*list of float*) – Upper and lower variable bounds in vector form.
- **z_map_constr_ineq** (*optional (sparse.csr_matrix)*) – Matrices that contain the relationship between metabolic reactions and different parts of the LP, such as reactions, metabolites or other (in)equalities. These matrices help keeping track of the parts of the LP that are affected by reaction knockouts and additions. When a reaction (i) knockout removes the variable or constraint (j), the respective matrix contains a coefficient 1 at this position. -1 marks additions. E.g.: If the knockout of reaction i corresponds to the removal of inequality constraint j, there is a matrix entry z_map_constr_ineq_(i,j) = 1. If these matrices are provided, they are updated. Otherwise, all reactions are assumed to be notknockable and thus all constraints on single variables put into lb and ub.
- **z_map_constr_eq** (*optional (sparse.csr_matrix)*) – Matrices that contain the relationship between metabolic reactions and different parts of the LP, such as reactions, metabolites or other (in)equalities. These matrices help keeping track of the parts of the LP that are affected by reaction knockouts and additions. When a reaction (i) knockout removes the variable or constraint (j), the respective matrix contains a coefficient 1 at this position. -1 marks additions. E.g.: If the knockout of reaction i corresponds to the removal of inequality constraint j, there is a matrix entry z_map_constr_ineq_(i,j) = 1. If these matrices are provided, they are updated. Otherwise, all reactions are assumed to be notknockable and thus all constraints on single variables put into lb and ub.
- **z_map_vars** (*optional (sparse.csr_matrix)*) – Matrices that contain the relationship between metabolic reactions and different parts of the LP, such as reactions, metabolites or other (in)equalities. These matrices help keeping track of the parts of the LP that are affected by reaction knockouts and additions. When a reaction (i) knockout removes the variable or constraint (j), the respective matrix contains a coefficient 1 at this position. -1 marks additions. E.g.: If the knockout of reaction i corresponds to the removal of inequality constraint j, there is a matrix entry z_map_constr_ineq_(i,j) = 1. If these matrices are provided, they are updated. Otherwise, all reactions are assumed to be notknockable and thus all constraints on single variables put into lb and ub.

Returns

(Tuple): A linear (in)equality system in the format: A_ineq, b_ineq, A_eq, b_eq, lb, ub and optionally also updated z_map_constr_ineq, z_map_constr_eq

straindesign.strainDesignProblem.**worker_compute**(*i*) → Tuple[int, float]

Helper function for determining bounds on linear expressions

straindesign.strainDesignProblem.**worker_init**(*A*, *A_ineq*, *b_ineq*, *A_eq*, *b_eq*, *lb*, *ub*, *solver*)

Helper function for determining bounds on linear expressions

straindesign.strainDesignSolutions

Container for strain design solutions (SDSolutions)

Module Contents

class straindesign.strainDesignSolutions.**SDSolutions**(*model*, *sd*, *status*, *sd_setup*)

Bases: object

Container for strain design solutions

Objects of this class are returned by strain design computations. This class contains the metabolic interventions on the gene, reaction or regulation level alongside with information about the strain design setup, including the model used and the strain design modules. Strain design solutions can be accessed either through the fields or through specific functions that preprocess or reformat strain designs for different purposes.

Instances of this class are not meant to be created by StrainDesign users.

Parameters

- **model** (*cobra.Model*) – A metabolic model that is an instance of the cobra.Model class.
- **sd** (*list of dict*) – A list of dicts every dict represents an intervention set. Keys in each dict are reaction/gene identifiers and the associated value determines if it is added (1), not added (0) or knocked out (-1). For regulatory interventions, (1) means active regulation and (0) means regulatory intervention not added. These will be translated to True and False.
- **status** (*str*) – Status string of the computation (e.g.: 'optimal')
- **sd_setup** (*dict*) – A dictionary containing information about the problem setup. This dict can/should contain the keys MODEL_ID, MODULES, MAX_SOLUTIONS, MAX_COST, TIME_LIMIT, SOLVER, KOCOST, KICOST, REGCOST, GKICOST, GKOCOST

  These entries can be set like this: sd_setup[straindesign.MODEL_ID] = model.id

Returns

(SDSolutions): Strain design solutions

get_gene_reac_sd_assoc(*i=None*)

Get reaction and gene-based strain design solutions, and show which reaction-based solution corresponds to which gene-based.

Often the association is not 1:1 but n:1.

get_gene_reac_sd_assoc_mark_no_ki(*i=None*)

Get reaction and gene-based strain design solutions, but also tag knock-ins that were not made with a 0

Often the association is not 1:1 but n:1.

get_gene_sd(*i=None*)

Get gene-based strain design solutions

get_gene_sd_mark_no_ki(*i=None*)

Get gene-based strain design solutions, but also tag knock-ins that were not made with a 0

get_num_sols()

Get number of solutions

get_reaction_sd(*i=None*)

Get reaction-based strain design solutions

Gene-based intervention sets are translated to the reaction level. This can be helpful to understand the impact of gene interventions. GPR-rules are accounted for automatically.

get_reaction_sd_bnds(*i=None*)

Get reaction-based strain design solutions represented by upper and lower bounds

Knocked-out reactions will show as upper and lower bounds of zero.

get_reaction_sd_mark_no_ki(*i=None*)

Get reaction-based strain design solutions, but also tag knock-ins that were not made with a 0

This can be helpful to analyze gene intervention sets in original metabolic models. GPR-rules are accounted for automatically.

get_strain_design_costs(*i=None*)

Get costs of i-th strain design or of all in a list

get_strain_designs(*i=None*)

Get i-th strain design (intervention set) or all in original format

classmethod load(*filename*)

Load strain design solutions from a file.

save(*filename*)

Save strain design solutions to a file.

straindesign.strainDesignSolutions.**get_subset**(*sd*, *i*)

SDSolutions internal function: getting a subset of solutions

straindesign.strainDesignSolutions.**gpr_eval**(*cj_terms*, *interv*)

SDSolutions internal function: evaluate a GPR term

straindesign.strainDesignSolutions.**strip_non_ki**(*sd*)

SDSolutions internal function: removing non-added reactions or genes

4.9.1.2 Package Contents

class straindesign.**DisableLogger**

Environment in which logging is disabled

CHAPTER

FIVE

REFERENCES:

Ebrahim, A., Lerman, J.A., Palsson, B.O. et al. - COBRApy: COnstraints-Based Reconstruction and Analysis for Python. BMC Syst Biol 7, 74 (2013)

Burgard, A. P., Pharkya, P., & Maranas, C. D. - Optknock: a bilevel programming framework for identifying gene knockout strategies for microbial strain optimization. Biotechnology and bioengineering, 84(6), 647–657 (2003)

Tepper N., Shlomi T. - Predicting metabolic engineering knockout strategies for chemical production: accounting for competing pathways, Bioinformatics. Volume 26, Issue 4, Pages 536–543 (2010)

Jensen K., Broeken V., Lærke Hansen A.S., et al. - OptCouple: Joint simulation of gene knockouts, insertions and medium modifications for prediction of growth-coupled strain designs. Metabolic Engineering Communications, Volume 8 (2019)

Bestuzheva K., Besançon M., Chen W.K. et al. - The SCIP Optimization Suite 8.0. Available at Optimization Online and as ZIB-Report 21-41, (2021)

Marco Terzer, Jörg Stelling, Large-scale computation of elementary flux modes with bit pattern trees, Bioinformatics, Volume 24, Issue 19, (2008), Pages 2229–2235,

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

G

I

J

L

M

N

P

R

S

V

W

Y