cobra Documentation

Release 0.17.1

The cobrapy core team

May 26, 2022

CONTENTS

For installation instructions, please see INSTALL.rst.

Many of the examples below are viewable as IPython notebooks, which can be viewed at nbviewer.

{

“cells”: [

{ “cell_type”: “markdown”, “metadata”: {}, “source”: [

“# Getting Started”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“## Loading a model and inspecting it”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

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

]

}, {

“cell_type”: “code”, “execution_count”: 1, “metadata”: {}, “outputs”: [], “source”: [

“from __future__ import print_functionn”, “n”, “import cobran”, “import cobra.testn”, “n”, “# “ecoli” and “salmonella” are also valid argumentsn”, “model = cobra.test.create_test_model(“textbook”)”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

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

]

}, {

“cell_type”: “code”, “execution_count”: 2, “metadata”: {

“scrolled”: true

}, “outputs”: [

{ “name”: “stdout”, “output_type”: “stream”, “text”: [

“95n”, “72n”, “137n”

]

}

], “source”: [

“print(len(model.reactions))n”, “print(len(model.metabolites))n”, “print(len(model.genes))”

]

```
        }, {
            "cell_type": "markdown", "metadata": {}, "source": [
                "When using [Jupyter notebook](https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/) this type of information is rendered as a table."
            ]
        }, {
            "cell_type": "code", "execution_count": 3, "metadata": {}, "outputs": [
                {
                  "data": {
                    "text/html": [ "n", " <table>n", " <tr>n", " <td><strong>Name</strong></td>n", " <td>e_coli_core</td>n", " </tr><tr>n", " <td><strong>Memory address</strong></td>n",
<<<<<<< HEAD
 " <td>0x01158878d0</td>n",
" </tr><tr>n", " <td><strong>Number of metabolites</strong></td>n", " <td>72</td>n", " </tr><tr>n", " <td><strong>Number of reactions</strong></td>n", " <td>95</td>n", " </tr><tr>n", " <td><strong>Objective expression</strong></td>n", " <td>1.0*Biomass_Ecoli_core - 1.0*Biomass_Ecoli_core_reverse_2cdba</td>n", " </tr><tr>n", " <td><strong>Compartments</strong></td>n", " <td>cytosol, extracellular</td>n", " </tr>n", " </table>"
      ], "text/plain": [
<<<<<<< HEAD
 "<Model e_coli_core at 0x1158878d0>"
                    ]
                  }, "execution_count": 3, "metadata": {}, "output_type": "execute_result"
                }
            ], "source": [
                "model"
            ]
        }, {
            "cell_type": "markdown", "metadata": {}, "source": [
                "Just like a regular list, objects in the *DictList* can be retrieved by index. For example, to get the 30th reaction in the model (at index 29 because of [0-indexing](https://en.wikipedia.org/wiki/Zero-based_numbering)):"
            ]
        }, {
            "cell_type": "code", "execution_count": 4, "metadata": {}, "outputs": [
                {
                  "data": {
                    "text/html": [ "n", " <table>n", " <tr>n", " <td><strong>Reaction identifier</strong></td><td>EX_glu__L_e</td>n", " </tr><tr>n", " <td><strong>Name</strong></td><td>L-Glutamate exchange</td>n", " </tr><tr>n", " <td><strong>Memory address</strong></td>n",
<<<<<<< HEAD
 " <td>0x011615e2e8</td>n",
```

” </tr><tr>n”, ” <td>Stoichiometry</td>n”, ” <td>n”, ” glu__L_e –> n”, ” L-Glutamate –> n”, ” </td>n”, ” </tr><tr>n”, ” <td>GPR</td><td></td>n”, ” </tr><tr>n”, ” <td>Lower bound</td><td>0.0</td>n”, ” </tr><tr>n”, ” <td>Upper bound</td><td>1000.0</td>n”, ” </tr>n”, ” </table>n”, ” “

], “text/plain”: [

<<<<<<< HEAD “<Reaction EX_glu__L_e at 0x11615e2e8>”

]

}, “execution_count”: 4, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“model.reactions[29]”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

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

]

}, {

“cell_type”: “code”, “execution_count”: 5, “metadata”: {}, “outputs”: [

{

“data”: {

“text/html”: [“n”, ” <table>n”, ” <tr>n”, ” <td>Metabolite identifier</td><td>atp_c</td>n”, ” </tr><tr>n”, ” <td>Name</td><td>ATP</td>n”, ” </tr><tr>n”, ” <td>Memory address</td>n”,

<<<<<<< HEAD ” <td>0x01160d4630</td>n”,

” </tr><tr>n”, ” <td>Formula</td><td>C10H12N5O13P3</td>n”, ” </tr><tr>n”, ” <td>Compartment</td><td>c</td>n”, ” </tr><tr>n”, ” <td>In 13 reaction(s)</td><td>n”,

<<<<<<< HEAD ” PPS, ADK1, ATPS4r, GLNS, SUCOAS, GLNabc, PGK, ATPM, PPCK, ACKr, PFK, Biomass_Ecoli_core, PYK</td>n”,

” </tr>n”, ” </table>”

], “text/plain”: [

<<<<<<< HEAD “<Metabolite atp_c at 0x1160d4630>”

]

}, “execution_count”: 5, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“model.metabolites.get_by_id(“atp_c”)”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“As an added bonus, users with an interactive shell such as IPython will be able to tab-complete to list elements inside a list. While this is not recommended behavior for most code because of the possibility for characters like “-” inside ids, this is very useful while in an interactive prompt:”

]

}, {

“cell_type”: “code”, “execution_count”: 6, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [“(-10.0, 1000.0)”

]

}, “execution_count”: 6, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“model.reactions.EX_glc__D_e.bounds”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“## Reactions”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“We will consider the reaction glucose 6-phosphate isomerase, which interconverts glucose 6-phosphate and fructose 6-phosphate. The reaction id for this reaction in our test model is PGI.”

]

}, {

“cell_type”: “code”, “execution_count”: 7, “metadata”: {}, “outputs”: [

{

“data”: {

“text/html”: [“n”, ” <table>n”, ” <tr>n”, ” <td>Reaction identifier</td><td>PGI</td>n”, ” </tr><tr>n”, ” <td>Name</td><td>glucose-6-phosphate isomerase</td>n”, ” </tr><tr>n”, ” <td>Memory address</td>n”,

<<<<<<< HEAD ” <td>0x0116188e48</td>n”,

” </tr><tr>n”, ” <td>Stoichiometry</td>n”, ” <td>n”, ” g6p_c <=> f6p_c n”, ” D-Glucose 6-phosphate <=> D-Fructose 6-phosphate n”, ” </td>n”, ” </tr><tr>n”, ” <td>GPR</td><td>b4025</td>n”, ” </tr><tr>n”, ” <td>Lower bound</td><td>-1000.0</td>n”, ” </tr><tr>n”, ” <td>Upper bound</td><td>1000.0</td>n”, ” </tr>n”, ” </table>n”, ” “

], “text/plain”: [

<<<<<<< HEAD “<Reaction PGI at 0x116188e48>”

]

}, “execution_count”: 7, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“pgi = model.reactions.get_by_id(“PGI”)n”, “pgi”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“We can view the full name and reaction catalyzed as strings”

]

}, {

“cell_type”: “code”, “execution_count”: 8, “metadata”: {}, “outputs”: [

{ “name”: “stdout”, “output_type”: “stream”, “text”: [

“glucose-6-phosphate isomerasen”, “g6p_c <=> f6p_cn”

]

}

], “source”: [

“print(pgi.name)n”, “print(pgi.reaction)”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“We can also view reaction upper and lower bounds. Because the *pgi.lower_bound* < 0, and *pgi.upper_bound* > 0, *pgi* is reversible.”

]

}, {

“cell_type”: “code”, “execution_count”: 9, “metadata”: {}, “outputs”: [

{ “name”: “stdout”, “output_type”: “stream”, “text”: [

“-1000.0 < pgi < 1000.0n”, “Truen”

]

}

], “source”: [

“print(pgi.lower_bound, “< pgi <”, pgi.upper_bound)n”, “print(pgi.reversibility)”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“The lower and upper bound of reactions can also be modified, and the reversibility attribute will automatically be updated. The preferred method for manipulating bounds is using *reaction.bounds*, e.g.”

]

}, {

“cell_type”: “code”, “execution_count”: 10, “metadata”: {}, “outputs”: [

{ “name”: “stdout”, “output_type”: “stream”, “text”: [

“0 < pgi < 1000.0n”, “Reversibility after modification: Falsen”, “Reversibility after resetting: Truen”

]

}

], “source”: [

“old_bounds = pgi.boundsn”, “pgi.bounds = (0, 1000.0)n”, “print(pgi.lower_bound, “< pgi <”, pgi.upper_bound)n”, “print(“Reversibility after modification:”, pgi.reversibility)n”, “pgi.bounds = old_boundsn”, “print(“Reversibility after resetting:”, pgi.reversibility)”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Bounds can also be modified one-at-a-time using *reaction.lower_bound* or *reaction.upper_bound*. This approach is less desirable than setting both bounds simultaneously with *reaction.bounds* because a user might accidently set a lower bound *higher* than an upper bound (or vice versa). Currently, cobrapy will automatically adjust the other bound (e.g. the bound the user didn’t manually adjust) so that this violation doesn’t occur, but this feature may be removed in the near future. “

]

}, {

“cell_type”: “code”, “execution_count”: 11, “metadata”: {}, “outputs”: [

{ “name”: “stdout”, “output_type”: “stream”, “text”: [

“Upper bound prior to setting new lower bound: 1000.0n”, “Upper bound after setting new lower bound: 1100n”

]

}

], “source”: [

“old_bounds = pgi.boundsn”, “print(‘Upper bound prior to setting new lower bound:’, pgi.upper_bound)n”, “pgi.lower_bound = 1100n”, “print(‘Upper bound after setting new lower bound:’, pgi.upper_bound)n”, “pgi.bounds = old_bounds”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“We can also ensure the reaction is mass balanced. This function will return elements which violate mass balance. If it comes back empty, then the reaction is mass balanced.”

]

}, {

“cell_type”: “code”, “execution_count”: 12, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [“{}”

]

}, “execution_count”: 12, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“pgi.check_mass_balance()”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

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

]

}, {

“cell_type”: “code”, “execution_count”: 13, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [“‘g6p_c + h_c <=> f6p_c’”

]

}, “execution_count”: 13, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“pgi.add_metabolites({model.metabolites.get_by_id(“h_c”): -1})n”, “pgi.reaction”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“The reaction is no longer mass balanced”

]

}, {

“cell_type”: “code”, “execution_count”: 14, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [“{‘charge’: -1.0, ‘H’: -1.0}”

]

}, “execution_count”: 14, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“pgi.check_mass_balance()”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

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

]

}, {

“cell_type”: “code”, “execution_count”: 15, “metadata”: {}, “outputs”: [

{ “name”: “stdout”, “output_type”: “stream”, “text”: [

“g6p_c <=> f6p_cn”, “{}n”

]

}

], “source”: [

“pgi.subtract_metabolites({model.metabolites.get_by_id(“h_c”): -1})n”, “print(pgi.reaction)n”, “print(pgi.check_mass_balance())”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“It is also possible to build the reaction from a string. However, care must be taken when doing this to ensure reaction id’s match those in the model. The direction of the arrow is also used to update the upper and lower bounds.”

]

}, {

“cell_type”: “code”, “execution_count”: 16, “metadata”: {}, “outputs”: [

{ “name”: “stdout”, “output_type”: “stream”, “text”: [

“unknown metabolite ‘green_eggs’ createdn”, “unknown metabolite ‘ham’ createdn”

]

}

], “source”: [

“pgi.reaction = “g6p_c –> f6p_c + h_c + green_eggs + ham””

]

}, {

"cell_type": "code", "execution_count": 17, "metadata": {}, "outputs": [

{

"data": {

"text/plain": ["'g6p_c –> f6p_c + green_eggs + h_c + ham'"

]

}, "execution_count": 17, "metadata": {}, "output_type": "execute_result"

}

], "source": [

"pgi.reaction"

]

}, {

"cell_type": "code", "execution_count": 18, "metadata": {}, "outputs": [

{

"data": {

"text/plain": ["'g6p_c <=> f6p_c'"

]

}, "execution_count": 18, "metadata": {}, "output_type": "execute_result"

}

], "source": [

"pgi.reaction = "g6p_c <=> f6p_c"n", "pgi.reaction"

]

}, {

"cell_type": "markdown", "metadata": {}, "source": [

"## Metabolites"

]

}, {

"cell_type": "markdown", "metadata": {}, "source": [

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

]

}, {

"cell_type": "code", "execution_count": 19, "metadata": {}, "outputs": [

{

"data": {

"text/html": ["n", " <table>n", " <tr>n", " <td>Metabolite identifier</td><td>atp_c</td>n", " </tr><tr>n", "

<td>Name</td><td>ATP</td>n”, ”
</tr><tr>n”, ” <td>Memory address</td>n”,

<<<<<<< HEAD ” <td>0x01160d4630</td>n”,

” </tr><tr>n”, ” <td>Formula</td><td>C10H12N5O13P3</td>n”, ” </tr><tr>n”, ” <td>Compartment</td><td>c</td>n”, ” </tr><tr>n”, ” <td>In 13 reaction(s)</td><td>n”,

<<<<<<< HEAD ” PPS, ADK1, ATPS4r, GLNS, SUCOAS, GLNabc, PGK, ATPM, PPCK, ACKr, PFK, Biomass_Ecoli_core, PYK</td>n”,

” </tr>n”, ” </table>”

], “text/plain”: [

<<<<<<< HEAD “<Metabolite atp_c at 0x1160d4630>”

]

}, “execution_count”: 19, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“atp = model.metabolites.get_by_id(“atp_c”)n”, “atp”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

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

]

}, {

“cell_type”: “code”, “execution_count”: 20, “metadata”: {}, “outputs”: [

{ “name”: “stdout”, “output_type”: “stream”, “text”: [

“ATPn”, “cn”

]

}

], “source”: [

“print(atp.name)n”, “print(atp.compartment)”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

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

]

}, {

“cell_type”: “code”, “execution_count”: 21, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [“-4”

```
                ]
            }, "execution_count": 21, "metadata": {}, "output_type": "execute_result"
        }
    ], "source": [
        "atp.charge"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "We can see the chemical formula for the metabolite as well."
    ]
}, {
    "cell_type": "code", "execution_count": 22, "metadata": {}, "outputs": [
        { "name": "stdout", "output_type": "stream", "text": [
            "C10H12N5O13P3n"
        ]
        }
    ], "source": [
        "print(atp.formula)"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "The reactions attribute gives a *frozenset* of all reactions using the given metabolite. We can use this to count the number of reactions which use atp."
    ]
}, {
    "cell_type": "code", "execution_count": 23, "metadata": {}, "outputs": [
        {
            "data": {
                "text/plain": [ "13"
                ]
            }, "execution_count": 23, "metadata": {}, "output_type": "execute_result"
        }
    ], "source": [
        "len(atp.reactions)"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "A metabolite like glucose 6-phosphate will participate in fewer reactions."
```

```
        ]
    }, {
        "cell_type": "code", "execution_count": 24, "metadata": {}, "outputs": [
            {
                "data": {  "text/plain": [
<<<<<<< HEAD "frozenset({<Reaction Biomass_Ecoli_core at 0x1161337b8>,n", " <Reaction G6PDH2r at 0x1160a3a20>,n", " <Reaction GLCpts at 0x1160a3da0>,n", " <Reaction PGI at 0x116188e48>})"
    " <Reaction GLCpts at 0x117a9d0f0>,n", " <Reaction PGI at 0x117afacc0>})"
>>>>>>> origin/devel
                ]
            }, "execution_count": 24, "metadata": {}, "output_type": "execute_result"
            }
        ], "source": [
            "model.metabolites.get_by_id("g6p_c").reactions"
        ]
    }, {
        "cell_type": "markdown", "metadata": {}, "source": [
            "## Genes"
        ]
    }, {
        "cell_type": "markdown", "metadata": {}, "source": [
            "The *gene_reaction_rule* is a boolean representation of the gene requirements for this reaction to be active as described in [Schellenberger et al 2011 Nature Protocols 6(9):1290-307](http://dx.doi.org/doi:10.1038/nprot.2011.308).n", "n", "The GPR is stored as the gene_reaction_rule for a Reaction object as a string."
        ]
    }, {
        "cell_type": "code", "execution_count": 25, "metadata": {}, "outputs": [
            {
                "data": {
                    "text/plain": [  "'b4025'"
                    ]
                }, "execution_count": 25, "metadata": {}, "output_type": "execute_result"
            }
        ], "source": [
            "gpr = pgi.gene_reaction_rulen", "gpr"
        ]
    }, {
        "cell_type": "markdown", "metadata": {}, "source": [
```

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

]

}, {

“cell_type”: “code”, “execution_count”: 26, “metadata”: {}, “outputs”: [

{

“data”: { “text/plain”: [

<<<<<<< HEAD “frozenset({<Gene b4025 at 0x11610a2b0>})”

]

}, “execution_count”: 26, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“pgi.genes”

]

}, {

“cell_type”: “code”, “execution_count”: 27, “metadata”: {}, “outputs”: [

{

“data”: {

“text/html”: [“n”, ” <table>n”, ” <tr>n”, ” <td>Gene identifier</td><td>b4025</td>n”, ” </tr><tr>n”, ” <td>Name</td><td>pgi</td>n”, ” </tr><tr>n”, ” <td>Memory address</td>n”,

<<<<<<< HEAD ” <td>0x011610a2b0</td>n”,

” </tr><tr>n”, ” <td>Functional</td><td>True</td>n”, ” </tr><tr>n”, ” <td>In 1 reaction(s)</td><td>n”, ” PGI</td>n”, ” </tr>n”, ” </table>”

], “text/plain”: [

<<<<<<< HEAD “<Gene b4025 at 0x11610a2b0>”

]

}, “execution_count”: 27, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“pgi_gene = model.genes.get_by_id(“b4025”)n”, “pgi_gene”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Each gene keeps track of the reactions it catalyzes”

]

}, {

“cell_type”: “code”, “execution_count”: 28, “metadata”: {}, “outputs”: [

```
                {
                    "data": {  "text/plain": [
<<<<<<< HEAD  "frozenset({<Reaction PGI at 0x116188e48>})"
                        ]
                    }, "execution_count":  28, "metadata":  {}, "output_type":  "exe-
                    cute_result"
                }
            ], "source": [
                "pgi_gene.reactions"
            ]
        }, {
            "cell_type": "markdown", "metadata": {}, "source": [
                "Altering the gene_reaction_rule will create new gene objects if necessary and
                update all relationships."
            ]
        }, {
            "cell_type": "code", "execution_count": 29, "metadata": {}, "outputs": [
                {
                    "data": {  "text/plain": [
<<<<<<< HEAD  "frozenset({<Gene eggs at 0x1160245c0>, <Gene spam at 0x116024080>})"
                        ]
                    }, "execution_count":  29, "metadata":  {}, "output_type":  "exe-
                    cute_result"
                }
            ], "source": [
                "pgi.gene_reaction_rule = "(spam or eggs)"n", "pgi.genes"
            ]
        }, {
            "cell_type": "code", "execution_count": 30, "metadata": {}, "outputs": [
                {
                    "data": {
                        "text/plain": [  "frozenset()"
                        ]
                    }, "execution_count":  30, "metadata":  {}, "output_type":  "exe-
                    cute_result"
                }
            ], "source": [
                "pgi_gene.reactions"
            ]
        }, {
```

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Newly created genes are also added to the model”

]

}, {

“cell_type”: “code”, “execution_count”: 31, “metadata”: {}, “outputs”: [

{

“data”: {

“text/html”: [“n”, ” <table>n”, ” <tr>n”, ” <td>Gene identifier</td><td>spam</td>n”, ” </tr><tr>n”, ” <td>Name</td><td></td>n”, ” </tr><tr>n”, ” <td>Memory address</td>n”,

<<<<<<< HEAD ” <td>0x0116024080</td>n”,

” </tr><tr>n”, ” <td>Functional</td><td>True</td>n”, ” </tr><tr>n”, ” <td>In 1 reaction(s)</td><td>n”, ” PGI</td>n”, ” </tr>n”, ” </table>”

], “text/plain”: [

<<<<<<< HEAD “<Gene spam at 0x116024080>”

]

}, “execution_count”: 31, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“model.genes.get_by_id(“spam”)”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“The *delete_model_genes* function will evaluate the GPR and set the upper and lower bounds to 0 if the reaction is knocked out. This function can preserve existing deletions or reset them using the *cumulative_deletions* flag.”

]

}, {

“cell_type”: “code”, “execution_count”: 32, “metadata”: {}, “outputs”: [

{ “name”: “stdout”, “output_type”: “stream”, “text”: [

“after 1 KO: -1000 < flux_PGI < 1000n”, “after 2 KO: 0 < flux_PGI < 0n”

]

}

], “source”: [

“cobra.manipulation.delete_model_genes(n”, ” model, [“spam”], cumulative_deletions=True)n”, “print(“after 1 KO: %4d < flux_PGI

```
        < %4d" % (pgi.lower_bound, pgi.upper_bound))n", "n", "cobra.manipulation.delete_model_genes(n", " model, ["eggs"], cumulative_deletions=True)n", "print("after 2 KO: %4d < flux_PGI < %4d" % (pgi.lower_bound, pgi.upper_bound))"
      ]
    }, {
      "cell_type": "markdown", "metadata": {}, "source": [
        "The undelete_model_genes can be used to reset a gene deletion"
      ]
    }, {
      "cell_type": "code", "execution_count": 33, "metadata": {}, "outputs": [
        { "name": "stdout", "output_type": "stream", "text": [
            "-1000 < pgi < 1000n"
          ]
        }
      ], "source": [
        "cobra.manipulation.undelete_model_genes(model)n",
        "print(pgi.lower_bound, "< pgi <", pgi.upper_bound)"
      ]
    }, {
      "cell_type": "markdown", "metadata": {}, "source": [
        "## Making changes reversibly using models as contexts"
      ]
    }, {
      "cell_type": "markdown", "metadata": {}, "source": [
        "Quite often, one wants to make small changes to a model and evaluate the impacts of these. For example, we may want to knock-out all reactions sequentially, and see what the impact of this is on the objective function. One way of doing this would be to create a new copy of the model before each knock-out with *model.copy()*. However, even with small models, this is a very slow approach as models are quite complex objects. Better then would be to do the knock-out, optimizing and then manually resetting the reaction bounds before proceeding with the next reaction. Since this is such a common scenario however, cobrapy allows us to use the model as a context, to have changes reverted automatically."
      ]
    }, {
      "cell_type": "code", "execution_count": 34, "metadata": {}, "outputs": [
        { "name": "stdout", "output_type": "stream", "text": [
            "ACALD blocked (bounds: (0, 0)), new growth rate 0.873922n", "ACALDt blocked (bounds: (0, 0)), new growth rate 0.873922n", "ACKr blocked (bounds: (0, 0)), new growth rate 0.873922n", "ACONTa blocked (bounds:
```

(0, 0)), new growth rate -0.000000n", "ACONTb blocked (bounds: (0, 0)), new growth rate -0.000000n"

]

}

], "source": [

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

]

}, {

"cell_type": "markdown", "metadata": {}, "source": [

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

]

}, {

"cell_type": "code", "execution_count": 35, "metadata": {}, "outputs": [

{

"data": {

"text/plain": ["[(-1000.0, 1000.0),n", " (-1000.0, 1000.0),n", " (-1000.0, 1000.0),n", " (-1000.0, 1000.0),n", " (-1000.0, 1000.0)]"

]

}, "execution_count": 35, "metadata": {}, "output_type": "execute_result"

}

], "source": [

"[reaction.bounds for reaction in model.reactions[:5]]"

]

}, {

"cell_type": "markdown", "metadata": {}, "source": [

"Nested contexts are also supported"

]

}, {

"cell_type": "code", "execution_count": 36, "metadata": {}, "outputs": [

{ "name": "stdout", "output_type": "stream", "text": [

"original objective: 1.0*Biomass_Ecoli_core - 1.0*Biomass_Ecoli_core_reverse_2cdban", "print objective in first context: -1.0*ATPM_reverse_5b752 + 1.0*ATPMn", "print objective in second context: 1.0*ACALD - 1.0*ACALD_reverse_fda2bn", "objective after exiting second context: - 1.0*ATPM_reverse_5b752 + 1.0*ATPMn", "back

```
                    to original objective: 1.0*Biomass_Ecoli_core -
                    1.0*Biomass_Ecoli_core_reverse_2cdban"

                ]

            }

        ], "source": [

            "print('original objective: ', model.objective.expression)\n", "with
            model:\n", " model.objective = 'ATPM'\n", " print('print objective
            in first context:', model.objective.expression)\n", " with model:\n",
            " model.objective = 'ACALD'\n", " print('print objective in sec-
            ond context:', model.objective.expression)\n", " print('objective af-
            ter exiting second context:',\n", " model.objective.expression)\n",
            "print('back to original objective:', model.objective.expression)"

        ]

    }, {

        "cell_type": "markdown", "metadata": {}, "source": [

            "Most methods that modify the model are supported like this includ-
            ing adding and removing reactions and metabolites and setting the
            objective. Supported methods and functions mention this in the cor-
            responding documentation."

        ]

    }, {

        "cell_type": "markdown", "metadata": {}, "source": [

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

        ]

    }, {

        "cell_type": "code", "execution_count": 37, "metadata": {}, "outputs": [],
        "source": [

            "with model as inner:\n", " inner.reactions.PFK.knock_out"

        ]

    }

], "metadata": {

    "kernelspec": { "display_name": "Python 3", "language": "python", "name":
        "python3"

    }, "language_info": {

        "codemirror_mode": { "name": "ipython", "version": 3

        }, "file_extension": ".py", "mimetype": "text/x-python", "name": "python",
        "nbconvert_exporter": "python", "pygments_lexer": "ipython3", "version":
        "3.6.5"

    }

}, "nbformat": 4, "nbformat_minor": 1

}
```

CHAPTER

ONE

GLOBAL CONFIGURATION

With cobra > 0.13.4, we introduce a global configuration object. For now, you can configure default reaction bounds and optimization solver which will be respected by newly created reactions and models.

1.1 The configuration object

You can get a configuration object*1* in the following way:

[1]:
```
import cobra
```

[2]:
```
cobra_config = cobra.Configuration()
```

1The configuration object is a singleton. That means only one instance can exist and it is respected everywhere in COBRApy.

1.2 Reaction bounds

The object has the following attributes which you can inspect but also change as desired.

[3]:
```
cobra_config.lower_bound
```

[3]:
```
-1000.0
```

[4]:
```
cobra_config.upper_bound
```

[4]:
```
1000.0
```

[5]:
```
cobra_config.bounds
```

[5]:
```
(-1000.0, 1000.0)
```

1.2.1 Changing bounds

If you modify the above values before creating a reaction they will be used.

[6]:
```
cobra_config.bounds = -10, 20
```

[7]:
```
cobra.Reaction("R1")
```

[7]:
```
<Reaction R1 at 0x7f0426135fd0>
```

Please note that by default reactions are irreversible. You can change this behavior by unsetting the lower bound argument.

```
[8]: cobra.Reaction("R2", lower_bound=None)
[8]: <Reaction R2 at 0x7f04260d4438>
```

N.B.: Most models define reaction bounds explicitly which takes precedence over the configured values.

```
[9]: from cobra.test import create_test_model

[10]: model = create_test_model("textbook")

[11]: model.reactions.ACt2r
[11]: <Reaction ACt2r at 0x7f042607c780>
```

1.3 Solver

You can define the default solver used by newly instantiated models. The default solver depends on your environment. In order we test for the availability of Gurobi, CPLEX, and GLPK. GLPK is assumed to always be present in the environment.

```
[12]: model.solver
[12]: <optlang.cplex_interface.Model at 0x7f04260d4b00>
```

1.3.1 Changing solver

```
[13]: cobra_config.solver = "glpk_exact"

[14]: new_model = create_test_model("textbook")

[15]: new_model.solver
[15]: <optlang.glpk_exact_interface.Model at 0x7f04260d47b8>
```

Changing global configuration values is mostly useful at the beginning of a work session.

CHAPTER

TWO

BUILDING A MODEL

This simple example demonstrates how to create a model, create a reaction, and then add the reaction to the model.

We'll use the '3OAS140' reaction from the STM_1.0 model:

1.0 malACP[c] + 1.0 h[c] + 1.0 ddcaACP[c] → 1.0 co2[c] + 1.0 ACP[c] + 1.0 3omrsACP[c]

First, create the model and reaction.

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

[2]:
```
from cobra import Model, Reaction, Metabolite
# Best practise: SBML compliant IDs
model = Model('example_model')

reaction = Reaction('3OAS140')
reaction.name = '3 oxoacyl acyl carrier protein synthase n C140 '
reaction.subsystem = 'Cell Envelope Biosynthesis'
reaction.lower_bound = 0.  # This is the default
reaction.upper_bound = 1000.  # This is the default
```

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

[3]:
```
ACP_c = Metabolite(
    'ACP_c',
    formula='C11H21N2O7PRS',
    name='acyl-carrier-protein',
    compartment='c')
omrsACP_c = Metabolite(
    '3omrsACP_c',
    formula='C25H45N2O9PRS',
    name='3-Oxotetradecanoyl-acyl-carrier-protein',
    compartment='c')
co2_c = Metabolite('co2_c', formula='CO2', name='CO2', compartment='c')
malACP_c = Metabolite(
    'malACP_c',
    formula='C14H22N2O10PRS',
    name='Malonyl-acyl-carrier-protein',
    compartment='c')
h_c = Metabolite('h_c', formula='H', name='H', compartment='c')
ddcaACP_c = Metabolite(
    'ddcaACP_c',
    formula='C23H43N2O8PRS',
    name='Dodecanoyl-ACP-n-C120ACP',
    compartment='c')
```

Adding metabolites to a reaction requires using a dictionary of the metabolites and their stoichiometric coefficients. A group of metabolites can be added all at once, or they can be added one at a time.

[4]:
```
reaction.add_metabolites({
    malACP_c: -1.0,
    h_c: -1.0,
    ddcaACP_c: -1.0,
    co2_c: 1.0,
    ACP_c: 1.0,
    omrsACP_c: 1.0
})

reaction.reaction  # This gives a string representation of the reaction
```

[4]:
```
'ddcaACP_c + h_c + malACP_c --> 3omrsACP_c + ACP_c + co2_c'
```

The gene_reaction_rule is a boolean representation of the gene requirements for this reaction to be active as described in Schellenberger et al 2011 Nature Protocols 6(9):1290-307. We will assign the gene reaction rule string, which will automatically create the corresponding gene objects.

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

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

At this point in time, the model is still empty

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

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

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

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

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

```
1 reaction
6 metabolites
2 genes
```

We can iterate through the model objects to observe the contents

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

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

print("")
print("Genes")
print("-----")
for x in model.genes:
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

```
-1.0*3OAS140_reverse_65ddc + 1.0*3OAS140
max
```

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

CHAPTER

THREE

READING AND WRITING MODELS

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

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

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

data_dir = cobra.test.data_dir

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

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

```
mini test files:
mini.json, mini.mat, mini.pickle, mini.yml, mini_cobra.xml, mini_fbc1.xml, mini_
↪fbc2.xml, mini_fbc2.xml.bz2, mini_fbc2.xml.gz
```

3.1 SBML

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

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

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

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

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

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

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

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

[4]:
```
cobra.io.read_sbml_model(join(data_dir, "mini_cobra.xml"))
```

[4]:
```
<Model mini_textbook at 0x112fa6b38>
```

[5]:
```
cobra.io.write_sbml_model(
    textbook_model, "test_cobra.xml", use_fbc_package=False)
```

3.2 JSON

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

[6]:
```
cobra.io.load_json_model(join(data_dir, "mini.json"))
```

[6]:
```
<Model mini_textbook at 0x113061080>
```

[7]:
```
cobra.io.save_json_model(textbook_model, "test.json")
```

3.3 YAML

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

[8]:
```
cobra.io.load_yaml_model(join(data_dir, "mini.yml"))
```

[8]:
```
<Model mini_textbook at 0x113013390>
```

[9]:
```
cobra.io.save_yaml_model(textbook_model, "test.yml")
```

3.4 MATLAB

Often, models may be imported and exported solely for the purposes of working with the same models in cobrapy and the MATLAB cobra toolbox. MATLAB has its own ".mat" format for storing variables. Reading and writing to these mat files from python requires scipy.

A mat file can contain multiple MATLAB variables. Therefore, the variable name of the model in the MATLAB file can be passed into the reading function:

[10]:
```
cobra.io.load_matlab_model(
    join(data_dir, "mini.mat"), variable_name="mini_textbook")
```

[10]:
```
<Model mini_textbook at 0x113000b70>
```

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

[11]:
```
cobra.io.load_matlab_model(join(data_dir, "mini.mat"))
```

[11]:
```
<Model mini_textbook at 0x113758438>
```

Saving models to mat files is also relatively straightforward

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

3.5 Pickle

Cobra models can be serialized using the python serialization format, pickle.

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

CHAPTER

FOUR

SIMULATING WITH FBA

Simulations using flux balance analysis can be solved using `Model.optimize()`. This will maximize or minimize (maximizing is the default) flux through the objective reactions.

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

4.1 Running FBA

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

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

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

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

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

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

[3]:
```
0.8739215069684307
```

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

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

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

[4]:
```
0.8739215069684307
```

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

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

[5]:
```
0.8739215069684307
```

4.1.1 Analyzing FBA solutions

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

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

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

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

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

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

CONSUMING REACTIONS -- Nicotinamide adenine dinucleotide - reduced (nadh_c)
---------------------------------------------------------------------------
%       FLUX  RXN ID      REACTION
----  ------  ----------  ----------------------------------------------------
100%   38.5   NADH16      4.0 h_c + nadh_c + q8_c --> 3.0 h_e + nad_c + q...
```

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

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

```
PRODUCING REACTIONS -- ATP (atp_c)
----------------------------------
%      FLUX  RXN ID      REACTION
---  ------  ----------  ----------------------------------------------------
67%  45.5    ATPS4r      adp_c + 4.0 h_e + pi_c <=> atp_c + h2o_c + 3.0 h_c
23%  16      PGK         3pg_c + atp_c <=> 13dpg_c + adp_c
7%    5.06   SUCOAS      atp_c + coa_c + succ_c <=> adp_c + pi_c + succoa_c
3%    1.76   PYK         adp_c + h_c + pep_c --> atp_c + pyr_c

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

4.2 Changing the Objectives

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

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

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

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

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

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

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

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

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

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

[12]:
```
174.99999999999966
```

We can also have more complicated objectives including quadratic terms.

4.3 Running FVA

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

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

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

[14]:
```
             maximum       minimum
ACALD  -2.208811e-30 -5.247085e-14
ACALDt  0.000000e+00 -5.247085e-14
ACKr    0.000000e+00 -8.024953e-14
ACONTa  2.000000e+01  2.000000e+01
ACONTb  2.000000e+01  2.000000e+01
ACt2r   0.000000e+00 -8.024953e-14
ADK1    3.410605e-13  0.000000e+00
AKGDH   2.000000e+01  2.000000e+01
AKGt2r  0.000000e+00 -2.902643e-14
ALCD2x  0.000000e+00 -4.547474e-14
```

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

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

[15]:
```
             maximum   minimum
ACALD   0.000000e+00 -2.692308
ACALDt  0.000000e+00 -2.692308
ACKr    6.635712e-30 -4.117647
ACONTa  2.000000e+01  8.461538
ACONTb  2.000000e+01  8.461538
ACt2r   0.000000e+00 -4.117647
ADK1    1.750000e+01  0.000000
AKGDH   2.000000e+01  2.500000
AKGt2r  2.651196e-16 -1.489362
ALCD2x  0.000000e+00 -2.333333
```

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

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

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

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

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

4.3.1 Running FVA in summary methods

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

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

```
IN FLUXES                    OUT FLUXES                    OBJECTIVES
---------------------------  ----------------------------  ------------
id          Flux  Range      id          Flux  Range       ATPM  175
--------  ------  ---------  --------  ------  ----------
o2_e          60  [55.9, 60]  co2_e         60  [54.2, 60]
glc__D_e      10  [9.5, 10]   h2o_e         60  [54.2, 60]
nh4_e          0  [0, 0.673]  for_e          0  [0, 5.83]
pi_e           0  [0, 0.171]  h_e            0  [0, 5.83]
                              ac_e           0  [0, 2.06]
                              acald_e        0  [0, 1.35]
                              pyr_e          0  [0, 1.35]
                              etoh_e         0  [0, 1.17]
                              lac__D_e       0  [0, 1.13]
                              succ_e         0  [0, 0.875]
                              akg_e          0  [0, 0.745]
                              glu__L_e       0  [0, 0.673]
```

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

4.4 Running pFBA

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

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

These functions should give approximately the same objective value.

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

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

4.5 Running geometric FBA

Geometric FBA finds a unique optimal flux distribution which is central to the range of possible fluxes. For more details on geometric FBA, please see K Smallbone, E Simeonidis (2009).

[22]:
```
geometric_fba_sol = cobra.flux_analysis.geometric_fba(model)
geometric_fba_sol
```

[22]:
```
<Solution 0.000 at 0x116dfcc88>
```

CHAPTER

FIVE

SIMULATING DELETIONS

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

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

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

5.1 Knocking out single genes and reactions

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

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

```
complete model:  <Solution 0.874 at 0x7f41bb363550>
pfk knocked out:  <Solution 0.704 at 0x7f41bb363710>
```

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

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

```
complete model:  <Solution 0.874 at 0x7f41bb35bf60>
pfkA knocked out:  <Solution 0.874 at 0x7f41bb35bd68>
pfkB knocked out:  <Solution 0.704 at 0x7f41bb35bf98>
```

5.2 Single Deletions

Perform all single gene deletions on a model

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

These can also be done for only a subset of genes

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

[5]:

ids	growth	status
(b0356)	0.873922	optimal
(b0351)	0.873922	optimal
(b1849)	0.873922	optimal
(b2296)	0.873922	optimal
(b2587)	0.873922	optimal
(b0726)	0.858307	optimal
(b1276)	0.873922	optimal
(b3115)	0.873922	optimal
(b1478)	0.873922	optimal
(b0474)	0.873922	optimal
(b1241)	0.873922	optimal
(b0118)	0.873922	optimal
(b0116)	0.782351	optimal
(b0727)	0.858307	optimal
(b3735)	0.374230	optimal
(b3733)	0.374230	optimal
(b3734)	0.374230	optimal
(b3736)	0.374230	optimal
(s0001)	0.211141	optimal
(b3732)	0.374230	optimal

This can also be done for reactions

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

[6]:

ids	growth	status
(Biomass_Ecoli_core)	0.000000e+00	optimal
(ETOHt2r)	8.739215e-01	optimal
(ATPM)	9.166475e-01	optimal
(ACALD)	8.739215e-01	optimal
(EX_ac_e)	8.739215e-01	optimal
(ALCD2x)	8.739215e-01	optimal
(ACt2r)	8.739215e-01	optimal
(ACALDt)	8.739215e-01	optimal
(ACONTb)	8.988837e-16	optimal
(AKGt2r)	8.739215e-01	optimal
(ACONTa)	7.726020e-15	optimal
(AKGDH)	8.583074e-01	optimal
(ACKr)	8.739215e-01	optimal
(ADK1)	8.739215e-01	optimal
(CO2t)	4.616696e-01	optimal
(D_LACt2)	8.739215e-01	optimal
(CYTBD)	2.116629e-01	optimal
(CS)	7.726020e-15	optimal
(ENO)	1.937120e-16	optimal
(ATPS4r)	3.742299e-01	optimal

5.3 Double Deletions

Double deletions run in a similar way.

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

[7]:
```
                growth   status
ids
(b2465, b2464)  0.8739  optimal
(b3919)         0.7040  optimal
(b2935)         0.8739  optimal
(b2935, b0008)  0.8739  optimal
(b2464)         0.8739  optimal
(b0008, b2464)  0.8739  optimal
(b3919, b2464)  0.7040  optimal
(b2935, b3919)  0.7040  optimal
(b2465, b3919)  0.7040  optimal
(b3919, b0008)  0.7040  optimal
(b2935, b2464)  0.8739  optimal
(b2465)         0.8739  optimal
(b2465, b2935)  0.8739  optimal
(b2465, b0008)  0.8739  optimal
(b0008)         0.8739  optimal
```

By default, the double deletion function will automatically use multiprocessing, splitting the task over up to 4 cores if they are available. The number of cores can be manually specified as well. Setting use of a single core will disable use of the multiprocessing library, which often aids debugging.

[8]:
```
start = time()  # start timer()
double_gene_deletion(
    ecoli_model, ecoli_model.genes[:25], processes=2)
t1 = time() - start
print("Double gene deletions for 200 genes completed in "
      "%.2f sec with 2 cores" % t1)

start = time()  # start timer()
double_gene_deletion(
    ecoli_model, ecoli_model.genes[:25], processes=1)
t2 = time() - start
print("Double gene deletions for 200 genes completed in "
      "%.2f sec with 1 core" % t2)

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

```
Double gene deletions for 200 genes completed in 2.53 sec with 2 cores
Double gene deletions for 200 genes completed in 4.09 sec with 1 core
Speedup of 1.62x
```

Double deletions can also be run for reactions.

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

[9]:
```
                   growth   status
ids
(ACt2r)            0.8739  optimal
(ACONTa, ACONTb)   0.0000  optimal
(ACONTb)           0.0000  optimal
(ADK1, ACONTa)     0.0000  optimal
(ADK1)             0.8739  optimal
(ACKr, ACt2r)      0.8739  optimal
(ACONTa)           0.0000  optimal
```

(continues on next page)

(continued from previous page)

```
(ADK1, ACONTb)     0.0000  optimal
(ACKr)             0.8739  optimal
(ACKr, ACONTa)     0.0000  optimal
(ACt2r, ACONTb)    0.0000  optimal
(ADK1, ACt2r)      0.8739  optimal
(ACKr, ACONTb)     0.0000  optimal
(ACONTa, ACt2r)    0.0000  optimal
(ACKr, ADK1)       0.8739  optimal
```

CHAPTER

SIX

PRODUCTION ENVELOPES

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

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

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

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

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

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

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

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

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

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

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

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

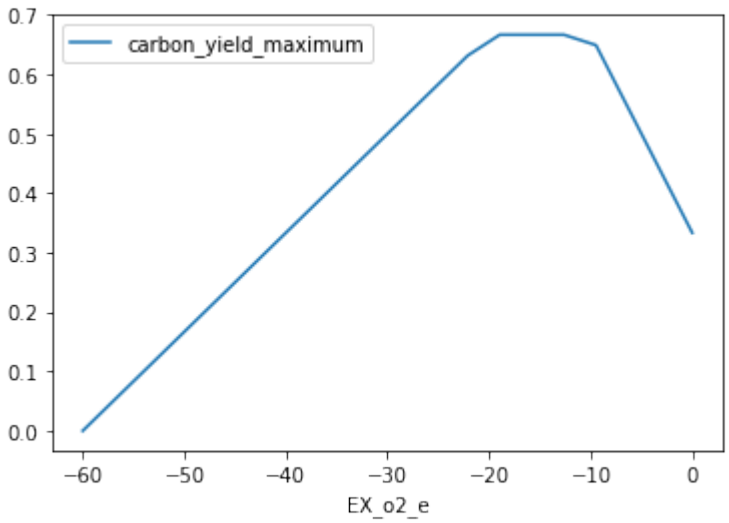
[5]:
```
prod_env.head()
```

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

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

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

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



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

CHAPTER

SEVEN

FLUX SAMPLING

7.1 Basic usage

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

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

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

[1]:
```
      ACALD    ACALDt      ACKr    ACONTa    ACONTb     ACt2r       ADK1  \
0 -2.060626 -0.766231 -1.746726  6.136642  6.136642 -1.746726  13.915541
1 -1.518217 -1.265778 -0.253608  9.081331  9.081331 -0.253608   7.194475
2 -3.790368 -1.292543 -0.457502  9.340755  9.340755 -0.457502  23.435794
3 -5.173189 -4.511308 -2.333962  7.364836  7.364836 -2.333962  11.725401
4 -6.787036 -5.645414 -1.521566  6.373250  6.373250 -1.521566   4.823373

      AKGDH    AKGt2r    ALCD2x  ...       RPI   SUCCt2_2     SUCCt3  \
0  2.174506 -0.242290 -1.294395  ... -6.117270  33.457990  34.319917
1  5.979050 -0.225992 -0.252439  ... -5.072733  39.902893  40.343192
2  1.652395 -0.333891 -2.497825  ... -0.674220   0.153276   1.506968
3  2.504044 -0.051420 -0.661881  ... -0.681200   7.506732   9.110446
4  3.452123 -0.126943 -1.141621  ... -0.510598   9.307459  10.941500

        SUCDi    SUCOAS      TALA      THD2      TKT1      TKT2       TPI
0  704.483302 -2.174506  6.109618  0.230408  6.109618  6.106540  3.122076
1  718.488475 -5.979050  4.991843  0.137019  4.991843  4.959315  4.172389
2  844.889698 -1.652395  0.673601  9.198001  0.673601  0.673352  7.770955
3  885.755585 -2.504044  0.656561  7.514520  0.656561  0.646653  8.450394
4  749.854462 -3.452123  0.474878  6.235982  0.474878  0.460514  8.908012

[5 rows x 95 columns]
```

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

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

```
One process:
CPU times: user 19.7 s, sys: 918 ms, total: 20.6 s
Wall time: 16.1 s
Two processes:
```

(continues on next page)

(continued from previous page)

```
CPU times: user 1.31 s, sys: 154 ms, total: 1.46 s
Wall time: 8.76 s
```

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

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

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

7.2 Advanced usage

7.2.1 Sampler objects

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

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

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

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

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

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

7.2.2 Sampling and validation

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

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

s2 = optgp.sample(100)
```

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

- "v" - valid point
- "l" - lower bound violation
- "u" - upper bound violation

- "e" - equality violation (meaning the point is not a steady state)

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

[8]:
```
import numpy as np

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

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

And for our generated samples:

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

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

Even though most models are numerically stable enought that the sampler should only generate valid samples we still urge to check this. `validate` is pretty fast and works quickly even for large models and many samples. If you find invalid samples you do not necessarily have to rerun the entire sampling but can exclude them from the sample DataFrame.

[10]:
```
s1_valid = s1[achr.validate(s1) == "v"]
len(s1_valid)
```

[10]:
```
100
```

7.2.3 Batch sampling

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

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

[11]:
```
counts = [np.mean(s.Biomass_Ecoli_core > 0.1) for s in optgp.batch(100, 10)]
print("Usually {:.2f}% +- {:.2f}% grow...".format(
    np.mean(counts) * 100.0, np.std(counts) * 100.0))
```

```
Usually 14.50% +- 2.16% grow...
```

7.3 Adding constraints

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

[12]:
```
co = model.problem.Constraint(model.reactions.Biomass_Ecoli_core.flux_expression,␣
↪lb=0.1)
model.add_cons_vars([co])
```

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

[13]:
```
s = sample(model, 10)
print(s.Biomass_Ecoli_core)
```

```
0    0.124471
1    0.151331
2    0.108145
3    0.144076
4    0.110480
5    0.109024
6    0.111399
7    0.139682
8    0.103511
9    0.116880
Name: Biomass_Ecoli_core, dtype: float64
```

As we can see our new constraint was respected.

CHAPTER

EIGHT

LOOPLESS FBA

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

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

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

8.1 Loopless solution

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

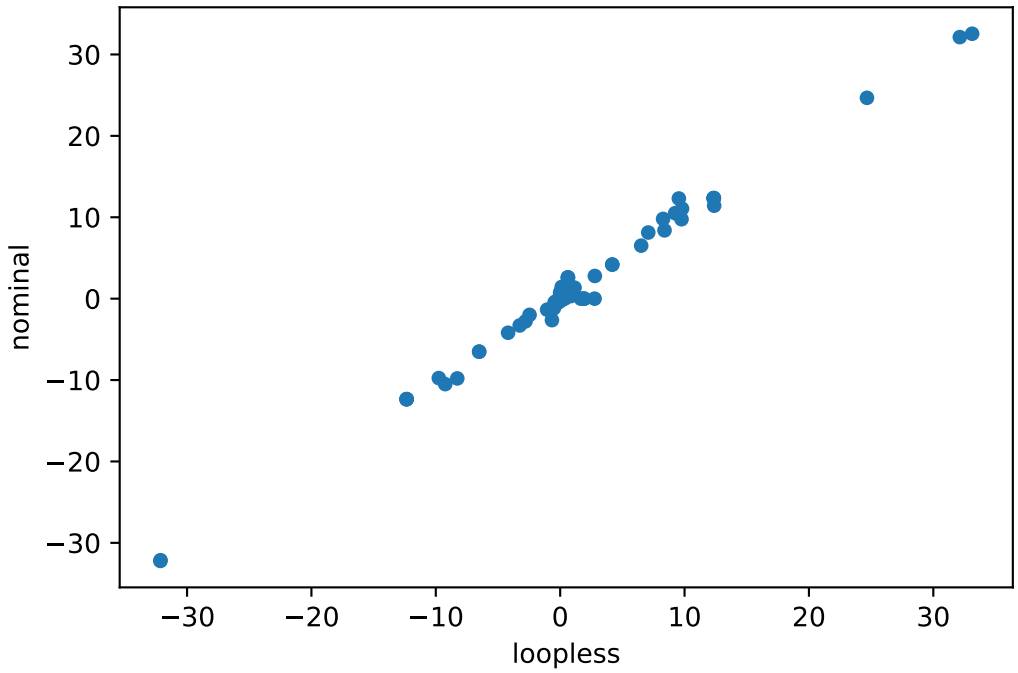
Using a larger model than the simple example above, this can be demonstrated as follows

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

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

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

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



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

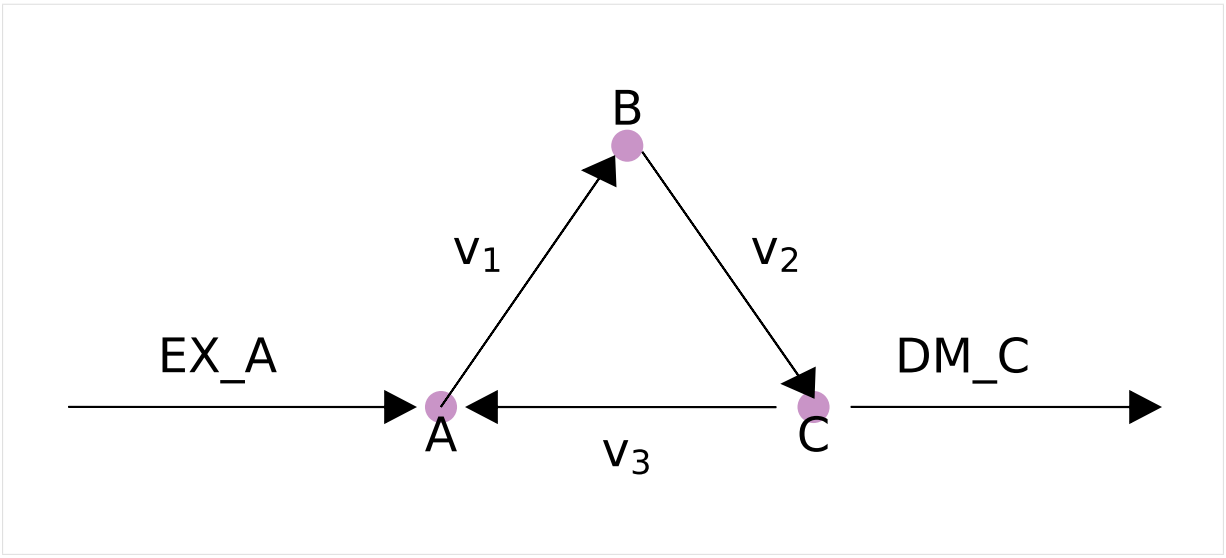
8.2 Loopless model

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

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

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

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



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

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

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

model.objective = 'DM_C'
```

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
    except:
        print('model is infeasible')
```

```
model is infeasible
```

```
cobra/util/solver.py:398 UserWarning: solver status is 'infeasible'
```

[10]:
```
solution = pfba(model)
print("parsimonious solution: status = " + solution.status)
print("loopless solution flux: v3 = %.1f" % solution.fluxes["v3"])
```

```
parsimonious solution: status = optimal
loopless solution flux: v3 = 1.0
```

8.3 Method

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

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

As proven in the original publication this will identify the "least-loopy" solution closest to the reference fluxes.

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

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

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

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

CHAPTER

NINE

CONSISTENCY TESTING

For most problems, multiple flux states can achieve the same optimum and thus we try to obtain a consistent network. By this, we mean that there will be mulitple blocked reactions in the network, which gives rise to this inconsistency. To solve this problem, we use algorithms which can detect all the blocked reactions and also give us consistent networks.

Let us take a toy network, like so:

$$v_1 : \rightarrow 2A \tag{9.1}$$
$$v_2 : A \leftrightarrow B \tag{9.2}$$
$$v_3 : A \rightarrow D \tag{9.3}$$
$$v_4 : A \rightarrow C \tag{9.4}$$
$$v_5 : C \rightarrow D \tag{9.5}$$
$$v_6 : D \rightarrow \tag{9.6}$$

Here, v_x, where $x \in \{1, 2, \ldots, 6\}$ represent the flux carried by the reactions as shown above.

[1]:
```
import cobra
```

[2]:
```
test_model = cobra.Model("test_model")
v1 = cobra.Reaction("v1")
v2 = cobra.Reaction("v2")
v3 = cobra.Reaction("v3")
v4 = cobra.Reaction("v4")
v5 = cobra.Reaction("v5")
v6 = cobra.Reaction("v6")

test_model.add_reactions([v1, v2, v3, v4, v5, v6])

v1.reaction = "-> 2 A"
v2.reaction = "A <-> B"
v3.reaction = "A -> D"
v4.reaction = "A -> C"
v5.reaction = "C -> D"
v6.reaction = "D ->"

v1.bounds = (0.0, 3.0)
v2.bounds = (-3.0, 3.0)
v3.bounds = (0.0, 3.0)
v4.bounds = (0.0, 3.0)
v5.bounds = (0.0, 3.0)
v6.bounds = (0.0, 3.0)

test_model.objective = v6
```

```
unknown metabolite 'A' created
unknown metabolite 'B' created
unknown metabolite 'D' created
unknown metabolite 'C' created
```

9.1 Using FVA

The first approach we can follow is to use FVA (Flux Variability Analysis) which among many other applications, is used to detect blocked reactions. The `cobra.flux_analysis.find_blocked_reactions()` function will return a list of all the blocked reactions obtained using FVA.

[3]:
```
cobra.flux_analysis.find_blocked_reactions(test_model)
```

[3]:
```
['v2']
```

As we see above, we are able to obtain the blocked reaction, which in this case is v_2.

9.2 Using FASTCC

The second approach to obtaining consistent network in `cobrapy` is to use FASTCC. Using this method, you can expect to efficiently obtain an accurate consistent network. For more details regarding the algorithm, please see Vlassis N, Pacheco MP, Sauter T (2014).

[4]:
```
consistent_model = cobra.flux_analysis.fastcc(test_model)
consistent_model.reactions
```

[4]:
```
[<Reaction v1 at 0x7fc71ddea5c0>,
 <Reaction v3 at 0x7fc71ddea630>,
 <Reaction v4 at 0x7fc71ddea668>,
 <Reaction v5 at 0x7fc71ddea6a0>,
 <Reaction v6 at 0x7fc71ddea6d8>]
```

Similar to the FVA approach, we are able to identify that v_2 is indeed the blocked reaction.

CHAPTER

TEN

GAPFILLLING

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

Minimize:

$$\sum_i c_i * z_i$$

subject to

$$Sv = 0$$

$$v^\star \geq t$$

$$l_i \leq v_i \leq u_i$$

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

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

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

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

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

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

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

[3]:
```
0.0
```

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

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

```
GF6PTA
F6PP
TKT2
FBP
MAN6PI
```

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

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

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

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

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

```
FBP
```

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

CHAPTER

ELEVEN

GROWTH MEDIA

The availability of nutrients has a major impact on metabolic fluxes and `cobrapy` provides some helpers to manage the exchanges between the external environment and your metabolic model. In experimental settings the "environment" is usually constituted by the growth medium, ergo the concentrations of all metabolites and co-factors available to the modeled organism. However, constraint-based metabolic models only consider fluxes. Thus, you can not simply use concentrations since fluxes have the unit `mmol / [gDW h]` (concentration per gram dry weight of cells and hour).

Also, you are setting an upper bound for the particular import flux and not the flux itself. There are some crude approximations. For instance, if you supply 1 mol of glucose every 24h to 1 gram of bacteria you might set the upper exchange flux for glucose to `1 mol / [1 gDW * 24 h]` since that is the nominal maximum that can be imported. There is no guarantee however that glucose will be consumed with that flux. Thus, the preferred data for exchange fluxes are direct flux measurements as the ones obtained from timecourse exa-metabolome measurements for instance.

So how does that look in COBRApy? The current growth medium of a model is managed by the `medium` attribute.

[1]:
```
from cobra.test import create_test_model

model = create_test_model("textbook")
model.medium
```

[1]:
```
{'EX_co2_e': 1000.0,
 'EX_glc__D_e': 10.0,
 'EX_h_e': 1000.0,
 'EX_h2o_e': 1000.0,
 'EX_nh4_e': 1000.0,
 'EX_o2_e': 1000.0,
 'EX_pi_e': 1000.0}
```

This will return a dictionary that contains the upper flux bounds for all active exchange fluxes (the ones having non-zero flux bounds). Right now we see that we have enabled aerobic growth. You can modify a growth medium of a model by assigning a dictionary to `model.medium` that maps exchange reactions to their respective upper import bounds. For now let us enforce anaerobic growth by shutting off the oxygen import.

[2]:
```
medium = model.medium
medium["EX_o2_e"] = 0.0
model.medium = medium

model.medium
```

[2]:
```
{'EX_co2_e': 1000.0,
 'EX_glc__D_e': 10.0,
 'EX_h_e': 1000.0,
 'EX_h2o_e': 1000.0,
 'EX_nh4_e': 1000.0,
 'EX_pi_e': 1000.0}
```

As we can see oxygen import is now removed from the list of active exchanges and we can verify that this also leads to a lower growth rate.

[3]:
```
model.slim_optimize()
```

[3]:
```
0.21166294973530736
```

There is a small trap here. `model.medium` can not be assigned to directly. So the following will not work:

[4]:
```
model.medium["EX_co2_e"] = 0.0
model.medium
```

[4]:
```
{'EX_co2_e': 1000.0,
 'EX_glc__D_e': 10.0,
 'EX_h_e': 1000.0,
 'EX_h2o_e': 1000.0,
 'EX_nh4_e': 1000.0,
 'EX_pi_e': 1000.0}
```

As you can see `EX_co2_e` is not set to zero. This is because model.medium is just a copy of the current exchange fluxes. Assigning to it directly with `model.medium[...] = ...` will **not** change the model. You have to assign an entire dictionary with the changed import flux upper bounds:

[5]:
```
medium = model.medium
medium["EX_co2_e"] = 0.0
model.medium = medium

model.medium  # now it worked
```

[5]:
```
{'EX_glc__D_e': 10.0,
 'EX_h_e': 1000.0,
 'EX_h2o_e': 1000.0,
 'EX_nh4_e': 1000.0,
 'EX_pi_e': 1000.0}
```

Setting the growth medium also connects to the context manager, so you can set a specific growth medium in a reversible manner.

[6]:
```
model = create_test_model("textbook")

with model:
    medium = model.medium
    medium["EX_o2_e"] = 0.0
    model.medium = medium
    print(model.slim_optimize())
print(model.slim_optimize())
model.medium
```

```
0.21166294973530736
0.8739215069684102
```

[6]:
```
{'EX_co2_e': 1000.0,
 'EX_glc__D_e': 10.0,
 'EX_h_e': 1000.0,
 'EX_h2o_e': 1000.0,
 'EX_nh4_e': 1000.0,
 'EX_o2_e': 1000.0,
 'EX_pi_e': 1000.0}
```

So the medium change is only applied within the `with` block and reverted automatically.

11.1 Minimal media

In some cases you might be interested in the smallest growth medium that can maintain a specific growth rate, the so called "minimal medium". For this we provide the function `minimal_medium` which by default obtains the medium with the lowest total import flux. This function needs two arguments: the model and the minimum growth rate (or other objective) the model has to achieve.

[7]:
```
from cobra.medium import minimal_medium

max_growth = model.slim_optimize()
minimal_medium(model, max_growth)
```

[7]:
```
EX_glc__D_e    10.000000
EX_nh4_e        4.765319
EX_o2_e        21.799493
EX_pi_e         3.214895
dtype: float64
```

So we see that growth is actually limited by glucose import.

Alternatively you might be interested in a minimal medium with the smallest number of active imports. This can be achieved by using the `minimize_components` argument (note that this uses a MIP formulation and will therefore be much slower).

[8]:
```
minimal_medium(model, 0.1, minimize_components=True)
```

[8]:
```
EX_glc__D_e    10.000000
EX_nh4_e        1.042503
EX_pi_e         0.703318
dtype: float64
```

When minimizing the number of import fluxes there may be many alternative solutions. To obtain several of those you can also pass a positive integer to `minimize_components` which will give you at most that many alternative solutions. Let us try that with our model and also use the `open_exchanges` argument which will assign a large upper bound to all import reactions in the model. The return type will be a `pandas.DataFrame`.

[9]:
```
minimal_medium(model, 0.8, minimize_components=8, open_exchanges=True)
```

[9]:

	0	1	2	3
EX_fru_e	0.000000	521.357767	0.000000	0.000000
EX_glc__D_e	0.000000	0.000000	0.000000	519.750758
EX_gln__L_e	0.000000	40.698058	18.848678	0.000000
EX_glu__L_e	348.101944	0.000000	0.000000	0.000000
EX_mal__L_e	0.000000	0.000000	1000.000000	0.000000
EX_nh4_e	0.000000	0.000000	0.000000	81.026921
EX_o2_e	500.000000	0.000000	0.000000	0.000000
EX_pi_e	66.431529	54.913419	12.583458	54.664344

So there are 4 alternative solutions in total. One aerobic and three anaerobic ones using different carbon sources.

11.2 Boundary reactions

Apart from exchange reactions there are other types of boundary reactions such as demand or sink reactions. `cobrapy` uses various heuristics to identify those and they can be accessed by using the appropriate attribute.

For exchange reactions:

[10]:
```
ecoli = create_test_model("ecoli")
ecoli.exchanges[0:5]
```

```
[10]: [<Reaction EX_12ppd__R_e at 0x131b4a58d0>,
       <Reaction EX_12ppd__S_e at 0x131b471c50>,
       <Reaction EX_14glucan_e at 0x131b471e10>,
       <Reaction EX_15dap_e at 0x131b471e48>,
       <Reaction EX_23camp_e at 0x131b471f98>]
```

For demand reactions:

```
[11]: ecoli.demands
```

```
[11]: [<Reaction DM_4CRSOL at 0x131b3162b0>,
       <Reaction DM_5DRIB at 0x131b4712e8>,
       <Reaction DM_AACALD at 0x131b471400>,
       <Reaction DM_AMOB at 0x131b4714e0>,
       <Reaction DM_MTHTHF at 0x131b4715f8>,
       <Reaction DM_OXAM at 0x131b4716d8>]
```

For sink reactions:

```
[12]: ecoli.sinks
```

```
[12]: []
```

All boundary reactions (any reaction that consumes or introduces mass into the system) can be obtained with the `boundary` attribute:

```
[13]: ecoli.boundary[0:10]
```

```
[13]: [<Reaction DM_4CRSOL at 0x131b3162b0>,
       <Reaction DM_5DRIB at 0x131b4712e8>,
       <Reaction DM_AACALD at 0x131b471400>,
       <Reaction DM_AMOB at 0x131b4714e0>,
       <Reaction DM_MTHTHF at 0x131b4715f8>,
       <Reaction DM_OXAM at 0x131b4716d8>,
       <Reaction EX_12ppd__R_e at 0x131b4a58d0>,
       <Reaction EX_12ppd__S_e at 0x131b471c50>,
       <Reaction EX_14glucan_e at 0x131b471e10>,
       <Reaction EX_15dap_e at 0x131b471e48>]
```

CHAPTER

TWELVE

SOLVERS

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

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

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

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

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

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

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

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

12.1 Internal solver interfaces

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

CHAPTER

THIRTEEN

TAILORED CONSTRAINTS, VARIABLES AND OBJECTIVES

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

13.1 Constraints

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

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

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

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

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

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

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

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

```
4.66274904774 4.66274904774 0.855110960926157
```

It is also possible to add many constraints at once. For large models, with constraints involving many reactions, the efficient way to do this is to first build a dictionary of the linear coefficients for every flux, and then add the constraint at once. For example, suppose we want to add a constrain on the sum of the absolute values of every flux in the network to be less than 100:

[5]:
```
coefficients = dict()
for rxn in model.reactions:
    coefficients[rxn.forward_variable] = 1.
    coefficients[rxn.reverse_variable] = 1.
constraint = model.problem.Constraint(0, lb=0, ub=100)
model.add_cons_vars(constraint)
```

(continues on next page)

(continued from previous page)

```
model.solver.update()
constraint.set_linear_coefficients(coefficients=coefficients)
```

13.2 Objectives

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

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

```
0.8739215069684307
```

The objectives mathematical expression is seen by

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

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

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

Consider the example problem:

$$\textbf{min } \tfrac{1}{2}\left(x^2 + y^2\right) - y$$

subject to

$$x + y = 2$$

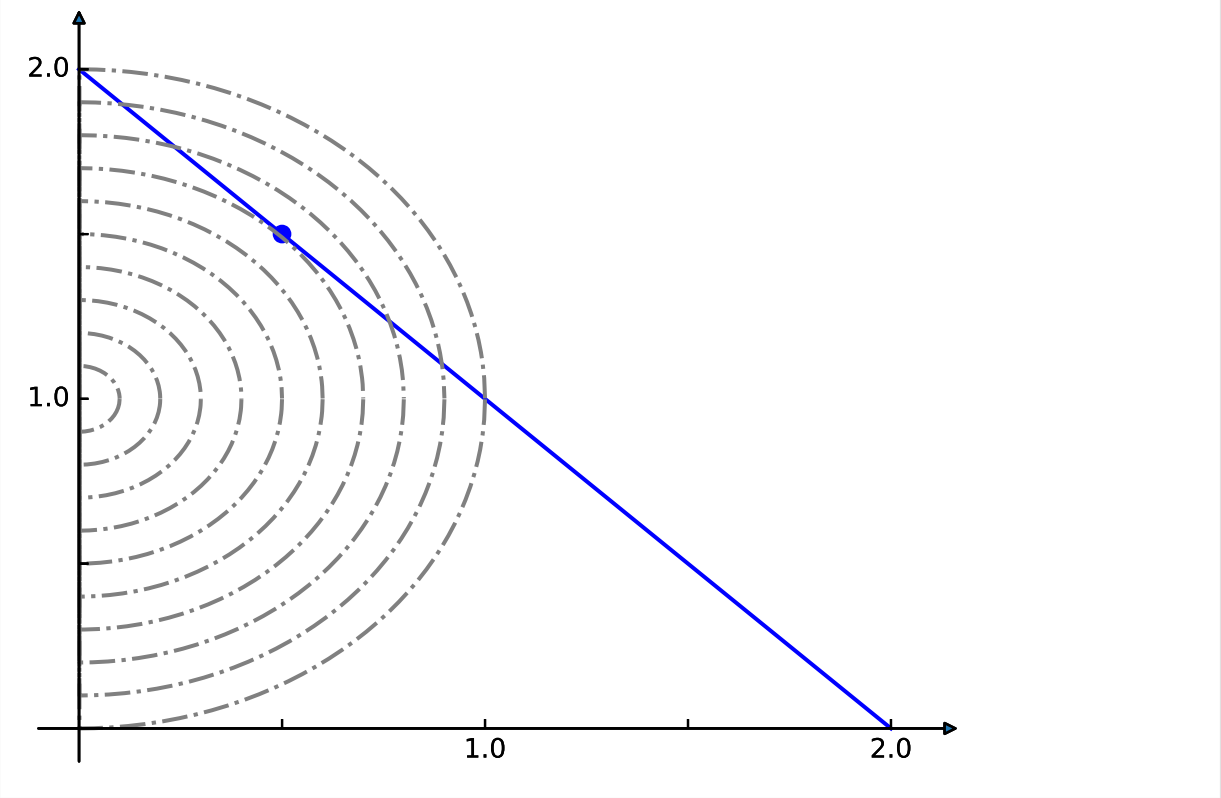
$$x \geq 0$$

$$y \geq 0$$

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

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

plot_helper.plot_qp2()
```



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

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

Next we add the quadratic objective

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

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

```
0.5 1.5
```

13.3 Variables

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

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

We use constraints to define what values this variable shall take

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

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

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

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

CHAPTER

FOURTEEN

DYNAMIC FLUX BALANCE ANALYSIS (DFBA) IN COBRAPY

The following notebook shows a simple, but slow example of implementing dFBA using COBRApy and scipy.integrate.solve_ivp. This notebook shows a static optimization approach (SOA) implementation and should not be considered production ready.

The model considers only basic Michaelis-Menten limited growth on glucose.

[1]:
```
import numpy as np
from tqdm import tqdm

from scipy.integrate import solve_ivp

import matplotlib.pyplot as plt
%matplotlib inline
```

Create or load a cobrapy model. Here, we use the 'textbook' e-coli core model.

[2]:
```
import cobra
from cobra.test import create_test_model
model = create_test_model('textbook')
```

14.1 Set up the dynamic system

Dynamic flux balance analysis couples a dynamic system in external cellular concentrations to a pseudo-steady state metabolic model.

In this notebook, we define the function `add_dynamic_bounds(model, y)` to convert the external metabolite concentrations into bounds on the boundary fluxes in the metabolic model.

[7]:
```
def add_dynamic_bounds(model, y):
    """Use external concentrations to bound the uptake flux of glucose."""
    biomass, glucose = y  # expand the boundary species
    glucose_max_import = -10 * glucose / (5 + glucose)
    model.reactions.EX_glc__D_e.lower_bound = glucose_max_import


def dynamic_system(t, y):
    """Calculate the time derivative of external species."""

    biomass, glucose = y  # expand the boundary species

    # Calculate the specific exchanges fluxes at the given external concentrations.
    with model:
        add_dynamic_bounds(model, y)

        cobra.util.add_lp_feasibility(model)
        feasibility = cobra.util.fix_objective_as_constraint(model)
```

(continues on next page)

(continued from previous page)

```
        lex_constraints = cobra.util.add_lexicographic_constraints(
            model, ['Biomass_Ecoli_core', 'EX_glc__D_e'], ['max', 'max'])

    # Since the calculated fluxes are specific rates, we multiply them by the
    # biomass concentration to get the bulk exchange rates.
    fluxes = lex_constraints.values
    fluxes *= biomass

    # This implementation is **not** efficient, so I display the current
    # simulation time using a progress bar.
    if dynamic_system.pbar is not None:
        dynamic_system.pbar.update(1)
        dynamic_system.pbar.set_description('t = {:.3f}'.format(t))

    return fluxes

dynamic_system.pbar = None


def infeasible_event(t, y):
    """
    Determine solution feasibility.

    Avoiding infeasible solutions is handled by solve_ivp's built-in event␣
↪detection.
    This function re-solves the LP to determine whether or not the solution is␣
↪feasible
    (and if not, how far it is from feasibility). When the sign of this function␣
↪changes
    from -epsilon to positive, we know the solution is no longer feasible.

    """

    with model:

        add_dynamic_bounds(model, y)

        cobra.util.add_lp_feasibility(model)
        feasibility = cobra.util.fix_objective_as_constraint(model)

    return feasibility - infeasible_event.epsilon

infeasible_event.epsilon = 1E-6
infeasible_event.direction = 1
infeasible_event.terminal = True
```

14.2 Run the dynamic FBA simulation

[4]:
```
ts = np.linspace(0, 15, 100)  # Desired integration resolution and interval
y0 = [0.1, 10]

with tqdm() as pbar:
    dynamic_system.pbar = pbar

    sol = solve_ivp(
        fun=dynamic_system,
        events=[infeasible_event],
        t_span=(ts.min(), ts.max()),
```

(continues on next page)

(continued from previous page)

```
        y0=y0,
        t_eval=ts,
        rtol=1e-6,
        atol=1e-8,
        method='BDF'
    )
```

```
t = 5.804: : 185it [00:16, 11.27it/s]
```

Because the culture runs out of glucose, the simulation terminates early. The exact time of this 'cell death' is recorded in `sol.t_events`.

[5]:
```
sol
```

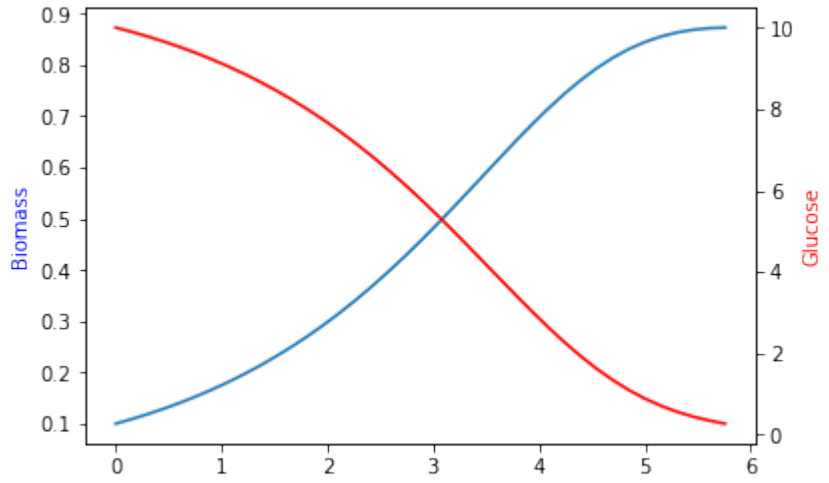
[5]:
```
  message: 'A termination event occurred.'
     nfev: 179
     njev: 2
      nlu: 14
      sol: None
   status: 1
  success: True
        t: array([0.        , 0.15151515, 0.3030303 , 0.45454545, 0.60606061,
       0.75757576, 0.90909091, 1.06060606, 1.21212121, 1.36363636,
       1.51515152, 1.66666667, 1.81818182, 1.96969697, 2.12121212,
       2.27272727, 2.42424242, 2.57575758, 2.72727273, 2.87878788,
       3.03030303, 3.18181818, 3.33333333, 3.48484848, 3.63636364,
       3.78787879, 3.93939394, 4.09090909, 4.24242424, 4.39393939,
       4.54545455, 4.6969697 , 4.84848485, 5.        , 5.15151515,
       5.3030303 , 5.45454545, 5.60606061, 5.75757576])
 t_events: [array([5.80191035])]
        y: array([[ 0.1       ,  0.10897602,  0.11871674,  0.12927916,  0.14072254,
         0.15310825,  0.16649936,  0.18095988,  0.19655403,  0.21334507,
         0.23139394,  0.25075753,  0.27148649,  0.29362257,  0.31719545,
         0.34221886,  0.36868605,  0.3965646 ,  0.42579062,  0.4562623 ,
         0.48783322,  0.52030582,  0.55342574,  0.58687742,  0.62028461,
         0.65321433,  0.685188  ,  0.71570065,  0.74425054,  0.77037369,
         0.79368263,  0.81390289,  0.83089676,  0.84467165,  0.85535715,
         0.8631722 ,  0.86843813,  0.8715096 ,  0.8727423 ],
       [10.        ,  9.8947027 ,  9.78040248,  9.65642157,  9.52205334,
         9.37656372,  9.21919615,  9.04917892,  8.86573366,  8.6680879 ,
         8.45549026,  8.22722915,  7.98265735,  7.72122137,  7.442497  ,
         7.14623236,  6.83239879,  6.50124888,  6.15338213,  5.78981735,
         5.41206877,  5.02222068,  4.62299297,  4.21779303,  3.81071525,
         3.40650104,  3.01042208,  2.6280723 ,  2.26504645,  1.92656158,
         1.61703023,  1.33965598,  1.09616507,  0.88670502,  0.70995892,
         0.56344028,  0.44387781,  0.34762375,  0.27100065]])
```

14.2.1 Plot timelines of biomass and glucose

[6]:
```
ax = plt.subplot(111)
ax.plot(sol.t, sol.y.T[:, 0])
ax2 = plt.twinx(ax)
ax2.plot(sol.t, sol.y.T[:, 1], color='r')

ax.set_ylabel('Biomass', color='b')
ax2.set_ylabel('Glucose', color='r')
```

[6]:
```
Text(0, 0.5, 'Glucose')
```


Biomass
Glucose
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
10
8
6
4
2
0
0
1
2
3
4
5
6

CHAPTER

FIFTEEN

USING THE COBRA TOOLBOX WITH COBRAPY

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

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

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

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

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

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

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

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

```
model =

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

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

[5]:
```
%%matlab --silent
warning('off'); % this works around a pymatbridge bug
```

(continues on next page)

(continued from previous page)

```
addpath(genpath('~/cobratoolbox/'));
initCobraToolbox();
```

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

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

```
ans =

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

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

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

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

[7]:
```
0.8739215069684909
```

CHAPTER

SIXTEEN

FAQ

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

16.1 How do I install cobrapy?

Please see the INSTALL.rst file.

16.2 How do I cite cobrapy?

Please cite the 2013 publication: 10.1186/1752-0509-7-74

16.3 How do I rename reactions or metabolites?

TL;DR Use `Model.repair` afterwards

When renaming metabolites or reactions, there are issues because cobra indexes based off of ID's, which can cause errors. For example:

[1]:
```
from __future__ import print_function
import cobra.test
model = cobra.test.create_test_model()

for metabolite in model.metabolites:
    metabolite.id = "test_" + metabolite.id

try:
    model.metabolites.get_by_id(model.metabolites[0].id)
except KeyError as e:
    print(repr(e))
```

The Model.repair function will rebuild the necessary indexes

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

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

16.4 How do I delete a gene?

That depends on what precisely you mean by delete a gene.

If you want to simulate the model with a gene knockout, use the `cobra.manipulation.delete_model_genes` function. The effects of this function are reversed by `cobra.manipulation.undelete_model_genes`.

[3]:
```
model = cobra.test.create_test_model()
PGI = model.reactions.get_by_id("PGI")
print("bounds before knockout:", (PGI.lower_bound, PGI.upper_bound))
cobra.manipulation.delete_model_genes(model, ["STM4221"])
print("bounds after knockouts", (PGI.lower_bound, PGI.upper_bound))
```

```
bounds before knockout: (-1000.0, 1000.0)
bounds after knockouts (0.0, 0.0)
```

If you want to actually remove all traces of a gene from a model, this is more difficult because this will require changing all the `gene_reaction_rule` strings for reactions involving the gene.

16.5 How do I change the reversibility of a Reaction?

`Reaction.reversibility` is a property in cobra which is computed when it is requested from the lower and upper bounds.

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

[4]:
```
True
```

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

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

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

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

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

[6]:
```
False
```

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

16.6.1 For optlang based solvers

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

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

16.6.2 For cobrapy's internal solvers

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

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

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

16.6.3 How do I visualize my flux solutions?

cobrapy works well with the escher package, which is well suited to this purpose. Consult the escher documentation for examples.

CHAPTER

SEVENTEEN

API REFERENCE

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

17.1 cobra

17.1.1 Subpackages

cobra.core

Subpackages

cobra.core.summary

Submodules

cobra.core.summary.metabolite_summary

Define the MetaboliteSummary class.

Module Contents

Classes

MetaboliteSummary	Define the metabolite summary.

class cobra.core.summary.metabolite_summary.**MetaboliteSummary**(*metabolite*, *model*, ***kwargs*)

Bases: *cobra.core.summary.Summary*

Define the metabolite summary.

metabolite
The metabolite to summarize.

Type *cobra.Metabolite*

See also:

Summary Parent that defines further attributes.

[1] Created with sphinx-autoapi

ReactionSummary, ModelSummary

_generate(*self*)

Returns **flux_summary** – The DataFrame of flux summary data.

Return type pandas.DataFrame

to_frame(*self*)

Returns

Return type A pandas.DataFrame of the summary.

_to_table(*self*)

Returns

Return type A string of the summary table.

cobra.core.summary.model_summary

Define the ModelSummary class.

Module Contents

Classes

ModelSummary	Define the model summary.

cobra.core.summary.model_summary.**logger**

class cobra.core.summary.model_summary.**ModelSummary**(*model*, ***kwargs*)

Bases: *cobra.core.summary.Summary*

Define the model summary.

See also:

Summary Parent that defines further attributes.

MetaboliteSummary, ReactionSummary

_generate(*self*)

Returns **flux_summary** – The DataFrame of flux summary data.

Return type pandas.DataFrame

to_frame(*self*)

Returns

Return type A pandas.DataFrame of the summary.

_to_table(*self*)

Returns

Return type A string of the summary table.

cobra.core.summary.summary

Define the Summary class.

Module Contents

Classes

Summary	Define the abstract base summary.

class cobra.core.summary.summary.**Summary**(*model*, *solution=None*, *threshold=None*, *fva=None*, *names=False*, *float_format='{:.3G}'.format*, ***kwargs*)

Bases: object

Define the abstract base summary.

model
The metabolic model in which to generate a summary description.

Type *cobra.Model*

solution
A solution that matches the given model.

Type *cobra.Solution*

threshold
Threshold below which fluxes are not reported.

Type float, optional

fva
The result of a flux variability analysis (FVA) involving reactions of interest if an FVA was requested.

Type pandas.DataFrame, optional

names
Whether or not to use object names rather than identifiers.

Type bool

float_format
Format string for displaying floats.

Type callable

to_frame()
Return a data frame representation of the summary.

abstract _generate(*self*)
Generate the summary for the required cobra object.

This is an abstract method and thus the subclass needs to implement it.

_process_flux_dataframe(*self*, *flux_dataframe*)
Process a flux DataFrame for convenient downstream analysis.

This method removes flux entries which are below the threshold and also adds information regarding the direction of the fluxes. It is used in both ModelSummary and MetaboliteSummary.

Parameters **flux_dataframe** (*pandas.DataFrame*) – The pandas.DataFrame to process.

Returns

Return type A processed pandas.DataFrame.

abstract to_frame(*self*)
Generate a pandas DataFrame.

This is an abstract method and thus the subclass needs to implement it.

abstract _to_table(*self*)
Generate a pretty-print table.

This is an abstract method and thus the subclass needs to implement it.

__str__(*self*)
Return str(self).

_repr_html_(*self*)

Package Contents

Classes

Summary	Define the abstract base summary.
MetaboliteSummary	Define the metabolite summary.
ModelSummary	Define the model summary.

class cobra.core.summary.**Summary**(*model*, *solution=None*, *threshold=None*, *fva=None*, *names=False*, *float_format='{:.3G}'.format*, ***kwargs*)
Bases: object

Define the abstract base summary.

model
The metabolic model in which to generate a summary description.

Type *cobra.Model*

solution
A solution that matches the given model.

Type *cobra.Solution*

threshold
Threshold below which fluxes are not reported.

Type float, optional

fva
The result of a flux variability analysis (FVA) involving reactions of interest if an FVA was requested.

Type pandas.DataFrame, optional

names
Whether or not to use object names rather than identifiers.

Type bool

float_format
Format string for displaying floats.

Type callable

to_frame()
Return a data frame representation of the summary.

abstract _generate(*self*)
Generate the summary for the required cobra object.

This is an abstract method and thus the subclass needs to implement it.

_process_flux_dataframe(*self*, *flux_dataframe*)
Process a flux DataFrame for convenient downstream analysis.

This method removes flux entries which are below the threshold and also adds information regarding the direction of the fluxes. It is used in both ModelSummary and MetaboliteSummary.

Parameters **flux_dataframe** (*pandas.DataFrame*) – The pandas.DataFrame to process.

Returns

Return type A processed pandas.DataFrame.

abstract to_frame(*self*)
Generate a pandas DataFrame.

This is an abstract method and thus the subclass needs to implement it.

abstract _to_table(*self*)
Generate a pretty-print table.

This is an abstract method and thus the subclass needs to implement it.

__str__(*self*)
Return str(self).

_repr_html_(*self*)

class cobra.core.summary.**MetaboliteSummary**(*metabolite*, *model*, ***kwargs*)
Bases: *cobra.core.summary.Summary*

Define the metabolite summary.

metabolite
The metabolite to summarize.

Type *cobra.Metabolite*

See also:

Summary Parent that defines further attributes.

ReactionSummary, *ModelSummary*

_generate(*self*)

Returns **flux_summary** – The DataFrame of flux summary data.

Return type pandas.DataFrame

to_frame(*self*)

Returns

Return type A pandas.DataFrame of the summary.

_to_table(*self*)

Returns

Return type A string of the summary table.

class cobra.core.summary.**ModelSummary**(*model*, ***kwargs*)
Bases: *cobra.core.summary.Summary*

Define the model summary.

See also:

Summary Parent that defines further attributes.

MetaboliteSummary, ReactionSummary

_generate(*self*)

Returns **flux_summary** – The DataFrame of flux summary data.

Return type pandas.DataFrame

to_frame(*self*)

Returns

Return type A pandas.DataFrame of the summary.

_to_table(*self*)

Returns

Return type A string of the summary table.

Submodules

cobra.core.configuration

Define the global configuration.

Module Contents

Classes

Configuration	Define the configuration to be singleton based.

class cobra.core.configuration.**Configuration**

Bases: six.with_metaclass()

Define the configuration to be singleton based.

cobra.core.dictlist

Module Contents

Classes

DictList	A combined dict and list

class cobra.core.dictlist.**DictList**(**args*)

Bases: list

A combined dict and list

This object behaves like a list, but has the O(1) speed benefits of a dict when looking up elements by their id.

has_id(*self*, *id*)

_check(*self*, *id*)

make sure duplicate id's are not added. This function is called before adding in elements.

_generate_index(*self*)

rebuild the _dict index

get_by_id (*self*, *id*)
return the element with a matching id

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

get_by_any (*self*, *iterable*)
Get a list of members using several different ways of indexing

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

Returns a list of members

Return type list

query (*self*, *search_function*, *attribute=None*)
Query the list

Parameters

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

Returns a new list of objects which match the query

Return type *DictList*

Examples

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

_replace_on_id (*self*, *new_object*)
Replace an object by another with the same id.

append (*self*, *object*)
append object to end

union (*self*, *iterable*)
adds elements with id's not already in the model

extend (*self*, *iterable*)
extend list by appending elements from the iterable

_extend_nocheck (*self*, *iterable*)
extends without checking for uniqueness

This function should only be used internally by DictList when it can guarantee elements are already unique (as in when coming from self or other DictList). It will be faster because it skips these checks.

__sub__ (*self*, *other*)
x.__sub__(y) <==> x - y

Parameters **other** (*iterable*) – other must contain only unique id's present in the list

__isub__ (*self*, *other*)
x.__sub__(y) <==> x -= y

Parameters **other** (*iterable*) – other must contain only unique id's present in the list

__add__ (*self*, *other*)
x.__add__(y) <==> x + y

Parameters **other** (*iterable*) – other must contain only unique id's which do not intersect with self

__iadd__ (*self*, *other*)
x.__iadd__(y) <==> x += y

Parameters **other** (*iterable*) – other must contain only unique id's whcih do not intersect with self

__reduce__ (*self*)
Helper for pickle.

__getstate__ (*self*)
gets internal state

This is only provided for backwards compatibility so older versions of cobrapy can load pickles generated with cobrapy. In reality, the "_dict" state is ignored when loading a pickle

__setstate__ (*self*, *state*)
sets internal state

Ignore the passed in state and recalculate it. This is only for compatibility with older pickles which did not correctly specify the initialization class

index (*self*, *id*, **args*)
Determine the position in the list

id: A string or a `Object`

__contains__ (*self*, *object*)
DictList.__contains__(object) <==> object in DictList

object: str or `Object`

__copy__ (*self*)

insert (*self*, *index*, *object*)
insert object before index

pop (*self*, **args*)
remove and return item at index (default last).

add (*self*, *x*)
Opposite of *remove*. Mirrors set.add

remove (*self*, *x*)

> **Warning:** Internal use only

reverse (*self*)
reverse *IN PLACE*

sort (*self*, *cmp=None*, *key=None*, *reverse=False*)
stable sort *IN PLACE*

cmp(x, y) -> -1, 0, 1

__getitem__ (*self*, *i*)
x.__getitem__(y) <==> x[y]

__setitem__ (*self*, *i*, *y*)
Set self[key] to value.

__delitem__ (*self*, *index*)
Delete self[key].

__getslice__ (*self*, *i*, *j*)

__setslice__ (*self*, *i*, *j*, *y*)

__delslice__ (*self*, *i*, *j*)

__getattr__ (*self*, *attr*)

__dir__ (*self*)
Default dir() implementation.

cobra.core.formula

Module Contents

Classes

Formula	Describes a Chemical Formula

cobra.core.formula.**element_re**

class cobra.core.formula.**Formula**(*formula=None*)
Bases: *cobra.core.object.Object*

Describes a Chemical Formula

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

__add__ (*self*, *other_formula*)
Combine two molecular formulas.

Parameters **other_formula** (Formula, *str*) – string for a chemical formula

Returns The combined formula

Return type *Formula*

parse_composition (*self*)
Breaks the chemical formula down by element.

property weight (*self*)
Calculate the mol mass of the compound

Returns the mol mass

Return type float

cobra.core.formula.**elements_and_molecular_weights**

cobra.core.gene

Module Contents

Classes

GPRCleaner	Parses compiled ast of a gene_reaction_rule and identifies genes
Gene	A Gene in a cobra model

Functions

ast2str(expr, level=0, names=None)	convert compiled ast to gene_reaction_rule str
eval_gpr(expr, knockouts)	evaluate compiled ast of gene_reaction_rule with knockouts
parse_gpr(str_expr)	parse gpr into AST

cobra.core.gene.**keywords**

cobra.core.gene.**keyword_re**

cobra.core.gene.**number_start_re**

cobra.core.gene.**replacements = [['.', '__COBRA_DOT__'], ["'", '__COBRA_SQUOTE__'], ['"',**

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

Parameters

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

Returns The gene reaction rule

Return type string

cobra.core.gene.**eval_gpr**(*expr*, *knockouts*)
evaluate compiled ast of gene_reaction_rule with knockouts

Parameters

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

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

Return type bool

class cobra.core.gene.**GPRCleaner**
Bases: ast.NodeTransformer

Parses compiled ast of a gene_reaction_rule and identifies genes

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

visit_Name(*self*, *node*)

visit_BinOp(*self*, *node*)

cobra.core.gene.**parse_gpr**(*str_expr*)
parse gpr into AST

Parameters **str_expr** (*string*) – string with the gene reaction rule to parse

Returns elements ast_tree and gene_ids as a set

Return type tuple

class cobra.core.gene.**Gene**(*id=None*, *name=''*, *functional=True*)
Bases: *cobra.core.species.Species*

A Gene in a cobra model

Parameters

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

property functional(*self*)
A flag indicating if the gene is functional.

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

knock_out(*self*)
Knockout gene by marking it as non-functional and setting all associated reactions bounds to zero.

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

remove_from_model(*self*, *model=None*, *make_dependent_reactions_nonfunctional=True*)
Removes the association

Parameters

- **model** (*cobra model*) – The model to remove the gene from
- **make_dependent_reactions_nonfunctional** (*bool*) – If True then replace the gene with 'False' in the gene association, else replace the gene with 'True'

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

_repr_html_(*self*)

cobra.core.group

Define the group class.

Module Contents

Classes

Group	Manage groups via this implementation of the SBML group specification.

class cobra.core.group.**Group**(*id*, *name=''*, *members=None*, *kind=None*)
Bases: *cobra.core.object.Object*

Manage groups via this implementation of the SBML group specification.

Group is a class for holding information regarding a pathways, subsystems, or other custom groupings of objects within a cobra.Model object.

Parameters

- **id** (*str*) – The identifier to associate with this group
- **name** (*str, optional*) – A human readable name for the group
- **members** (*iterable, optional*) – A list object containing references to cobra.Model-associated objects that belong to the group.
- **kind** (*{"collection", "classification", "partonomy"}, optional*) – The kind of group, as specified for the Groups feature in the SBML level 3 package specification. Can be any of "classification", "partonomy", or "collection". The default is "collection". Please consult the SBML level 3 package specification to ensure you are using the proper value for kind. In short, members of a "classification" group should have an "is-a" relationship to the group (e.g. member is-a polar compound, or member is-a transporter). Members of a "partonomy" group should have a "part-of" relationship (e.g. member is part-of glycolysis). Members of a "collection" group do not have an implied relationship between the members, so use this value for kind when in doubt (e.g. member is a gap-filled reaction, or member is involved in a disease phenotype).

KIND_TYPES = ['collection', 'classification', 'partonomy']

__len__(*self*)

property members(*self*)

property kind(*self*)

add_members(*self*, *new_members*)
Add objects to the group.

Parameters **new_members** (*list*) – A list of cobrapy objects to add to the group.

remove_members(*self*, *to_remove*)
Remove objects from the group.

Parameters **to_remove** (*list*) – A list of cobra objects to remove from the group

cobra.core.metabolite

Define the Metabolite class.

Module Contents

Classes

Metabolite	Metabolite is a class for holding information regarding

cobra.core.metabolite.**element_re**

class cobra.core.metabolite.**Metabolite**(*id=None*, *formula=None*, *name=''*, *charge=None*, *compartment=None*)
Bases: *cobra.core.species.Species*

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

Parameters

- **id** (*str*) – the identifier to associate with the metabolite
- **formula** (*str*) – Chemical formula (e.g. H2O)
- **name** (*str*) – A human readable name.
- **charge** (*float*) – The charge number of the metabolite
- **compartment** (*str or None*) – Compartment of the metabolite.

_set_id_with_model(*self*, *value*)

property constraint(*self*)
Get the constraints associated with this metabolite from the solve

Returns the optlang constraint for this metabolite

Return type optlang.<interface>.Constraint

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

property formula_weight(*self*)
Calculate the formula weight

property y(*self*)
The shadow price for the metabolite in the most recent solution

Shadow prices are computed from the dual values of the bounds in the solution.

property shadow_price(*self*)
The shadow price in the most recent solution.

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

Warning:

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

Raises

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

Examples

```
>>> import cobra
>>> import cobra.test
>>> model = cobra.test.create_test_model("textbook")
>>> solution = model.optimize()
>>> model.metabolites.glc__D_e.shadow_price
-0.09166474637510488
>>> solution.shadow_prices.glc__D_e
-0.09166474637510883
```

remove_from_model (*self*, *destructive=False*)
Removes the association from self.model

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

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

summary (*self*, *solution=None*, *threshold=0.01*, *fva=None*, *names=False*, *float_format='{:.3g}'.format*)
Create a summary of the producing and consuming fluxes.

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

Parameters

- **solution** (*cobra.Solution, optional*) – A previous model solution to use for generating the summary. If None, the summary method will resolve the model. Note that the solution object must match the model, i.e., changes to the model such as changed bounds, added or removed reactions are not taken into account by this method (default None).
- **threshold** (*float, optional*) – Threshold below which fluxes are not reported. May not be smaller than the model tolerance (default 0.01).
- **fva** (*pandas.DataFrame or float, optional*) – Whether or not to include flux variability analysis in the output. If given, fva should either be a previous FVA solution matching the model or a float between 0 and 1 representing the fraction of the optimum objective to be searched (default None).
- **names** (*bool, optional*) – Emit reaction and metabolite names rather than identifiers (default False).
- **float_format** (*callable, optional*) – Format string for floats (default '{:3G}'.format).

Returns

Return type cobra.MetaboliteSummary

See also:

Reaction.summary(), Model.summary()

_repr_html_ (*self*)

cobra.core.model

Define the Model class.

Module Contents

Classes

Model	Class representation for a cobra model

cobra.core.model.**logger**

cobra.core.model.**configuration**

class cobra.core.model.**Model**(*id_or_model=None*, *name=None*)
Bases: *cobra.core.object.Object*

Class representation for a cobra model

Parameters

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

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

Type *DictList*

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

Type *DictList*

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

Type *DictList*

groups
A DictList where the key is the group identifier and the value a Group

Type *DictList*

solution
The last obtained solution from optimizing the model.

Type *Solution*

__setstate__(*self*, *state*)
Make sure all cobra.Objects in the model point to the model.

__getstate__(*self*)
Get state for serialization.

Ensures that the context stack is cleared prior to serialization, since partial functions cannot be pickled reliably.

property solver(*self*)
Get or set the attached solver instance.

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

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

Examples

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

property tolerance (*self*)

property description (*self*)

get_metabolite_compartments (*self*)
Return all metabolites' compartments.

property compartments (*self*)

property medium (*self*)

__add__ (*self*, *other_model*)
Add the content of another model to this model (+).

The model is copied as a new object, with a new model identifier, and copies of all the reactions in the other model are added to this model. The objective is the sum of the objective expressions for the two models.

__iadd__ (*self*, *other_model*)
Incrementally add the content of another model to this model (+=).

Copies of all the reactions in the other model are added to this model. The objective is the sum of the objective expressions for the two models.

copy (*self*)
Provides a partial 'deepcopy' of the Model. All of the Metabolite, Gene, and Reaction objects are created anew but in a faster fashion than deepcopy

add_metabolites (*self*, *metabolite_list*)
Will add a list of metabolites to the model object and add new constraints accordingly.

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

Parameters **metabolite_list** (A list of *cobra.core.Metabolite* objects) –

remove_metabolites (*self*, *metabolite_list*, *destructive=False*)
Remove a list of metabolites from the the object.

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

Parameters

- **metabolite_list** (*list*) – A list with *cobra.Metabolite* objects as elements.
- **destructive** (*bool*) – If False then the metabolite is removed from all associated reactions. If True then all associated reactions are removed from the Model.

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

Parameters

- **reaction** (cobra.Reaction) – The reaction to add

- **(0.6) Use ~cobra.Model.add_reactions instead** (*Deprecated*) –

add_boundary (*self*, *metabolite*, *type='exchange'*, *reaction_id=None*, *lb=None*, *ub=None*, *sbo_term=None*)

Add a boundary reaction for a given metabolite.

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

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

Parameters

- **metabolite** (*cobra.Metabolite*) – Any given metabolite. The compartment is not checked but you are encouraged to stick to the definition of exchanges and sinks.
- **type** (*str, {"exchange", "demand", "sink"}*) – Using one of the pre-defined reaction types is easiest. If you want to create your own kind of boundary reaction choose any other string, e.g., 'my-boundary'.
- **reaction_id** (*str, optional*) – The ID of the resulting reaction. This takes precedence over the auto-generated identifiers but beware that it might make boundary reactions harder to identify afterwards when using *model.boundary* or specifically *model.exchanges* etc.
- **lb** (*float, optional*) – The lower bound of the resulting reaction.
- **ub** (*float, optional*) – The upper bound of the resulting reaction.
- **sbo_term** (*str, optional*) – A correct SBO term is set for the available types. If a custom type is chosen, a suitable SBO term should also be set.

Returns The created boundary reaction.

Return type *cobra.Reaction*

Examples

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

add_reactions (*self*, *reaction_list*)

Add reactions to the model.

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

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

Parameters **reaction_list** (*list*) – A list of *cobra.Reaction* objects

remove_reactions (*self*, *reactions*, *remove_orphans=False*)
Remove reactions from the model.

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

Parameters

- **reactions** (*list*) – A list with reactions (*cobra.Reaction*), or their id's, to remove
- **remove_orphans** (*bool*) – Remove orphaned genes and metabolites from the model as well

add_groups (*self*, *group_list*)
Add groups to the model.

Groups with identifiers identical to a group already in the model are ignored.

If any group contains members that are not in the model, these members are added to the model as well. Only metabolites, reactions, and genes can have groups.

Parameters **group_list** (*list*) – A list of *cobra.Group* objects to add to the model.

remove_groups (*self*, *group_list*)
Remove groups from the model.

Members of each group are not removed from the model (i.e. metabolites, reactions, and genes in the group stay in the model after any groups containing them are removed).

Parameters **group_list** (*list*) – A list of *cobra.Group* objects to remove from the model.

get_associated_groups (*self*, *element*)
Returns a list of groups that an element (reaction, metabolite, gene) is associated with.

Parameters **element** (*cobra.Reaction*, *cobra.Metabolite*, or *cobra.Gene*) –

Returns All groups that the provided object is a member of

Return type list of *cobra.Group*

add_cons_vars (*self*, *what*, ***kwargs*)
Add constraints and variables to the model's mathematical problem.

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

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

Parameters

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

remove_cons_vars (*self*, *what*)
Remove variables and constraints from the model's mathematical problem.

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

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

property problem (*self*)
The interface to the model's underlying mathematical problem.

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

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

Return type optlang.interface

property variables (*self*)
The mathematical variables in the cobra model.

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

Returns A container with all associated variables.

Return type optlang.container.Container

property constraints (*self*)
The constraints in the cobra model.

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

Returns A container with all associated constraints.

Return type optlang.container.Container

property boundary (*self*)
Boundary reactions in the model. Reactions that either have no substrate or product.

property exchanges (*self*)
Exchange reactions in model. Reactions that exchange mass with the exterior. Uses annotations and heuristics to exclude non-exchanges such as sink reactions.

property demands (*self*)
Demand reactions in model. Irreversible reactions that accumulate or consume a metabolite in the inside of the model.

property sinks (*self*)
Sink reactions in model. Reversible reactions that accumulate or consume a metabolite in the inside of the model.

_populate_solver (*self*, *reaction_list*, *metabolite_list=None*)
Populate attached solver with constraints and variables that model the provided reactions.

slim_optimize (*self*, *error_value=float('nan')*, *message=None*)
Optimize model without creating a solution object.

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

Parameters

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

Returns The objective value.

Return type float

optimize (*self*, *objective_sense=None*, *raise_error=False*)
Optimize the model using flux balance analysis.

Parameters

- **objective_sense** (*{None, 'maximize' 'minimize'}, optional*) – Whether fluxes should be maximized or minimized. In case of None, the previous direction is used.
- **raise_error** (*bool*) –

  If true, raise an OptimizationError if solver status is not optimal.

Notes

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

repair (*self*, *rebuild_index=True*, *rebuild_relationships=True*)
Update all indexes and pointers in a model

Parameters

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

property objective (*self*)
Get or set the solver objective

Before introduction of the optlang based problems, this function returned the objective reactions as a list. With optlang, the objective is not limited a simple linear summation of individual reaction fluxes, making that return value ambiguous. Henceforth, use *cobra.util.solver.linear_reaction_coefficients* to get a dictionary of reactions with their linear coefficients (empty if there are none)

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

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

property objective_direction (*self*)
Get or set the objective direction.

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

summary (*self*, *solution=None*, *threshold=0.01*, *fva=None*, *names=False*, *float_format='{:.3g}'.format*)
Create a summary of the exchange fluxes of the model.

Parameters

- **solution** (*cobra.Solution, optional*) – A previous model solution to use for generating the summary. If None, the summary method will resolve the model. Note that the solution object must match the model, i.e., changes to the model such as changed bounds, added or removed reactions are not taken into account by this method (default None).
- **threshold** (*float, optional*) – Threshold below which fluxes are not reported. May not be smaller than the model tolerance (default 0.01).

- **fva** (*pandas.DataFrame or float, optional*) – Whether or not to include flux variability analysis in the output. If given, fva should either be a previous FVA solution matching the model or a float between 0 and 1 representing the fraction of the optimum objective to be searched (default None).
- **names** (*bool, optional*) – Emit reaction and metabolite names rather than identifiers (default False).
- **float_format** (*callable, optional*) – Format string for floats (default `'{:3G}'.format`).

Returns

Return type cobra.ModelSummary

See also:

`Reaction.summary()`, `Metabolite.summary()`

__enter__(*self*)
Record all future changes to the model, undoing them when a call to __exit__ is received

__exit__(*self*, *type*, *value*, *traceback*)
Pop the top context manager and trigger the undo functions

merge(*self*, *right*, *prefix_existing=None*, *inplace=True*, *objective='left'*)
Merge two models to create a model with the reactions from both models.

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

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

prefix_existing [string] Prefix the reaction identifier in the right that already exist in the left model with this string.

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

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

_repr_html_(*self*)

cobra.core.object

Module Contents

Classes

Object	Defines common behavior of object in cobra.core

class cobra.core.object.**Object**(*id=None*, *name=''*)
Bases: object

Defines common behavior of object in cobra.core

property id(*self*)

_set_id_with_model(*self*, *value*)

__getstate__(*self*)
To prevent excessive replication during deepcopy.

__repr__ (*self*)
Return repr(self).

__str__ (*self*)
Return str(self).

cobra.core.reaction

Define the Reaction class.

Module Contents

Classes

Reaction	Reaction is a class for holding information regarding

cobra.core.reaction.**config**

cobra.core.reaction.**and_or_search**

cobra.core.reaction.**uppercase_AND**

cobra.core.reaction.**uppercase_OR**

cobra.core.reaction.**gpr_clean**

cobra.core.reaction.**compartment_finder**

cobra.core.reaction.**_reversible_arrow_finder**

cobra.core.reaction.**_forward_arrow_finder**

cobra.core.reaction.**_reverse_arrow_finder**

class cobra.core.reaction.**Reaction**(*id=None*, *name=''*, *subsystem=''*, *lower_bound=0.0*, *upper_bound=None*)

Bases: *cobra.core.object.Object*

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

Reactions are by default irreversible with bounds *(0.0, cobra.Configuration().upper_bound)* if no bounds are provided on creation. To create an irreversible reaction use *lower_bound=None*, resulting in reaction bounds of *(cobra.Configuration().lower_bound, cobra.Configuration().upper_bound)*.

Parameters

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

__radd__

_set_id_with_model(*self*, *value*)

property reverse_id(*self*)
Generate the id of reverse_variable from the reaction's id.

property flux_expression(*self*)
Forward flux expression

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

Return type sympy expression

property forward_variable(*self*)
An optlang variable representing the forward flux

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

Return type optlang.interface.Variable

property reverse_variable(*self*)
An optlang variable representing the reverse flux

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

Return type optlang.interface.Variable

property objective_coefficient(*self*)
Get the coefficient for this reaction in a linear objective (float)

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

__copy__(*self*)

__deepcopy__(*self*, *memo*)

static _check_bounds(*lb*, *ub*)

update_variable_bounds(*self*)

property lower_bound(*self*)
Get or set the lower bound

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

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

property upper_bound(*self*)
Get or set the upper bound

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

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

property bounds(*self*)
Get or set the bounds directly from a tuple

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

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

property flux(*self*)
The flux value in the most recent solution.

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

Warning:

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

Raises

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

Examples

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

property reduced_cost (*self*)

The reduced cost in the most recent solution.

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

Warning:

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

Raises

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

Examples

```
>>> import cobra.test
>>> model = cobra.test.create_test_model("textbook")
>>> solution = model.optimize()
>>> model.reactions.PFK.reduced_cost
-8.673617379884035e-18
>>> solution.reduced_costs.PFK
-8.6736173798840355e-18
```

property metabolites (*self*)

property genes (*self*)

property gene_reaction_rule (*self*)

property gene_name_reaction_rule (*self*)
Display gene_reaction_rule with names intead.

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

property functional (*self*)
All required enzymes for reaction are functional.

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

Return type bool

property x (*self*)
The flux through the reaction in the most recent solution.

Flux values are computed from the primal values of the variables in the solution.

property y (*self*)
The reduced cost of the reaction in the most recent solution.

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

property reversibility (*self*)
Whether the reaction can proceed in both directions (reversible)

This is computed from the current upper and lower bounds.

property boundary (*self*)
Whether or not this reaction is an exchange reaction.

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

property model (*self*)
returns the model the reaction is a part of

_update_awareness (*self*)
Make sure all metabolites and genes that are associated with this reaction are aware of it.

remove_from_model (*self*, *remove_orphans=False*)
Removes the reaction from a model.

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

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

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

delete (*self*, *remove_orphans=False*)
Removes the reaction from a model.

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

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

Deprecated, use *reaction.remove_from_model* instead.

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

__setstate__(*self*, *state*)
Probably not necessary to set _model as the cobra.Model that contains self sets the _model attribute for all metabolites and genes in the reaction.

However, to increase performance speed we do want to let the metabolite and gene know that they are employed in this reaction

copy(*self*)
Copy a reaction

The referenced metabolites and genes are also copied.

__add__(*self*, *other*)
Add two reactions

The stoichiometry will be the combined stoichiometry of the two reactions, and the gene reaction rule will be both rules combined by an and. All other attributes (i.e. reaction bounds) will match those of the first reaction

__iadd__(*self*, *other*)

__sub__(*self*, *other*)

__isub__(*self*, *other*)

__imul__(*self*, *coefficient*)
Scale coefficients in a reaction by a given value

E.g. A -> B becomes 2A -> 2B.

If coefficient is less than zero, the reaction is reversed and the bounds are swapped.

__mul__(*self*, *coefficient*)

property reactants(*self*)
Return a list of reactants for the reaction.

property products(*self*)
Return a list of products for the reaction

get_coefficient(*self*, *metabolite_id*)
Return the stoichiometric coefficient of a metabolite.

Parameters **metabolite_id** (*str or cobra.Metabolite*) –

get_coefficients(*self*, *metabolite_ids*)
Return the stoichiometric coefficients for a list of metabolites.

Parameters **metabolite_ids** (*iterable*) – Containing str or ``cobra.Metabolite``s.

add_metabolites(*self*, *metabolites_to_add*, *combine=True*, *reversibly=True*)
Add metabolites and stoichiometric coefficients to the reaction. If the final coefficient for a metabolite is 0 then it is removed from the reaction.

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

Parameters

- **metabolites_to_add** (*dict*) – Dictionary with metabolite objects or metabolite identifiers as keys and coefficients as values. If keys are strings (name of a metabolite) the reaction must already be part of a model and a metabolite with the given name must exist in the model.

- **combine** (*bool*) – Describes behavior a metabolite already exists in the reaction. True causes the coefficients to be added. False causes the coefficient to be replaced.
- **reversibly** (*bool*) – Whether to add the change to the context to make the change reversibly or not (primarily intended for internal use).

subtract_metabolites (*self*, *metabolites*, *combine=True*, *reversibly=True*)
Subtract metabolites from a reaction.

That means add the metabolites with -1*coefficient. If the final coefficient for a metabolite is 0 then the metabolite is removed from the reaction.

Notes

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

Parameters

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

property reaction (*self*)
Human readable reaction string

build_reaction_string (*self*, *use_metabolite_names=False*)
Generate a human readable reaction string

check_mass_balance (*self*)
Compute mass and charge balance for the reaction

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

property compartments (*self*)
lists compartments the metabolites are in

get_compartments (*self*)
lists compartments the metabolites are in

_associate_gene (*self*, *cobra_gene*)
Associates a cobra.Gene object with a cobra.Reaction.

Parameters **cobra_gene** (*cobra.core.Gene.Gene*) –

_dissociate_gene (*self*, *cobra_gene*)
Dissociates a cobra.Gene object with a cobra.Reaction.

Parameters **cobra_gene** (*cobra.core.Gene.Gene*) –

knock_out (*self*)
Knockout reaction by setting its bounds to zero.

build_reaction_from_string (*self*, *reaction_str*, *verbose=True*, *fwd_arrow=None*, *rev_arrow=None*, *reversible_arrow=None*, *term_split='+'*)
Builds reaction from reaction equation reaction_str using parser

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

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

Parameters

- **reaction_str** (*string*) – a string containing a reaction formula (equation)
- **verbose** (*bool*) – setting verbosity of function
- **fwd_arrow** (*re.compile*) – for forward irreversible reaction arrows
- **rev_arrow** (*re.compile*) – for backward irreversible reaction arrows
- **reversible_arrow** (*re.compile*) – for reversible reaction arrows
- **term_split** (*string*) – dividing individual metabolite entries

summary (*self*, *solution=None*, *threshold=0.01*, *fva=None*, *names=False*, *float_format='{:.3g}'.format*)

Create a summary of the producing and consuming fluxes of the reaction.

Parameters

- **solution** (cobra.Solution, *optional*) – A previous model solution to use for generating the summary. If None, the summary method will resolve the model. Note that the solution object must match the model, i.e., changes to the model such as changed bounds, added or removed reactions are not taken into account by this method (default None).
- **threshold** (*float, optional*) – Threshold below which fluxes are not reported. May not be smaller than the model tolerance (default 0.01).
- **fva** (*pandas.DataFrame or float, optional*) – Whether or not to include flux variability analysis in the output. If given, fva should either be a previous FVA solution matching the model or a float between 0 and 1 representing the fraction of the optimum objective to be searched (default None).
- **names** (*bool, optional*) – Emit reaction and metabolite names rather than identifiers (default False).
- **float_format** (*callable, optional*) – Format string for floats (default `'{:3G}'.format`).

Returns

Return type cobra.ReactionSummary

See also:

`Metabolite.summary()`, `Model.summary()`

__str__ (*self*)

Return str(self).

_repr_html_ (*self*)

cobra.core.singleton

Define the singleton meta class.

Module Contents

Classes

Singleton	Implementation of the singleton pattern as a meta class.

class cobra.core.singleton.**Singleton**
Bases: type

Implementation of the singleton pattern as a meta class.

_instances

__call__(*cls*, **args*, ***kwargs*)
Override an inheriting class' call.

cobra.core.solution

Provide unified interfaces to optimization solutions.

Module Contents

Classes

Solution	A unified interface to a *cobra.Model* optimization solution.
LegacySolution	Legacy support for an interface to a *cobra.Model* optimization solution.

Functions

get_solution(model, reactions=None, metabolites=None, raise_error=False)	Generate a solution representation of the current solver state.

class cobra.core.solution.**Solution**(*objective_value*, *status*, *fluxes*, *reduced_costs=None*, *shadow_prices=None*, ***kwargs*)
Bases: object

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

Notes

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

objective_value
The (optimal) value for the objective function.

Type float

status
The solver status related to the solution.

Type str

fluxes
Contains the reaction fluxes (primal values of variables).

Type pandas.Series

reduced_costs
Contains reaction reduced costs (dual values of variables).

Type pandas.Series

shadow_prices
Contains metabolite shadow prices (dual values of constraints).

Type pandas.Series

get_primal_by_id

__repr__(*self*)
String representation of the solution instance.

_repr_html_(*self*)

__getitem__(*self*, *reaction_id*)
Return the flux of a reaction.

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

to_frame(*self*)
Return the fluxes and reduced costs as a data frame

class cobra.core.solution.**LegacySolution**(*f*, *x=None*, *x_dict=None*, *y=None*, *y_dict=None*, *solver=None*, *the_time=0*, *status='NA'*, ***kwargs*)
Bases: object

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

f
The objective value

Type float

solver
A string indicating which solver package was used.

Type str

x
List or Array of the fluxes (primal values).

Type iterable

x_dict
A dictionary of reaction IDs that maps to the respective primal values.

Type dict

y
List or Array of the dual values.

Type iterable

y_dict
A dictionary of reaction IDs that maps to the respective dual values.

Type dict

> **Warning:** The LegacySolution class and its interface is deprecated.

__repr__ (*self*)
String representation of the solution instance.

__getitem__ (*self*, *reaction_id*)
Return the flux of a reaction.

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

dress_results (*self*, *model*)
Method could be intended as a decorator.

> **Warning:** deprecated

cobra.core.solution.**get_solution** (*model*, *reactions=None*, *metabolites=None*, *raise_error=False*)
Generate a solution representation of the current solver state.

Parameters

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

Returns

Return type *cobra.Solution*

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

cobra.core.species

Module Contents

Classes

Species	Species is a class for holding information regarding

class cobra.core.species.**Species** (*id=None*, *name=None*)
Bases: *cobra.core.object.Object*

Species is a class for holding information regarding a chemical Species

Parameters

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

property reactions(*self*)

__getstate__(*self*)
Remove the references to container reactions when serializing to avoid problems associated with recursion.

copy(*self*)
When copying a reaction, it is necessary to deepcopy the components so the list references aren't carried over.

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

This should be fixed with self.__deepcopy__ if possible

property model(*self*)

Package Contents

Classes

Configuration	Define the configuration to be singleton based.
DictList	A combined dict and list
Gene	A Gene in a cobra model
Metabolite	Metabolite is a class for holding information regarding
Model	Class representation for a cobra model
Object	Defines common behavior of object in cobra.core
Reaction	Reaction is a class for holding information regarding
Group	Manage groups via this implementation of the SBML group specification.
Solution	A unified interface to a *cobra.Model* optimization solution.
LegacySolution	Legacy support for an interface to a *cobra.Model* optimization solution.
Species	Species is a class for holding information regarding
MetaboliteSummary	Define the metabolite summary.
Summary	Define the abstract base summary.

Functions

get_solution(model, reactions=None, metabolites=None, raise_error=False)	Generate a solution representation of the current solver state.

class cobra.core.**Configuration**
Bases: six.with_metaclass()

Define the configuration to be singleton based.

class cobra.core.**DictList**(**args*)
Bases: list

A combined dict and list

This object behaves like a list, but has the O(1) speed benefits of a dict when looking up elements by their id.

has_id (*self*, *id*)

_check (*self*, *id*)
make sure duplicate id's are not added. This function is called before adding in elements.

_generate_index (*self*)
rebuild the _dict index

get_by_id (*self*, *id*)
return the element with a matching id

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

get_by_any (*self*, *iterable*)
Get a list of members using several different ways of indexing

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

Returns a list of members

Return type list

query (*self*, *search_function*, *attribute=None*)
Query the list

Parameters

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

Returns a new list of objects which match the query

Return type *DictList*

Examples

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

_replace_on_id (*self*, *new_object*)
Replace an object by another with the same id.

append (*self*, *object*)
append object to end

union (*self*, *iterable*)
adds elements with id's not already in the model

extend(*self*, *iterable*)
extend list by appending elements from the iterable

_extend_nocheck(*self*, *iterable*)
extends without checking for uniqueness

This function should only be used internally by DictList when it can guarantee elements are already unique (as in when coming from self or other DictList). It will be faster because it skips these checks.

__sub__(*self*, *other*)
x.__sub__(y) <==> x - y

Parameters **other** (*iterable*) – other must contain only unique id's present in the list

__isub__(*self*, *other*)
x.__sub__(y) <==> x -= y

Parameters **other** (*iterable*) – other must contain only unique id's present in the list

__add__(*self*, *other*)
x.__add__(y) <==> x + y

Parameters **other** (*iterable*) – other must contain only unique id's which do not intersect with self

__iadd__(*self*, *other*)
x.__iadd__(y) <==> x += y

Parameters **other** (*iterable*) – other must contain only unique id's whcih do not intersect with self

__reduce__(*self*)
Helper for pickle.

__getstate__(*self*)
gets internal state

This is only provided for backwards compatibility so older versions of cobrapy can load pickles generated with cobrapy. In reality, the "_dict" state is ignored when loading a pickle

__setstate__(*self*, *state*)
sets internal state

Ignore the passed in state and recalculate it. This is only for compatibility with older pickles which did not correctly specify the initialization class

index(*self*, *id*, **args*)
Determine the position in the list

id: A string or a `Object`

__contains__(*self*, *object*)
DictList.__contains__(object) <==> object in DictList

object: str or `Object`

__copy__(*self*)

insert(*self*, *index*, *object*)
insert object before index

pop(*self*, **args*)
remove and return item at index (default last).

add(*self*, *x*)
Opposite of *remove*. Mirrors set.add

remove (*self*, *x*)

> **Warning:** Internal use only

reverse (*self*)
reverse *IN PLACE*

sort (*self*, *cmp=None*, *key=None*, *reverse=False*)
stable sort *IN PLACE*

cmp(x, y) -> -1, 0, 1

__getitem__ (*self*, *i*)
x.__getitem__(y) <==> x[y]

__setitem__ (*self*, *i*, *y*)
Set self[key] to value.

__delitem__ (*self*, *index*)
Delete self[key].

__getslice__ (*self*, *i*, *j*)

__setslice__ (*self*, *i*, *j*, *y*)

__delslice__ (*self*, *i*, *j*)

__getattr__ (*self*, *attr*)

__dir__ (*self*)
Default dir() implementation.

class cobra.core.**Gene**(*id=None*, *name=''*, *functional=True*)
Bases: *cobra.core.species.Species*

A Gene in a cobra model

Parameters

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

property functional (*self*)
A flag indicating if the gene is functional.

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

knock_out (*self*)
Knockout gene by marking it as non-functional and setting all associated reactions bounds to zero.

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

remove_from_model (*self*, *model=None*, *make_dependent_reactions_nonfunctional=True*)
Removes the association

Parameters

- **model** (*cobra model*) – The model to remove the gene from
- **make_dependent_reactions_nonfunctional** (*bool*) – If True then replace the gene with 'False' in the gene association, else replace the gene with 'True'

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

_repr_html_(*self*)

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

Bases: *cobra.core.species.Species*

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

Parameters

- **id** (*str*) – the identifier to associate with the metabolite
- **formula** (*str*) – Chemical formula (e.g. H2O)
- **name** (*str*) – A human readable name.
- **charge** (*float*) – The charge number of the metabolite
- **compartment** (*str or None*) – Compartment of the metabolite.

_set_id_with_model(*self*, *value*)

property constraint(*self*)

Get the constraints associated with this metabolite from the solve

Returns the optlang constraint for this metabolite

Return type optlang.<interface>.Constraint

property elements(*self*)

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

property formula_weight(*self*)

Calculate the formula weight

property y(*self*)

The shadow price for the metabolite in the most recent solution

Shadow prices are computed from the dual values of the bounds in the solution.

property shadow_price(*self*)

The shadow price in the most recent solution.

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

Warning:

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

Raises

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

Examples

```
>>> import cobra
>>> import cobra.test
>>> model = cobra.test.create_test_model("textbook")
>>> solution = model.optimize()
>>> model.metabolites.glc__D_e.shadow_price
-0.09166474637510488
>>> solution.shadow_prices.glc__D_e
-0.09164746375104883
```

remove_from_model (*self*, *destructive=False*)
Removes the association from self.model

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

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

summary (*self*, *solution=None*, *threshold=0.01*, *fva=None*, *names=False*, *float_format='{:.3g}'.format*)
Create a summary of the producing and consuming fluxes.

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

Parameters

- **solution** (*cobra.Solution, optional*) – A previous model solution to use for generating the summary. If None, the summary method will resolve the model. Note that the solution object must match the model, i.e., changes to the model such as changed bounds, added or removed reactions are not taken into account by this method (default None).
- **threshold** (*float, optional*) – Threshold below which fluxes are not reported. May not be smaller than the model tolerance (default 0.01).
- **fva** (*pandas.DataFrame or float, optional*) – Whether or not to include flux variability analysis in the output. If given, fva should either be a previous FVA solution matching the model or a float between 0 and 1 representing the fraction of the optimum objective to be searched (default None).
- **names** (*bool, optional*) – Emit reaction and metabolite names rather than identifiers (default False).
- **float_format** (*callable, optional*) – Format string for floats (default '{:3G}'.format).

Returns

Return type cobra.MetaboliteSummary

See also:

Reaction.summary(), *Model.summary()*

_repr_html_ (*self*)

class cobra.core.**Model** (*id_or_model=None*, *name=None*)
Bases: *cobra.core.object.Object*

Class representation for a cobra model

Parameters

- **id_or_model** (*Model, string*) – Either an existing Model object in which case a new model object is instantiated with the same properties as the original model, or an identifier to associate with the model as a string.

- **name** (*string*) – Human readable name for the model

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

Type *DictList*

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

Type *DictList*

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

Type *DictList*

groups
A DictList where the key is the group identifier and the value a Group

Type *DictList*

solution
The last obtained solution from optimizing the model.

Type *Solution*

__setstate__(*self*, *state*)
Make sure all cobra.Objects in the model point to the model.

__getstate__(*self*)
Get state for serialization.

Ensures that the context stack is cleared prior to serialization, since partial functions cannot be pickled reliably.

property solver(*self*)
Get or set the attached solver instance.

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

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

Examples

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

property tolerance(*self*)

property description(*self*)

get_metabolite_compartments(*self*)
Return all metabolites' compartments.

property compartments(*self*)

property medium(*self*)

__add__(*self*, *other_model*)
Add the content of another model to this model (+).

The model is copied as a new object, with a new model identifier, and copies of all the reactions in the other model are added to this model. The objective is the sum of the objective expressions for the two models.

__iadd__ (*self*, *other_model*)
Incrementally add the content of another model to this model (+=).

Copies of all the reactions in the other model are added to this model. The objective is the sum of the objective expressions for the two models.

copy (*self*)
Provides a partial 'deepcopy' of the Model. All of the Metabolite, Gene, and Reaction objects are created anew but in a faster fashion than deepcopy

add_metabolites (*self*, *metabolite_list*)
Will add a list of metabolites to the model object and add new constraints accordingly.

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

Parameters **metabolite_list** (A list of *cobra.core.Metabolite* objects) –

remove_metabolites (*self*, *metabolite_list*, *destructive=False*)
Remove a list of metabolites from the the object.

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

Parameters

- **metabolite_list** (*list*) – A list with *cobra.Metabolite* objects as elements.
- **destructive** (*bool*) – If False then the metabolite is removed from all associated reactions. If True then all associated reactions are removed from the Model.

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

Parameters

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

add_boundary (*self*, *metabolite*, *type='exchange'*, *reaction_id=None*, *lb=None*, *ub=None*, *sbo_term=None*)
Add a boundary reaction for a given metabolite.

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

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

Parameters

- **metabolite** (cobra.Metabolite) – Any given metabolite. The compartment is not checked but you are encouraged to stick to the definition of exchanges and sinks.
- **type** (*str, {"exchange", "demand", "sink"}*) – Using one of the pre-defined reaction types is easiest. If you want to create your own kind of boundary reaction choose any other string, e.g., 'my-boundary'.
- **reaction_id** (*str, optional*) – The ID of the resulting reaction. This takes precedence over the auto-generated identifiers but beware that it might make boundary reactions harder to identify afterwards when using *model.boundary* or specifically *model.exchanges* etc.

- **lb** (*float, optional*) – The lower bound of the resulting reaction.
- **ub** (*float, optional*) – The upper bound of the resulting reaction.
- **sbo_term** (*str, optional*) – A correct SBO term is set for the available types. If a custom type is chosen, a suitable SBO term should also be set.

Returns The created boundary reaction.

Return type *cobra.Reaction*

Examples

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

add_reactions (*self*, *reaction_list*)
Add reactions to the model.

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

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

Parameters **reaction_list** (*list*) – A list of *cobra.Reaction* objects

remove_reactions (*self*, *reactions*, *remove_orphans=False*)
Remove reactions from the model.

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

Parameters

- **reactions** (*list*) – A list with reactions (*cobra.Reaction*), or their id's, to remove
- **remove_orphans** (*bool*) – Remove orphaned genes and metabolites from the model as well

add_groups (*self*, *group_list*)
Add groups to the model.

Groups with identifiers identical to a group already in the model are ignored.

If any group contains members that are not in the model, these members are added to the model as well. Only metabolites, reactions, and genes can have groups.

Parameters **group_list** (*list*) – A list of *cobra.Group* objects to add to the model.

remove_groups (*self*, *group_list*)
Remove groups from the model.

Members of each group are not removed from the model (i.e. metabolites, reactions, and genes in the group stay in the model after any groups containing them are removed).

Parameters **group_list** (*list*) – A list of *cobra.Group* objects to remove from the model.

get_associated_groups (*self*, *element*)
Returns a list of groups that an element (reaction, metabolite, gene) is associated with.

Parameters **element** (*cobra.Reaction*, *cobra.Metabolite*, or *cobra.Gene*) –

Returns All groups that the provided object is a member of

Return type list of *cobra.Group*

add_cons_vars(*self*, *what*, ***kwargs*)

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

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

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

Parameters

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

remove_cons_vars(*self*, *what*)

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

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

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

property problem(*self*)

The interface to the model's underlying mathematical problem.

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

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

Return type optlang.interface

property variables(*self*)

The mathematical variables in the cobra model.

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

Returns A container with all associated variables.

Return type optlang.container.Container

property constraints(*self*)

The constraints in the cobra model.

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

Returns A container with all associated constraints.

Return type optlang.container.Container

property boundary(*self*)

Boundary reactions in the model. Reactions that either have no substrate or product.

property exchanges (*self*)
Exchange reactions in model. Reactions that exchange mass with the exterior. Uses annotations and heuristics to exclude non-exchanges such as sink reactions.

property demands (*self*)
Demand reactions in model. Irreversible reactions that accumulate or consume a metabolite in the inside of the model.

property sinks (*self*)
Sink reactions in model. Reversible reactions that accumulate or consume a metabolite in the inside of the model.

_populate_solver (*self*, *reaction_list*, *metabolite_list=None*)
Populate attached solver with constraints and variables that model the provided reactions.

slim_optimize (*self*, *error_value=float('nan')*, *message=None*)
Optimize model without creating a solution object.

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

Parameters

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

Returns The objective value.

Return type float

optimize (*self*, *objective_sense=None*, *raise_error=False*)
Optimize the model using flux balance analysis.

Parameters

- **objective_sense** (*{None, 'maximize' 'minimize'}, optional*) – Whether fluxes should be maximized or minimized. In case of None, the previous direction is used.
- **raise_error** (*bool*) –

  If true, raise an OptimizationError if solver status is not optimal.

Notes

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

repair (*self*, *rebuild_index=True*, *rebuild_relationships=True*)
Update all indexes and pointers in a model

Parameters

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

property objective(*self*)
Get or set the solver objective

Before introduction of the optlang based problems, this function returned the objective reactions as a list. With optlang, the objective is not limited a simple linear summation of individual reaction fluxes, making that return value ambiguous. Henceforth, use *cobra.util.solver.linear_reaction_coefficients* to get a dictionary of reactions with their linear coefficients (empty if there are none)

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

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

property objective_direction(*self*)
Get or set the objective direction.

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

summary(*self*, *solution=None*, *threshold=0.01*, *fva=None*, *names=False*, *float_format='{:.3g}'.format*)
Create a summary of the exchange fluxes of the model.

Parameters

- **solution** (*cobra.Solution, optional*) – A previous model solution to use for generating the summary. If None, the summary method will resolve the model. Note that the solution object must match the model, i.e., changes to the model such as changed bounds, added or removed reactions are not taken into account by this method (default None).
- **threshold** (*float, optional*) – Threshold below which fluxes are not reported. May not be smaller than the model tolerance (default 0.01).
- **fva** (*pandas.DataFrame or float, optional*) – Whether or not to include flux variability analysis in the output. If given, fva should either be a previous FVA solution matching the model or a float between 0 and 1 representing the fraction of the optimum objective to be searched (default None).
- **names** (*bool, optional*) – Emit reaction and metabolite names rather than identifiers (default False).
- **float_format** (*callable, optional*) – Format string for floats (default `'{:3G}'.format`).

Returns

Return type cobra.ModelSummary

See also:

Reaction.summary(), *Metabolite.summary()*

__enter__(*self*)
Record all future changes to the model, undoing them when a call to __exit__ is received

__exit__(*self*, *type*, *value*, *traceback*)
Pop the top context manager and trigger the undo functions

merge(*self*, *right*, *prefix_existing=None*, *inplace=True*, *objective='left'*)
Merge two models to create a model with the reactions from both models.

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

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

prefix_existing [string] Prefix the reaction identifier in the right that already exist in the left model with this string.

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

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

_repr_html_(*self*)

class cobra.core.**Object**(*id=None*, *name=''*)
Bases: *object*

Defines common behavior of object in cobra.core

property id(*self*)

_set_id_with_model(*self*, *value*)

__getstate__(*self*)
To prevent excessive replication during deepcopy.

__repr__(*self*)
Return repr(self).

__str__(*self*)
Return str(self).

class cobra.core.**Reaction**(*id=None*, *name=''*, *subsystem=''*, *lower_bound=0.0*, *upper_bound=None*)
Bases: *cobra.core.object.Object*

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

Reactions are by default irreversible with bounds *(0.0, cobra.Configuration().upper_bound)* if no bounds are provided on creation. To create an irreversible reaction use *lower_bound=None*, resulting in reaction bounds of *(cobra.Configuration().lower_bound, cobra.Configuration().upper_bound)*.

Parameters

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

__radd__

_set_id_with_model(*self*, *value*)

property reverse_id(*self*)
Generate the id of reverse_variable from the reaction's id.

property flux_expression(*self*)
Forward flux expression

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

Return type sympy expression

property forward_variable(*self*)
An optlang variable representing the forward flux

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

Return type optlang.interface.Variable

property reverse_variable(*self*)
An optlang variable representing the reverse flux

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

Return type optlang.interface.Variable

property objective_coefficient(*self*)
Get the coefficient for this reaction in a linear objective (float)

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

__copy__(*self*)

__deepcopy__(*self*, *memo*)

static _check_bounds(*lb*, *ub*)

update_variable_bounds(*self*)

property lower_bound(*self*)
Get or set the lower bound

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

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

property upper_bound(*self*)
Get or set the upper bound

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

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

property bounds(*self*)
Get or set the bounds directly from a tuple

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

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

property flux(*self*)
The flux value in the most recent solution.

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

Warning:

- Accessing reaction fluxes through a *Solution* object is the safer, preferred, and only guaranteed to be correct way. You can see how to do so easily in the examples.

- Reaction flux is retrieved from the currently defined *self._model.solver*. The solver status is checked but there are no guarantees that the current solver state is the one you are looking for.
- If you modify the underlying model after an optimization, you will retrieve the old optimization values.

Raises

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

Examples

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

property reduced_cost(*self*)

The reduced cost in the most recent solution.

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

Warning:

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

Raises

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

Examples

```
>>> import cobra.test
>>> model = cobra.test.create_test_model("textbook")
>>> solution = model.optimize()
>>> model.reactions.PFK.reduced_cost
-8.673617379884035e-18
>>> solution.reduced_costs.PFK
-8.6736173798840355e-18
```

property metabolites (*self*)

property genes (*self*)

property gene_reaction_rule (*self*)

property gene_name_reaction_rule (*self*)
Display gene_reaction_rule with names intead.

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

property functional (*self*)
All required enzymes for reaction are functional.

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

Return type bool

property x (*self*)
The flux through the reaction in the most recent solution.

Flux values are computed from the primal values of the variables in the solution.

property y (*self*)
The reduced cost of the reaction in the most recent solution.

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

property reversibility (*self*)
Whether the reaction can proceed in both directions (reversible)

This is computed from the current upper and lower bounds.

property boundary (*self*)
Whether or not this reaction is an exchange reaction.

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

property model (*self*)
returns the model the reaction is a part of

_update_awareness (*self*)
Make sure all metabolites and genes that are associated with this reaction are aware of it.

remove_from_model (*self*, *remove_orphans=False*)
Removes the reaction from a model.

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

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

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

delete (*self*, *remove_orphans=False*)
Removes the reaction from a model.

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

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

Deprecated, use *reaction.remove_from_model* instead.

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

__setstate__(*self*, *state*)
Probably not necessary to set _model as the cobra.Model that contains self sets the _model attribute for all metabolites and genes in the reaction.

However, to increase performance speed we do want to let the metabolite and gene know that they are employed in this reaction

copy(*self*)
Copy a reaction

The referenced metabolites and genes are also copied.

__add__(*self*, *other*)
Add two reactions

The stoichiometry will be the combined stoichiometry of the two reactions, and the gene reaction rule will be both rules combined by an and. All other attributes (i.e. reaction bounds) will match those of the first reaction

__iadd__(*self*, *other*)

__sub__(*self*, *other*)

__isub__(*self*, *other*)

__imul__(*self*, *coefficient*)
Scale coefficients in a reaction by a given value

E.g. A -> B becomes 2A -> 2B.

If coefficient is less than zero, the reaction is reversed and the bounds are swapped.

__mul__(*self*, *coefficient*)

property reactants(*self*)
Return a list of reactants for the reaction.

property products(*self*)
Return a list of products for the reaction

get_coefficient(*self*, *metabolite_id*)
Return the stoichiometric coefficient of a metabolite.

Parameters **metabolite_id** (*str or cobra.Metabolite*) –

get_coefficients(*self*, *metabolite_ids*)
Return the stoichiometric coefficients for a list of metabolites.

Parameters **metabolite_ids** (*iterable*) – Containing str or ``cobra.Metabolite``s.

add_metabolites(*self*, *metabolites_to_add*, *combine=True*, *reversibly=True*)
Add metabolites and stoichiometric coefficients to the reaction. If the final coefficient for a metabolite is 0 then it is removed from the reaction.

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

Parameters

- **metabolites_to_add** (*dict*) – Dictionary with metabolite objects or metabolite identifiers as keys and coefficients as values. If keys are strings (name of a metabolite) the reaction must already be part of a model and a metabolite with the given name must exist in the model.

- **combine** (*bool*) – Describes behavior a metabolite already exists in the reaction. True causes the coefficients to be added. False causes the coefficient to be replaced.
- **reversibly** (*bool*) – Whether to add the change to the context to make the change reversibly or not (primarily intended for internal use).

subtract_metabolites (*self*, *metabolites*, *combine=True*, *reversibly=True*)
Subtract metabolites from a reaction.

That means add the metabolites with -1*coefficient. If the final coefficient for a metabolite is 0 then the metabolite is removed from the reaction.

Notes

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

Parameters

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

property reaction (*self*)
Human readable reaction string

build_reaction_string (*self*, *use_metabolite_names=False*)
Generate a human readable reaction string

check_mass_balance (*self*)
Compute mass and charge balance for the reaction

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

property compartments (*self*)
lists compartments the metabolites are in

get_compartments (*self*)
lists compartments the metabolites are in

_associate_gene (*self*, *cobra_gene*)
Associates a cobra.Gene object with a cobra.Reaction.

Parameters **cobra_gene** (*cobra.core.Gene.Gene*) –

_dissociate_gene (*self*, *cobra_gene*)
Dissociates a cobra.Gene object with a cobra.Reaction.

Parameters **cobra_gene** (*cobra.core.Gene.Gene*) –

knock_out (*self*)
Knockout reaction by setting its bounds to zero.

build_reaction_from_string (*self*, *reaction_str*, *verbose=True*, *fwd_arrow=None*, *rev_arrow=None*, *reversible_arrow=None*, *term_split='+'*)
Builds reaction from reaction equation reaction_str using parser

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

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

Parameters

- **reaction_str** (*string*) – a string containing a reaction formula (equation)
- **verbose** (*bool*) – setting verbosity of function
- **fwd_arrow** (*re.compile*) – for forward irreversible reaction arrows
- **rev_arrow** (*re.compile*) – for backward irreversible reaction arrows
- **reversible_arrow** (*re.compile*) – for reversible reaction arrows
- **term_split** (*string*) – dividing individual metabolite entries

summary(*self*, *solution=None*, *threshold=0.01*, *fva=None*, *names=False*, *float_format='{:.3g}'.format*)

Create a summary of the producing and consuming fluxes of the reaction.

Parameters

- **solution** (*cobra.Solution, optional*) – A previous model solution to use for generating the summary. If None, the summary method will resolve the model. Note that the solution object must match the model, i.e., changes to the model such as changed bounds, added or removed reactions are not taken into account by this method (default None).
- **threshold** (*float, optional*) – Threshold below which fluxes are not reported. May not be smaller than the model tolerance (default 0.01).
- **fva** (*pandas.DataFrame or float, optional*) – Whether or not to include flux variability analysis in the output. If given, fva should either be a previous FVA solution matching the model or a float between 0 and 1 representing the fraction of the optimum objective to be searched (default None).
- **names** (*bool, optional*) – Emit reaction and metabolite names rather than identifiers (default False).
- **float_format** (*callable, optional*) – Format string for floats (default `'{:3G}'.format`).

Returns

Return type cobra.ReactionSummary

See also:

Metabolite.summary(), *Model.summary()*

__str__(*self*)

Return str(self).

_repr_html_(*self*)

class cobra.core.**Group**(*id*, *name=''*, *members=None*, *kind=None*)

Bases: *cobra.core.object.Object*

Manage groups via this implementation of the SBML group specification.

Group is a class for holding information regarding a pathways, subsystems, or other custom groupings of objects within a cobra.Model object.

Parameters

- **id** (*str*) – The identifier to associate with this group
- **name** (*str, optional*) – A human readable name for the group

- **members** (*iterable, optional*) – A list object containing references to cobra.Model-associated objects that belong to the group.
- **kind** (*{"collection", "classification", "partonomy"}, optional*) – The kind of group, as specified for the Groups feature in the SBML level 3 package specification. Can be any of "classification", "partonomy", or "collection". The default is "collection". Please consult the SBML level 3 package specification to ensure you are using the proper value for kind. In short, members of a "classification" group should have an "is-a" relationship to the group (e.g. member is-a polar compound, or member is-a transporter). Members of a "partonomy" group should have a "part-of" relationship (e.g. member is part-of glycolysis). Members of a "collection" group do not have an implied relationship between the members, so use this value for kind when in doubt (e.g. member is a gap-filled reaction, or member is involved in a disease phenotype).

KIND_TYPES = ['collection', 'classification', 'partonomy']

__len__(*self*)

property members(*self*)

property kind(*self*)

add_members(*self*, *new_members*)
Add objects to the group.

Parameters **new_members** (*list*) – A list of cobrapy objects to add to the group.

remove_members(*self*, *to_remove*)
Remove objects from the group.

Parameters **to_remove** (*list*) – A list of cobra objects to remove from the group

class cobra.core.**Solution**(*objective_value*, *status*, *fluxes*, *reduced_costs=None*, *shadow_prices=None*, ***kwargs*)
Bases: *object*

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

Notes

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

objective_value
The (optimal) value for the objective function.

Type float

status
The solver status related to the solution.

Type str

fluxes
Contains the reaction fluxes (primal values of variables).

Type pandas.Series

reduced_costs
Contains reaction reduced costs (dual values of variables).

Type pandas.Series

shadow_prices
Contains metabolite shadow prices (dual values of constraints).

Type pandas.Series

get_primal_by_id

__repr__ (*self*)
String representation of the solution instance.

_repr_html_ (*self*)

__getitem__ (*self*, *reaction_id*)
Return the flux of a reaction.

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

to_frame (*self*)
Return the fluxes and reduced costs as a data frame

class cobra.core.**LegacySolution** (*f*, *x=None*, *x_dict=None*, *y=None*, *y_dict=None*, *solver=None*, *the_time=0*, *status='NA'*, ***kwargs*)
Bases: *object*

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

f
The objective value

Type float

solver
A string indicating which solver package was used.

Type str

x
List or Array of the fluxes (primal values).

Type iterable

x_dict
A dictionary of reaction IDs that maps to the respective primal values.

Type dict

y
List or Array of the dual values.

Type iterable

y_dict
A dictionary of reaction IDs that maps to the respective dual values.

Type dict

Warning: The LegacySolution class and its interface is deprecated.

__repr__ (*self*)
String representation of the solution instance.

__getitem__ (*self*, *reaction_id*)
Return the flux of a reaction.

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

dress_results (*self*, *model*)
Method could be intended as a decorator.

Warning: deprecated

cobra.core.**get_solution**(*model*, *reactions=None*, *metabolites=None*, *raise_error=False*)
Generate a solution representation of the current solver state.

Parameters

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

Returns

Return type *cobra.Solution*

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

---

class cobra.core.**Species**(*id=None*, *name=None*)
Bases: *cobra.core.object.Object*

Species is a class for holding information regarding a chemical Species

Parameters

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

property reactions(*self*)

__getstate__(*self*)
Remove the references to container reactions when serializing to avoid problems associated with recursion.

copy(*self*)
When copying a reaction, it is necessary to deepcopy the components so the list references aren't carried over.

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

This should be fixed with self.__deepcopy__ if possible

property model(*self*)

class cobra.core.**MetaboliteSummary**(*metabolite*, *model*, ***kwargs*)
Bases: *cobra.core.summary.Summary*

Define the metabolite summary.

metabolite
The metabolite to summarize.

Type *cobra.Metabolite*

See also:

Summary Parent that defines further attributes.

ReactionSummary, ModelSummary

_generate(*self*)

Returns **flux_summary** – The DataFrame of flux summary data.

Return type pandas.DataFrame

to_frame(*self*)

Returns

Return type A pandas.DataFrame of the summary.

_to_table(*self*)

Returns

Return type A string of the summary table.

class cobra.core.**Summary**(*model*, *solution=None*, *threshold=None*, *fva=None*, *names=False*, *float_format='{:.3G}'.format*, ***kwargs*)

Bases: *object*

Define the abstract base summary.

model
The metabolic model in which to generate a summary description.

Type *cobra.Model*

solution
A solution that matches the given model.

Type *cobra.Solution*

threshold
Threshold below which fluxes are not reported.

Type float, optional

fva
The result of a flux variability analysis (FVA) involving reactions of interest if an FVA was requested.

Type pandas.DataFrame, optional

names
Whether or not to use object names rather than identifiers.

Type bool

float_format
Format string for displaying floats.

Type callable

to_frame()
Return a data frame representation of the summary.

abstract _generate(*self*)
Generate the summary for the required cobra object.

This is an abstract method and thus the subclass needs to implement it.

_process_flux_dataframe(*self*, *flux_dataframe*)
Process a flux DataFrame for convenient downstream analysis.

This method removes flux entries which are below the threshold and also adds information regarding the direction of the fluxes. It is used in both ModelSummary and MetaboliteSummary.

Parameters **flux_dataframe** (*pandas.DataFrame*) – The pandas.DataFrame to process.

Returns

Return type A processed pandas.DataFrame.

abstract to_frame(*self*)
Generate a pandas DataFrame.

This is an abstract method and thus the subclass needs to implement it.

abstract _to_table(*self*)
Generate a pretty-print table.

This is an abstract method and thus the subclass needs to implement it.

__str__(*self*)
Return str(self).

_repr_html_(*self*)

cobra.flux_analysis

Submodules

cobra.flux_analysis.deletion

Module Contents

Functions

_reactions_knockouts_with_restore(model, reactions)	
_get_growth(model)	
_reaction_deletion(model, ids)	
_gene_deletion(model, ids)	
_reaction_deletion_worker(ids)	
_gene_deletion_worker(ids)	
_init_worker(model)	
_multi_deletion(model, entity, element_lists, method='fba', solution=None, processes=None, **kwargs)	Provide a common interface for single or multiple knockouts.
_entities_ids(entities)	
_element_lists(entities, *ids)	
single_reaction_deletion(model, reaction_list=None, method='fba', solution=None, processes=None, **kwargs)	Knock out each reaction from a given list.
single_gene_deletion(model, gene_list=None, method='fba', solution=None, processes=None, **kwargs)	Knock out each gene from a given list.
double_reaction_deletion(model, reaction_list1=None, reaction_list2=None, method='fba', solution=None, processes=None, **kwargs)	Knock out each reaction pair from the combinations of two given lists.
double_gene_deletion(model, gene_list1=None, gene_list2=None, method='fba', solution=None, processes=None, **kwargs)	Knock out each gene pair from the combination of two given lists.

cobra.flux_analysis.deletion.**LOGGER**

cobra.flux_analysis.deletion.**CONFIGURATION**

cobra.flux_analysis.deletion.**_reactions_knockouts_with_restore**(*model*, *reactions*)

cobra.flux_analysis.deletion.**_get_growth**(*model*)

cobra.flux_analysis.deletion.**_reaction_deletion**(*model*, *ids*)

cobra.flux_analysis.deletion.**_gene_deletion**(*model*, *ids*)

cobra.flux_analysis.deletion.**_reaction_deletion_worker**(*ids*)

cobra.flux_analysis.deletion.**_gene_deletion_worker**(*ids*)

cobra.flux_analysis.deletion.**_init_worker**(*model*)

cobra.flux_analysis.deletion.**_multi_deletion**(*model*, *entity*, *element_lists*, *method='fba'*, *solution=None*, *processes=None*, ***kwargs*)

Provide a common interface for single or multiple knockouts.

Parameters

- **model** (cobra.Model) – The metabolic model to perform deletions in.
- **entity** (*'gene' or 'reaction'*) – The entity to knockout (cobra.Gene or cobra.Reaction).
- **element_lists** (*list*) – List of iterables ``cobra.Reaction``s or ``cobra.Gene``s (or their IDs) to be deleted.
- **method** (*{"fba", "moma", "linear moma", "room", "linear room"}, optional*) – Method used to predict the growth rate.
- **solution** (*cobra.Solution, optional*) – A previous solution to use as a reference for (linear) MOMA or ROOM.
- **processes** (*int, optional*) – The number of parallel processes to run. Can speed up the computations if the number of knockouts to perform is large. If not passed, will be set to the number of CPUs found.
- **kwargs** – Passed on to underlying simulation functions.

Returns

A representation of all combinations of entity deletions. The columns are 'growth' and 'status', where

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

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

status [str] The solution's status.

Return type pandas.DataFrame

cobra.flux_analysis.deletion.**_entities_ids**(*entities*)

cobra.flux_analysis.deletion.**_element_lists**(*entities*, **ids*)

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

Knock out each reaction from a given list.

Parameters

- **model** (cobra.Model) – The metabolic model to perform deletions in.
- **reaction_list** (*iterable, optional*) – ``cobra.Reaction``s to be deleted. If not passed, all the reactions from the model are used.
- **method** (*{"fba", "moma", "linear moma", "room", "linear room"}, optional*) – Method used to predict the growth rate.
- **solution** (*cobra.Solution, optional*) – A previous solution to use as a reference for (linear) MOMA or ROOM.

- **processes** (*int, optional*) – The number of parallel processes to run. Can speed up the computations if the number of knockouts to perform is large. If not passed, will be set to the number of CPUs found.
- **kwargs** – Keyword arguments are passed on to underlying simulation functions such as `add_room`.

Returns

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

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

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

status [str] The solution's status.

Return type pandas.DataFrame

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

Knock out each gene from a given list.

Parameters

- **model** (*cobra.Model*) – The metabolic model to perform deletions in.
- **gene_list** (*iterable*) – ``cobra.Gene``s to be deleted. If not passed, all the genes from the model are used.
- **method** (*{"fba", "moma", "linear moma", "room", "linear room"}, optional*) – Method used to predict the growth rate.
- **solution** (*cobra.Solution, optional*) – A previous solution to use as a reference for (linear) MOMA or ROOM.
- **processes** (*int, optional*) – The number of parallel processes to run. Can speed up the computations if the number of knockouts to perform is large. If not passed, will be set to the number of CPUs found.
- **kwargs** – Keyword arguments are passed on to underlying simulation functions such as `add_room`.

Returns

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

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

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

status [str] The solution's status.

Return type pandas.DataFrame

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

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

We say 'pair' here but the order order does not matter.

Parameters

- **model** (*cobra.Model*) – The metabolic model to perform deletions in.

- **reaction_list1** (*iterable, optional*) – First iterable of ``cobra.Reaction``s to be deleted. If not passed, all the reactions from the model are used.
- **reaction_list2** (*iterable, optional*) – Second iterable of ``cobra.Reaction``s to be deleted. If not passed, all the reactions from the model are used.
- **method** (*{"fba", "moma", "linear moma", "room", "linear room"}, optional*) – Method used to predict the growth rate.
- **solution** (cobra.Solution, *optional*) – A previous solution to use as a reference for (linear) MOMA or ROOM.
- **processes** (*int, optional*) – The number of parallel processes to run. Can speed up the computations if the number of knockouts to perform is large. If not passed, will be set to the number of CPUs found.
- **kwargs** – Keyword arguments are passed on to underlying simulation functions such as add_room.

Returns

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

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

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

status [str] The solution's status.

Return type pandas.DataFrame

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

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

We say 'pair' here but the order order does not matter.

Parameters

- **model** (cobra.Model) – The metabolic model to perform deletions in.
- **gene_list1** (*iterable, optional*) – First iterable of ``cobra.Gene``s to be deleted. If not passed, all the genes from the model are used.
- **gene_list2** (*iterable, optional*) – Second iterable of ``cobra.Gene``s to be deleted. If not passed, all the genes from the model are used.
- **method** (*{"fba", "moma", "linear moma", "room", "linear room"}, optional*) – Method used to predict the growth rate.
- **solution** (cobra.Solution, *optional*) – A previous solution to use as a reference for (linear) MOMA or ROOM.
- **processes** (*int, optional*) – The number of parallel processes to run. Can speed up the computations if the number of knockouts to perform is large. If not passed, will be set to the number of CPUs found.
- **kwargs** – Keyword arguments are passed on to underlying simulation functions such as add_room.

Returns

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

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

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

status [str] The solution's status.

Return type pandas.DataFrame

cobra.flux_analysis.fastcc

Provide an implementation of FASTCC.

Module Contents

Functions

_find_sparse_mode(model, rxns, flux_threshold, zero_cutoff)	Perform the LP required for FASTCC.
_flip_coefficients(model, rxns)	Flip the coefficients for optimizing in reverse direction.
fastcc(model, flux_threshold=1.0, zero_cutoff=None)	Check consistency of a metabolic network using FASTCC[1].

cobra.flux_analysis.fastcc.**_find_sparse_mode**(*model*, *rxns*, *flux_threshold*, *zero_cutoff*)

Perform the LP required for FASTCC.

Parameters

- **model** (*cobra.core.Model*) – The cobra model to perform FASTCC on.
- **rxns** (*list of cobra.core.Reactions*) – The reactions to use for LP.
- **flux_threshold** (*float*) – The upper threshold an auxiliary variable can have.
- **zero_cutoff** (*float*) – The cutoff below which flux is considered zero.

Returns **result** – The list of reactions to consider as consistent.

Return type list

cobra.flux_analysis.fastcc.**_flip_coefficients**(*model*, *rxns*)

Flip the coefficients for optimizing in reverse direction.

cobra.flux_analysis.fastcc.**fastcc**(*model*, *flux_threshold=1.0*, *zero_cutoff=None*)

Check consistency of a metabolic network using FASTCC[1].

FASTCC (Fast Consistency Check) is an algorithm for rapid and efficient consistency check in metabolic networks. FASTCC is a pure LP implementation and is low on computation resource demand. FASTCC also circumvents the problem associated with reversible reactions for the purpose. Given a global model, it will generate a consistent global model i.e., remove blocked reactions. For more details on FASTCC, please check[1].

Parameters

- **model** (*cobra.Model*) – The constraint-based model to operate on.
- **flux_threshold** (*float, optional (default 1.0)*) – The flux threshold to consider.

[1] Vlassis N, Pacheco MP, Sauter T (2014) Fast Reconstruction of Compact Context-Specific Metabolic Network Models. PLoS Comput Biol 10(1): e1003424. doi:10.1371/journal.pcbi.1003424

- **zero_cutoff** (*float, optional*) – The cutoff to consider for zero flux (default model.tolerance).

Returns The consistent constraint-based model.

Return type *cobra.Model*

Notes

The LP used for FASTCC is like so: maximize: sum_{i in J} z_i s.t. : z_i in [0, varepsilon] forall i in J, z_i in mathbb{R}_+

v_i ge z_i forall i in J Sv = 0 v in B

References

cobra.flux_analysis.gapfilling

Module Contents

Classes

GapFiller	Class for performing gap filling.

Functions

gapfill(model, universal=None, lower_bound=0.05, penalties=None, demand_reactions=True, exchange_reactions=False, iterations=1)	Perform gapfilling on a model.

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

Bases: object

Class for performing gap filling.

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

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

v_o >= t lb_i <= v_i <= ub_i v_i = 0 if z_i = 0

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

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

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

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

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

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

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

Parameters

- **model** (*cobra.Model*) – The model to perform gap filling on.
- **universal** (*cobra.Model*) – A universal model with reactions that can be used to complete the model.
- **lower_bound** (*float*) – The minimally accepted flux for the objective in the filled model.
- **penalties** (*dict, None*) – A dictionary with keys being 'universal' (all reactions included in the universal model), 'exchange' and 'demand' (all additionally added exchange and demand reactions) for the three reaction types. Can also have reaction identifiers for reaction specific costs. Defaults are 1, 100 and 1 respectively.
- **integer_threshold** (*float*) – The threshold at which a value is considered non-zero (aka integrality threshold). If gapfilled models fail to validate, you may want to lower this value.
- **exchange_reactions** (*bool*) – Consider adding exchange (uptake) reactions for all metabolites in the model.
- **demand_reactions** (*bool*) – Consider adding demand reactions for all metabolites.

References

extend_model (*self*, *exchange_reactions=False*, *demand_reactions=True*)

Extend gapfilling model.

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

Parameters

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

update_costs (*self*)

Update the coefficients for the indicator variables in the objective.

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

add_switches_and_objective (*self*)

Update gapfilling model with switches and the indicator objective.

fill (*self*, *iterations=1*)

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

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

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

Return type iterable

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

validate(*self*, *reactions*)

cobra.flux_analysis.gapfilling.**gapfill**(*model*, *universal=None*, *lower_bound=0.05*, *penalties=None*, *demand_reactions=True*, *exchange_reactions=False*, *iterations=1*)

Perform gapfilling on a model.

See documentation for the class GapFiller.

Parameters

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

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

Return type iterable

Examples

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

cobra.flux_analysis.geometric

Provide an implementation of geometric FBA.

Module Contents

Functions

geometric_fba(model, epsilon=1e-06, max_tries=200, processes=None)	Perform geometric FBA to obtain a unique, centered flux distribution.

cobra.flux_analysis.geometric.**LOGGER**

cobra.flux_analysis.geometric.**geometric_fba**(*model*, *epsilon=1e-06*, *max_tries=200*, *processes=None*)

Perform geometric FBA to obtain a unique, centered flux distribution.

Geometric FBA[1] formulates the problem as a polyhedron and then solves it by bounding the convex hull of the polyhedron. The bounding forms a box around the convex hull which reduces with every iteration and extracts a unique solution in this way.

Parameters

- **model** (*cobra.Model*) – The model to perform geometric FBA on.
- **epsilon** (*float, optional*) – The convergence tolerance of the model (default 1E-06).
- **max_tries** (*int, optional*) – Maximum number of iterations (default 200).
- **processes** (*int, optional*) – The number of parallel processes to run. If not explicitly passed, will be set from the global configuration singleton.

Returns The solution object containing all the constraints required for geometric FBA.

Return type *cobra.Solution*

References

cobra.flux_analysis.helpers

Helper functions for all flux analysis methods.

Module Contents

Functions

normalize_cutoff(model, zero_cutoff=None)	Return a valid zero cutoff value.

cobra.flux_analysis.helpers.**LOGGER**

cobra.flux_analysis.helpers.**normalize_cutoff**(*model*, *zero_cutoff=None*)

Return a valid zero cutoff value.

[1] Smallbone, Kieran & Simeonidis, Vangelis. (2009). Flux balance analysis: A geometric perspective. Journal of theoretical biology.258. 311-5. 10.1016/j.jtbi.2009.01.027.

cobra.flux_analysis.loopless

Provides functions to remove thermodynamically infeasible loops.

Module Contents

Functions

add_loopless(model, zero_cutoff=None)	Modify a model so all feasible flux distributions are loopless.
_add_cycle_free(model, fluxes)	Add constraints for CycleFreeFlux.
loopless_solution(model, fluxes=None)	Convert an existing solution to a loopless one.
loopless_fva_iter(model, reaction, solution=False, zero_cutoff=None)	Plugin to get a loopless FVA solution from single FVA iteration.

cobra.flux_analysis.loopless.**LOGGER**

cobra.flux_analysis.loopless.**add_loopless**(*model*, *zero_cutoff=None*)

Modify a model so all feasible flux distributions are loopless.

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

Parameters

- **model** (*cobra.Model*) – The model to which to add the constraints.
- **zero_cutoff** (*positive float, optional*) – Cutoff used for null space. Coefficients with an absolute value smaller than *zero_cutoff* are considered to be zero (default model.tolerance).

Returns

Return type Nothing

References

cobra.flux_analysis.loopless.**_add_cycle_free**(*model*, *fluxes*)

Add constraints for CycleFreeFlux.

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

Convert an existing solution to a loopless one.

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

Parameters

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

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

Return type *cobra.Solution*

Notes

The returned flux solution has the following properties:

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

References

cobra.flux_analysis.loopless.**loopless_fva_iter**(*model*, *reaction*, *solution=False*, *zero_cutoff=None*)

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

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

reaction

3. the model contains an auxiliary variable called "fva_old_objective" denoting the previous objective

Parameters

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

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

Return type single float or dict

cobra.flux_analysis.moma

Provide minimization of metabolic adjustment (MOMA).

Module Contents

Functions

moma(model, solution=None, linear=True)	Compute a single solution based on (linear) MOMA.
add_moma(model, solution=None, linear=True)	Add constraints and objective representing for MOMA.

cobra.flux_analysis.moma.**moma**(*model*, *solution=None*, *linear=True*)

Compute a single solution based on (linear) MOMA.

Compute a new flux distribution that is at a minimal distance to a previous reference solution. Minimization

of metabolic adjustment (MOMA) is generally used to assess the impact of knock-outs. Thus the typical usage is to provide a wildtype flux distribution as reference and a model in knock-out state.

Parameters

- **model** (cobra.Model) – The model state to compute a MOMA-based solution for.
- **solution** (cobra.Solution, *optional*) – A (wildtype) reference solution.
- **linear** (*bool*, *optional*) – Whether to use the linear MOMA formulation or not (default True).

Returns A flux distribution that is at a minimal distance compared to the reference solution.

Return type *cobra.Solution*

See also:

add_moma() add MOMA constraints and objective

cobra.flux_analysis.moma.**add_moma**(*model*, *solution=None*, *linear=True*)

Add constraints and objective representing for MOMA.

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

Parameters

- **model** (cobra.Model) – The model to add MOMA constraints and objective to.
- **solution** (cobra.Solution, *optional*) – A previous solution to use as a reference. If no solution is given, one will be computed using pFBA.
- **linear** (*bool*, *optional*) – Whether to use the linear MOMA formulation or not (default True).

Notes

In the original MOMA[1] specification one looks for the flux distribution of the deletion (v^d) closest to the fluxes without the deletion (v). In math this means:

minimize sum_i (v^d_i - v_i)^2 s.t. Sv^d = 0

lb_i <= v^d_i <= ub_i

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

minimize sum_i (v^t_i)^2 s.t. Sv^d = 0

v^t = v^d_i - v_i lb_i <= v^d_i <= ub_i

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

In the case of linear MOMA[2], we instead minimize sum_i abs(v^t_i). The linear MOMA is typically significantly faster. Also quadratic MOMA tends to give flux distributions in which all fluxes deviate from the reference fluxes a little bit whereas linear MOMA tends to give flux distributions where the majority of fluxes are the same reference with few fluxes deviating a lot (typical effect of L2 norm vs L1 norm).

The former objective function is saved in the optlang solver interface as "moma_old_objective" and this can be used to immediately extract the value of the former objective after MOMA optimization.

See also:

[1] Segrè, Daniel, Dennis Vitkup, and George M. Church. "Analysis of Optimality in Natural and Perturbed Metabolic Networks." Proceedings of the National Academy of Sciences 99, no. 23 (November 12, 2002): 15112. https://doi.org/10.1073/pnas.232349399.

[2] Becker, Scott A, Adam M Feist, Monica L Mo, Gregory Hannum, Bernhard Ø Palsson, and Markus J Herrgard. "Quantitative Prediction of Cellular Metabolism with Constraint-Based Models: The COBRA Toolbox." Nature Protocols 2 (March 29, 2007): 727.

pfba() parsimonious FBA

References

cobra.flux_analysis.parsimonious

Module Contents

Functions

optimize_minimal_flux(*args, **kwargs)	
pfba(model, fraction_of_optimum=1.0, objective=None, reactions=None)	Perform basic pFBA (parsimonious Enzyme Usage Flux Balance Analysis)
add_pfba(model, objective=None, fraction_of_optimum=1.0)	Add pFBA objective

cobra.flux_analysis.parsimonious.**LOGGER**

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

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

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

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

Parameters

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

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

Return type *cobra.Solution*

References

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

Add pFBA objective

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

See also:

pfba()

Parameters

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

cobra.flux_analysis.phenotype_phase_plane

Module Contents

Functions

production_envelope(model, reactions, objective=None, carbon_sources=None, points=20, threshold=None)	Calculate the objective value conditioned on all combinations of
add_envelope(model, reactions, grid, c_input, c_output, threshold)	
total_yield(input_fluxes, input_elements, output_flux, output_elements)	Compute total output per input unit.
reaction_elements(reaction)	Split metabolites into the atoms times their stoichiometric coefficients.
reaction_weight(reaction)	Return the metabolite weight times its stoichiometric coefficient.
total_components_flux(flux, components, consumption=True)	Compute the total components consumption or production flux.
find_carbon_sources(model)	Find all active carbon source reactions.

cobra.flux_analysis.phenotype_phase_plane.**LOGGER**

cobra.flux_analysis.phenotype_phase_plane.**production_envelope**(*model*, *reactions*, *objective=None*, *carbon_sources=None*, *points=20*, *threshold=None*)

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

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

Parameters

- **model** (cobra.Model) – The model to compute the production envelope for.
- **reactions** (*list or string*) – A list of reactions, reaction identifiers or a single reaction.
- **objective** (*string, dict, model.solver.interface.Objective, optional*) – The objective (reaction) to use for the production envelope. Use the model's current objective if left missing.
- **carbon_sources** (*list or string, optional*) – One or more reactions or reaction identifiers that are the source of carbon for computing carbon (mol carbon

in output over mol carbon in input) and mass yield (gram product over gram output). Only objectives with a carbon containing input and output metabolite is supported. Will identify active carbon sources in the medium if none are specified.

- **points** (*int, optional*) – The number of points to calculate production for.
- **threshold** (*float, optional*) – A cut-off under which flux values will be considered to be zero (default model.tolerance).

Returns

A data frame with one row per evaluated point and

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

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

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

Return type pandas.DataFrame

Examples

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

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

cobra.flux_analysis.phenotype_phase_plane.**total_yield**(*input_fluxes*, *input_elements*, *output_flux*, *output_elements*)

Compute total output per input unit.

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

Parameters

- **input_fluxes** (*list*) – A list of input reaction fluxes in the same order as the input_components.
- **input_elements** (*list*) – A list of reaction components which are in turn list of numbers.
- **output_flux** (*float*) – The output flux value.
- **output_elements** (*list*) – A list of stoichiometrically weighted output reaction components.

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

Return type float

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

Split metabolites into the atoms times their stoichiometric coefficients.

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

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

Return type list

cobra.flux_analysis.phenotype_phase_plane.**reaction_weight**(*reaction*)
Return the metabolite weight times its stoichiometric coefficient.

cobra.flux_analysis.phenotype_phase_plane.**total_components_flux**(*flux*, *components*, *consumption=True*)
Compute the total components consumption or production flux.

Parameters

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

cobra.flux_analysis.phenotype_phase_plane.**find_carbon_sources**(*model*)
Find all active carbon source reactions.

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

Returns The medium reactions with carbon input flux.

Return type list

cobra.flux_analysis.reaction

functions for analyzing / creating objective functions

Module Contents

Functions

assess(model, reaction, flux_coefficient_cutoff=0.001, solver=None)	Assesses production capacity.
assess_component(model, reaction, side, flux_coefficient_cutoff=0.001, solver=None)	Assesses the ability of the model to provide sufficient precursors,
_optimize_or_value(model, value=0.0, solver=None)	
assess_precursors(model, reaction, flux_coefficient_cutoff=0.001, solver=None)	Assesses the ability of the model to provide sufficient precursors for
assess_products(model, reaction, flux_coefficient_cutoff=0.001, solver=None)	Assesses whether the model has the capacity to absorb the products of

cobra.flux_analysis.reaction.**assess**(*model*, *reaction*, *flux_coefficient_cutoff=0.001*, *solver=None*)
Assesses production capacity.

Assesses the capacity of the model to produce the precursors for the reaction and absorb the production of the reaction while the reaction is operating at, or above, the specified cutoff.

Parameters

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

Returns True if the model can produce the precursors and absorb the products for the reaction operating at, or above, flux_coefficient_cutoff. Otherwise, a dictionary of {'precursor': Status, 'product': Status}. Where Status is the results from assess_precursors and assess_products, respectively.

Return type bool or dict

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

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

Parameters

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

Returns True if the precursors can be simultaneously produced at the specified cutoff. False, if the model has the capacity to produce each individual precursor at the specified threshold but not all precursors at the required level simultaneously. Otherwise a dictionary of the required and the produced fluxes for each reactant that is not produced in sufficient quantities.

Return type bool or dict

cobra.flux_analysis.reaction.**_optimize_or_value**(*model*, *value=0.0*, *solver=None*)

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

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

Deprecated: use assess_component instead

Parameters

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

Returns True if the precursors can be simultaneously produced at the specified cutoff. False, if the model has the capacity to produce each individual precursor at the specified threshold but not all precursors at the required level simultaneously. Otherwise a dictionary of

the required and the produced fluxes for each reactant that is not produced in sufficient quantities.

Return type bool or dict

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

Assesses whether the model has the capacity to absorb the products of a reaction at a given flux rate.

Useful for identifying which components might be blocking a reaction from achieving a specific flux rate.

Deprecated: use assess_component instead

Parameters

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

Returns True if the model has the capacity to absorb all the reaction products being simultaneously given the specified cutoff. False, if the model has the capacity to absorb each individual product but not all products at the required level simultaneously. Otherwise a dictionary of the required and the capacity fluxes for each product that is not absorbed in sufficient quantities.

Return type bool or dict

cobra.flux_analysis.room

Provide regulatory on/off minimization (ROOM).

Module Contents

Functions

room(model, solution=None, linear=False, delta=0.03, epsilon=0.001)	Compute a single solution based on regulatory on/off minimization (ROOM).
add_room(model, solution=None, linear=False, delta=0.03, epsilon=0.001)	Add constraints and objective for ROOM.

cobra.flux_analysis.room.**room**(*model*, *solution=None*, *linear=False*, *delta=0.03*, *epsilon=0.001*)

Compute a single solution based on regulatory on/off minimization (ROOM).

Compute a new flux distribution that minimizes the number of active reactions needed to accommodate a previous reference solution. Regulatory on/off minimization (ROOM) is generally used to assess the impact of knock-outs. Thus the typical usage is to provide a wildtype flux distribution as reference and a model in knock-out state.

Parameters

- **model** (cobra.Model) – The model state to compute a ROOM-based solution for.
- **solution** (cobra.Solution, *optional*) – A (wildtype) reference solution.

- **linear** (*bool, optional*) – Whether to use the linear ROOM formulation or not (default False).
- **delta** (*float, optional*) – The relative tolerance range (additive) (default 0.03).
- **epsilon** (*float, optional*) – The absolute tolerance range (multiplicative) (default 0.001).

Returns A flux distribution with minimal active reaction changes compared to the reference.

Return type *cobra.Solution*

See also:

add_room() add ROOM constraints and objective

cobra.flux_analysis.room.**add_room**(*model*, *solution=None*, *linear=False*, *delta=0.03*, *epsilon=0.001*)

Add constraints and objective for ROOM.

This function adds variables and constraints for applying regulatory on/off minimization (ROOM) to the model.

Parameters

- **model** (cobra.Model) – The model to add ROOM constraints and objective to.
- **solution** (cobra.Solution, *optional*) – A previous solution to use as a reference. If no solution is given, one will be computed using pFBA.
- **linear** (*bool, optional*) – Whether to use the linear ROOM formulation or not (default False).
- **delta** (*float, optional*) – The relative tolerance range which is additive in nature (default 0.03).
- **epsilon** (*float, optional*) – The absolute range of tolerance which is multiplicative (default 0.001).

Notes

The formulation used here is the same as stated in the original paper[1]. The mathematical expression is given below:

minimize sum_{i=1}^m y^i s.t. Sv = 0

v_min <= v <= v_max v_j = 0 j A for 1 <= i <= m v_i - y_i(v_{max,i} - w_i^u) <= w_i^u (1) v_i - y_i(v_{min,i} - w_i^l) <= w_i^l (2) y_i {0,1} (3) w_i^u = w_i + deltaǀw_iǀ + epsilon w_i^l = w_i - deltaǀw_iǀ - epsilon

So, for the linear version of the ROOM , constraint (3) is relaxed to 0 <= y_i <= 1.

See also:

pfba() parsimonious FBA

[1] Tomer Shlomi, Omer Berkman and Eytan Ruppin, "Regulatory on/off minimization of metabolic flux changes after genetic perturbations", PNAS 2005 102 (21) 7695-7700; doi:10.1073/pnas.0406346102

References

cobra.flux_analysis.variability

Module Contents

Functions

_init_worker(model, loopless, sense)	Initialize a global model object for multiprocessing.
_fva_step(reaction_id)	
flux_variability_analysis(model, reaction_list=None, loopless=False, fraction_of_optimum=1.0, pfba_factor=None, processes=None)	Determine the minimum and maximum possible flux value for each reaction.
find_blocked_reactions(model, reaction_list=None, zero_cutoff=None, open_exchanges=False, processes=None)	Find reactions that cannot carry any flux.
find_essential_genes(model, threshold=None, processes=None)	Return a set of essential genes.
find_essential_reactions(model, threshold=None, processes=None)	Return a set of essential reactions.

cobra.flux_analysis.variability.**LOGGER**

cobra.flux_analysis.variability.**CONFIGURATION**

cobra.flux_analysis.variability.**_init_worker**(*model*, *loopless*, *sense*)
Initialize a global model object for multiprocessing.

cobra.flux_analysis.variability.**_fva_step**(*reaction_id*)

cobra.flux_analysis.variability.**flux_variability_analysis**(*model*, *reaction_list=None*, *loopless=False*, *fraction_of_optimum=1.0*, *pfba_factor=None*, *processes=None*)
Determine the minimum and maximum possible flux value for each reaction.

Parameters

- **model** (cobra.Model) – The model for which to run the analysis. It will *not* be modified.
- **reaction_list** (*list of cobra.Reaction or str, optional*) – The reactions for which to obtain min/max fluxes. If None will use all reactions in the model (default).
- **loopless** (*boolean, optional*) – Whether to return only loopless solutions. This is significantly slower. Please also refer to the notes.
- **fraction_of_optimum** (*float, optional*) – Must be <= 1.0. Requires that the objective value is at least the fraction times maximum objective value. A value of 0.85 for instance means that the objective has to be at least at 85% percent of its maximum.
- **pfba_factor** (*float, optional*) – Add an additional constraint to the model that requires the total sum of absolute fluxes must not be larger than this value times the smallest possible sum of absolute fluxes, i.e., by setting the value to 1.1 the total sum of absolute fluxes must not be more than 10% larger than the pFBA solution.

Since the pFBA solution is the one that optimally minimizes the total flux sum, the `pfba_factor` should, if set, be larger than one. Setting this value may lead to more realistic predictions of the effective flux bounds.

- **processes** (*int, optional*) – The number of parallel processes to run. If not explicitly passed, will be set from the global configuration singleton.

Returns A data frame with reaction identifiers as the index and two columns: - maximum: indicating the highest possible flux - minimum: indicating the lowest possible flux

Return type pandas.DataFrame

Notes

This implements the fast version as described in[1]. Please note that the flux distribution containing all minimal/maximal fluxes does not have to be a feasible solution for the model. Fluxes are minimized/maximized individually and a single minimal flux might require all others to be suboptimal.

Using the loopless option will lead to a significant increase in computation time (about a factor of 100 for large models). However, the algorithm used here (see[2]) is still more than 1000x faster than the "naive" version using `add_loopless(model)`. Also note that if you have included constraints that force a loop (for instance by setting all fluxes in a loop to be non-zero) this loop will be included in the solution.

References

`cobra.flux_analysis.variability.`**`find_blocked_reactions`**(*model*, *reaction_list=None*, *zero_cutoff=None*, *open_exchanges=False*, *processes=None*)

Find reactions that cannot carry any flux.

The question whether or not a reaction is blocked is highly dependent on the current exchange reaction settings for a COBRA model. Hence an argument is provided to open all exchange reactions.

Notes

Sink and demand reactions are left untouched. Please modify them manually.

Parameters

- **model** (`cobra.Model`) – The model to analyze.
- **reaction_list** (*list, optional*) – List of reactions to consider, the default includes all model reactions.
- **zero_cutoff** (*float, optional*) – Flux value which is considered to effectively be zero (default model.tolerance).
- **open_exchanges** (*bool, optional*) – Whether or not to open all exchange reactions to very high flux ranges.
- **processes** (*int, optional*) – The number of parallel processes to run. Can speed up the computations if the number of reactions is large. If not explicitly passed, it will be set from the global configuration singleton.

Returns List with the identifiers of blocked reactions.

Return type list

[1] Computationally efficient flux variability analysis. Gudmundsson S, Thiele I. BMC Bioinformatics. 2010 Sep 29;11:489. doi: 10.1186/1471-2105-11-489, PMID: 20920235

[2] CycleFreeFlux: efficient removal of thermodynamically infeasible loops from flux distributions. Desouki AA, Jarre F, Gelius-Dietrich G, Lercher MJ. Bioinformatics. 2015 Jul 1;31(13):2159-65. doi: 10.1093/bioinformatics/btv096.

cobra.flux_analysis.variability.**find_essential_genes**(*model*, *threshold=None*, *processes=None*)

Return a set of essential genes.

A gene is considered essential if restricting the flux of all reactions that depend on it to zero causes the objective, e.g., the growth rate, to also be zero, below the threshold, or infeasible.

Parameters

- **model** (cobra.Model) – The model to find the essential genes for.
- **threshold** (*float, optional*) – Minimal objective flux to be considered viable. By default this is 1% of the maximal objective.
- **processes** (*int, optional*) – The number of parallel processes to run. If not passed, will be set to the number of CPUs found.
- **processes** – The number of parallel processes to run. Can speed up the computations if the number of knockouts to perform is large. If not explicitly passed, it will be set from the global configuration singleton.

Returns Set of essential genes

Return type set

cobra.flux_analysis.variability.**find_essential_reactions**(*model*, *threshold=None*, *processes=None*)

Return a set of essential reactions.

A reaction is considered essential if restricting its flux to zero causes the objective, e.g., the growth rate, to also be zero, below the threshold, or infeasible.

Parameters

- **model** (cobra.Model) – The model to find the essential reactions for.
- **threshold** (*float, optional*) – Minimal objective flux to be considered viable. By default this is 1% of the maximal objective.
- **processes** (*int, optional*) – The number of parallel processes to run. Can speed up the computations if the number of knockouts to perform is large. If not explicitly passed, it will be set from the global configuration singleton.

Returns Set of essential reactions

Return type set

Package Contents

Functions

double_gene_deletion(model, gene_list1=None, gene_list2=None, method='fba', solution=None, processes=None, **kwargs)	Knock out each gene pair from the combination of two given lists.
double_reaction_deletion(model, reaction_list1=None, reaction_list2=None, method='fba', solution=None, processes=None, **kwargs)	Knock out each reaction pair from the combinations of two given lists.
single_gene_deletion(model, gene_list=None, method='fba', solution=None, processes=None, **kwargs)	Knock out each gene from a given list.

Continued on next page

Table 34 – continued from previous page

single_reaction_deletion(model, reaction_list=None, method='fba', solution=None, processes=None, **kwargs)	Knock out each reaction from a given list.
fastcc(model, flux_threshold=1.0, zero_cutoff=None)	Check consistency of a metabolic network using FASTCC [1]_.
gapfill(model, universal=None, lower_bound=0.05, penalties=None, demand_reactions=True, exchange_reactions=False, iterations=1)	Perform gapfilling on a model.
geometric_fba(model, epsilon=1e-06, max_tries=200, processes=None)	Perform geometric FBA to obtain a unique, centered flux distribution.
loopless_solution(model, fluxes=None)	Convert an existing solution to a loopless one.
add_loopless(model, zero_cutoff=None)	Modify a model so all feasible flux distributions are loopless.
add_moma(model, solution=None, linear=True)	Add constraints and objective representing for MOMA.
moma(model, solution=None, linear=True)	Compute a single solution based on (linear) MOMA.
pfba(model, fraction_of_optimum=1.0, objective=None, reactions=None)	Perform basic pFBA (parsimonious Enzyme Usage Flux Balance Analysis)
find_blocked_reactions(model, reaction_list=None, zero_cutoff=None, open_exchanges=False, processes=None)	Find reactions that cannot carry any flux.
find_essential_genes(model, threshold=None, processes=None)	Return a set of essential genes.
find_essential_reactions(model, threshold=None, processes=None)	Return a set of essential reactions.
flux_variability_analysis(model, reaction_list=None, loopless=False, fraction_of_optimum=1.0, pfba_factor=None, processes=None)	Determine the minimum and maximum possible flux value for each reaction.
production_envelope(model, reactions, objective=None, carbon_sources=None, points=20, threshold=None)	Calculate the objective value conditioned on all combinations of
add_room(model, solution=None, linear=False, delta=0.03, epsilon=0.001)	Add constraints and objective for ROOM.
room(model, solution=None, linear=False, delta=0.03, epsilon=0.001)	Compute a single solution based on regulatory on/off minimization (ROOM).

cobra.flux_analysis.**double_gene_deletion**(*model*, *gene_list1=None*, *gene_list2=None*, *method='fba'*, *solution=None*, *processes=None*, ***kwargs*)

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

We say 'pair' here but the order order does not matter.

Parameters

- **model** (*cobra.Model*) – The metabolic model to perform deletions in.
- **gene_list1** (*iterable, optional*) – First iterable of ``cobra.Gene``s to be deleted. If not passed, all the genes from the model are used.
- **gene_list2** (*iterable, optional*) – Second iterable of ``cobra.Gene``s to be deleted. If not passed, all the genes from the model are used.
- **method** (*{"fba", "moma", "linear moma", "room", "linear room"}, optional*) – Method used to predict the growth rate.
- **solution** (*cobra.Solution, optional*) – A previous solution to use as a reference for (linear) MOMA or ROOM.

- **processes** (*int, optional*) – The number of parallel processes to run. Can speed up the computations if the number of knockouts to perform is large. If not passed, will be set to the number of CPUs found.
- **kwargs** – Keyword arguments are passed on to underlying simulation functions such as `add_room`.

Returns

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

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

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

status [str] The solution's status.

Return type pandas.DataFrame

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

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

We say 'pair' here but the order order does not matter.

Parameters

- **model** (*cobra.Model*) – The metabolic model to perform deletions in.
- **reaction_list1** (*iterable, optional*) – First iterable of ``cobra.Reaction``s to be deleted. If not passed, all the reactions from the model are used.
- **reaction_list2** (*iterable, optional*) – Second iterable of ``cobra.Reaction``s to be deleted. If not passed, all the reactions from the model are used.
- **method** (*{"fba", "moma", "linear moma", "room", "linear room"}, optional*) – Method used to predict the growth rate.
- **solution** (*cobra.Solution, optional*) – A previous solution to use as a reference for (linear) MOMA or ROOM.
- **processes** (*int, optional*) – The number of parallel processes to run. Can speed up the computations if the number of knockouts to perform is large. If not passed, will be set to the number of CPUs found.
- **kwargs** – Keyword arguments are passed on to underlying simulation functions such as `add_room`.

Returns

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

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

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

status [str] The solution's status.

Return type pandas.DataFrame

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

Knock out each gene from a given list.

Parameters

- **model** (cobra.Model) – The metabolic model to perform deletions in.
- **gene_list** (*iterable*) – ``cobra.Gene``s to be deleted. If not passed, all the genes from the model are used.
- **method** (*{"fba", "moma", "linear moma", "room", "linear room"}, optional*) – Method used to predict the growth rate.
- **solution** (cobra.Solution, *optional*) – A previous solution to use as a reference for (linear) MOMA or ROOM.
- **processes** (*int, optional*) – The number of parallel processes to run. Can speed up the computations if the number of knockouts to perform is large. If not passed, will be set to the number of CPUs found.
- **kwargs** – Keyword arguments are passed on to underlying simulation functions such as add_room.

Returns

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

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

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

status [str] The solution's status.

Return type pandas.DataFrame

cobra.flux_analysis.**single_reaction_deletion**(*model*, *reaction_list=None*, *method='fba'*, *solution=None*, *processes=None*, ***kwargs*)

Knock out each reaction from a given list.

Parameters

- **model** (cobra.Model) – The metabolic model to perform deletions in.
- **reaction_list** (*iterable, optional*) – ``cobra.Reaction``s to be deleted. If not passed, all the reactions from the model are used.
- **method** (*{"fba", "moma", "linear moma", "room", "linear room"}, optional*) – Method used to predict the growth rate.
- **solution** (cobra.Solution, *optional*) – A previous solution to use as a reference for (linear) MOMA or ROOM.
- **processes** (*int, optional*) – The number of parallel processes to run. Can speed up the computations if the number of knockouts to perform is large. If not passed, will be set to the number of CPUs found.
- **kwargs** – Keyword arguments are passed on to underlying simulation functions such as add_room.

Returns

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

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

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

status [str] The solution's status.

Return type pandas.DataFrame

cobra.flux_analysis.**fastcc**(*model*, *flux_threshold=1.0*, *zero_cutoff=None*)

Check consistency of a metabolic network using FASTCC [1]_.

FASTCC (Fast Consistency Check) is an algorithm for rapid and efficient consistency check in metabolic networks. FASTCC is a pure LP implementation and is low on computation resource demand. FASTCC also circumvents the problem associated with reversible reactions for the purpose. Given a global model, it will generate a consistent global model i.e., remove blocked reactions. For more details on FASTCC, please check [1]_.

Parameters

- **model** (cobra.Model) – The constraint-based model to operate on.
- **flux_threshold** (*float, optional (default 1.0)*) – The flux threshold to consider.
- **zero_cutoff** (*float, optional*) – The cutoff to consider for zero flux (default model.tolerance).

Returns The consistent constraint-based model.

Return type *cobra.Model*

Notes

The LP used for FASTCC is like so: maximize: sum_{i in J} z_i s.t. : z_i in [0, varepsilon] forall i in J, z_i in mathbb{R}_+

v_i ge z_i forall i in J Sv = 0 v in B

References

cobra.flux_analysis.**gapfill**(*model*, *universal=None*, *lower_bound=0.05*, *penalties=None*, *demand_reactions=True*, *exchange_reactions=False*, *iterations=1*)

Perform gapfilling on a model.

See documentation for the class GapFiller.

Parameters

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

- **demand_reactions** (*bool*) – Consider adding demand reactions for all metabolites.

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

Return type iterable

Examples

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

cobra.flux_analysis.**geometric_fba**(*model*, *epsilon=1e-06*, *max_tries=200*, *processes=None*)

Perform geometric FBA to obtain a unique, centered flux distribution.

Geometric FBA [1]_ formulates the problem as a polyhedron and then solves it by bounding the convex hull of the polyhedron. The bounding forms a box around the convex hull which reduces with every iteration and extracts a unique solution in this way.

Parameters

- **model** (*cobra.Model*) – The model to perform geometric FBA on.
- **epsilon** (*float, optional*) – The convergence tolerance of the model (default 1E-06).
- **max_tries** (*int, optional*) – Maximum number of iterations (default 200).
- **processes** (*int, optional*) – The number of parallel processes to run. If not explicitly passed, will be set from the global configuration singleton.

Returns The solution object containing all the constraints required for geometric FBA.

Return type *cobra.Solution*

References

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

Convert an existing solution to a loopless one.

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

Parameters

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

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

Return type *cobra.Solution*

Notes

The returned flux solution has the following properties:

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

References

cobra.flux_analysis.**add_loopless**(*model*, *zero_cutoff=None*)

Modify a model so all feasible flux distributions are loopless.

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

Parameters

- **model** (cobra.Model) – The model to which to add the constraints.
- **zero_cutoff** (*positive float, optional*) – Cutoff used for null space. Coefficients with an absolute value smaller than *zero_cutoff* are considered to be zero (default model.tolerance).

Returns

Return type Nothing

References

cobra.flux_analysis.**add_moma**(*model*, *solution=None*, *linear=True*)

Add constraints and objective representing for MOMA.

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

Parameters

- **model** (cobra.Model) – The model to add MOMA constraints and objective to.
- **solution** (cobra.Solution, *optional*) – A previous solution to use as a reference. If no solution is given, one will be computed using pFBA.
- **linear** (*bool, optional*) – Whether to use the linear MOMA formulation or not (default True).

Notes

In the original MOMA [1]_ specification one looks for the flux distribution of the deletion (v^d) closest to the fluxes without the deletion (v). In math this means:

minimize sum_i (v^d_i - v_i)^2 s.t. Sv^d = 0

lb_i <= v^d_i <= ub_i

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

minimize sum_i (v^t_i)^2 s.t. Sv^d = 0

v^t = v^d_i - v_i lb_i <= v^d_i <= ub_i

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

In the case of linear MOMA [2]_, we instead minimize sum_i abs(v^t_i). The linear MOMA is typically significantly faster. Also quadratic MOMA tends to give flux distributions in which all fluxes deviate from the reference fluxes a little bit whereas linear MOMA tends to give flux distributions where the majority of fluxes are the same reference with few fluxes deviating a lot (typical effect of L2 norm vs L1 norm).

The former objective function is saved in the optlang solver interface as `"moma_old_objective"` and this can be used to immediately extract the value of the former objective after MOMA optimization.

See also:

pfba() parsimonious FBA

References

cobra.flux_analysis.**moma**(*model*, *solution=None*, *linear=True*)

Compute a single solution based on (linear) MOMA.

Compute a new flux distribution that is at a minimal distance to a previous reference solution. Minimization of metabolic adjustment (MOMA) is generally used to assess the impact of knock-outs. Thus the typical usage is to provide a wildtype flux distribution as reference and a model in knock-out state.

Parameters

- **model** (cobra.Model) – The model state to compute a MOMA-based solution for.
- **solution** (cobra.Solution, *optional*) – A (wildtype) reference solution.
- **linear** (*bool*, *optional*) – Whether to use the linear MOMA formulation or not (default True).

Returns A flux distribution that is at a minimal distance compared to the reference solution.

Return type *cobra.Solution*

See also:

add_moma() add MOMA constraints and objective

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

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

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

Parameters

- **model** (cobra.Model) – The model

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

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

Return type *cobra.Solution*

References

cobra.flux_analysis.**find_blocked_reactions**(*model*, *reaction_list=None*, *zero_cutoff=None*, *open_exchanges=False*, *processes=None*)

Find reactions that cannot carry any flux.

The question whether or not a reaction is blocked is highly dependent on the current exchange reaction settings for a COBRA model. Hence an argument is provided to open all exchange reactions.

Notes

Sink and demand reactions are left untouched. Please modify them manually.

Parameters

- **model** (cobra.Model) – The model to analyze.
- **reaction_list** (*list, optional*) – List of reactions to consider, the default includes all model reactions.
- **zero_cutoff** (*float, optional*) – Flux value which is considered to effectively be zero (default model.tolerance).
- **open_exchanges** (*bool, optional*) – Whether or not to open all exchange reactions to very high flux ranges.
- **processes** (*int, optional*) – The number of parallel processes to run. Can speed up the computations if the number of reactions is large. If not explicitly passed, it will be set from the global configuration singleton.

Returns List with the identifiers of blocked reactions.

Return type list

cobra.flux_analysis.**find_essential_genes**(*model*, *threshold=None*, *processes=None*)

Return a set of essential genes.

A gene is considered essential if restricting the flux of all reactions that depend on it to zero causes the objective, e.g., the growth rate, to also be zero, below the threshold, or infeasible.

Parameters

- **model** (cobra.Model) – The model to find the essential genes for.
- **threshold** (*float, optional*) – Minimal objective flux to be considered viable. By default this is 1% of the maximal objective.

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

Returns Set of essential genes

Return type set

cobra.flux_analysis.**find_essential_reactions**(*model*, *threshold=None*, *processes=None*)

Return a set of essential reactions.

A reaction is considered essential if restricting its flux to zero causes the objective, e.g., the growth rate, to also be zero, below the threshold, or infeasible.

Parameters

- **model** (cobra.Model) – The model to find the essential reactions for.
- **threshold** (*float, optional*) – Minimal objective flux to be considered viable. By default this is 1% of the maximal objective.
- **processes** (*int, optional*) – The number of parallel processes to run. Can speed up the computations if the number of knockouts to perform is large. If not explicitly passed, it will be set from the global configuration singleton.

Returns Set of essential reactions

Return type set

cobra.flux_analysis.**flux_variability_analysis**(*model*, *reaction_list=None*, *loopless=False*, *fraction_of_optimum=1.0*, *pfba_factor=None*, *processes=None*)

Determine the minimum and maximum possible flux value for each reaction.

Parameters

- **model** (cobra.Model) – The model for which to run the analysis. It will *not* be modified.
- **reaction_list** (*list of cobra.Reaction or str, optional*) – The reactions for which to obtain min/max fluxes. If None will use all reactions in the model (default).
- **loopless** (*boolean, optional*) – Whether to return only loopless solutions. This is significantly slower. Please also refer to the notes.
- **fraction_of_optimum** (*float, optional*) – Must be <= 1.0. Requires that the objective value is at least the fraction times maximum objective value. A value of 0.85 for instance means that the objective has to be at least at 85% percent of its maximum.
- **pfba_factor** (*float, optional*) – Add an additional constraint to the model that requires the total sum of absolute fluxes must not be larger than this value times the smallest possible sum of absolute fluxes, i.e., by setting the value to 1.1 the total sum of absolute fluxes must not be more than 10% larger than the pFBA solution. Since the pFBA solution is the one that optimally minimizes the total flux sum, the pfba_factor should, if set, be larger than one. Setting this value may lead to more realistic predictions of the effective flux bounds.
- **processes** (*int, optional*) – The number of parallel processes to run. If not explicitly passed, will be set from the global configuration singleton.

Returns A data frame with reaction identifiers as the index and two columns: - maximum: indicating the highest possible flux - minimum: indicating the lowest possible flux

Return type pandas.DataFrame

Notes

This implements the fast version as described in [1]_. Please note that the flux distribution containing all minimal/maximal fluxes does not have to be a feasible solution for the model. Fluxes are minimized/maximized individually and a single minimal flux might require all others to be suboptimal.

Using the loopless option will lead to a significant increase in computation time (about a factor of 100 for large models). However, the algorithm used here (see [2]_) is still more than 1000x faster than the "naive" version using `add_loopless(model)`. Also note that if you have included constraints that force a loop (for instance by setting all fluxes in a loop to be non-zero) this loop will be included in the solution.

References

`cobra.flux_analysis.`**`production_envelope`**(*model*, *reactions*, *objective=None*, *carbon_sources=None*, *points=20*, *threshold=None*)

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

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

Parameters

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

Returns

A data frame with one row per evaluated point and

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

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

- flux: the objective flux
- carbon_yield: if carbon source is defined and the product is a single metabolite (mol carbon product per mol carbon feeding source)

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

Return type pandas.DataFrame

Examples

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

cobra.flux_analysis.**add_room**(*model*, *solution=None*, *linear=False*, *delta=0.03*, *epsilon=0.001*)

Add constraints and objective for ROOM.

This function adds variables and constraints for applying regulatory on/off minimization (ROOM) to the model.

Parameters

- **model** (cobra.Model) – The model to add ROOM constraints and objective to.
- **solution** (cobra.Solution, *optional*) – A previous solution to use as a reference. If no solution is given, one will be computed using pFBA.
- **linear** (*bool*, *optional*) – Whether to use the linear ROOM formulation or not (default False).
- **delta** (*float*, *optional*) – The relative tolerance range which is additive in nature (default 0.03).
- **epsilon** (*float*, *optional*) – The absolute range of tolerance which is multiplicative (default 0.001).

Notes

The formulation used here is the same as stated in the original paper [1]_. The mathematical expression is given below:

minimize sum_{i=1}^m y^i s.t. Sv = 0

v_min <= v <= v_max v_j = 0 j A for 1 <= i <= m v_i - y_i(v_{max,i} - w_i^u) <= w_i^u (1) v_i - y_i(v_{min,i} - w_i^l) <= w_i^l (2) y_i {0,1} (3) w_i^u = w_i + deltalw_il + epsilon w_i^l = w_i - deltalw_il - epsilon

So, for the linear version of the ROOM , constraint (3) is relaxed to 0 <= y_i <= 1.

See also:

pfba() parsimonious FBA

References

cobra.flux_analysis.**room**(*model*, *solution=None*, *linear=False*, *delta=0.03*, *epsilon=0.001*)
Compute a single solution based on regulatory on/off minimization (ROOM).

Compute a new flux distribution that minimizes the number of active reactions needed to accommodate a previous reference solution. Regulatory on/off minimization (ROOM) is generally used to assess the impact of knock-outs. Thus the typical usage is to provide a wildtype flux distribution as reference and a model in knock-out state.

Parameters

- **model** (*cobra.Model*) – The model state to compute a ROOM-based solution for.
- **solution** (*cobra.Solution, optional*) – A (wildtype) reference solution.
- **linear** (*bool, optional*) – Whether to use the linear ROOM formulation or not (default False).
- **delta** (*float, optional*) – The relative tolerance range (additive) (default 0.03).
- **epsilon** (*float, optional*) – The absolute tolerance range (multiplicative) (default 0.001).

Returns A flux distribution with minimal active reaction changes compared to the reference.

Return type *cobra.Solution*

See also:

add_room() add ROOM constraints and objective

cobra.io

Submodules

cobra.io.dict

Module Contents

Functions

_fix_type(value)	convert possible types to str, float, and bool
_update_optional(cobra_object, new_dict, optional_attribute_dict, ordered_keys)	update new_dict with optional attributes from cobra_object
metabolite_to_dict(metabolite)	
metabolite_from_dict(metabolite)	
gene_to_dict(gene)	
gene_from_dict(gene)	
reaction_to_dict(reaction)	
reaction_from_dict(reaction, model)	
model_to_dict(model, sort=False)	Convert model to a dict.
model_from_dict(obj)	Build a model from a dict.

cobra.io.dict.**_REQUIRED_REACTION_ATTRIBUTES = ['id', 'name', 'metabolites', 'lower_bound**

cobra.io.dict.**_ORDERED_OPTIONAL_REACTION_KEYS = ['objective_coefficient', 'subsystem', '**

cobra.io.dict.**_OPTIONAL_REACTION_ATTRIBUTES**

cobra.io.dict._**REQUIRED_METABOLITE_ATTRIBUTES = ['id', 'name', 'compartment']**

cobra.io.dict._**ORDERED_OPTIONAL_METABOLITE_KEYS = ['charge', 'formula', '_bound', 'notes**

cobra.io.dict._**OPTIONAL_METABOLITE_ATTRIBUTES**

cobra.io.dict._**REQUIRED_GENE_ATTRIBUTES = ['id', 'name']**

cobra.io.dict._**ORDERED_OPTIONAL_GENE_KEYS = ['notes', 'annotation']**

cobra.io.dict._**OPTIONAL_GENE_ATTRIBUTES**

cobra.io.dict._**ORDERED_OPTIONAL_MODEL_KEYS = ['name', 'compartments', 'notes', 'annotati**

cobra.io.dict._**OPTIONAL_MODEL_ATTRIBUTES**

cobra.io.dict._**fix_type**(*value*)
convert possible types to str, float, and bool

cobra.io.dict._**update_optional**(*cobra_object*, *new_dict*, *optional_attribute_dict*, *ordered_keys*)
update new_dict with optional attributes from cobra_object

cobra.io.dict.**metabolite_to_dict**(*metabolite*)

cobra.io.dict.**metabolite_from_dict**(*metabolite*)

cobra.io.dict.**gene_to_dict**(*gene*)

cobra.io.dict.**gene_from_dict**(*gene*)

cobra.io.dict.**reaction_to_dict**(*reaction*)

cobra.io.dict.**reaction_from_dict**(*reaction*, *model*)

cobra.io.dict.**model_to_dict**(*model*, *sort=False*)
Convert model to a dict.

Parameters

- **model** (cobra.Model) – The model to reformulate as a dict.
- **sort** (*bool, optional*) – Whether to sort the metabolites, reactions, and genes or maintain the order defined in the model.

Returns A dictionary with elements, 'genes', 'compartments', 'id', 'metabolites', 'notes' and 'reactions'; where 'metabolites', 'genes' and 'metabolites' are in turn lists with dictionaries holding all attributes to form the corresponding object.

Return type OrderedDict

See also:

cobra.io.model_from_dict()

cobra.io.dict.**model_from_dict**(*obj*)
Build a model from a dict.

Models stored in json are first formulated as a dict that can be read to cobra model using this function.

Parameters **obj** (*dict*) – A dictionary with elements, 'genes', 'compartments', 'id', 'metabolites', 'notes' and 'reactions'; where 'metabolites', 'genes' and 'metabolites' are in turn lists with dictionaries holding all attributes to form the corresponding object.

Returns The generated model.

Return type cora.core.Model

See also:

cobra.io.model_to_dict()

cobra.io.json

Module Contents

Functions

to_json(model, sort=False, **kwargs)	Return the model as a JSON document.
from_json(document)	Load a cobra model from a JSON document.
save_json_model(model, filename, sort=False, pretty=False, **kwargs)	Write the cobra model to a file in JSON format.
load_json_model(filename)	Load a cobra model from a file in JSON format.

cobra.io.json.**JSON_SPEC = 1**

cobra.io.json.**to_json**(*model*, *sort=False*, ***kwargs*)
Return the model as a JSON document.

`kwargs` are passed on to `json.dumps`.

Parameters

- **model** (*cobra.Model*) – The cobra model to represent.
- **sort** (*bool, optional*) – Whether to sort the metabolites, reactions, and genes or maintain the order defined in the model.

Returns String representation of the cobra model as a JSON document.

Return type str

See also:

save_json_model() Write directly to a file.

json.dumps() Base function.

cobra.io.json.**from_json**(*document*)
Load a cobra model from a JSON document.

Parameters **document** (*str*) – The JSON document representation of a cobra model.

Returns The cobra model as represented in the JSON document.

Return type *cobra.Model*

See also:

load_json_model() Load directly from a file.

cobra.io.json.**save_json_model**(*model*, *filename*, *sort=False*, *pretty=False*, ***kwargs*)
Write the cobra model to a file in JSON format.

`kwargs` are passed on to `json.dump`.

Parameters

- **model** (*cobra.Model*) – The cobra model to represent.
- **filename** (*str or file-like*) – File path or descriptor that the JSON representation should be written to.
- **sort** (*bool, optional*) – Whether to sort the metabolites, reactions, and genes or maintain the order defined in the model.

- **pretty** (*bool, optional*) – Whether to format the JSON more compactly (default) or in a more verbose but easier to read fashion. Can be partially overwritten by the `kwargs`.

See also:

to_json() Return a string representation.

json.dump() Base function.

cobra.io.json.**load_json_model**(*filename*)
Load a cobra model from a file in JSON format.

Parameters **filename** (*str or file-like*) – File path or descriptor that contains the JSON document describing the cobra model.

Returns The cobra model as represented in the JSON document.

Return type *cobra.Model*

See also:

from_json() Load from a string.

cobra.io.json.**json_schema**

cobra.io.mat

Module Contents

Functions

_get_id_compartment(id)	extract the compartment from the id string
_cell(x)	translate an array x into a MATLAB cell array
load_matlab_model(infile_path, variable_name=None, inf=inf)	Load a cobra model stored as a .mat file
save_matlab_model(model, file_name, varname=None)	Save the cobra model as a .mat file.
create_mat_metabolite_id(model)	
create_mat_dict(model)	create a dict mapping model attributes to arrays
from_mat_struct(mat_struct, model_id=None, inf=inf)	create a model from the COBRA toolbox struct
_check(result)	ensure success of a pymatbridge operation
model_to_pymatbridge(model, variable_name='model', matlab=None)	send the model to a MATLAB workspace through pymatbridge

cobra.io.mat.**scipy_sparse**

cobra.io.mat.**_bracket_re**

cobra.io.mat.**_underscore_re**

cobra.io.mat.**_get_id_compartment**(*id*)
extract the compartment from the id string

cobra.io.mat.**_cell**(*x*)
translate an array x into a MATLAB cell array

cobra.io.mat.**load_matlab_model**(*infile_path*, *variable_name=None*, *inf=inf*)
Load a cobra model stored as a .mat file

Parameters

- **infile_path** (*str*) – path to the file to to read
- **variable_name** (*str, optional*) – The variable name of the model in the .mat file. If this is not specified, then the first MATLAB variable which looks like a COBRA model will be used
- **inf** (*value*) – The value to use for infinite bounds. Some solvers do not handle infinite values so for using those, set this to a high numeric value.

Returns The resulting cobra model

Return type cobra.core.Model.Model

cobra.io.mat.**save_matlab_model**(*model*, *file_name*, *varname=None*)
Save the cobra model as a .mat file.

This .mat file can be used directly in the MATLAB version of COBRA.

Parameters

- **model** (*cobra.core.Model.Model object*) – The model to save
- **file_name** (*str or file-like object*) – The file to save to
- **varname** (*string*) – The name of the variable within the workspace

cobra.io.mat.**create_mat_metabolite_id**(*model*)

cobra.io.mat.**create_mat_dict**(*model*)
create a dict mapping model attributes to arrays

cobra.io.mat.**from_mat_struct**(*mat_struct*, *model_id=None*, *inf=inf*)
create a model from the COBRA toolbox struct

The struct will be a dict read in by scipy.io.loadmat

cobra.io.mat.**_check**(*result*)
ensure success of a pymatbridge operation

cobra.io.mat.**model_to_pymatbridge**(*model*, *variable_name='model'*, *matlab=None*)
send the model to a MATLAB workspace through pymatbridge

This model can then be manipulated through the COBRA toolbox

Parameters

- **variable_name** (*str*) – The variable name to which the model will be assigned in the MATLAB workspace
- **matlab** (*None or pymatbridge.Matlab instance*) – The MATLAB workspace to which the variable will be sent. If this is None, then this will be sent to the same environment used in IPython magics.

cobra.io.sbml

SBML import and export using python-libsbml.

- The SBML importer supports all versions of SBML and the fbc package.
- The SBML exporter writes SBML L3 models.
- Annotation information is stored on the cobrapy objects
- Information from the group package is read

Parsing of fbc models was implemented as efficient as possible, whereas (discouraged) fallback solutions are not optimized for efficiency.

Notes are only supported in a minimal way relevant for constraint-based models. I.e., structured information from notes in the form

key: value

is read into the Object.notes dictionary when reading SBML files. On writing the Object.notes dictionary is serialized to the SBML notes information.

Annotations are read in the Object.annotation fields.

Some SBML related issues are still open, please refer to the respective issue: - update annotation format and support qualifiers (depends on decision

for new annotation format; https://github.com/opencobra/cobrapy/issues/684)

- **write compartment annotations and notes (depends on updated first-class** compartments; see https://github.com/opencobra/cobrapy/issues/760)
- **support compression on file handles (depends on solution for** https://github.com/opencobra/cobrapy/issues/812)

Module Contents

Functions

_escape_non_alphanum(nonASCII)	converts a non alphanumeric character to a string representation of
_number_to_chr(numberStr)	converts an ascii number to a character
_clip(sid, prefix)	Clips a prefix from the beginning of a string if it exists.
_f_gene(sid, prefix='G_')	Clips gene prefix from id.
_f_gene_rev(sid, prefix='G_')	Adds gene prefix to id.
_f_specie(sid, prefix='M_')	Clips specie/metabolite prefix from id.
_f_specie_rev(sid, prefix='M_')	Adds specie/metabolite prefix to id.
_f_reaction(sid, prefix='R_')	Clips reaction prefix from id.
_f_reaction_rev(sid, prefix='R_')	Adds reaction prefix to id.
_f_group(sid, prefix='G_')	Clips group prefix from id.
_f_group_rev(sid, prefix='G_')	Adds group prefix to id.
read_sbml_model(filename, number=float, f_replace=F_REPLACE, **kwargs)	Reads SBML model from given filename.
_get_doc_from_filename(filename)	Get SBMLDocument from given filename.
_sbml_to_model(doc, number=float, f_replace=F_REPLACE, set_missing_bounds=False, **kwargs)	Creates cobra model from SBMLDocument.
write_sbml_model(cobra_model, filename, f_replace=F_REPLACE, **kwargs)	Writes cobra model to filename.
_model_to_sbml(cobra_model, f_replace=None, units=True)	Convert Cobra model to SBMLDocument.
_create_bound(model, reaction, bound_type, f_replace, units=None, flux_udef=None)	Creates bound in model for given reaction.
_create_parameter(model, pid, value, sbo=None, constant=True, units=None, flux_udef=None)	Create parameter in SBML model.
_check_required(sbase, value, attribute)	Get required attribute from SBase.
_check(value, message)	Checks the libsbml return value and logs error messages.
_parse_notes_dict(sbase)	Creates dictionary of COBRA notes.
_sbase_notes_dict(sbase, notes)	Set SBase notes based on dictionary.
_parse_annotations(sbase)	Parses cobra annotations from a given SBase object.

Continued on next page

Table 38 – continued from previous page

_parse_annotation_info(uri)	Parses provider and term from given identifiers annotation uri.
_sbase_annotations(sbase, annotation)	Set SBase annotations based on cobra annotations.
validate_sbml_model(filename, check_model=True, internal_consistency=True, check_units_consistency=False, check_modeling_practice=False, **kwargs)	Validate SBML model and returns the model along with a list of errors.
_error_string(error, k=None)	String representation of SBMLError.

exception cobra.io.sbml.**CobraSBMLError**
Bases: Exception

SBML error class.

cobra.io.sbml.**LOGGER**

cobra.io.sbml.**config**

cobra.io.sbml.**LOWER_BOUND_ID = cobra_default_lb**

cobra.io.sbml.**UPPER_BOUND_ID = cobra_default_ub**

cobra.io.sbml.**ZERO_BOUND_ID = cobra_0_bound**

cobra.io.sbml.**BOUND_MINUS_INF = minus_inf**

cobra.io.sbml.**BOUND_PLUS_INF = plus_inf**

cobra.io.sbml.**SBO_FBA_FRAMEWORK = SBO:0000624**

cobra.io.sbml.**SBO_DEFAULT_FLUX_BOUND = SBO:0000626**

cobra.io.sbml.**SBO_FLUX_BOUND = SBO:0000625**

cobra.io.sbml.**SBO_EXCHANGE_REACTION = SBO:0000627**

cobra.io.sbml.**LONG_SHORT_DIRECTION**

cobra.io.sbml.**SHORT_LONG_DIRECTION**

cobra.io.sbml.**Unit**

cobra.io.sbml.**UNITS_FLUX = ['mmol_per_gDW_per_hr', None]**

cobra.io.sbml.**SBML_DOT = __SBML_DOT__**

cobra.io.sbml.**pattern_notes**

cobra.io.sbml.**pattern_to_sbml**

cobra.io.sbml.**pattern_from_sbml**

cobra.io.sbml.**_escape_non_alphanum**(*nonASCII*)
converts a non alphanumeric character to a string representation of its ascii number

cobra.io.sbml.**_number_to_chr**(*numberStr*)
converts an ascii number to a character

cobra.io.sbml.**_clip**(*sid*, *prefix*)
Clips a prefix from the beginning of a string if it exists.

cobra.io.sbml.**_f_gene**(*sid*, *prefix='G_'*)
Clips gene prefix from id.

cobra.io.sbml.**_f_gene_rev**(*sid*, *prefix='G_'*)
Adds gene prefix to id.

cobra.io.sbml.**_f_specie**(*sid*, *prefix='M_'*)
Clips specie/metabolite prefix from id.

cobra.io.sbml.**_f_specie_rev**(*sid*, *prefix='M_'*)
Adds specie/metabolite prefix to id.

cobra.io.sbml.**_f_reaction**(*sid*, *prefix='R_'*)
Clips reaction prefix from id.

cobra.io.sbml.**_f_reaction_rev**(*sid*, *prefix='R_'*)
Adds reaction prefix to id.

cobra.io.sbml.**_f_group**(*sid*, *prefix='G_'*)
Clips group prefix from id.

cobra.io.sbml.**_f_group_rev**(*sid*, *prefix='G_'*)
Adds group prefix to id.

cobra.io.sbml.**F_GENE = F_GENE**

cobra.io.sbml.**F_GENE_REV = F_GENE_REV**

cobra.io.sbml.**F_SPECIE = F_SPECIE**

cobra.io.sbml.**F_SPECIE_REV = F_SPECIE_REV**

cobra.io.sbml.**F_REACTION = F_REACTION**

cobra.io.sbml.**F_REACTION_REV = F_REACTION_REV**

cobra.io.sbml.**F_GROUP = F_GROUP**

cobra.io.sbml.**F_GROUP_REV = F_GROUP_REV**

cobra.io.sbml.**F_REPLACE**

cobra.io.sbml.**read_sbml_model**(*filename*, *number=float*, *f_replace=F_REPLACE*, ***kwargs*)
Reads SBML model from given filename.

If the given filename ends with the suffix ''.gz'' (for example, ''myfile.xml.gz''),' the file is assumed to be compressed in gzip format and will be automatically decompressed upon reading. Similarly, if the given filename ends with ''.zip'' or ''.bz2'',' the file is assumed to be compressed in zip or bzip2 format (respectively). Files whose names lack these suffixes will be read uncompressed. Note that if the file is in zip format but the archive contains more than one file, only the first file in the archive will be read and the rest ignored.

To read a gzip/zip file, libSBML needs to be configured and linked with the zlib library at compile time. It also needs to be linked with the bzip2 library to read files in bzip2 format. (Both of these are the default configurations for libSBML.)

This function supports SBML with FBC-v1 and FBC-v2. FBC-v1 models are converted to FBC-v2 models before reading.

The parser tries to fall back to information in notes dictionaries if information is not available in the FBC packages, e.g., CHARGE, FORMULA on species, or GENE_ASSOCIATION, SUBSYSTEM on reactions.

Parameters

- **filename** (*path to SBML file, or SBML string, or SBML file handle*) – SBML which is read into cobra model
- **number** (*data type of stoichiometry: {float, int}*) – In which data type should the stoichiometry be parsed.
- **f_replace** (*dict of replacement functions for id replacement*) – Dictionary of replacement functions for gene, specie, and reaction. By default the following id changes are performed on import: clip **G_** from genes, clip **M_** from species, clip **R_** from reactions If no replacements should be performed, set f_replace={}, None

Returns

Return type *cobra.core.Model*

Notes

Provided file handles cannot be opened in binary mode, i.e., use

with open(path, "r" as f): read_sbml_model(f)

File handles to compressed files are not supported yet.

cobra.io.sbml.**_get_doc_from_filename**(*filename*)
Get SBMLDocument from given filename.

Parameters **filename** (*path to SBML, or SBML string, or filehandle*) –

Returns

Return type libsbml.SBMLDocument

cobra.io.sbml.**_sbml_to_model**(*doc*, *number=float*, *f_replace=F_REPLACE*, *set_missing_bounds=False*, ***kwargs*)
Creates cobra model from SBMLDocument.

Parameters

- **doc** (*libsbml.SBMLDocument*) –
- **number** (*data type of stoichiometry: {float, int}*) – In which data type should the stoichiometry be parsed.
- **f_replace** (*dict of replacement functions for id replacement*) –
- **set_missing_bounds** (*flag to set missing bounds*) –

Returns

Return type *cobra.core.Model*

cobra.io.sbml.**write_sbml_model**(*cobra_model*, *filename*, *f_replace=F_REPLACE*, ***kwargs*)
Writes cobra model to filename.

The created model is SBML level 3 version 1 (L1V3) with fbc package v2 (fbc-v2).

If the given filename ends with the suffix ".gz" (for example, "myfile.xml.gz"), libSBML assumes the caller wants the file to be written compressed in gzip format. Similarly, if the given filename ends with ".zip" or ".bz2", libSBML assumes the caller wants the file to be compressed in zip or bzip2 format (respectively). Files whose names lack these suffixes will be written uncompressed. Special considerations for the zip format: If the given filename ends with ".zip", the file placed in the zip archive will have the suffix ".xml" or ".sbml". For example, the file in the zip archive will be named "test.xml" if the given filename is "test.xml.zip" or "test.zip". Similarly, the filename in the archive will be "test.sbml" if the given filename is "test.sbml.zip".

Parameters

- **cobra_model** (cobra.core.Model) – Model instance which is written to SBML
- **filename** (*string*) – path to which the model is written
- **f_replace** (*dict of replacement functions for id replacement*) –

cobra.io.sbml.**_model_to_sbml**(*cobra_model*, *f_replace=None*, *units=True*)
Convert Cobra model to SBMLDocument.

Parameters

- **cobra_model** (cobra.core.Model) – Cobra model instance

- **f_replace** (*dict of replacement functions*) – Replacement to apply on identifiers.
- **units** (*boolean*) – Should the FLUX_UNITS be written in the SBMLDocument.

Returns

Return type libsbml.SBMLDocument

cobra.io.sbml.**_create_bound**(*model*, *reaction*, *bound_type*, *f_replace*, *units=None*, *flux_udef=None*)

Creates bound in model for given reaction.

Adds the parameters for the bounds to the SBML model.

Parameters

- **model** (*libsbml.Model*) – SBML model instance
- **reaction** (cobra.core.Reaction) – Cobra reaction instance from which the bounds are read.
- **bound_type** (*{LOWER_BOUND, UPPER_BOUND}*) – Type of bound
- **f_replace** (*dict of id replacement functions*) –
- **units** (*flux units*) –

Returns

Return type Id of bound parameter.

cobra.io.sbml.**_create_parameter**(*model*, *pid*, *value*, *sbo=None*, *constant=True*, *units=None*, *flux_udef=None*)

Create parameter in SBML model.

cobra.io.sbml.**_check_required**(*sbase*, *value*, *attribute*)

Get required attribute from SBase.

Parameters

- **sbase** (*libsbml.SBase*) –
- **value** (*existing value*) –
- **attribute** (*name of attribute*) –

Returns

Return type attribute value (or value if already set)

cobra.io.sbml.**_check**(*value*, *message*)

Checks the libsbml return value and logs error messages.

If 'value' is None, logs an error message constructed using 'message' and then exits with status code 1. If 'value' is an integer, it assumes it is a libSBML return status code. If the code value is LIBSBML_OPERATION_SUCCESS, returns without further action; if it is not, prints an error message constructed using 'message' along with text from libSBML explaining the meaning of the code, and exits with status code 1.

cobra.io.sbml.**_parse_notes_dict**(*sbase*)

Creates dictionary of COBRA notes.

Parameters **sbase** (*libsbml.SBase*) –

Returns

Return type dict of notes

cobra.io.sbml.**_sbase_notes_dict**(*sbase*, *notes*)

Set SBase notes based on dictionary.

Parameters

- **sbase** (*libsbml.SBase*) – SBML object to set notes on
- **notes** (*notes object*) – notes information from cobra object

cobra.io.sbml.**URL_IDENTIFIERS_PATTERN**

cobra.io.sbml.**URL_IDENTIFIERS_PREFIX = https://identifiers.org**

cobra.io.sbml.**QUALIFIER_TYPES**

cobra.io.sbml.**_parse_annotations**(*sbase*)
Parses cobra annotations from a given SBase object.

Annotations are dictionaries with the providers as keys.

Parameters **sbase** (*libsbml.SBase*) – SBase from which the SBML annotations are read

Returns

- *dict (annotation dictionary)*
- **FIXME** (*annotation format must be updated (this is a big collection of*) – fixes) - see: https://github.com/opencobra/cobrapy/issues/684)

cobra.io.sbml.**_parse_annotation_info**(*uri*)
Parses provider and term from given identifiers annotation uri.

Parameters **uri** (*str*) – uri (identifiers.org url)

Returns

Return type (provider, identifier) if resolvable, None otherwise

cobra.io.sbml.**_sbase_annotations**(*sbase*, *annotation*)
Set SBase annotations based on cobra annotations.

Parameters

- **sbase** (*libsbml.SBase*) – SBML object to annotate
- **annotation** (*cobra annotation structure*) – cobra object with annotation information
- **FIXME** (*annotation format must be updated*) – (https://github.com/opencobra/cobrapy/issues/684)

cobra.io.sbml.**validate_sbml_model**(*filename*, *check_model=True*, *internal_consistency=True*, *check_units_consistency=False*, *check_modeling_practice=False*, ***kwargs*)
Validate SBML model and returns the model along with a list of errors.

Parameters

- **filename** (*str*) – The filename (or SBML string) of the SBML model to be validated.
- **internal_consistency** (*boolean {True, False}*) – Check internal consistency.
- **check_units_consistency** (*boolean {True, False}*) – Check consistency of units.
- **check_modeling_practice** (*boolean {True, False}*) – Check modeling practise.
- **check_model** (*boolean {True, False}*) – Whether to also check some basic model properties such as reaction boundaries and compartment formulas.

Returns

- *(model, errors)*

- **model** (Model object) – The cobra model if the file could be read successfully or None otherwise.
- **errors** (*dict*) – Warnings and errors grouped by their respective types.

Raises *CobraSBMLError* –

cobra.io.sbml.**_error_string**(*error*, *k=None*)
String representation of SBMLError.

Parameters

- **error** (*libsbml.SBMLError*) –
- **k** (*index of error*) –

Returns

Return type string representation of error

cobra.io.yaml

Module Contents

Classes

MyYAML

Functions

to_yaml(model, sort=False, **kwargs)	Return the model as a YAML document.
from_yaml(document)	Load a cobra model from a YAML document.
save_yaml_model(model, filename, sort=False, **kwargs)	Write the cobra model to a file in YAML format.
load_yaml_model(filename)	Load a cobra model from a file in YAML format.

cobra.io.yaml.**YAML_SPEC = 1.2**

class cobra.io.yaml.**MyYAML**(*: Any*, **, typ: Optional[Text] = None*, *pure: Any = False*, *output: Any = None*, *plug_ins: Any = None*)
Bases: ruamel.yaml.main.YAML

dump(*self*, *data*, *stream=None*, ***kwargs*)

cobra.io.yaml.**yaml**

cobra.io.yaml.**to_yaml**(*model*, *sort=False*, ***kwargs*)
Return the model as a YAML document.

kwargs are passed on to yaml.dump.

Parameters

- **model** (cobra.Model) – The cobra model to represent.
- **sort** (*bool, optional*) – Whether to sort the metabolites, reactions, and genes or maintain the order defined in the model.

Returns String representation of the cobra model as a YAML document.

Return type str

See also:

save_yaml_model() Write directly to a file.

ruamel.yaml.dump() Base function.

cobra.io.yaml.**from_yaml**(*document*)
Load a cobra model from a YAML document.

Parameters **document** (*str*) – The YAML document representation of a cobra model.

Returns The cobra model as represented in the YAML document.

Return type *cobra.Model*

See also:

load_yaml_model() Load directly from a file.

cobra.io.yaml.**save_yaml_model**(*model*, *filename*, *sort=False*, ***kwargs*)
Write the cobra model to a file in YAML format.

`kwargs` are passed on to `yaml.dump`.

Parameters

- **model** (*cobra.Model*) – The cobra model to represent.
- **filename** (*str or file-like*) – File path or descriptor that the YAML representation should be written to.
- **sort** (*bool, optional*) – Whether to sort the metabolites, reactions, and genes or maintain the order defined in the model.

See also:

to_yaml() Return a string representation.

ruamel.yaml.dump() Base function.

cobra.io.yaml.**load_yaml_model**(*filename*)
Load a cobra model from a file in YAML format.

Parameters **filename** (*str or file-like*) – File path or descriptor that contains the YAML document describing the cobra model.

Returns The cobra model as represented in the YAML document.

Return type *cobra.Model*

See also:

from_yaml() Load from a string.

Package Contents

Functions

model_from_dict(obj)	Build a model from a dict.
model_to_dict(model, sort=False)	Convert model to a dict.
from_json(document)	Load a cobra model from a JSON document.
load_json_model(filename)	Load a cobra model from a file in JSON format.
save_json_model(model, filename, sort=False, pretty=False, **kwargs)	Write the cobra model to a file in JSON format.
to_json(model, sort=False, **kwargs)	Return the model as a JSON document.

Continued on next page

Table 41 – continued from previous page

load_matlab_model(infile_path, variable_name=None, inf=inf)	Load a cobra model stored as a .mat file
save_matlab_model(model, file_name, varname=None)	Save the cobra model as a .mat file.
read_sbml_model(filename, number=float, f_replace=F_REPLACE, **kwargs)	Reads SBML model from given filename.
write_sbml_model(cobra_model, filename, f_replace=F_REPLACE, **kwargs)	Writes cobra model to filename.
validate_sbml_model(filename, check_model=True, internal_consistency=True, check_units_consistency=False, check_modeling_practice=False, **kwargs)	Validate SBML model and returns the model along with a list of errors.
from_yaml(document)	Load a cobra model from a YAML document.
load_yaml_model(filename)	Load a cobra model from a file in YAML format.
save_yaml_model(model, filename, sort=False, **kwargs)	Write the cobra model to a file in YAML format.
to_yaml(model, sort=False, **kwargs)	Return the model as a YAML document.

cobra.io.**model_from_dict**(*obj*)

Build a model from a dict.

Models stored in json are first formulated as a dict that can be read to cobra model using this function.

Parameters **obj** (*dict*) – A dictionary with elements, 'genes', 'compartments', 'id', 'metabolites', 'notes' and 'reactions'; where 'metabolites', 'genes' and 'metabolites' are in turn lists with dictionaries holding all attributes to form the corresponding object.

Returns The generated model.

Return type cora.core.Model

See also:

cobra.io.model_to_dict()

cobra.io.**model_to_dict**(*model*, *sort=False*)

Convert model to a dict.

Parameters

- **model** (cobra.Model) – The model to reformulate as a dict.
- **sort** (*bool, optional*) – Whether to sort the metabolites, reactions, and genes or maintain the order defined in the model.

Returns A dictionary with elements, 'genes', 'compartments', 'id', 'metabolites', 'notes' and 'reactions'; where 'metabolites', 'genes' and 'metabolites' are in turn lists with dictionaries holding all attributes to form the corresponding object.

Return type OrderedDict

See also:

cobra.io.model_from_dict()

cobra.io.**from_json**(*document*)

Load a cobra model from a JSON document.

Parameters **document** (*str*) – The JSON document representation of a cobra model.

Returns The cobra model as represented in the JSON document.

Return type *cobra.Model*

See also:

load_json_model() Load directly from a file.

cobra.io.**load_json_model**(*filename*)
Load a cobra model from a file in JSON format.

Parameters **filename** (*str or file-like*) – File path or descriptor that contains the JSON document describing the cobra model.

Returns The cobra model as represented in the JSON document.

Return type *cobra.Model*

See also:

from_json() Load from a string.

cobra.io.**save_json_model**(*model*, *filename*, *sort=False*, *pretty=False*, ***kwargs*)
Write the cobra model to a file in JSON format.

`kwargs` are passed on to `json.dump`.

Parameters

- **model** (`cobra.Model`) – The cobra model to represent.
- **filename** (*str or file-like*) – File path or descriptor that the JSON representation should be written to.
- **sort** (*bool, optional*) – Whether to sort the metabolites, reactions, and genes or maintain the order defined in the model.
- **pretty** (*bool, optional*) – Whether to format the JSON more compactly (default) or in a more verbose but easier to read fashion. Can be partially overwritten by the `kwargs`.

See also:

to_json() Return a string representation.

json.dump() Base function.

cobra.io.**to_json**(*model*, *sort=False*, ***kwargs*)
Return the model as a JSON document.

`kwargs` are passed on to `json.dumps`.

Parameters

- **model** (`cobra.Model`) – The cobra model to represent.
- **sort** (*bool, optional*) – Whether to sort the metabolites, reactions, and genes or maintain the order defined in the model.

Returns String representation of the cobra model as a JSON document.

Return type str

See also:

save_json_model() Write directly to a file.

json.dumps() Base function.

cobra.io.**load_matlab_model**(*infile_path*, *variable_name=None*, *inf=inf*)
Load a cobra model stored as a .mat file

Parameters

- **infile_path** (*str*) – path to the file to to read

- **variable_name** (*str, optional*) – The variable name of the model in the .mat file. If this is not specified, then the first MATLAB variable which looks like a COBRA model will be used
- **inf** (*value*) – The value to use for infinite bounds. Some solvers do not handle infinite values so for using those, set this to a high numeric value.

Returns The resulting cobra model

Return type cobra.core.Model.Model

cobra.io.**save_matlab_model**(*model*, *file_name*, *varname=None*)

Save the cobra model as a .mat file.

This .mat file can be used directly in the MATLAB version of COBRA.

Parameters

- **model** (*cobra.core.Model.Model object*) – The model to save
- **file_name** (*str or file-like object*) – The file to save to
- **varname** (*string*) – The name of the variable within the workspace

cobra.io.**read_sbml_model**(*filename*, *number=float*, *f_replace=F_REPLACE*, ***kwargs*)

Reads SBML model from given filename.

If the given filename ends with the suffix ''.gz" (for example, ''myfile.xml.gz'),' the file is assumed to be compressed in gzip format and will be automatically decompressed upon reading. Similarly, if the given filename ends with ''.zip" or ''.bz2',' the file is assumed to be compressed in zip or bzip2 format (respectively). Files whose names lack these suffixes will be read uncompressed. Note that if the file is in zip format but the archive contains more than one file, only the first file in the archive will be read and the rest ignored.

To read a gzip/zip file, libSBML needs to be configured and linked with the zlib library at compile time. It also needs to be linked with the bzip2 library to read files in bzip2 format. (Both of these are the default configurations for libSBML.)

This function supports SBML with FBC-v1 and FBC-v2. FBC-v1 models are converted to FBC-v2 models before reading.

The parser tries to fall back to information in notes dictionaries if information is not available in the FBC packages, e.g., CHARGE, FORMULA on species, or GENE_ASSOCIATION, SUBSYSTEM on reactions.

Parameters

- **filename** (*path to SBML file, or SBML string, or SBML file handle*) – SBML which is read into cobra model
- **number** (*data type of stoichiometry: {float, int}*) – In which data type should the stoichiometry be parsed.
- **f_replace** (*dict of replacement functions for id replacement*) – Dictionary of replacement functions for gene, specie, and reaction. By default the following id changes are performed on import: clip **G_** from genes, clip **M_** from species, clip **R_** from reactions If no replacements should be performed, set f_replace={}, None

Returns

Return type *cobra.core.Model*

Notes

Provided file handles cannot be opened in binary mode, i.e., use

with open(path, "r" as f): read_sbml_model(f)

File handles to compressed files are not supported yet.

cobra.io.**write_sbml_model**(*cobra_model*, *filename*, *f_replace=F_REPLACE*, ***kwargs*)
Writes cobra model to filename.

The created model is SBML level 3 version 1 (L1V3) with fbc package v2 (fbc-v2).

If the given filename ends with the suffix ".gz" (for example, "myfile.xml.gz"), libSBML assumes the caller wants the file to be written compressed in gzip format. Similarly, if the given filename ends with ".zip" or ".bz2", libSBML assumes the caller wants the file to be compressed in zip or bzip2 format (respectively). Files whose names lack these suffixes will be written uncompressed. Special considerations for the zip format: If the given filename ends with ".zip", the file placed in the zip archive will have the suffix ".xml" or ".sbml". For example, the file in the zip archive will be named "test.xml" if the given filename is "test.xml.zip" or "test.zip". Similarly, the filename in the archive will be "test.sbml" if the given filename is "test.sbml.zip".

Parameters

- **cobra_model** (cobra.core.Model) – Model instance which is written to SBML
- **filename** (*string*) – path to which the model is written
- **f_replace** (*dict of replacement functions for id replacement*) –

cobra.io.**validate_sbml_model**(*filename*, *check_model=True*, *internal_consistency=True*, *check_units_consistency=False*, *check_modeling_practice=False*, ***kwargs*)
Validate SBML model and returns the model along with a list of errors.

Parameters

- **filename** (*str*) – The filename (or SBML string) of the SBML model to be validated.
- **internal_consistency** (*boolean {True, False}*) – Check internal consistency.
- **check_units_consistency** (*boolean {True, False}*) – Check consistency of units.
- **check_modeling_practice** (*boolean {True, False}*) – Check modeling practise.
- **check_model** (*boolean {True, False}*) – Whether to also check some basic model properties such as reaction boundaries and compartment formulas.

Returns

- *(model, errors)*
- **model** (Model object) – The cobra model if the file could be read successfully or None otherwise.
- **errors** (*dict*) – Warnings and errors grouped by their respective types.

Raises ***CobraSBMLError*** –

cobra.io.**from_yaml**(*document*)
Load a cobra model from a YAML document.

Parameters **document** (*str*) – The YAML document representation of a cobra model.

Returns The cobra model as represented in the YAML document.

Return type *cobra.Model*

See also:

load_yaml_model() Load directly from a file.

cobra.io.**load_yaml_model**(*filename*)
Load a cobra model from a file in YAML format.

Parameters **filename** (*str or file-like*) – File path or descriptor that contains the YAML document describing the cobra model.

Returns The cobra model as represented in the YAML document.

Return type *cobra.Model*

See also:

from_yaml() Load from a string.

cobra.io.**save_yaml_model**(*model*, *filename*, *sort=False*, ***kwargs*)
Write the cobra model to a file in YAML format.

`kwargs` are passed on to `yaml.dump`.

Parameters

- **model** (*cobra.Model*) – The cobra model to represent.
- **filename** (*str or file-like*) – File path or descriptor that the YAML representation should be written to.
- **sort** (*bool, optional*) – Whether to sort the metabolites, reactions, and genes or maintain the order defined in the model.

See also:

to_yaml() Return a string representation.

ruamel.yaml.dump() Base function.

cobra.io.**to_yaml**(*model*, *sort=False*, ***kwargs*)
Return the model as a YAML document.

`kwargs` are passed on to `yaml.dump`.

Parameters

- **model** (*cobra.Model*) – The cobra model to represent.
- **sort** (*bool, optional*) – Whether to sort the metabolites, reactions, and genes or maintain the order defined in the model.

Returns String representation of the cobra model as a YAML document.

Return type str

See also:

save_yaml_model() Write directly to a file.

ruamel.yaml.dump() Base function.

cobra.manipulation

Submodules

cobra.manipulation.annotate

Module Contents

Functions

add_SBO(model)	adds SBO terms for demands and exchanges

cobra.manipulation.annotate.**add_SBO**(*model*)
adds SBO terms for demands and exchanges

This works for models which follow the standard convention for constructing and naming these reactions.

The reaction should only contain the single metabolite being exchanged, and the id should be EX_metid or DM_metid

cobra.manipulation.delete

Module Contents

Classes

_GeneRemover	A `NodeVisitor` subclass that walks the abstract syntax tree and

Functions

prune_unused_metabolites(cobra_model)	Remove metabolites that are not involved in any reactions and
prune_unused_reactions(cobra_model)	Remove reactions with no assigned metabolites, returns pruned model
undelete_model_genes(cobra_model)	Undoes the effects of a call to delete_model_genes in place.
get_compiled_gene_reaction_rules(cobra_model)	Generates a dict of compiled gene_reaction_rules
find_gene_knockout_reactions(cobra_model, gene_list, compiled_gene_reaction_rules=None)	identify reactions which will be disabled when the genes are knocked out
delete_model_genes(cobra_model, gene_list, cumulative_deletions=True, disable_orphans=False)	delete_model_genes will set the upper and lower bounds for reactions
remove_genes(cobra_model, gene_list, remove_reactions=True)	remove genes entirely from the model

cobra.manipulation.delete.**prune_unused_metabolites**(*cobra_model*)
Remove metabolites that are not involved in any reactions and returns pruned model

Parameters **cobra_model** (class:~*cobra.core.Model.Model* object) – the model to remove unused metabolites from

Returns

- **output_model** (class:~*cobra.core.Model.Model* object) – input model with unused metabolites removed
- **inactive_metabolites** (list of class:~*cobra.core.reaction.Reaction*) – list of metabolites that were removed

cobra.manipulation.delete.**prune_unused_reactions**(*cobra_model*)
Remove reactions with no assigned metabolites, returns pruned model

Parameters **cobra_model** (class:~*cobra.core.Model.Model* object) – the model to remove unused reactions from

Returns

- **output_model** (class:~*cobra.core.Model.Model* object) – input model with unused reactions removed
- **reactions_to_prune** (list of class:~*cobra.core.reaction.Reaction*) – list of reactions that were removed

cobra.manipulation.delete.**undelete_model_genes**(*cobra_model*)
Undoes the effects of a call to delete_model_genes in place.

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

cobra.manipulation.delete.**get_compiled_gene_reaction_rules**(*cobra_model*)
Generates a dict of compiled gene_reaction_rules

Any gene_reaction_rule expressions which cannot be compiled or do not evaluate after compiling will be excluded. The result can be used in the find_gene_knockout_reactions function to speed up evaluation of these rules.

cobra.manipulation.delete.**find_gene_knockout_reactions**(*cobra_model*, *gene_list*, *compiled_gene_reaction_rules=None*)
identify reactions which will be disabled when the genes are knocked out

cobra_model: `Model`

gene_list: iterable of `Gene`

compiled_gene_reaction_rules: dict of {reaction_id: compiled_string} If provided, this gives pre-compiled gene_reaction_rule strings. The compiled rule strings can be evaluated much faster. If a rule is not provided, the regular expression evaluation will be used. Because not all gene_reaction_rule strings can be evaluated, this dict must exclude any rules which can not be used with eval.

cobra.manipulation.delete.**delete_model_genes**(*cobra_model*, *gene_list*, *cumulative_deletions=True*, *disable_orphans=False*)
delete_model_genes will set the upper and lower bounds for reactions catalysed by the genes in gene_list if deleting the genes means that the reaction cannot proceed according to cobra_model.reactions[:].gene_reaction_rule

cumulative_deletions: False or True. If True then any previous deletions will be maintained in the model.

class cobra.manipulation.delete.**_GeneRemover**(*target_genes*)
Bases: `ast.NodeTransformer`

A `NodeVisitor` subclass that walks the abstract syntax tree and allows modification of nodes.

The *NodeTransformer* will walk the AST and use the return value of the visitor methods to replace or remove the old node. If the return value of the visitor method is `None`, the node will be removed from its location, otherwise it is replaced with the return value. The return value may be the original node in which case no replacement takes place.

Here is an example transformer that rewrites all occurrences of name lookups (`foo`) to `data['foo']`:

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Index(value=Str(s=node.id)),
            ctx=node.ctx
        )
```

Keep in mind that if the node you're operating on has child nodes you must either transform the child nodes yourself or call the `generic_visit()` method for the node first.

For nodes that were part of a collection of statements (that applies to all statement nodes), the visitor may also return a list of nodes rather than just a single node.

Usually you use the transformer like this:

```
node = YourTransformer().visit(node)
```

visit_Name(*self*, *node*)

visit_BoolOp(*self*, *node*)

cobra.manipulation.delete.**remove_genes**(*cobra_model*, *gene_list*, *remove_reactions=True*)

remove genes entirely from the model

This will also simplify all gene_reaction_rules with this gene inactivated.

cobra.manipulation.modify

Module Contents

Classes

_GeneEscaper	A `NodeVisitor` subclass that walks the abstract syntax tree and

Functions

_escape_str_id(id_str)	make a single string id SBML compliant
escape_ID(cobra_model)	makes all ids SBML compliant
rename_genes(cobra_model, rename_dict)	renames genes in a model from the rename_dict

cobra.manipulation.modify.**_renames = [['.', '_DOT_'], ['(', '_LPAREN_'], [')', '_RPAREN_**

cobra.manipulation.modify.**_escape_str_id**(*id_str*)

make a single string id SBML compliant

class cobra.manipulation.modify.**_GeneEscaper**

Bases: `ast.NodeTransformer`

A `NodeVisitor` subclass that walks the abstract syntax tree and allows modification of nodes.

The *NodeTransformer* will walk the AST and use the return value of the visitor methods to replace or remove the old node. If the return value of the visitor method is `None`, the node will be removed from its location, otherwise it is replaced with the return value. The return value may be the original node in which case no replacement takes place.

Here is an example transformer that rewrites all occurrences of name lookups (`foo`) to `data['foo']`:

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Index(value=Str(s=node.id)),
            ctx=node.ctx
        )
```

Keep in mind that if the node you're operating on has child nodes you must either transform the child nodes yourself or call the `generic_visit()` method for the node first.

For nodes that were part of a collection of statements (that applies to all statement nodes), the visitor may also return a list of nodes rather than just a single node.

Usually you use the transformer like this:

```
node = YourTransformer().visit(node)
```

visit_Name(*self*, *node*)

cobra.manipulation.modify.**escape_ID**(*cobra_model*)
makes all ids SBML compliant

cobra.manipulation.modify.**rename_genes**(*cobra_model*, *rename_dict*)
renames genes in a model from the rename_dict

cobra.manipulation.validate

Module Contents

Functions

check_mass_balance(model)	
check_metabolite_compartment_formula(model)	

cobra.manipulation.validate.**NOT_MASS_BALANCED_TERMS**

cobra.manipulation.validate.**check_mass_balance**(*model*)

cobra.manipulation.validate.**check_metabolite_compartment_formula**(*model*)

Package Contents

Functions

add_SBO(model)	adds SBO terms for demands and exchanges
delete_model_genes(cobra_model, gene_list, cumulative_deletions=True, disable_orphans=False)	delete_model_genes will set the upper and lower bounds for reactions
find_gene_knockout_reactions(cobra_model, gene_list, compiled_gene_reaction_rules=None)	identify reactions which will be disabled when the genes are knocked out
remove_genes(cobra_model, gene_list, remove_reactions=True)	remove genes entirely from the model

Continued on next page

Table 48 – continued from previous page

undelete_model_genes(cobra_model)	Undoes the effects of a call to delete_model_genes in place.
escape_ID(cobra_model)	makes all ids SBML compliant
get_compiled_gene_reaction_rules(cobra_model)	Generates a dict of compiled gene_reaction_rules
check_mass_balance(model)	
check_metabolite_compartment_formula(model)	

cobra.manipulation.**add_SBO**(*model*)
adds SBO terms for demands and exchanges

This works for models which follow the standard convention for constructing and naming these reactions.

The reaction should only contain the single metabolite being exchanged, and the id should be EX_metid or DM_metid

cobra.manipulation.**delete_model_genes**(*cobra_model*, *gene_list*, *cumulative_deletions=True*, *disable_orphans=False*)
delete_model_genes will set the upper and lower bounds for reactions catalysed by the genes in gene_list if deleting the genes means that the reaction cannot proceed according to cobra_model.reactions[:].gene_reaction_rule

cumulative_deletions: False or True. If True then any previous deletions will be maintained in the model.

cobra.manipulation.**find_gene_knockout_reactions**(*cobra_model*, *gene_list*, *compiled_gene_reaction_rules=None*)
identify reactions which will be disabled when the genes are knocked out

cobra_model: `Model`

gene_list: iterable of `Gene`

compiled_gene_reaction_rules: dict of {reaction_id: compiled_string} If provided, this gives pre-compiled gene_reaction_rule strings. The compiled rule strings can be evaluated much faster. If a rule is not provided, the regular expression evaluation will be used. Because not all gene_reaction_rule strings can be evaluated, this dict must exclude any rules which can not be used with eval.

cobra.manipulation.**remove_genes**(*cobra_model*, *gene_list*, *remove_reactions=True*)
remove genes entirely from the model

This will also simplify all gene_reaction_rules with this gene inactivated.

cobra.manipulation.**undelete_model_genes**(*cobra_model*)
Undoes the effects of a call to delete_model_genes in place.

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

cobra.manipulation.**escape_ID**(*cobra_model*)
makes all ids SBML compliant

cobra.manipulation.**get_compiled_gene_reaction_rules**(*cobra_model*)
Generates a dict of compiled gene_reaction_rules

Any gene_reaction_rule expressions which cannot be compiled or do not evaluate after compiling will be excluded. The result can be used in the find_gene_knockout_reactions function to speed up evaluation of these rules.

cobra.manipulation.**check_mass_balance**(*model*)

cobra.manipulation.**check_metabolite_compartment_formula**(*model*)

cobra.medium

Imports for the media module.

Submodules

cobra.medium.annotations

Lists and annotations for compartment names and reactions.

Please send a PR if you want to add something here :)

Module Contents

cobra.medium.annotations.**excludes**
A list of sub-strings in reaction IDs that usually indicate that the reaction is *not* a reaction of the specified type.

cobra.medium.annotations.**sbo_terms**
SBO term identifiers for various boundary types.

cobra.medium.annotations.**compartment_shortlist**
A list of common compartment abbreviations and alternative names.

cobra.medium.boundary_types

Contains function to identify the type of boundary reactions.

This module uses various heuristics to decide whether a boundary reaction is an exchange, demand or sink reaction. It mostly orientates on the following paper:

Thiele, I., & Palsson, B. Ø. (2010, January). A protocol for generating a high-quality genome-scale metabolic reconstruction. Nature protocols. Nature Publishing Group. http://doi.org/10.1038/nprot.2009.203

Module Contents

Functions

find_external_compartment(model)	Find the external compartment in the model.
is_boundary_type(reaction, boundary_type, external_compartment)	Check whether a reaction is an exchange reaction.
find_boundary_types(model, boundary_type, external_compartment=None)	Find specific boundary reactions.

cobra.medium.boundary_types.**LOGGER**

cobra.medium.boundary_types.**find_external_compartment**(*model*)
Find the external compartment in the model.

Uses a simple heuristic where the external compartment should be the one with the most exchange reactions.

Parameters **model** (cobra.Model) – A cobra model.

Returns The putative external compartment.

Return type str

cobra.medium.boundary_types.**is_boundary_type**(*reaction*, *boundary_type*, *external_compartment*)

Check whether a reaction is an exchange reaction.

Parameters

- **reaction** (cobra.Reaction) – The reaction to check.
- **boundary_type** (*str*) – What boundary type to check for. Must be one of "exchange", "demand", or "sink".
- **external_compartment** (*str*) – The id for the external compartment.

Returns Whether the reaction looks like the requested type. Might be based on a heuristic.

Return type boolean

cobra.medium.boundary_types.**find_boundary_types**(*model*, *boundary_type*, *external_compartment=None*)

Find specific boundary reactions.

Parameters

- **model** (cobra.Model) – A cobra model.
- **boundary_type** (*str*) – What boundary type to check for. Must be one of "exchange", "demand", or "sink".
- **external_compartment** (*str or None*) – The id for the external compartment. If None it will be detected automatically.

Returns A list of likely boundary reactions of a user defined type.

Return type list of cobra.reaction

cobra.medium.minimal_medium

Contains functions and helpers to obtain minimal growth media.

Module Contents

Functions

add_linear_obj(model)	Add a linear version of a minimal medium to the model solver.
add_mip_obj(model)	Add a mixed-integer version of a minimal medium to the model.
_as_medium(exchanges, tolerance=1e-06, exports=False)	Convert a solution to medium.
minimal_medium(model, min_objective_value=0.1, exports=False, minimize_components=False, open_exchanges=False)	Find the minimal growth medium for the model.

cobra.medium.minimal_medium.**LOGGER**

cobra.medium.minimal_medium.**add_linear_obj**(*model*)

Add a linear version of a minimal medium to the model solver.

Changes the optimization objective to finding the growth medium requiring the smallest total import flux:

```
minimize sum |r_i| for r_i in import_reactions
```

Parameters **model** (cobra.Model) – The model to modify.

cobra.medium.minimal_medium.**add_mip_obj**(*model*)

Add a mixed-integer version of a minimal medium to the model.

Changes the optimization objective to finding the medium with the least components:

```
minimize size(R) where R part of import_reactions
```

Parameters **model** (*cobra.model*) – The model to modify.

cobra.medium.minimal_medium.**_as_medium**(*exchanges*, *tolerance=1e-06*, *exports=False*)

Convert a solution to medium.

Parameters

- **exchanges** (*list of cobra.reaction*) – The exchange reactions to consider.
- **tolerance** (*positive double*) – The absolute tolerance for fluxes. Fluxes with an absolute value smaller than this number will be ignored.
- **exports** (*bool*) – Whether to return export fluxes as well.

Returns The "medium", meaning all active import fluxes in the solution.

Return type pandas.Series

cobra.medium.minimal_medium.**minimal_medium**(*model*, *min_objective_value=0.1*, *exports=False*, *minimize_components=False*, *open_exchanges=False*)

Find the minimal growth medium for the model.

Finds the minimal growth medium for the model which allows for model as well as individual growth. Here, a minimal medium can either be the medium requiring the smallest total import flux or the medium requiring the least components (ergo ingredients), which will be much slower due to being a mixed integer problem (MIP).

Parameters

- **model** (*cobra.model*) – The model to modify.
- **min_objective_value** (*positive float or array-like object*) – The minimum growth rate (objective) that has to be achieved.
- **exports** (*boolean*) – Whether to include export fluxes in the returned medium. Defaults to False which will only return import fluxes.
- **minimize_components** (*boolean or positive int*) – Whether to minimize the number of components instead of the total import flux. Might be more intuitive if set to True but may also be slow to calculate for large communities. If set to a number *n* will return up to *n* alternative solutions all with the same number of components.
- **open_exchanges** (*boolean or number*) – Whether to ignore currently set bounds and make all exchange reactions in the model possible. If set to a number all exchange reactions will be opened with (-number, number) as bounds.

Returns A series giving the import flux for each required import reaction and (optionally) the associated export fluxes. All exchange fluxes are oriented into the import reaction e.g. positive fluxes denote imports and negative fluxes exports. If *minimize_components* is a number larger 1 may return a DataFrame where each column is a minimal medium. Returns None if the minimization is infeasible (for instance if min_growth > maximum growth rate).

Return type pandas.Series, pandas.DataFrame or None

Notes

Due to numerical issues the *minimize_components* option will usually only minimize the number of "large" import fluxes. Specifically, the detection limit is given by `integrality_tolerance * max_bound` where `max_bound` is the largest bound on an import reaction. Thus, if you are interested in small import fluxes as well you may have to adjust the integrality tolerance at first with *model.solver.configuration.tolerances.integrality = 1e-7* for instance. However, this will be *very* slow for large models especially with GLPK.

Package Contents

Functions

find_boundary_types(model, boundary_type, external_compartment=None)	Find specific boundary reactions.
find_external_compartment(model)	Find the external compartment in the model.
is_boundary_type(reaction, boundary_type, external_compartment)	Check whether a reaction is an exchange reaction.
minimal_medium(model, min_objective_value=0.1, exports=False, minimize_components=False, open_exchanges=False)	Find the minimal growth medium for the model.

cobra.medium.**find_boundary_types**(*model*, *boundary_type*, *external_compartment=None*)

Find specific boundary reactions.

Parameters

- **model** (*cobra.Model*) – A cobra model.
- **boundary_type** (*str*) – What boundary type to check for. Must be one of "exchange", "demand", or "sink".
- **external_compartment** (*str or None*) – The id for the external compartment. If None it will be detected automatically.

Returns A list of likely boundary reactions of a user defined type.

Return type list of cobra.reaction

cobra.medium.**find_external_compartment**(*model*)

Find the external compartment in the model.

Uses a simple heuristic where the external compartment should be the one with the most exchange reactions.

Parameters **model** (*cobra.Model*) – A cobra model.

Returns The putative external compartment.

Return type str

cobra.medium.**is_boundary_type**(*reaction*, *boundary_type*, *external_compartment*)

Check whether a reaction is an exchange reaction.

Parameters

- **reaction** (*cobra.Reaction*) – The reaction to check.
- **boundary_type** (*str*) – What boundary type to check for. Must be one of "exchange", "demand", or "sink".
- **external_compartment** (*str*) – The id for the external compartment.

Returns Whether the reaction looks like the requested type. Might be based on a heuristic.

Return type boolean

cobra.medium.**sbo_terms**
SBO term identifiers for various boundary types.

cobra.medium.**minimal_medium**(*model*, *min_objective_value=0.1*, *exports=False*, *minimize_components=False*, *open_exchanges=False*)
Find the minimal growth medium for the model.

Finds the minimal growth medium for the model which allows for model as well as individual growth. Here, a minimal medium can either be the medium requiring the smallest total import flux or the medium requiring the least components (ergo ingredients), which will be much slower due to being a mixed integer problem (MIP).

Parameters

- **model** (*cobra.model*) – The model to modify.
- **min_objective_value** (*positive float or array-like object*) – The minimum growth rate (objective) that has to be achieved.
- **exports** (*boolean*) – Whether to include export fluxes in the returned medium. Defaults to False which will only return import fluxes.
- **minimize_components** (*boolean or positive int*) – Whether to minimize the number of components instead of the total import flux. Might be more intuitive if set to True but may also be slow to calculate for large communities. If set to a number *n* will return up to *n* alternative solutions all with the same number of components.
- **open_exchanges** (*boolean or number*) – Whether to ignore currently set bounds and make all exchange reactions in the model possible. If set to a number all exchange reactions will be opened with (-number, number) as bounds.

Returns A series giving the import flux for each required import reaction and (optionally) the associated export fluxes. All exchange fluxes are oriented into the import reaction e.g. positive fluxes denote imports and negative fluxes exports. If *minimize_components* is a number larger 1 may return a DataFrame where each column is a minimal medium. Returns None if the minimization is infeasible (for instance if min_growth > maximum growth rate).

Return type pandas.Series, pandas.DataFrame or None

Notes

Due to numerical issues the *minimize_components* option will usually only minimize the number of "large" import fluxes. Specifically, the detection limit is given by `integrality_tolerance * max_bound` where `max_bound` is the largest bound on an import reaction. Thus, if you are interested in small import fluxes as well you may have to adjust the integrality tolerance at first with *model.solver.configuration.tolerances.integrality = 1e-7* for instance. However, this will be *very* slow for large models especially with GLPK.

cobra.sampling

Submodules

cobra.sampling.achr

Provide ACHR sampler.

Module Contents

Classes

ACHRSampler	Artificial Centering Hit-and-Run sampler.

class cobra.sampling.achr.**ACHRSampler**(*model*, *thinning=100*, *nproj=None*, *seed=None*)

Bases: *cobra.sampling.hr_sampler.HRSampler*

Artificial Centering Hit-and-Run sampler.

A sampler with low memory footprint and good convergence.

Parameters

- **model** (*cobra.Model*) – The cobra model from which to generate samples.
- **thinning** (*int, optional*) – The thinning factor of the generated sampling chain. A thinning of 10 means samples are returned every 10 steps.
- **nproj** (*int > 0, optional*) – How often to reproject the sampling point into the feasibility space. Avoids numerical issues at the cost of lower sampling. If you observe many equality constraint violations with *sampler.validate* you should lower this number.
- **seed** (*int > 0, optional*) – Sets the random number seed. Initialized to the current time stamp if None.

model

The cobra model from which the samples get generated.

Type *cobra.Model*

thinning

The currently used thinning factor.

Type int

n_samples

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

Type int

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

Type collections.namedtuple

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

Type numpy.matrix

retries
The overall of sampling retries the sampler has observed. Larger values indicate numerical instabilities.

Type int

seed
Sets the random number seed. Initialized to the current time stamp if None.

Type int > 0, optional

nproj
How often to reproject the sampling point into the feasibility space.

Type int

fwd_idx
Has one entry for each reaction in the model containing the index of the respective forward variable.

Type numpy.array

rev_idx
Has one entry for each reaction in the model containing the index of the respective reverse variable.

Type numpy.array

prev
The current/last flux sample generated.

Type numpy.array

center
The center of the sampling space as estimated by the mean of all previously generated samples.

Type numpy.array

Notes

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

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

[2] https://github.com/opencobra/cobratoolbox

References

__single_iteration(*self*)

sample(*self*, *n*, *fluxes=True*)
Generate a set of samples.

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

Parameters

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

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

Return type numpy.matrix

Notes

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

cobra.sampling.hr_sampler

Provide base class for Hit-and-Run samplers.

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

Module Contents

Classes

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

Functions

shared_np_array(shape, data=None, integer=False)	Create a new numpy array that resides in shared memory.
step(sampler, x, delta, fraction=None, tries=0)	Sample a new feasible point from the point *x* in direction *delta*.

cobra.sampling.hr_sampler.**LOGGER**

cobra.sampling.hr_sampler.**MAX_TRIES = 100**

cobra.sampling.hr_sampler.**Problem**
Defines the matrix representation of a sampling problem.

cobra.sampling.hr_sampler.**equalities**
All equality constraints in the model.

Type numpy.array

cobra.sampling.hr_sampler.**b**

The right side of the equality constraints.

Type numpy.array

cobra.sampling.hr_sampler.**inequalities**
All inequality constraints in the model.

Type numpy.array

cobra.sampling.hr_sampler.**bounds**
The lower and upper bounds for the inequality constraints.

Type numpy.array

cobra.sampling.hr_sampler.**variable_bounds**
The lower and upper bounds for the variables.

Type numpy.array

cobra.sampling.hr_sampler.**homogeneous**
Indicates whether the sampling problem is homogenous, e.g. whether there exist no non-zero fixed variables or constraints.

Type boolean

cobra.sampling.hr_sampler.**nullspace**
A matrix containing the nullspace of the equality constraints. Each column is one basis vector.

Type numpy.matrix

cobra.sampling.hr_sampler.**shared_np_array**(*shape*, *data=None*, *integer=False*)
Create a new numpy array that resides in shared memory.

Parameters

- **shape** (*tuple of ints*) – The shape of the new array.
- **data** (*numpy.array*) – Data to copy to the new array. Has to have the same shape.
- **integer** (*boolean*) – Whether to use an integer array. Defaults to False which means float array.

class cobra.sampling.hr_sampler.**HRSampler**(*model*, *thinning*, *nproj=None*, *seed=None*)
Bases: object

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

Parameters

- **model** (cobra.Model) – The cobra model from which to generate samples.
- **thinning** (*int*) – The thinning factor of the generated sampling chain. A thinning of 10 means samples are returned every 10 steps.
- **nproj** (*int > 0, optional*) – How often to reproject the sampling point into the feasibility space. Avoids numerical issues at the cost of lower sampling. If you observe many equality constraint violations with *sampler.validate* you should lower this number.
- **seed** (*int > 0, optional*) – The random number seed that should be used.

model
The cobra model from which the sampes get generated.

Type *cobra.Model*

feasibility_tol
The tolerance used for checking equalities feasibility.

Type float

bounds_tol
The tolerance used for checking bounds feasibility.

Type float

thinning
The currently used thinning factor.

Type int

n_samples
The total number of samples that have been generated by this sampler instance.

Type int

retries
The overall of sampling retries the sampler has observed. Larger values indicate numerical instabilities.

Type int

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

Type collections.namedtuple

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

Type numpy.matrix

nproj
How often to reproject the sampling point into the feasibility space.

Type int

seed
Sets the random number seed. Initialized to the current time stamp if None.

Type int > 0, optional

fwd_idx
Has one entry for each reaction in the model containing the index of the respective forward variable.

Type numpy.array

rev_idx
Has one entry for each reaction in the model containing the index of the respective reverse variable.

Type numpy.array

__build_problem(*self*)
Build the matrix representation of the sampling problem.

generate_fva_warmup(*self*)
Generate the warmup points for the sampler.

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

_reproject(*self*, *p*)
Reproject a point into the feasibility region.

This function is guaranteed to return a new feasible point. However, no guarantees in terms of proximity to the original point can be made.

Parameters **p** (*numpy.array*) – The current sample point.

Returns A new feasible point. If *p* was feasible it wil return p.

Return type numpy.array

_random_point (*self*)
Find an approximately random point in the flux cone.

_is_redundant (*self*, *matrix*, *cutoff=None*)
Identify rdeundant rows in a matrix that can be removed.

_bounds_dist (*self*, *p*)
Get the lower and upper bound distances. Negative is bad.

sample (*self*, *n*, *fluxes=True*)
Abstract sampling function.

Should be overwritten by child classes.

batch (*self*, *batch_size*, *batch_num*, *fluxes=True*)
Create a batch generator.

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

Parameters

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

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

validate (*self*, *samples*)
Validate a set of samples for equality and inequality feasibility.

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

Parameters **samples** (*numpy.matrix*) – Must be of dimension (n_samples x n_reactions). Contains the samples to be validated. Samples must be from fluxes.

Returns

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

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

Return type numpy.array

cobra.sampling.hr_sampler.**step** (*sampler*, *x*, *delta*, *fraction=None*, *tries=0*)
Sample a new feasible point from the point *x* in direction *delta*.

cobra.sampling.optgp

Provide OptGP sampler.

Module Contents

Classes

OptGPSampler	A parallel optimized sampler.

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

Bases: *cobra.sampling.hr_sampler.HRSampler*

A parallel optimized sampler.

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

Parameters

- **model** (*cobra.Model*) – The cobra model from which to generate samples.
- **processes** (*int, optional (default Configuration.processes)*) – The number of processes used during sampling.
- **thinning** (*int, optional*) – The thinning factor of the generated sampling chain. A thinning of 10 means samples are returned every 10 steps.
- **nproj** (*int > 0, optional*) – How often to reproject the sampling point into the feasibility space. Avoids numerical issues at the cost of lower sampling. If you observe many equality constraint violations with *sampler.validate* you should lower this number.
- **seed** (*int > 0, optional*) – Sets the random number seed. Initialized to the current time stamp if None.

model
The cobra model from which the samples get generated.

Type *cobra.Model*

thinning
The currently used thinning factor.

Type int

n_samples
The total number of samples that have been generated by this sampler instance.

Type int

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

Type collections.namedtuple

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

Type numpy.matrix

[1] Megchelenbrink W, Huynen M, Marchiori E (2014) optGpSampler: An Improved Tool for Uniformly Sampling the Solution-Space of Genome-Scale Metabolic Networks. PLoS ONE 9(2): e86587. https://doi.org/10.1371/journal.pone.0086587

retries
The overall of sampling retries the sampler has observed. Larger values indicate numerical instabilities.

Type int

seed
Sets the random number seed. Initialized to the current time stamp if None.

Type int > 0, optional

nproj
How often to reproject the sampling point into the feasibility space.

Type int

fwd_idx
Has one entry for each reaction in the model containing the index of the respective forward variable.

Type numpy.array

rev_idx
Has one entry for each reaction in the model containing the index of the respective reverse variable.

Type numpy.array

prev
The current/last flux sample generated.

Type numpy.array

center
The center of the sampling space as estimated by the mean of all previously generated samples.

Type numpy.array

Notes

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

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

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

References

sample(*self*, *n*, *fluxes=True*)
Generate a set of samples.

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

Parameters

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

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

Return type numpy.matrix

Notes

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

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

__getstate__(*self*)
Return the object for serialization.

cobra.sampling.sampling

Module implementing flux sampling for cobra models.

Module Contents

Functions

sample(model, n, method='optgp', thinning=100, processes=1, seed=None)	Sample valid flux distributions from a cobra model.

cobra.sampling.sampling.**sample**(*model*, *n*, *method='optgp'*, *thinning=100*, *processes=1*, *seed=None*)
Sample valid flux distributions from a cobra model.

The function samples valid flux distributions from a cobra model. Currently we support two methods:

1. **'optgp' (default) which uses the OptGPSampler that supports parallel** sampling[1]. Requires large numbers of samples to be performant (n < 1000). For smaller samples 'achr' might be better suited.

or

2. 'achr' which uses artificial centering hit-and-run. This is a single process method with good convergence[2].

Parameters

- **model** (*cobra.Model*) – The model from which to sample flux distributions.
- **n** (*int*) – The number of samples to obtain. When using 'optgp' this must be a multiple of *processes*, otherwise a larger number of samples will be returned.
- **method** (*str, optional*) – The sampling algorithm to use.
- **thinning** (*int, optional*) – The thinning factor of the generated sampling chain. A thinning of 10 means samples are returned every 10 steps. Defaults to 100 which in benchmarks gives approximately uncorrelated samples. If set to one will return all iterates.
- **processes** (*int, optional*) – Only used for 'optgp'. The number of processes used to generate samples.

[1] Megchelenbrink W, Huynen M, Marchiori E (2014) optGpSampler: An Improved Tool for Uniformly Sampling the Solution-Space of Genome-Scale Metabolic Networks. PLoS ONE 9(2): e86587.

[2] Direction Choice for Accelerated Convergence in Hit-and-Run Sampling David E. Kaufman Robert L. Smith Operations Research 199846:1 , 84-95

- **seed** (*int > 0, optional*) – The random number seed to be used. Initialized to current time stamp if None.

Returns The generated flux samples. Each row corresponds to a sample of the fluxes and the columns are the reactions.

Return type pandas.DataFrame

Notes

The samplers have a correction method to ensure equality feasibility for long-running chains, however this will only work for homogeneous models, meaning models with no non-zero fixed variables or constraints (right-hand side of the equalities are zero).

References

Package Contents

Classes

HRSampler	The abstract base class for hit-and-run samplers.
ACHRSampler	Artificial Centering Hit-and-Run sampler.
OptGPSampler	A parallel optimized sampler.

Functions

shared_np_array(shape, data=None, integer=False)	Create a new numpy array that resides in shared memory.
step(sampler, x, delta, fraction=None, tries=0)	Sample a new feasible point from the point *x* in direction *delta*.
sample(model, n, method='optgp', thinning=100, processes=1, seed=None)	Sample valid flux distributions from a cobra model.

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

Bases: object

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

Parameters

- **model** (*cobra.Model*) – The cobra model from which to generate samples.
- **thinning** (*int*) – The thinning factor of the generated sampling chain. A thinning of 10 means samples are returned every 10 steps.
- **nproj** (*int > 0, optional*) – How often to reproject the sampling point into the feasibility space. Avoids numerical issues at the cost of lower sampling. If you observe many equality constraint violations with *sampler.validate* you should lower this number.
- **seed** (*int > 0, optional*) – The random number seed that should be used.

model

The cobra model from which the sampes get generated.

Type *cobra.Model*

feasibility_tol
The tolerance used for checking equalities feasibility.

Type float

bounds_tol
The tolerance used for checking bounds feasibility.

Type float

thinning
The currently used thinning factor.

Type int

n_samples
The total number of samples that have been generated by this sampler instance.

Type int

retries
The overall of sampling retries the sampler has observed. Larger values indicate numerical instabilities.

Type int

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

Type collections.namedtuple

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

Type numpy.matrix

nproj
How often to reproject the sampling point into the feasibility space.

Type int

seed
Sets the random number seed. Initialized to the current time stamp if None.

Type int > 0, optional

fwd_idx
Has one entry for each reaction in the model containing the index of the respective forward variable.

Type numpy.array

rev_idx
Has one entry for each reaction in the model containing the index of the respective reverse variable.

Type numpy.array

__build_problem(*self*)
Build the matrix representation of the sampling problem.

generate_fva_warmup(*self*)
Generate the warmup points for the sampler.

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

_reproject(*self*, *p*)
Reproject a point into the feasibility region.

This function is guaranteed to return a new feasible point. However, no guarantees in terms of proximity to the original point can be made.

Parameters **p** (*numpy.array*) – The current sample point.

Returns A new feasible point. If *p* was feasible it wil return p.

Return type numpy.array

_random_point (*self*)
Find an approximately random point in the flux cone.

_is_redundant (*self*, *matrix*, *cutoff=None*)
Identify rdeundant rows in a matrix that can be removed.

_bounds_dist (*self*, *p*)
Get the lower and upper bound distances. Negative is bad.

sample (*self*, *n*, *fluxes=True*)
Abstract sampling function.

Should be overwritten by child classes.

batch (*self*, *batch_size*, *batch_num*, *fluxes=True*)
Create a batch generator.

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

Parameters

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

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

validate (*self*, *samples*)
Validate a set of samples for equality and inequality feasibility.

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

Parameters **samples** (*numpy.matrix*) – Must be of dimension (n_samples x n_reactions). Contains the samples to be validated. Samples must be from fluxes.

Returns

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

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

Return type numpy.array

cobra.sampling.**shared_np_array** (*shape*, *data=None*, *integer=False*)
Create a new numpy array that resides in shared memory.

Parameters

- **shape** (*tuple of ints*) – The shape of the new array.

- **data** (*numpy.array*) – Data to copy to the new array. Has to have the same shape.
- **integer** (*boolean*) – Whether to use an integer array. Defaults to False which means float array.

cobra.sampling.**step**(*sampler*, *x*, *delta*, *fraction=None*, *tries=0*)
Sample a new feasible point from the point *x* in direction *delta*.

class cobra.sampling.**ACHRSampler**(*model*, *thinning=100*, *nproj=None*, *seed=None*)
Bases: *cobra.sampling.hr_sampler.HRSampler*

Artificial Centering Hit-and-Run sampler.

A sampler with low memory footprint and good convergence.

Parameters

- **model** (*cobra.Model*) – The cobra model from which to generate samples.
- **thinning** (*int, optional*) – The thinning factor of the generated sampling chain. A thinning of 10 means samples are returned every 10 steps.
- **nproj** (*int > 0, optional*) – How often to reproject the sampling point into the feasibility space. Avoids numerical issues at the cost of lower sampling. If you observe many equality constraint violations with *sampler.validate* you should lower this number.
- **seed** (*int > 0, optional*) – Sets the random number seed. Initialized to the current time stamp if None.

model
The cobra model from which the samples get generated.

Type *cobra.Model*

thinning
The currently used thinning factor.

Type int

n_samples
The total number of samples that have been generated by this sampler instance.

Type int

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

Type collections.namedtuple

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

Type numpy.matrix

retries
The overall of sampling retries the sampler has observed. Larger values indicate numerical instabilities.

Type int

seed
Sets the random number seed. Initialized to the current time stamp if None.

Type int > 0, optional

nproj
How often to reproject the sampling point into the feasibility space.

Type int

fwd_idx
Has one entry for each reaction in the model containing the index of the respective forward variable.

Type numpy.array

rev_idx
Has one entry for each reaction in the model containing the index of the respective reverse variable.

Type numpy.array

prev
The current/last flux sample generated.

Type numpy.array

center
The center of the sampling space as estimated by the mean of all previously generated samples.

Type numpy.array

Notes

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

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

References

__single_iteration(*self*)

sample(*self*, *n*, *fluxes=True*)
Generate a set of samples.

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

Parameters

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

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

Return type numpy.matrix

Notes

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

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

Bases: *cobra.sampling.hr_sampler.HRSampler*

A parallel optimized sampler.

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

Parameters

- **model** (*cobra.Model*) – The cobra model from which to generate samples.
- **processes** (*int, optional (default Configuration.processes)*) – The number of processes used during sampling.
- **thinning** (*int, optional*) – The thinning factor of the generated sampling chain. A thinning of 10 means samples are returned every 10 steps.
- **nproj** (*int > 0, optional*) – How often to reproject the sampling point into the feasibility space. Avoids numerical issues at the cost of lower sampling. If you observe many equality constraint violations with *sampler.validate* you should lower this number.
- **seed** (*int > 0, optional*) – Sets the random number seed. Initialized to the current time stamp if None.

model
The cobra model from which the samples get generated.

Type *cobra.Model*

thinning
The currently used thinning factor.

Type int

n_samples
The total number of samples that have been generated by this sampler instance.

Type int

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

Type collections.namedtuple

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

Type numpy.matrix

retries
The overall of sampling retries the sampler has observed. Larger values indicate numerical instabilities.

Type int

seed
Sets the random number seed. Initialized to the current time stamp if None.

Type int > 0, optional

nproj
How often to reproject the sampling point into the feasibility space.

Type int

fwd_idx
Has one entry for each reaction in the model containing the index of the respective forward variable.

Type numpy.array

rev_idx
Has one entry for each reaction in the model containing the index of the respective reverse variable.

Type numpy.array

prev
The current/last flux sample generated.

Type numpy.array

center
The center of the sampling space as estimated by the mean of all previously generated samples.

Type numpy.array

Notes

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

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

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

References

sample(*self*, *n*, *fluxes=True*)
Generate a set of samples.

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

Parameters

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

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

Return type numpy.matrix

Notes

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

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

__getstate__(*self*)
Return the object for serialization.

cobra.sampling.**sample**(*model*, *n*, *method='optgp'*, *thinning=100*, *processes=1*, *seed=None*)
Sample valid flux distributions from a cobra model.

The function samples valid flux distributions from a cobra model. Currently we support two methods:

1. **'optgp' (default) which uses the OptGPSampler that supports parallel** sampling [1]_. Requires large numbers of samples to be performant (n < 1000). For smaller samples 'achr' might be better suited.

or

2. 'achr' which uses artificial centering hit-and-run. This is a single process method with good convergence [2]_.

Parameters

- **model** (*cobra.Model*) – The model from which to sample flux distributions.
- **n** (*int*) – The number of samples to obtain. When using 'optgp' this must be a multiple of *processes*, otherwise a larger number of samples will be returned.
- **method** (*str, optional*) – The sampling algorithm to use.
- **thinning** (*int, optional*) – The thinning factor of the generated sampling chain. A thinning of 10 means samples are returned every 10 steps. Defaults to 100 which in benchmarks gives approximately uncorrelated samples. If set to one will return all iterates.
- **processes** (*int, optional*) – Only used for 'optgp'. The number of processes used to generate samples.
- **seed** (*int > 0, optional*) – The random number seed to be used. Initialized to current time stamp if None.

Returns The generated flux samples. Each row corresponds to a sample of the fluxes and the columns are the reactions.

Return type pandas.DataFrame

Notes

The samplers have a correction method to ensure equality feasibility for long-running chains, however this will only work for homogeneous models, meaning models with no non-zero fixed variables or constraints (right-hand side of the equalities are zero).

References

cobra.test

Subpackages

cobra.test.test_core

Subpackages

cobra.test.test_core.test_summary

Submodules

cobra.test.test_core.test_summary.test_metabolite_summary

Test functionalities of MetaboliteSummary.

Module Contents

Functions

test_metabolite_summary_to_table_previ(model, opt_solver, met)	Test metabolite summary._to_table() of previous solution.
test_metabolite_summary_to_frame_previ(model, opt_solver, met)	Test metabolite summary.to_frame() of previous solution.
test_metabolite_summary_to_table(model, opt_solver, met, names)	Test metabolite summary._to_table().
test_metabolite_summary_to_frame(model, opt_solver, met, names)	Test metabolite summary.to_frame().
test_metabolite_summary_to_table_with_fva(model, opt_solver, fraction, met)	Test metabolite summary._to_table() (using FVA).
test_metabolite_summary_to_frame_with_fva(model, opt_solver, fraction, met)	Test metabolite summary.to_frame() (using FVA).

cobra.test.test_core.test_summary.test_metabolite_summary.**test_metabolite_summary_to_tab**

Test metabolite summary._to_table() of previous solution.

cobra.test.test_core.test_summary.test_metabolite_summary.**test_metabolite_summary_to_fra**

Test metabolite summary.to_frame() of previous solution.

cobra.test.test_core.test_summary.test_metabolite_summary.**test_metabolite_summary_to_tab**

Test metabolite summary._to_table().

cobra.test.test_core.test_summary.test_metabolite_summary.**test_metabolite_summary_to_fra**

Test metabolite summary.to_frame().

cobra.test.test_core.test_summary.test_metabolite_summary.**test_metabolite_summary_to_tab**

Test metabolite summary._to_table() (using FVA).

cobra.test.test_core.test_summary.test_metabolite_summary.**test_metabolite_summary_to_fra**

Test metabolite summary.to_frame() (using FVA).

cobra.test.test_core.test_summary.test_model_summary

Test functionalities of ModelSummary.

Module Contents

Functions

test_model_summary_to_table_previous_solution(model, opt_solver, names)	Test Summary._to_table() of previous solution.
test_model_summary_to_frame_previous_solution(model, opt_solver, names)	Test Summary.to_frame() of previous solution.
test_model_summary_to_table(model, opt_solver, names)	Test model.summary()._to_table().
test_model_summary_to_frame(model, opt_solver, names)	Test model.summary().to_frame().
test_model_summary_to_table_with_fva(model, opt_solver, fraction)	Test model summary._to_table() (using FVA).
test_model_summary_to_frame_with_fva(model, opt_solver, fraction)	Test model summary.to_frame() (using FVA).

cobra.test.test_core.test_summary.test_model_summary.**test_model_summary_to_table_previou**

Test Summary._to_table() of previous solution.

cobra.test.test_core.test_summary.test_model_summary.**test_model_summary_to_frame_previou**

Test Summary.to_frame() of previous solution.

cobra.test.test_core.test_summary.test_model_summary.**test_model_summary_to_table**(*model*, *opt_solver*, *names*)

Test model.summary()._to_table().

cobra.test.test_core.test_summary.test_model_summary.**test_model_summary_to_frame**(*model*, *opt_solver*, *names*)

Test model.summary().to_frame().

cobra.test.test_core.test_summary.test_model_summary.**test_model_summary_to_table_with_fv**

Test model summary._to_table() (using FVA).

cobra.test.test_core.test_summary.test_model_summary.**test_model_summary_to_frame_with_fv**

Test model summary.to_frame() (using FVA).

cobra.test.test_core.test_summary.test_reaction_summary

Test functionalities of ReactionSummary.

Module Contents

Functions

test_reaction_summary_to_table(model, rxn, names)	Test reaction summary._to_table().
test_reaction_summary_to_frame(model, rxn, names)	Test reaction summary.to_frame().

cobra.test.test_core.test_summary.test_reaction_summary.**test_reaction_summary_to_table**(*m*
r
n

Test reaction summary._to_table().

cobra.test.test_core.test_summary.test_reaction_summary.**test_reaction_summary_to_frame**(*m*
r
n

Test reaction summary.to_frame().

Package Contents

Functions

captured_output()	A context manager to test the IO summary methods.
check_line(output, expected_entries, pattern=re.compile('\s'))	Ensure each expected entry is in the output.
check_in_line(output, expected_entries, pattern=re.compile('\s'))	Ensure each expected entry is contained in the output.

cobra.test.test_core.test_summary.**captured_output**()

A context manager to test the IO summary methods.

cobra.test.test_core.test_summary.**check_line**(*output*, *expected_entries*, *pattern=re.compile('\s')*)

Ensure each expected entry is in the output.

cobra.test.test_core.test_summary.**check_in_line**(*output*, *expected_entries*, *pattern=re.compile('\s')*)

Ensure each expected entry is contained in the output.

Submodules

cobra.test.test_core.conftest

Module level fixtures

Module Contents

Functions

solved_model(request, model)	

cobra.test.test_core.conftest.**solver_trials**

cobra.test.test_core.conftest.**solved_model**(*request*, *model*)

cobra.test.test_core.test_configuration

Test functions of configuration.py

Module Contents

Functions

test_default_bounds()	Verify the default bounds.
test_bounds()	Test changing bounds.
test_solver()	Test assignment of different solvers.
test_default_tolerance(model)	Verify the default solver tolerance.
test_toy_model_tolerance_with_different_default()	Verify that different default tolerance is respected by Model.
test_tolerance_assignment(model)	Test assignment of solver tolerance.

cobra.test.test_core.test_configuration.**test_default_bounds**()
Verify the default bounds.

cobra.test.test_core.test_configuration.**test_bounds**()
Test changing bounds.

cobra.test.test_core.test_configuration.**test_solver**()
Test assignment of different solvers.

cobra.test.test_core.test_configuration.**test_default_tolerance**(*model*)
Verify the default solver tolerance.

cobra.test.test_core.test_configuration.**test_toy_model_tolerance_with_different_default**
Verify that different default tolerance is respected by Model.

cobra.test.test_core.test_configuration.**test_tolerance_assignment**(*model*)
Test assignment of solver tolerance.

cobra.test.test_core.test_core_reaction

Test functions of reaction.py

Module Contents

Functions

test_gpr()	
test_gpr_modification(model)	
test_gene_knock_out(model)	
test_str()	
test_str_from_model(model)	
test_add_metabolite_from_solved_model(solved_model)	
test_add_metabolite_benchmark(model, benchmark, solver)	
test_add_metabolite(model)	
test_subtract_metabolite_benchmark(model, benchmark, solver)	
test_subtract_metabolite(model, solver)	
test_mass_balance(model)	
test_build_from_string(model)	
test_bounds_setter(model)	
test_copy(model)	
test_iadd(model)	
test_add(model)	
test_radd(model)	
test_mul(model)	
test_sub(model)	
test_removal_from_model_retains_bounds(model)	
test_set_bounds_scenario_1(model)	
test_set_bounds_scenario_3(model)	
test_set_bounds_scenario_4(model)	
test_set_upper_before_lower_bound_to_0(model)	
test_set_bounds_scenario_2(model)	
test_change_bounds(model)	
test_make_irreversible(model)	
test_make_reversible(model)	
test_make_irreversible_irreversible_to_the_other_side(model)	
test_make_lhs_irreversible_reversible(model)	
test_model_less_reaction(model)	
test_knockout(model)	
test_reaction_without_model()	
test_weird_left_to_right_reaction_issue(tiny_toy_model)	
test_one_left_to_right_reaction_set_positive_ub(tiny_toy_model)	
test_irrev_reaction_set_negative_lb(model)	
test_twist_irrev_right_to_left_reaction_to_left_to_right(model)	
test_set_lb_higher_than_ub_sets_ub_to_new_lb(model)	
test_set_ub_lower_than_lb_sets_lb_to_new_ub(model)	
test_add_metabolites_combine_true(model)	
test_add_metabolites_combine_false(model)	
test_reaction_imul(model)	
test_remove_from_model(model)	

Continued on next page

Table 65 – continued from previous page

test_change_id_is_reflected_in_solver(model)	
*test_repr_html_*(model)	

cobra.test.test_core.test_core_reaction.**config**

cobra.test.test_core.test_core_reaction.**stable_optlang = ['glpk', 'cplex', 'gurobi']**

cobra.test.test_core.test_core_reaction.**test_gpr**()

cobra.test.test_core.test_core_reaction.**test_gpr_modification**(*model*)

cobra.test.test_core.test_core_reaction.**test_gene_knock_out**(*model*)

cobra.test.test_core.test_core_reaction.**test_str**()

cobra.test.test_core.test_core_reaction.**test_str_from_model**(*model*)

cobra.test.test_core.test_core_reaction.**test_add_metabolite_from_solved_model**(*solved_model*)

cobra.test.test_core.test_core_reaction.**test_add_metabolite_benchmark**(*model*, *benchmark*, *solver*)

cobra.test.test_core.test_core_reaction.**test_add_metabolite**(*model*)

cobra.test.test_core.test_core_reaction.**test_subtract_metabolite_benchmark**(*model*, *benchmark*, *solver*)

cobra.test.test_core.test_core_reaction.**test_subtract_metabolite**(*model*, *solver*)

cobra.test.test_core.test_core_reaction.**test_mass_balance**(*model*)

cobra.test.test_core.test_core_reaction.**test_build_from_string**(*model*)

cobra.test.test_core.test_core_reaction.**test_bounds_setter**(*model*)

cobra.test.test_core.test_core_reaction.**test_copy**(*model*)

cobra.test.test_core.test_core_reaction.**test_iadd**(*model*)

cobra.test.test_core.test_core_reaction.**test_add**(*model*)

cobra.test.test_core.test_core_reaction.**test_radd**(*model*)

cobra.test.test_core.test_core_reaction.**test_mul**(*model*)

cobra.test.test_core.test_core_reaction.**test_sub**(*model*)

cobra.test.test_core.test_core_reaction.**test_removal_from_model_retains_bounds**(*model*)

cobra.test.test_core.test_core_reaction.**test_set_bounds_scenario_1**(*model*)

cobra.test.test_core.test_core_reaction.**test_set_bounds_scenario_3**(*model*)

cobra.test.test_core.test_core_reaction.**test_set_bounds_scenario_4**(*model*)

cobra.test.test_core.test_core_reaction.**test_set_upper_before_lower_bound_to_0**(*model*)

cobra.test.test_core.test_core_reaction.**test_set_bounds_scenario_2**(*model*)

cobra.test.test_core.test_core_reaction.**test_change_bounds**(*model*)

cobra.test.test_core.test_core_reaction.**test_make_irreversible**(*model*)

cobra.test.test_core.test_core_reaction.**test_make_reversible**(*model*)

cobra.test.test_core.test_core_reaction.**test_make_irreversible_irreversible_to_the_other**

cobra.test.test_core.test_core_reaction.**test_make_lhs_irreversible_reversible**(*model*)

cobra.test.test_core.test_core_reaction.**test_model_less_reaction**(*model*)

cobra.test.test_core.test_core_reaction.**test_knockout**(*model*)

cobra.test.test_core.test_core_reaction.**test_reaction_without_model**()

cobra.test.test_core.test_core_reaction.**test_weird_left_to_right_reaction_issue**(*tiny_toy_m*

cobra.test.test_core.test_core_reaction.**test_one_left_to_right_reaction_set_positive_ub**

cobra.test.test_core.test_core_reaction.**test_irrev_reaction_set_negative_lb**(*model*)

cobra.test.test_core.test_core_reaction.**test_twist_irrev_right_to_left_reaction_to_left_**

cobra.test.test_core.test_core_reaction.**test_set_lb_higher_than_ub_sets_ub_to_new_lb**(*mod*

cobra.test.test_core.test_core_reaction.**test_set_ub_lower_than_lb_sets_lb_to_new_ub**(*mode*

cobra.test.test_core.test_core_reaction.**test_add_metabolites_combine_true**(*model*)

cobra.test.test_core.test_core_reaction.**test_add_metabolites_combine_false**(*model*)

cobra.test.test_core.test_core_reaction.**test_reaction_imul**(*model*)

cobra.test.test_core.test_core_reaction.**test_remove_from_model**(*model*)

cobra.test.test_core.test_core_reaction.**test_change_id_is_reflected_in_solver**(*model*)

cobra.test.test_core.test_core_reaction.**test_repr_html_**(*model*)

cobra.test.test_core.test_dictlist

Test functions of dictlist.py

Module Contents

Functions

dict_list()	
test_contains(dict_list)	
test_index(dict_list)	
test_independent()	
test_get_by_any(dict_list)	
test_append(dict_list)	
test_insert(dict_list)	
test_extend(dict_list)	
test_iadd(dict_list)	
test_add(dict_list)	
test_sub(dict_list)	
test_isub(dict_list)	
test_init_copy(dict_list)	
test_slice(dict_list)	
test_copy(dict_list)	
test_deepcopy(dict_list)	
test_pickle(dict_list)	
test_query(dict_list)	
test_removal()	
test_set()	
test_sort_and_reverse()	
test_dir(dict_list)	

Continued on next page

Table 66 – continued from previous page

test_union(dict_list)

cobra.test.test_core.test_dictlist.**dict_list**()

cobra.test.test_core.test_dictlist.**test_contains**(*dict_list*)

cobra.test.test_core.test_dictlist.**test_index**(*dict_list*)

cobra.test.test_core.test_dictlist.**test_independent**()

cobra.test.test_core.test_dictlist.**test_get_by_any**(*dict_list*)

cobra.test.test_core.test_dictlist.**test_append**(*dict_list*)

cobra.test.test_core.test_dictlist.**test_insert**(*dict_list*)

cobra.test.test_core.test_dictlist.**test_extend**(*dict_list*)

cobra.test.test_core.test_dictlist.**test_iadd**(*dict_list*)

cobra.test.test_core.test_dictlist.**test_add**(*dict_list*)

cobra.test.test_core.test_dictlist.**test_sub**(*dict_list*)

cobra.test.test_core.test_dictlist.**test_isub**(*dict_list*)

cobra.test.test_core.test_dictlist.**test_init_copy**(*dict_list*)

cobra.test.test_core.test_dictlist.**test_slice**(*dict_list*)

cobra.test.test_core.test_dictlist.**test_copy**(*dict_list*)

cobra.test.test_core.test_dictlist.**test_deepcopy**(*dict_list*)

cobra.test.test_core.test_dictlist.**test_pickle**(*dict_list*)

cobra.test.test_core.test_dictlist.**test_query**(*dict_list*)

cobra.test.test_core.test_dictlist.**test_removal**()

cobra.test.test_core.test_dictlist.**test_set**()

cobra.test.test_core.test_dictlist.**test_sort_and_reverse**()

cobra.test.test_core.test_dictlist.**test_dir**(*dict_list*)

cobra.test.test_core.test_dictlist.**test_union**(*dict_list*)

cobra.test.test_core.test_gene

Test functions of gene.py

Module Contents

Functions

*test_repr_html_*(model)

cobra.test.test_core.test_gene.**test_repr_html_**(*model*)

cobra.test.test_core.test_group

Test functions of model.py

Module Contents

Functions

test_group_add_elements(model)	
test_group_kind()	

cobra.test.test_core.test_group.**test_group_add_elements**(*model*)

cobra.test.test_core.test_group.**test_group_kind**()

cobra.test.test_core.test_metabolite

Test functions of metabolite.py

Module Contents

Functions

test_metabolite_formula()	
test_formula_element_setting(model)	
test_set_id(solved_model)	
test_remove_from_model(solved_model)	
*test_repr_html_*(model)	

cobra.test.test_core.test_metabolite.**test_metabolite_formula**()

cobra.test.test_core.test_metabolite.**test_formula_element_setting**(*model*)

cobra.test.test_core.test_metabolite.**test_set_id**(*solved_model*)

cobra.test.test_core.test_metabolite.**test_remove_from_model**(*solved_model*)

cobra.test.test_core.test_metabolite.**test_repr_html_**(*model*)

cobra.test.test_core.test_model

Test functions of model.py

Module Contents

Functions

same_ex(ex1, ex2)	Compare to expressions for mathematical equality.
test_add_remove_reaction_benchmark(model, benchmark, solver)	
test_add_metabolite(model)	
test_remove_metabolite_subtractive(model)	
test_remove_metabolite_destructive(model)	
test_compartments(model)	
test_add_reaction(model)	
test_add_reaction_context(model)	
test_add_reaction_from_other_model(model)	
test_model_remove_reaction(model)	
test_reaction_remove(model)	
test_reaction_delete(model)	
test_remove_gene(model)	
test_group_model_reaction_association(model)	
test_group_members_add_to_model(model)	
test_group_loss_of_elements(model)	
test_exchange_reactions(model)	
test_add_boundary(model, metabolites, reaction_type, prefix)	
test_add_boundary_context(model, metabolites, reaction_type, prefix)	
test_add_existing_boundary(model, metabolites, reaction_type)	
test_copy_benchmark(model, solver, benchmark)	
test_copy_benchmark_large_model(large_model, solver, benchmark)	
test_copy(model)	
test_copy_with_groups(model)	
test_deepcopy_benchmark(model, benchmark)	
test_deepcopy(model)	
test_add_reaction_orphans(model)	
test_merge_models(model, tiny_toy_model)	
test_change_objective_benchmark(model, benchmark, solver)	
test_get_objective_direction(model)	
test_set_objective_direction(model)	
test_slim_optimize(model)	
test_optimize(model, solver)	
test_change_objective(model)	
test_problem_properties(model)	
test_solution_data_frame(model)	
test_context_manager(model)	

Continued on next page

Table 70 – continued from previous page

test_objective_coefficient_reflects_changed_objective(model)	
test_change_objective_through_objective_coefficient(model)	
test_transfer_objective(model)	
test_model_from_other_model(model)	
test_add_reactions(model)	
test_add_reactions_single_existing(model)	
test_add_reactions_duplicate(model)	
test_add_cobra_reaction(model)	
test_all_objects_point_to_all_other_correct_objects(model)	
test_objects_point_to_correct_other_after_copy(model)	
test_remove_reactions(model)	
test_objective(model)	
test_change_objective(model)	
test_set_reaction_objective(model)	
test_set_reaction_objective_str(model)	
test_invalid_objective_raises(model)	
test_solver_change(model)	
test_no_change_for_same_solver(model)	
test_invalid_solver_change_raises(model)	
test_change_solver_to_cplex_and_check_copy_works(model)	
test_copy_preserves_existing_solution(solved_model)	
*test_repr_html_*(model)	

cobra.test.test_core.test_model.**stable_optlang = ['glpk', 'cplex', 'gurobi']**

cobra.test.test_core.test_model.**optlang_solvers**

cobra.test.test_core.test_model.**same_ex**(*ex1*, *ex2*)
Compare to expressions for mathematical equality.

cobra.test.test_core.test_model.**test_add_remove_reaction_benchmark**(*model*, *benchmark*, *solver*)

cobra.test.test_core.test_model.**test_add_metabolite**(*model*)

cobra.test.test_core.test_model.**test_remove_metabolite_subtractive**(*model*)

cobra.test.test_core.test_model.**test_remove_metabolite_destructive**(*model*)

cobra.test.test_core.test_model.**test_compartments**(*model*)

cobra.test.test_core.test_model.**test_add_reaction**(*model*)

cobra.test.test_core.test_model.**test_add_reaction_context**(*model*)

cobra.test.test_core.test_model.**test_add_reaction_from_other_model**(*model*)

cobra.test.test_core.test_model.**test_model_remove_reaction**(*model*)

cobra.test.test_core.test_model.**test_reaction_remove**(*model*)

cobra.test.test_core.test_model.**test_reaction_delete**(*model*)

cobra.test.test_core.test_model.**test_remove_gene**(*model*)

cobra.test.test_core.test_model.**test_group_model_reaction_association**(*model*)

cobra.test.test_core.test_model.**test_group_members_add_to_model**(*model*)

cobra.test.test_core.test_model.**test_group_loss_of_elements**(*model*)

cobra.test.test_core.test_model.**test_exchange_reactions**(*model*)

cobra.test.test_core.test_model.**test_add_boundary**(*model*, *metabolites*, *reaction_type*, *prefix*)

cobra.test.test_core.test_model.**test_add_boundary_context**(*model*, *metabolites*, *reaction_type*, *prefix*)

cobra.test.test_core.test_model.**test_add_existing_boundary**(*model*, *metabolites*, *reaction_type*)

cobra.test.test_core.test_model.**test_copy_benchmark**(*model*, *solver*, *benchmark*)

cobra.test.test_core.test_model.**test_copy_benchmark_large_model**(*large_model*, *solver*, *benchmark*)

cobra.test.test_core.test_model.**test_copy**(*model*)

cobra.test.test_core.test_model.**test_copy_with_groups**(*model*)

cobra.test.test_core.test_model.**test_deepcopy_benchmark**(*model*, *benchmark*)

cobra.test.test_core.test_model.**test_deepcopy**(*model*)

cobra.test.test_core.test_model.**test_add_reaction_orphans**(*model*)

cobra.test.test_core.test_model.**test_merge_models**(*model*, *tiny_toy_model*)

cobra.test.test_core.test_model.**test_change_objective_benchmark**(*model*, *benchmark*, *solver*)

cobra.test.test_core.test_model.**test_get_objective_direction**(*model*)

cobra.test.test_core.test_model.**test_set_objective_direction**(*model*)

cobra.test.test_core.test_model.**test_slim_optimize**(*model*)

cobra.test.test_core.test_model.**test_optimize**(*model*, *solver*)

cobra.test.test_core.test_model.**test_change_objective**(*model*)

cobra.test.test_core.test_model.**test_problem_properties**(*model*)

cobra.test.test_core.test_model.**test_solution_data_frame**(*model*)

cobra.test.test_core.test_model.**test_context_manager**(*model*)

cobra.test.test_core.test_model.**test_objective_coefficient_reflects_changed_objective**(*mo*

cobra.test.test_core.test_model.**test_change_objective_through_objective_coefficient**(*mode*

cobra.test.test_core.test_model.**test_transfer_objective**(*model*)

cobra.test.test_core.test_model.**test_model_from_other_model**(*model*)

cobra.test.test_core.test_model.**test_add_reactions**(*model*)

cobra.test.test_core.test_model.**test_add_reactions_single_existing**(*model*)

cobra.test.test_core.test_model.**test_add_reactions_duplicate**(*model*)

cobra.test.test_core.test_model.**test_add_cobra_reaction**(*model*)

cobra.test.test_core.test_model.**test_all_objects_point_to_all_other_correct_objects**(*mode*

cobra.test.test_core.test_model.**test_objects_point_to_correct_other_after_copy**(*model*)

cobra.test.test_core.test_model.**test_remove_reactions**(*model*)

cobra.test.test_core.test_model.**test_objective**(*model*)

cobra.test.test_core.test_model.**test_change_objective**(*model*)

cobra.test.test_core.test_model.**test_set_reaction_objective**(*model*)

cobra.test.test_core.test_model.**test_set_reaction_objective_str**(*model*)

cobra.test.test_core.test_model.**test_invalid_objective_raises**(*model*)

cobra.test.test_core.test_model.**test_solver_change**(*model*)

cobra.test.test_core.test_model.**test_no_change_for_same_solver**(*model*)

cobra.test.test_core.test_model.**test_invalid_solver_change_raises**(*model*)

cobra.test.test_core.test_model.**test_change_solver_to_cplex_and_check_copy_works**(*model*)

cobra.test.test_core.test_model.**test_copy_preserves_existing_solution**(*solved_model*)

cobra.test.test_core.test_model.**test_repr_html_**(*model*)

cobra.test.test_core.test_solution

Test functions of solution.py

Module Contents

Functions

test_solution_contains_only_reaction_specific_values(solved_model)	

cobra.test.test_core.test_solution.**test_solution_contains_only_reaction_specific_values**

cobra.test.test_io

Submodules

cobra.test.test_io.conftest

Contains module level fixtures and utility functions.

Module Contents

Functions

mini_model(data_directory)	Fixture for mini model.
compare_models(model_1, model_2)	Compare two models (only for testing purposes).

cobra.test.test_io.conftest.**mini_model**(*data_directory*)
Fixture for mini model.

cobra.test.test_io.conftest.**compare_models**(*model_1*, *model_2*)
Compare two models (only for testing purposes).

cobra.test.test_io.test_annotation

Module Contents

Functions

_check_sbml_annotations(model)	Checks the annotations from the annotation.xml.
test_read_sbml_annotations(data_directory)	Test reading and writing annotations.
test_read_write_sbml_annotations(data_directory, tmp_path)	Test reading and writing annotations.

cobra.test.test_io.test_annotation.**_check_sbml_annotations**(*model*)
Checks the annotations from the annotation.xml.

cobra.test.test_io.test_annotation.**test_read_sbml_annotations**(*data_directory*)
Test reading and writing annotations.

cobra.test.test_io.test_annotation.**test_read_write_sbml_annotations**(*data_directory*, *tmp_path*)
Test reading and writing annotations.

cobra.test.test_io.test_io_order

Module Contents

Functions

tmp_path(tmpdir_factory)	
minimized_shuffle(small_model)	
minimized_sorted(minimized_shuffle)	
minimized_reverse(minimized_shuffle)	
template(request, minimized_shuffle, minimized_reverse, minimized_sorted)	
attribute(request)	
get_ids(iterable)	
test_io_order(attribute, read, write, ext, template, tmp_path)	

cobra.test.test_io.test_io_order.**LOGGER**

cobra.test.test_io.test_io_order.**tmp_path**(*tmpdir_factory*)

cobra.test.test_io.test_io_order.**minimized_shuffle**(*small_model*)

cobra.test.test_io.test_io_order.**minimized_sorted**(*minimized_shuffle*)

cobra.test.test_io.test_io_order.**minimized_reverse**(*minimized_shuffle*)

cobra.test.test_io.test_io_order.**template**(*request*, *minimized_shuffle*, *minimized_reverse*, *minimized_sorted*)

cobra.test.test_io.test_io_order.**attribute**(*request*)

cobra.test.test_io.test_io_order.**get_ids**(*iterable*)

cobra.test.test_io.test_io_order.**test_io_order**(*attribute*, *read*, *write*, *ext*, *template*, *tmp_path*)

cobra.test.test_io.test_json

Test functionalities of json.py

Module Contents

Functions

test_validate_json(data_directory)	Validate file according to JSON-schema.
test_load_json_model(data_directory, mini_model)	Test the reading of JSON model.
test_save_json_model(tmpdir, mini_model)	Test the writing of JSON model.

cobra.test.test_io.test_json.**test_validate_json**(*data_directory*)
Validate file according to JSON-schema.

cobra.test.test_io.test_json.**test_load_json_model**(*data_directory*, *mini_model*)
Test the reading of JSON model.

cobra.test.test_io.test_json.**test_save_json_model**(*tmpdir*, *mini_model*)
Test the writing of JSON model.

cobra.test.test_io.test_mat

Test functionalities provided by mat.py

Module Contents

Functions

raven_model(data_directory)	Fixture for RAVEN model.
test_load_matlab_model(data_directory, mini_model, raven_model)	Test the reading of MAT model.
test_save_matlab_model(tmpdir, mini_model, raven_model)	Test the writing of MAT model.

cobra.test.test_io.test_mat.**scipy**

cobra.test.test_io.test_mat.**raven_model**(*data_directory*)
Fixture for RAVEN model.

cobra.test.test_io.test_mat.**test_load_matlab_model**(*data_directory*, *mini_model*, *raven_model*)
Test the reading of MAT model.

cobra.test.test_io.test_mat.**test_save_matlab_model**(*tmpdir*, *mini_model*, *raven_model*)
Test the writing of MAT model.

cobra.test.test_io.test_pickle

Test data storage and recovery using pickle.

Module Contents

Functions

test_read_pickle(data_directory, mini_model, load_function)	Test the reading of model from pickle.
test_write_pickle(tmpdir, mini_model, dump_function)	Test the writing of model to pickle.

cobra.test.test_io.test_pickle.**cload**

cobra.test.test_io.test_pickle.**test_read_pickle**(*data_directory*, *mini_model*, *load_function*)

Test the reading of model from pickle.

cobra.test.test_io.test_pickle.**test_write_pickle**(*tmpdir*, *mini_model*, *dump_function*)

Test the writing of model to pickle.

cobra.test.test_io.test_sbml

Testing SBML functionality based on libsbml.

Module Contents

Classes

TestCobraIO	Tests the read and write functions.

Functions

test_validate(trial, data_directory)	Test validation function.
io_trial(request, data_directory)	
test_filehandle(data_directory, tmp_path)	Test reading and writing to file handle.
test_from_sbml_string(data_directory)	Test reading from SBML string.
test_model_history(tmp_path)	Testing reading and writing of ModelHistory.
test_groups(data_directory, tmp_path)	Testing reading and writing of groups
test_missing_flux_bounds1(data_directory)	
test_missing_flux_bounds2(data_directory)	
test_validate(trial, data_directory)	Test validation function.
test_validation_warnings(data_directory)	Test the validation warnings.
test_infinity_bounds(data_directory, tmp_path)	Test infinity bound example.
test_boundary_conditions(data_directory)	Test infinity bound example.
test_gprs(data_directory, tmp_path)	Test that GPRs are written and read correctly
test_identifiers_annotation()	

Continued on next page

Table 79 – continued from previous page

test_smbl_with_notes(data_directory, tmp_path)	Test that NOTES in the RECON 2.2 style are written and read correctly

cobra.test.test_io.test_sbml.**config**

cobra.test.test_io.test_sbml.**jsonschema**

cobra.test.test_io.test_sbml.**IOTrial**

cobra.test.test_io.test_sbml.**trials**

cobra.test.test_io.test_sbml.**trial_names**

cobra.test.test_io.test_sbml.**test_validate**(*trial*, *data_directory*)
Test validation function.

class cobra.test.test_io.test_sbml.**TestCobraIO**
Tests the read and write functions.

classmethod compare_models(*cls*, *name*, *model1*, *model2*)

classmethod extra_comparisons(*cls*, *name*, *model1*, *model2*)

test_read_1(*self*, *io_trial*)

test_read_2(*self*, *io_trial*)

test_write_1(*self*, *io_trial*)

test_write_2(*self*, *io_trial*)

cobra.test.test_io.test_sbml.**io_trial**(*request*, *data_directory*)

cobra.test.test_io.test_sbml.**test_filehandle**(*data_directory*, *tmp_path*)
Test reading and writing to file handle.

cobra.test.test_io.test_sbml.**test_from_sbml_string**(*data_directory*)
Test reading from SBML string.

cobra.test.test_io.test_sbml.**test_model_history**(*tmp_path*)
Testing reading and writing of ModelHistory.

cobra.test.test_io.test_sbml.**test_groups**(*data_directory*, *tmp_path*)
Testing reading and writing of groups

cobra.test.test_io.test_sbml.**test_missing_flux_bounds1**(*data_directory*)

cobra.test.test_io.test_sbml.**test_missing_flux_bounds2**(*data_directory*)

cobra.test.test_io.test_sbml.**test_validate**(*data_directory*)
Test the validation code.

cobra.test.test_io.test_sbml.**test_validation_warnings**(*data_directory*)
Test the validation warnings.

cobra.test.test_io.test_sbml.**test_infinity_bounds**(*data_directory*, *tmp_path*)
Test infinity bound example.

cobra.test.test_io.test_sbml.**test_boundary_conditions**(*data_directory*)
Test infinity bound example.

cobra.test.test_io.test_sbml.**test_gprs**(*data_directory*, *tmp_path*)
Test that GPRs are written and read correctly

cobra.test.test_io.test_sbml.**test_identifiers_annotation**()

cobra.test.test_io.test_sbml.**test_smbl_with_notes**(*data_directory*, *tmp_path*)
Test that NOTES in the RECON 2.2 style are written and read correctly

cobra.test.test_io.test_yaml

Test functionalities provided by yaml.py

Module Contents

Functions

test_load_yaml_model(data_directory, mini_model)	Test the reading of YAML model.
test_save_yaml_model(tmpdir, mini_model)	

cobra.test.test_io.test_yaml.**test_load_yaml_model**(*data_directory*, *mini_model*)
Test the reading of YAML model.

cobra.test.test_io.test_yaml.**test_save_yaml_model**(*tmpdir*, *mini_model*)

Submodules

cobra.test.conftest

Module Contents

Functions

pytest_addoption(parser)
data_directory()
empty_once()
empty_model(empty_once)
small_model()
model(small_model)
large_once()
large_model(large_once)
medium_model()
salmonella(medium_model)
solved_model(data_directory)
tiny_toy_model()
fva_results(data_directory)
pfba_fva_results(data_directory)
opt_solver(request)
metabolites(model, request)

cobra.test.conftest.**pytest_addoption**(*parser*)

cobra.test.conftest.**data_directory**()

cobra.test.conftest.**empty_once**()

cobra.test.conftest.**empty_model**(*empty_once*)

cobra.test.conftest.**small_model**()

cobra.test.conftest.**model**(*small_model*)

cobra.test.conftest.**large_once**()

cobra.test.conftest.**large_model**(*large_once*)

cobra.test.conftest.**medium_model**()

cobra.test.conftest.**salmonella**(*medium_model*)

cobra.test.conftest.**solved_model**(*data_directory*)

cobra.test.conftest.**tiny_toy_model**()

cobra.test.conftest.**fva_results**(*data_directory*)

cobra.test.conftest.**pfba_fva_results**(*data_directory*)

cobra.test.conftest.**stable_optlang = ['glpk', 'cplex', 'gurobi']**

cobra.test.conftest.**all_solvers**

cobra.test.conftest.**opt_solver**(*request*)

cobra.test.conftest.**metabolites**(*model*, *request*)

cobra.test.test_manipulation

Module Contents

Classes

TestManipulation	Test functions in cobra.manipulation

class cobra.test.test_manipulation.**TestManipulation**

Test functions in cobra.manipulation

test_escape_ids(*self*, *model*)

test_rename_gene(*self*, *model*)

test_gene_knockout_computation(*self*, *salmonella*)

test_remove_genes(*self*)

test_sbo_annotation(*self*, *model*)

test_validate_formula_compartment(*self*, *model*)

test_validate_mass_balance(*self*, *model*)

test_prune_unused_mets_output_type(*self*, *model*)

test_prune_unused_mets_functionality(*self*, *model*)

test_prune_unused_rxns_output_type(*self*, *model*)

test_prune_unused_rxns_functionality(*self*, *model*)

cobra.test.test_medium

Module Contents

Classes

TestModelMedium
TestTypeDetection
TestMinimalMedia
TestErrorsAndExceptions

class cobra.test.test_medium.**TestModelMedium**

test_model_medium(*self*, *model*)

class cobra.test.test_medium.**TestTypeDetection**

test_external_compartment(*self*, *model*)

test_multi_external(*self*, *model*)

test_exchange(*self*, *model*)

test_demand(*self*, *model*)

test_sink(*self*, *model*)

test_sbo_terms(*self*, *model*)

class cobra.test.test_medium.**TestMinimalMedia**

test_medium_linear(*self*, *model*)

test_medium_mip(*self*, *model*)

test_medium_alternative_mip(*self*, *model*)

test_benchmark_medium_linear(*self*, *model*, *benchmark*)

test_benchmark_medium_mip(*self*, *model*, *benchmark*)

test_medium_exports(*self*, *model*)

test_open_exchanges(*self*, *model*)

class cobra.test.test_medium.**TestErrorsAndExceptions**

test_no_boundary_reactions(*self*, *empty_model*)

test_no_names_or_boundary_reactions(*self*, *empty_model*)

test_bad_exchange(*self*, *model*)

Package Contents

Functions

read_sbml_model(filename, number=float, f_replace=F_REPLACE, **kwargs)	Reads SBML model from given filename.
create_test_model(model_name='salmonella')	Returns a cobra model for testing
test_all(args=None)	alias for running all unit-tests on installed cobra

cobra.test.**read_sbml_model** (*filename*, *number=float*, *f_replace=F_REPLACE*, ***kwargs*)

Reads SBML model from given filename.

If the given filename ends with the suffix ''.gz" (for example, ''myfile.xml.gz'),' the file is assumed to be compressed in gzip format and will be automatically decompressed upon reading. Similarly, if the given filename ends with ''.zip" or ''.bz2',' the file is assumed to be compressed in zip or bzip2 format (respectively). Files whose names lack these suffixes will be read uncompressed. Note that if the file is in zip format but the archive contains more than one file, only the first file in the archive will be read and the rest ignored.

To read a gzip/zip file, libSBML needs to be configured and linked with the zlib library at compile time. It also needs to be linked with the bzip2 library to read files in bzip2 format. (Both of these are the default configurations for libSBML.)

This function supports SBML with FBC-v1 and FBC-v2. FBC-v1 models are converted to FBC-v2 models before reading.

The parser tries to fall back to information in notes dictionaries if information is not available in the FBC packages, e.g., CHARGE, FORMULA on species, or GENE_ASSOCIATION, SUBSYSTEM on reactions.

Parameters

- **filename** (*path to SBML file, or SBML string, or SBML file handle*) – SBML which is read into cobra model
- **number** (*data type of stoichiometry: {float, int}*) – In which data type should the stoichiometry be parsed.
- **f_replace** (*dict of replacement functions for id replacement*) – Dictionary of replacement functions for gene, specie, and reaction. By default the following id changes are performed on import: clip G_ from genes, clip M_ from species, clip R_ from reactions If no replacements should be performed, set f_replace={}, None

Returns

Return type *cobra.core.Model*

Notes

Provided file handles cannot be opened in binary mode, i.e., use

with open(path, "r" as f): read_sbml_model(f)

File handles to compressed files are not supported yet.

cobra.test.**pytest**

cobra.test.**cobra_directory**

cobra.test.**cobra_location**

cobra.test.**data_dir**

cobra.test.**create_test_model**(*model_name='salmonella'*)
Returns a cobra model for testing

model_name: str One of 'ecoli', 'textbook', or 'salmonella', or the path to a pickled cobra.Model

cobra.test.**test_all**(*args=None*)
alias for running all unit-tests on installed cobra

cobra.util

Submodules

cobra.util.array

Module Contents

Functions

create_stoichiometric_matrix(model, array_type='dense', dtype=None)	Return a stoichiometric array representation of the given model.
nullspace(A, atol=1e-13, rtol=0)	Compute an approximate basis for the nullspace of A.
constraint_matrices(model, array_type='dense', include_vars=False, zero_tol=1e-06)	Create a matrix representation of the problem.

cobra.util.array.**create_stoichiometric_matrix**(*model*, *array_type='dense'*, *dtype=None*)
Return a stoichiometric array representation of the given model.

The the columns represent the reactions and rows represent metabolites. S[i,j] therefore contains the quantity of metabolite *i* produced (negative for consumed) by reaction *j*.

Parameters

- **model** (*cobra.Model*) – The cobra model to construct the matrix for.
- **array_type** (*string*) – The type of array to construct. if 'dense', return a standard numpy.array, 'dok', or 'lil' will construct a sparse array using scipy of the corresponding type and 'DataFrame' will give a pandas *DataFrame* with metabolite indices and reaction columns
- **dtype** (*data-type*) – The desired data-type for the array. If not given, defaults to float.

Returns The stoichiometric matrix for the given model.

Return type matrix of class *dtype*

cobra.util.array.**nullspace**(*A*, *atol=1e-13*, *rtol=0*)
Compute an approximate basis for the nullspace of A. The algorithm used by this function is based on the singular value decomposition of *A*.

Parameters

- **A** (*numpy.ndarray*) – A should be at most 2-D. A 1-D array with length k will be treated as a 2-D with shape (1, k)
- **atol** (*float*) – The absolute tolerance for a zero singular value. Singular values smaller than *atol* are considered to be zero.
- **rtol** (*float*) – The relative tolerance. Singular values less than rtol*smax are considered to be zero, where smax is the largest singular value.

- **both atol and rtol are positive, the combined tolerance is the** (*If*) –
- **of the two; that is::** (*maximum*) –
- **= max(atol, rtol * smax)** (*tol*) –
- **values smaller than tol are considered to be zero.** (*Singular*) –

Returns If *A* is an array with shape (m, k), then *ns* will be an array with shape (k, n), where n is the estimated dimension of the nullspace of *A*. The columns of *ns* are a basis for the nullspace; each element in numpy.dot(A, ns) will be approximately zero.

Return type numpy.ndarray

Notes

Taken from the numpy cookbook.

cobra.util.array.**constraint_matrices**(*model*, *array_type='dense'*, *include_vars=False*, *zero_tol=1e-06*)

Create a matrix representation of the problem.

This is used for alternative solution approaches that do not use optlang. The function will construct the equality matrix, inequality matrix and bounds for the complete problem.

Notes

To accomodate non-zero equalities the problem will add the variable "const_one" which is a variable that equals one.

Parameters

- **model** (cobra.Model) – The model from which to obtain the LP problem.
- **array_type** (*string*) – The type of array to construct. if 'dense', return a standard numpy.array, 'dok', or 'lil' will construct a sparse array using scipy of the corresponding type and 'DataFrame' will give a pandas *DataFrame* with metabolite indices and reaction columns.
- **zero_tol** (*float*) – The zero tolerance used to judge whether two bounds are the same.

Returns

A named tuple consisting of 6 matrices and 2 vectors: - "equalities" is a matrix S such that S*vars = b. It includes a row

for each constraint and one column for each variable.

- "b" the right side of the equality equation such that S*vars = b.
- "inequalities" is a matrix M such that lb <= M*vars <= ub. It contains a row for each inequality and as many columns as variables.
- "bounds" is a compound matrix [lb ub] containing the lower and upper bounds for the inequality constraints in M.
- "variable_fixed" is a boolean vector indicating whether the variable at that index is fixed (lower bound == upper_bound) and is thus bounded by an equality constraint.
- "variable_bounds" is a compound matrix [lb ub] containing the lower and upper bounds for all variables.

Return type collections.namedtuple

cobra.util.context

Module Contents

Classes

HistoryManager	Record a list of actions to be taken at a later time. Used to

Functions

get_context(obj)	Search for a context manager
resettable(f)	A decorator to simplify the context management of simple object

class cobra.util.context.**HistoryManager**

Bases: object

Record a list of actions to be taken at a later time. Used to implement context managers that allow temporary changes to a *Model*.

__call__(*self*, *operation*)

Add the corresponding method to the history stack.

Parameters **operation** (*function*) – A function to be called at a later time

reset(*self*)

Trigger executions for all items in the stack in reverse order

cobra.util.context.**get_context**(*obj*)

Search for a context manager

cobra.util.context.**resettable**(*f*)

A decorator to simplify the context management of simple object attributes. Gets the value of the attribute prior to setting it, and stores a function to set the value to the old value in the HistoryManager.

cobra.util.solver

Additional helper functions for the optlang solvers.

All functions integrate well with the context manager, meaning that all operations defined here are automatically reverted when used in a *with model:* block.

The functions defined here together with the existing model functions should allow you to implement custom flux analysis methods with ease.

Module Contents

Functions

linear_reaction_coefficients(model, reactions=None)	Coefficient for the reactions in a linear objective.
_valid_atoms(model, expression)	Check whether a sympy expression references the correct variables.
set_objective(model, value, additive=False)	Set the model objective.
interface_to_str(interface)	Give a string representation for an optlang interface.
get_solver_name(mip=False, qp=False)	Select a solver for a given optimization problem.
choose_solver(model, solver=None, qp=False)	Choose a solver given a solver name and model.
add_cons_vars_to_problem(model, what, **kwargs)	Add variables and constraints to a Model's solver object.
remove_cons_vars_from_problem(model, what)	Remove variables and constraints from a Model's solver object.
add_absolute_expression(model, expression, name='abs_var', ub=None, difference=0, add=True)	Add the absolute value of an expression to the model.
fix_objective_as_constraint(model, fraction=1, bound=None, name='fixed_objective_{}')	Fix current objective as an additional constraint.
check_solver_status(status, raise_error=False)	Perform standard checks on a solver's status.
assert_optimal(model, message='optimization failed')	Assert model solver status is optimal.
add_lp_feasibility(model)	Add a new objective and variables to ensure a feasible solution.
add_lexicographic_constraints(model, objectives, objective_direction='max')	Successively optimize separate targets in a specific order.

cobra.util.solver.**solvers**

cobra.util.solver.**qp_solvers = ['cplex', 'gurobi']**

cobra.util.solver.**has_primals**

cobra.util.solver.**linear_reaction_coefficients**(*model*, *reactions=None*)

Coefficient for the reactions in a linear objective.

Parameters

- **model** (*cobra model*) – the model object that defined the objective
- **reactions** (*list*) – an optional list for the reactions to get the coefficients for. All reactions if left missing.

Returns A dictionary where the key is the reaction object and the value is the corresponding coefficient. Empty dictionary if there are no linear terms in the objective.

Return type dict

cobra.util.solver.**_valid_atoms**(*model*, *expression*)

Check whether a sympy expression references the correct variables.

Parameters

- **model** (*cobra.Model*) – The model in which to check for variables.
- **expression** (*sympy.Basic*) – A sympy expression.

Returns True if all referenced variables are contained in model, False otherwise.

Return type boolean

cobra.util.solver.**set_objective**(*model*, *value*, *additive=False*)
Set the model objective.

Parameters

- **model** (*cobra model*) – The model to set the objective for
- **value** (*model.problem.Objective,*) – e.g. optlang.glpk_interface.Objective, sympy.Basic or dict

  If the model objective is linear, the value can be a new Objective object or a dictionary with linear coefficients where each key is a reaction and the element the new coefficient (float).

  If the objective is not linear and *additive* is true, only values of class Objective.
- **additive** (*boolmodel.reactions.Biomass_Ecoli_core.bounds = (0.1, 0.1)*) – If true, add the terms to the current objective, otherwise start with an empty objective.

cobra.util.solver.**interface_to_str**(*interface*)
Give a string representation for an optlang interface.

Parameters **interface** (*string, ModuleType*) – Full name of the interface in optlang or cobra representation. For instance 'optlang.glpk_interface' or 'optlang-glpk'.

Returns The name of the interface as a string

Return type string

cobra.util.solver.**get_solver_name**(*mip=False*, *qp=False*)
Select a solver for a given optimization problem.

Parameters

- **mip** (*bool*) – Does the solver require mixed integer linear programming capabilities?
- **qp** (*bool*) – Does the solver require quadratic programming capabilities?

Returns The name of feasible solver.

Return type string

Raises ***SolverNotFound*** – If no suitable solver could be found.

cobra.util.solver.**choose_solver**(*model*, *solver=None*, *qp=False*)
Choose a solver given a solver name and model.

This will choose a solver compatible with the model and required capabilities. Also respects model.solver where it can.

Parameters

- **model** (*a cobra model*) – The model for which to choose the solver.
- **solver** (*str, optional*) – The name of the solver to be used.
- **qp** (*boolean, optional*) – Whether the solver needs Quadratic Programming capabilities.

Returns **solver** – Returns a valid solver for the problem.

Return type an optlang solver interface

Raises ***SolverNotFound*** – If no suitable solver could be found.

cobra.util.solver.**add_cons_vars_to_problem**(*model*, *what*, ***kwargs*)
Add variables and constraints to a Model's solver object.

Useful for variables and constraints that can not be expressed with reactions and lower/upper bounds. Will integrate with the Model's context manager in order to revert changes upon leaving the context.

Parameters

- **model** (*a cobra model*) – The model to which to add the variables and constraints.
- **what** (*list or tuple of optlang variables or constraints.*) – The variables or constraints to add to the model. Must be of class *model.problem.Variable* or *model.problem.Constraint*.
- ****kwargs** (*keyword arguments*) – passed to solver.add()

cobra.util.solver.**remove_cons_vars_from_problem**(*model*, *what*)

Remove variables and constraints from a Model's solver object.

Useful to temporarily remove variables and constraints from a Models's solver object.

Parameters

- **model** (*a cobra model*) – The model from which to remove the variables and constraints.
- **what** (*list or tuple of optlang variables or constraints.*) – The variables or constraints to remove from the model. Must be of class *model.problem.Variable* or *model.problem.Constraint*.

cobra.util.solver.**add_absolute_expression**(*model*, *expression*, *name='abs_var'*, *ub=None*, *difference=0*, *add=True*)

Add the absolute value of an expression to the model.

Also defines a variable for the absolute value that can be used in other objectives or constraints.

Parameters

- **model** (*a cobra model*) – The model to which to add the absolute expression.
- **expression** (*A sympy expression*) – Must be a valid expression within the Model's solver object. The absolute value is applied automatically on the expression.
- **name** (*string*) – The name of the newly created variable.
- **ub** (*positive float*) – The upper bound for the variable.
- **difference** (*positive float*) – The difference between the expression and the variable.
- **add** (*bool*) – Whether to add the variable to the model at once.

Returns A named tuple with variable and two constraints (upper_constraint, lower_constraint) describing the new variable and the constraints that assign the absolute value of the expression to it.

Return type namedtuple

cobra.util.solver.**fix_objective_as_constraint**(*model*, *fraction=1*, *bound=None*, *name='fixed_objective_{}'*)

Fix current objective as an additional constraint.

When adding constraints to a model, such as done in pFBA which minimizes total flux, these constraints can become too powerful, resulting in solutions that satisfy optimality but sacrifices too much for the original objective function. To avoid that, we can fix the current objective value as a constraint to ignore solutions that give a lower (or higher depending on the optimization direction) objective value than the original model.

When done with the model as a context, the modification to the objective will be reverted when exiting that context.

Parameters

- **model** (cobra.Model) – The model to operate on
- **fraction** (*float*) – The fraction of the optimum the objective is allowed to reach.

- **bound** (*float, None*) – The bound to use instead of fraction of maximum optimal value. If not None, fraction is ignored.
- **name** (*str*) – Name of the objective. May contain one *{}* placeholder which is filled with the name of the old objective.

Returns

Return type The value of the optimized objective * fraction

cobra.util.solver.**check_solver_status**(*status*, *raise_error=False*)
Perform standard checks on a solver's status.

cobra.util.solver.**assert_optimal**(*model*, *message='optimization failed'*)
Assert model solver status is optimal.

Do nothing if model solver status is optimal, otherwise throw appropriate exception depending on the status.

Parameters

- **model** (cobra.Model) – The model to check the solver status for.
- **message** (*str (optional)*) – Message to for the exception if solver status was not optimal.

cobra.util.solver.**add_lp_feasibility**(*model*)
Add a new objective and variables to ensure a feasible solution.

The optimized objective will be zero for a feasible solution and otherwise represent the distance from feasibility (please see [1]_ for more information).

Parameters **model** (cobra.Model) – The model whose feasibility is to be tested.

References

"DFBAlab: A Fast and Reliable MATLAB Code for Dynamic Flux Balance Analysis." BMC Bioinformatics 15, no. 1 (December 18, 2014): 409. https://doi.org/10.1186/s12859-014-0409-8.

cobra.util.solver.**add_lexicographic_constraints**(*model*, *objectives*, *objective_direction='max'*)
Successively optimize separate targets in a specific order.

For each objective, optimize the model and set the optimal value as a constraint. Proceed in the order of the objectives given. Due to the specific order this is called lexicographic FBA [1]_. This procedure is useful for returning unique solutions for a set of important fluxes. Typically this is applied to exchange fluxes.

Parameters

- **model** (cobra.Model) – The model to be optimized.
- **objectives** (*list*) – A list of reactions (or objectives) in the model for which unique fluxes are to be determined.
- **objective_direction** (*str or list, optional*) – The desired objective direction for each reaction (if a list) or the objective direction to use for all reactions (default maximize).

Returns **optimized_fluxes** – A vector containing the optimized fluxes for each of the given reactions in *objectives*.

Return type pandas.Series

References

"DFBAlab: A Fast and Reliable MATLAB Code for Dynamic Flux Balance Analysis." BMC Bioinformatics 15, no. 1 (December 18, 2014): 409. https://doi.org/10.1186/s12859-014-0409-8.

cobra.util.util

Module Contents

Classes

AutoVivification	Implementation of perl's autovivification feature. Checkout

Functions

format_long_string(string, max_length=50)	
show_versions()	Print dependency information.

cobra.util.util.**format_long_string**(*string*, *max_length=50*)

class cobra.util.util.**AutoVivification**
Bases: dict

Implementation of perl's autovivification feature. Checkout http://stackoverflow.com/a/652284/280182

__getitem__(*self*, *item*)
x.__getitem__(y) <==> x[y]

cobra.util.util.**show_versions**()
Print dependency information.

Package Contents

Classes

HistoryManager	Record a list of actions to be taken at a later time. Used to
AutoVivification	Implementation of perl's autovivification feature. Checkout

Functions

create_stoichiometric_matrix(model, array_type='dense', dtype=None)	Return a stoichiometric array representation of the given model.
nullspace(A, atol=1e-13, rtol=0)	Compute an approximate basis for the nullspace of A.
constraint_matrices(model, array_type='dense', include_vars=False, zero_tol=1e-06)	Create a matrix representation of the problem.
get_context(obj)	Search for a context manager

Continued on next page

Table 92 – continued from previous page

resettable(f)	A decorator to simplify the context management of simple object
get_context(obj)	Search for a context manager
linear_reaction_coefficients(model, reactions=None)	Coefficient for the reactions in a linear objective.
_valid_atoms(model, expression)	Check whether a sympy expression references the correct variables.
set_objective(model, value, additive=False)	Set the model objective.
interface_to_str(interface)	Give a string representation for an optlang interface.
get_solver_name(mip=False, qp=False)	Select a solver for a given optimization problem.
choose_solver(model, solver=None, qp=False)	Choose a solver given a solver name and model.
add_cons_vars_to_problem(model, what, **kwargs)	Add variables and constraints to a Model's solver object.
remove_cons_vars_from_problem(model, what)	Remove variables and constraints from a Model's solver object.
add_absolute_expression(model, expression, name='abs_var', ub=None, difference=0, add=True)	Add the absolute value of an expression to the model.
fix_objective_as_constraint(model, fraction=1, bound=None, name='fixed_objective_{}')	Fix current objective as an additional constraint.
check_solver_status(status, raise_error=False)	Perform standard checks on a solver's status.
assert_optimal(model, message='optimization failed')	Assert model solver status is optimal.
add_lp_feasibility(model)	Add a new objective and variables to ensure a feasible solution.
add_lexicographic_constraints(model, objectives, objective_direction='max')	Successively optimize separate targets in a specific order.
format_long_string(string, max_length=50)	
show_versions()	Print dependency information.

cobra.util.**create_stoichiometric_matrix**(*model*, *array_type='dense'*, *dtype=None*)

Return a stoichiometric array representation of the given model.

The the columns represent the reactions and rows represent metabolites. S[i,j] therefore contains the quantity of metabolite *i* produced (negative for consumed) by reaction *j*.

Parameters

- **model** (*cobra.Model*) – The cobra model to construct the matrix for.
- **array_type** (*string*) – The type of array to construct. if 'dense', return a standard numpy.array, 'dok', or 'lil' will construct a sparse array using scipy of the corresponding type and 'DataFrame' will give a pandas *DataFrame* with metabolite indices and reaction columns
- **dtype** (*data-type*) – The desired data-type for the array. If not given, defaults to float.

Returns The stoichiometric matrix for the given model.

Return type matrix of class *dtype*

cobra.util.**nullspace**(*A*, *atol=1e-13*, *rtol=0*)

Compute an approximate basis for the nullspace of A. The algorithm used by this function is based on the singular value decomposition of *A*.

Parameters

- **A** (*numpy.ndarray*) – A should be at most 2-D. A 1-D array with length k will be treated as a 2-D with shape (1, k)

- **atol** (*float*) – The absolute tolerance for a zero singular value. Singular values smaller than *atol* are considered to be zero.
- **rtol** (*float*) – The relative tolerance. Singular values less than rtol*smax are considered to be zero, where smax is the largest singular value.
- **both atol and rtol are positive, the combined tolerance is the** (*If*) –
- **of the two; that is::** (*maximum*) –
- **= max(atol, rtol * smax)** (*tol*) –
- **values smaller than tol are considered to be zero.** (*Singular*) –

Returns If *A* is an array with shape (m, k), then *ns* will be an array with shape (k, n), where n is the estimated dimension of the nullspace of *A*. The columns of *ns* are a basis for the nullspace; each element in numpy.dot(A, ns) will be approximately zero.

Return type numpy.ndarray

Notes

Taken from the numpy cookbook.

cobra.util.**constraint_matrices**(*model*, *array_type='dense'*, *include_vars=False*, *zero_tol=1e-06*)

Create a matrix representation of the problem.

This is used for alternative solution approaches that do not use optlang. The function will construct the equality matrix, inequality matrix and bounds for the complete problem.

Notes

To accomodate non-zero equalities the problem will add the variable "const_one" which is a variable that equals one.

Parameters

- **model** (cobra.Model) – The model from which to obtain the LP problem.
- **array_type** (*string*) – The type of array to construct. if 'dense', return a standard numpy.array, 'dok', or 'lil' will construct a sparse array using scipy of the corresponding type and 'DataFrame' will give a pandas *DataFrame* with metabolite indices and reaction columns.
- **zero_tol** (*float*) – The zero tolerance used to judge whether two bounds are the same.

Returns

A named tuple consisting of 6 matrices and 2 vectors: - "equalities" is a matrix S such that S*vars = b. It includes a row

for each constraint and one column for each variable.

- "b" the right side of the equality equation such that S*vars = b.
- "inequalities" is a matrix M such that lb <= M*vars <= ub. It contains a row for each inequality and as many columns as variables.
- "bounds" is a compound matrix [lb ub] containing the lower and upper bounds for the inequality constraints in M.

- "variable_fixed" is a boolean vector indicating whether the variable at that index is fixed (lower bound == upper_bound) and is thus bounded by an equality constraint.
- "variable_bounds" is a compound matrix [lb ub] containing the lower and upper bounds for all variables.

Return type collections.namedtuple

class cobra.util.**HistoryManager**
Bases: object

Record a list of actions to be taken at a later time. Used to implement context managers that allow temporary changes to a *Model*.

__call__(*self*, *operation*)
Add the corresponding method to the history stack.

Parameters **operation** (*function*) – A function to be called at a later time

reset(*self*)
Trigger executions for all items in the stack in reverse order

cobra.util.**get_context**(*obj*)
Search for a context manager

cobra.util.**resettable**(*f*)
A decorator to simplify the context management of simple object attributes. Gets the value of the attribute prior to setting it, and stores a function to set the value to the old value in the HistoryManager.

cobra.util.**OPTLANG_TO_EXCEPTIONS_DICT**

exception cobra.util.**OptimizationError**(*message*)
Bases: Exception

Common base class for all non-exit exceptions.

exception cobra.util.**SolverNotFound**
Bases: Exception

A simple Exception when a solver can not be found.

cobra.util.**get_context**(*obj*)
Search for a context manager

cobra.util.**solvers**

cobra.util.**qp_solvers = ['cplex', 'gurobi']**

cobra.util.**has_primals**

cobra.util.**linear_reaction_coefficients**(*model*, *reactions=None*)
Coefficient for the reactions in a linear objective.

Parameters

- **model** (*cobra model*) – the model object that defined the objective
- **reactions** (*list*) – an optional list for the reactions to get the coefficients for. All reactions if left missing.

Returns A dictionary where the key is the reaction object and the value is the corresponding coefficient. Empty dictionary if there are no linear terms in the objective.

Return type dict

cobra.util.**_valid_atoms**(*model*, *expression*)
Check whether a sympy expression references the correct variables.

Parameters

- **model** (cobra.Model) – The model in which to check for variables.

- **expression** (*sympy.Basic*) – A sympy expression.

Returns True if all referenced variables are contained in model, False otherwise.

Return type boolean

cobra.util.**set_objective**(*model*, *value*, *additive=False*)
Set the model objective.

Parameters

- **model** (*cobra model*) – The model to set the objective for
- **value** (*model.problem.Objective,*) – e.g. optlang.glpk_interface.Objective, sympy.Basic or dict

  If the model objective is linear, the value can be a new Objective object or a dictionary with linear coefficients where each key is a reaction and the element the new coefficient (float).

  If the objective is not linear and *additive* is true, only values of class Objective.
- **additive** (*boolmodel.reactions.Biomass_Ecoli_core.bounds = (0.1, 0.1)*) – If true, add the terms to the current objective, otherwise start with an empty objective.

cobra.util.**interface_to_str**(*interface*)
Give a string representation for an optlang interface.

Parameters **interface** (*string, ModuleType*) – Full name of the interface in optlang or cobra representation. For instance 'optlang.glpk_interface' or 'optlang-glpk'.

Returns The name of the interface as a string

Return type string

cobra.util.**get_solver_name**(*mip=False*, *qp=False*)
Select a solver for a given optimization problem.

Parameters

- **mip** (*bool*) – Does the solver require mixed integer linear programming capabilities?
- **qp** (*bool*) – Does the solver require quadratic programming capabilities?

Returns The name of feasible solver.

Return type string

Raises ***SolverNotFound*** – If no suitable solver could be found.

cobra.util.**choose_solver**(*model*, *solver=None*, *qp=False*)
Choose a solver given a solver name and model.

This will choose a solver compatible with the model and required capabilities. Also respects model.solver where it can.

Parameters

- **model** (*a cobra model*) – The model for which to choose the solver.
- **solver** (*str, optional*) – The name of the solver to be used.
- **qp** (*boolean, optional*) – Whether the solver needs Quadratic Programming capabilities.

Returns **solver** – Returns a valid solver for the problem.

Return type an optlang solver interface

Raises ***SolverNotFound*** – If no suitable solver could be found.

cobra.util.**add_cons_vars_to_problem**(*model*, *what*, ***kwargs*)
Add variables and constraints to a Model's solver object.

Useful for variables and constraints that can not be expressed with reactions and lower/upper bounds. Will integrate with the Model's context manager in order to revert changes upon leaving the context.

Parameters

- **model** (*a cobra model*) – The model to which to add the variables and constraints.
- **what** (*list or tuple of optlang variables or constraints.*) – The variables or constraints to add to the model. Must be of class *model.problem.Variable* or *model.problem.Constraint*.
- ****kwargs** (*keyword arguments*) – passed to solver.add()

cobra.util.**remove_cons_vars_from_problem**(*model*, *what*)
Remove variables and constraints from a Model's solver object.

Useful to temporarily remove variables and constraints from a Models's solver object.

Parameters

- **model** (*a cobra model*) – The model from which to remove the variables and constraints.
- **what** (*list or tuple of optlang variables or constraints.*) – The variables or constraints to remove from the model. Must be of class *model.problem.Variable* or *model.problem.Constraint*.

cobra.util.**add_absolute_expression**(*model*, *expression*, *name='abs_var'*, *ub=None*, *difference=0*, *add=True*)
Add the absolute value of an expression to the model.

Also defines a variable for the absolute value that can be used in other objectives or constraints.

Parameters

- **model** (*a cobra model*) – The model to which to add the absolute expression.
- **expression** (*A sympy expression*) – Must be a valid expression within the Model's solver object. The absolute value is applied automatically on the expression.
- **name** (*string*) – The name of the newly created variable.
- **ub** (*positive float*) – The upper bound for the variable.
- **difference** (*positive float*) – The difference between the expression and the variable.
- **add** (*bool*) – Whether to add the variable to the model at once.

Returns A named tuple with variable and two constraints (upper_constraint, lower_constraint) describing the new variable and the constraints that assign the absolute value of the expression to it.

Return type namedtuple

cobra.util.**fix_objective_as_constraint**(*model*, *fraction=1*, *bound=None*, *name='fixed_objective_{}'*)
Fix current objective as an additional constraint.

When adding constraints to a model, such as done in pFBA which minimizes total flux, these constraints can become too powerful, resulting in solutions that satisfy optimality but sacrifices too much for the original objective function. To avoid that, we can fix the current objective value as a constraint to ignore solutions that give a lower (or higher depending on the optimization direction) objective value than the original model.

When done with the model as a context, the modification to the objective will be reverted when exiting that context.

Parameters

- **model** (cobra.Model) – The model to operate on
- **fraction** (*float*) – The fraction of the optimum the objective is allowed to reach.
- **bound** (*float, None*) – The bound to use instead of fraction of maximum optimal value. If not None, fraction is ignored.
- **name** (*str*) – Name of the objective. May contain one *{}* placeholder which is filled with the name of the old objective.

Returns

Return type The value of the optimized objective * fraction

cobra.util.**check_solver_status**(*status*, *raise_error=False*)
Perform standard checks on a solver's status.

cobra.util.**assert_optimal**(*model*, *message='optimization failed'*)
Assert model solver status is optimal.

Do nothing if model solver status is optimal, otherwise throw appropriate exception depending on the status.

Parameters

- **model** (cobra.Model) – The model to check the solver status for.
- **message** (*str (optional)*) – Message to for the exception if solver status was not optimal.

cobra.util.**add_lp_feasibility**(*model*)
Add a new objective and variables to ensure a feasible solution.

The optimized objective will be zero for a feasible solution and otherwise represent the distance from feasibility (please see [1]_ for more information).

Parameters **model** (cobra.Model) – The model whose feasibility is to be tested.

References

"DFBAlab: A Fast and Reliable MATLAB Code for Dynamic Flux Balance Analysis." BMC Bioinformatics 15, no. 1 (December 18, 2014): 409. https://doi.org/10.1186/s12859-014-0409-8.

cobra.util.**add_lexicographic_constraints**(*model*, *objectives*, *objective_direction='max'*)
Successively optimize separate targets in a specific order.

For each objective, optimize the model and set the optimal value as a constraint. Proceed in the order of the objectives given. Due to the specific order this is called lexicographic FBA [1]_. This procedure is useful for returning unique solutions for a set of important fluxes. Typically this is applied to exchange fluxes.

Parameters

- **model** (cobra.Model) – The model to be optimized.
- **objectives** (*list*) – A list of reactions (or objectives) in the model for which unique fluxes are to be determined.
- **objective_direction** (*str or list, optional*) – The desired objective direction for each reaction (if a list) or the objective direction to use for all reactions (default maximize).

Returns **optimized_fluxes** – A vector containing the optimized fluxes for each of the given reactions in *objectives*.

Return type pandas.Series

References

"DFBAlab: A Fast and Reliable MATLAB Code for Dynamic Flux Balance Analysis." BMC Bioinformatics 15, no. 1 (December 18, 2014): 409. https://doi.org/10.1186/s12859-014-0409-8.

cobra.util.**format_long_string**(*string*, *max_length=50*)

class cobra.util.**AutoVivification**
Bases: dict

Implementation of perl's autovivification feature. Checkout http://stackoverflow.com/a/652284/280182

__getitem__(*self*, *item*)
x.__getitem__(y) <==> x[y]

cobra.util.**show_versions**()
Print dependency information.

17.1.2 Submodules

cobra.exceptions

Module Contents

exception cobra.exceptions.**OptimizationError**(*message*)
Bases: Exception

Common base class for all non-exit exceptions.

exception cobra.exceptions.**Infeasible**(*message*)
Bases: *cobra.exceptions.OptimizationError*

Common base class for all non-exit exceptions.

exception cobra.exceptions.**Unbounded**(*message*)
Bases: *cobra.exceptions.OptimizationError*

Common base class for all non-exit exceptions.

exception cobra.exceptions.**FeasibleButNotOptimal**(*message*)
Bases: *cobra.exceptions.OptimizationError*

Common base class for all non-exit exceptions.

exception cobra.exceptions.**UndefinedSolution**(*message*)
Bases: *cobra.exceptions.OptimizationError*

Common base class for all non-exit exceptions.

exception cobra.exceptions.**SolverNotFound**
Bases: Exception

A simple Exception when a solver can not be found.

cobra.exceptions.**OPTLANG_TO_EXCEPTIONS_DICT**

17.1.3 Package Contents

Classes

Configuration	Define the configuration to be singleton based.
DictList	A combined dict and list
Gene	A Gene in a cobra model
Metabolite	Metabolite is a class for holding information regarding
Model	Class representation for a cobra model
Object	Defines common behavior of object in cobra.core
Reaction	Reaction is a class for holding information regarding
Solution	A unified interface to a *cobra.Model* optimization solution.
Species	Species is a class for holding information regarding

Functions

_warn_format(message, category, filename, lineno, file=None, line=None)	
show_versions()	Print dependency information.

cobra.**_cobra_path**

cobra.**_warning_base = %s:%s [1;31m%s[0m: %s**

cobra.**_warn_format**(*message*, *category*, *filename*, *lineno*, *file=None*, *line=None*)

cobra.**formatwarning**

class cobra.**Configuration**
Bases: six.with_metaclass()

Define the configuration to be singleton based.

class cobra.**DictList**(**args*)
Bases: list

A combined dict and list

This object behaves like a list, but has the O(1) speed benefits of a dict when looking up elements by their id.

has_id(*self*, *id*)

_check(*self*, *id*)
make sure duplicate id's are not added. This function is called before adding in elements.

_generate_index(*self*)
rebuild the _dict index

get_by_id(*self*, *id*)
return the element with a matching id

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

get_by_any(*self*, *iterable*)
Get a list of members using several different ways of indexing

Parameters **iterable** (*list (if not, turned into single element list)*) – list where each element is either int (referring to an index in in this

DictList), string (a id of a member in this DictList) or member of this DictList for pass-through

Returns a list of members

Return type list

query (*self*, *search_function*, *attribute=None*)
Query the list

Parameters

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

Returns a new list of objects which match the query

Return type *DictList*

Examples

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

_replace_on_id (*self*, *new_object*)
Replace an object by another with the same id.

append (*self*, *object*)
append object to end

union (*self*, *iterable*)
adds elements with id's not already in the model

extend (*self*, *iterable*)
extend list by appending elements from the iterable

_extend_nocheck (*self*, *iterable*)
extends without checking for uniqueness

This function should only be used internally by DictList when it can guarantee elements are already unique (as in when coming from self or other DictList). It will be faster because it skips these checks.

__sub__ (*self*, *other*)
x.__sub__(y) <==> x - y

Parameters **other** (*iterable*) – other must contain only unique id's present in the list

__isub__ (*self*, *other*)
x.__sub__(y) <==> x -= y

Parameters **other** (*iterable*) – other must contain only unique id's present in the list

__add__ (*self*, *other*)
x.__add__(y) <==> x + y

Parameters **other** (*iterable*) – other must contain only unique id's which do not intersect with self

__iadd__ (*self*, *other*)
x.__iadd__(y) <==> x += y

Parameters **other** (*iterable*) – other must contain only unique id's whcih do not intersect with self

__reduce__ (*self*)
Helper for pickle.

__getstate__ (*self*)
gets internal state

This is only provided for backwards compatibility so older versions of cobrapy can load pickles generated with cobrapy. In reality, the "_dict" state is ignored when loading a pickle

__setstate__ (*self*, *state*)
sets internal state

Ignore the passed in state and recalculate it. This is only for compatibility with older pickles which did not correctly specify the initialization class

index (*self*, *id*, **args*)
Determine the position in the list

id: A string or a `Object`

__contains__ (*self*, *object*)
DictList.__contains__(object) <==> object in DictList

object: str or `Object`

__copy__ (*self*)

insert (*self*, *index*, *object*)
insert object before index

pop (*self*, **args*)
remove and return item at index (default last).

add (*self*, *x*)
Opposite of *remove*. Mirrors set.add

remove (*self*, *x*)

> **Warning:** Internal use only

reverse (*self*)
reverse *IN PLACE*

sort (*self*, *cmp=None*, *key=None*, *reverse=False*)
stable sort *IN PLACE*

cmp(x, y) -> -1, 0, 1

__getitem__ (*self*, *i*)
x.__getitem__(y) <==> x[y]

__setitem__ (*self*, *i*, *y*)
Set self[key] to value.

__delitem__ (*self*, *index*)
Delete self[key].

__getslice__ (*self*, *i*, *j*)

__setslice__(*self*, *i*, *j*, *y*)

__delslice__(*self*, *i*, *j*)

__getattr__(*self*, *attr*)

__dir__(*self*)
Default dir() implementation.

class cobra.**Gene**(*id=None*, *name=''*, *functional=True*)
Bases: *cobra.core.species.Species*

A Gene in a cobra model

Parameters

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

property functional(*self*)
A flag indicating if the gene is functional.

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

knock_out(*self*)
Knockout gene by marking it as non-functional and setting all associated reactions bounds to zero.

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

remove_from_model(*self*, *model=None*, *make_dependent_reactions_nonfunctional=True*)
Removes the association

Parameters

- **model** (*cobra model*) – The model to remove the gene from
- **make_dependent_reactions_nonfunctional** (*bool*) – If True then replace the gene with 'False' in the gene association, else replace the gene with 'True'

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

_repr_html_(*self*)

class cobra.**Metabolite**(*id=None*, *formula=None*, *name=''*, *charge=None*, *compartment=None*)
Bases: *cobra.core.species.Species*

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

Parameters

- **id** (*str*) – the identifier to associate with the metabolite
- **formula** (*str*) – Chemical formula (e.g. H2O)
- **name** (*str*) – A human readable name.
- **charge** (*float*) – The charge number of the metabolite
- **compartment** (*str or None*) – Compartment of the metabolite.

_set_id_with_model(*self*, *value*)

property constraint(*self*)
Get the constraints associated with this metabolite from the solve

Returns the optlang constraint for this metabolite

Return type optlang.<interface>.Constraint

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

property formula_weight (*self*)
Calculate the formula weight

property y (*self*)
The shadow price for the metabolite in the most recent solution

Shadow prices are computed from the dual values of the bounds in the solution.

property shadow_price (*self*)
The shadow price in the most recent solution.

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

Warning:

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

Raises

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

Examples

```
>>> import cobra
>>> import cobra.test
>>> model = cobra.test.create_test_model("textbook")
>>> solution = model.optimize()
>>> model.metabolites.glc__D_e.shadow_price
-0.09166474637510488
>>> solution.shadow_prices.glc__D_e
-0.09166474637510883
```

remove_from_model (*self*, *destructive=False*)
Removes the association from self.model

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

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

summary (*self*, *solution=None*, *threshold=0.01*, *fva=None*, *names=False*, *float_format='{:.3g}'.format*)
Create a summary of the producing and consuming fluxes.

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

Parameters

- **solution** (*cobra.Solution, optional*) – A previous model solution to use for generating the summary. If None, the summary method will resolve the model. Note that the solution object must match the model, i.e., changes to the model such as changed bounds, added or removed reactions are not taken into account by this method (default None).
- **threshold** (*float, optional*) – Threshold below which fluxes are not reported. May not be smaller than the model tolerance (default 0.01).
- **fva** (*pandas.DataFrame or float, optional*) – Whether or not to include flux variability analysis in the output. If given, fva should either be a previous FVA solution matching the model or a float between 0 and 1 representing the fraction of the optimum objective to be searched (default None).
- **names** (*bool, optional*) – Emit reaction and metabolite names rather than identifiers (default False).
- **float_format** (*callable, optional*) – Format string for floats (default `'{:3G}'.format`).

Returns

Return type cobra.MetaboliteSummary

See also:

Reaction.summary(), *Model.summary()*

_repr_html_(*self*)

class cobra.**Model**(*id_or_model=None*, *name=None*)

Bases: *cobra.core.object.Object*

Class representation for a cobra model

Parameters

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

reactions

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

Type *DictList*

metabolites

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

Type *DictList*

genes

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

Type *DictList*

groups

A DictList where the key is the group identifier and the value a Group

Type *DictList*

solution

The last obtained solution from optimizing the model.

Type *Solution*

__setstate__(*self*, *state*)

Make sure all cobra.Objects in the model point to the model.

__getstate__ (*self*)
Get state for serialization.

Ensures that the context stack is cleared prior to serialization, since partial functions cannot be pickled reliably.

property solver (*self*)
Get or set the attached solver instance.

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

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

Examples

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

property tolerance (*self*)

property description (*self*)

get_metabolite_compartments (*self*)
Return all metabolites' compartments.

property compartments (*self*)

property medium (*self*)

__add__ (*self*, *other_model*)
Add the content of another model to this model (+).

The model is copied as a new object, with a new model identifier, and copies of all the reactions in the other model are added to this model. The objective is the sum of the objective expressions for the two models.

__iadd__ (*self*, *other_model*)
Incrementally add the content of another model to this model (+=).

Copies of all the reactions in the other model are added to this model. The objective is the sum of the objective expressions for the two models.

copy (*self*)
Provides a partial 'deepcopy' of the Model. All of the Metabolite, Gene, and Reaction objects are created anew but in a faster fashion than deepcopy

add_metabolites (*self*, *metabolite_list*)
Will add a list of metabolites to the model object and add new constraints accordingly.

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

Parameters **metabolite_list** (A list of *cobra.core.Metabolite* objects) –

remove_metabolites (*self*, *metabolite_list*, *destructive=False*)
Remove a list of metabolites from the the object.

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

Parameters

- **metabolite_list** (*list*) – A list with *cobra.Metabolite* objects as elements.

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

add_reaction(*self*, *reaction*)

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

Parameters

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

add_boundary(*self*, *metabolite*, *type='exchange'*, *reaction_id=None*, *lb=None*, *ub=None*, *sbo_term=None*)

Add a boundary reaction for a given metabolite.

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

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

Parameters

- **metabolite** (cobra.Metabolite) – Any given metabolite. The compartment is not checked but you are encouraged to stick to the definition of exchanges and sinks.
- **type** (*str, {"exchange", "demand", "sink"}*) – Using one of the pre-defined reaction types is easiest. If you want to create your own kind of boundary reaction choose any other string, e.g., 'my-boundary'.
- **reaction_id** (*str, optional*) – The ID of the resulting reaction. This takes precedence over the auto-generated identifiers but beware that it might make boundary reactions harder to identify afterwards when using *model.boundary* or specifically *model.exchanges* etc.
- **lb** (*float, optional*) – The lower bound of the resulting reaction.
- **ub** (*float, optional*) – The upper bound of the resulting reaction.
- **sbo_term** (*str, optional*) – A correct SBO term is set for the available types. If a custom type is chosen, a suitable SBO term should also be set.

Returns The created boundary reaction.

Return type *cobra.Reaction*

Examples

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

(continues on next page)

(continued from previous page)

```
>>> demand.build_reaction_string()
'atp_c --> '
```

add_reactions (*self*, *reaction_list*)
Add reactions to the model.

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

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

Parameters **reaction_list** (*list*) – A list of *cobra.Reaction* objects

remove_reactions (*self*, *reactions*, *remove_orphans=False*)
Remove reactions from the model.

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

Parameters

- **reactions** (*list*) – A list with reactions (*cobra.Reaction*), or their id's, to remove
- **remove_orphans** (*bool*) – Remove orphaned genes and metabolites from the model as well

add_groups (*self*, *group_list*)
Add groups to the model.

Groups with identifiers identical to a group already in the model are ignored.

If any group contains members that are not in the model, these members are added to the model as well. Only metabolites, reactions, and genes can have groups.

Parameters **group_list** (*list*) – A list of *cobra.Group* objects to add to the model.

remove_groups (*self*, *group_list*)
Remove groups from the model.

Members of each group are not removed from the model (i.e. metabolites, reactions, and genes in the group stay in the model after any groups containing them are removed).

Parameters **group_list** (*list*) – A list of *cobra.Group* objects to remove from the model.

get_associated_groups (*self*, *element*)
Returns a list of groups that an element (reaction, metabolite, gene) is associated with.

Parameters **element** (*cobra.Reaction*, *cobra.Metabolite*, or *cobra.Gene*) –

Returns All groups that the provided object is a member of

Return type list of *cobra.Group*

add_cons_vars (*self*, *what*, ***kwargs*)
Add constraints and variables to the model's mathematical problem.

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

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

Parameters

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

remove_cons_vars(*self*, *what*)
Remove variables and constraints from the model's mathematical problem.

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

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

property problem(*self*)
The interface to the model's underlying mathematical problem.

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

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

Return type optlang.interface

property variables(*self*)
The mathematical variables in the cobra model.

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

Returns A container with all associated variables.

Return type optlang.container.Container

property constraints(*self*)
The constraints in the cobra model.

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

Returns A container with all associated constraints.

Return type optlang.container.Container

property boundary(*self*)
Boundary reactions in the model. Reactions that either have no substrate or product.

property exchanges(*self*)
Exchange reactions in model. Reactions that exchange mass with the exterior. Uses annotations and heuristics to exclude non-exchanges such as sink reactions.

property demands(*self*)
Demand reactions in model. Irreversible reactions that accumulate or consume a metabolite in the inside of the model.

property sinks(*self*)
Sink reactions in model. Reversible reactions that accumulate or consume a metabolite in the inside of the model.

_populate_solver(*self*, *reaction_list*, *metabolite_list=None*)
Populate attached solver with constraints and variables that model the provided reactions.

slim_optimize(*self*, *error_value=float('nan')*, *message=None*)
Optimize model without creating a solution object.

Creating a full solution object implies fetching shadow prices and flux values for all reactions and metabolites from the solver object. This necessarily takes some time and in cases where only one or two values are of interest, it is recommended to instead use this function which does not create a solution object returning only the value of the objective. Note however that the *optimize()* function

uses efficient means to fetch values so if you need fluxes/shadow prices for more than say 4 reactions/metabolites, then the total speed increase of *slim_optimize* versus *optimize* is expected to be small or even negative depending on how you fetch the values after optimization.

Parameters

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

Returns The objective value.

Return type float

optimize (*self*, *objective_sense=None*, *raise_error=False*)

Optimize the model using flux balance analysis.

Parameters

- **objective_sense** (*{None, 'maximize' 'minimize'}, optional*) – Whether fluxes should be maximized or minimized. In case of None, the previous direction is used.
- **raise_error** (*bool*) –

  If true, raise an OptimizationError if solver status is not optimal.

Notes

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

repair (*self*, *rebuild_index=True*, *rebuild_relationships=True*)

Update all indexes and pointers in a model

Parameters

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

property objective (*self*)

Get or set the solver objective

Before introduction of the optlang based problems, this function returned the objective reactions as a list. With optlang, the objective is not limited a simple linear summation of individual reaction fluxes, making that return value ambiguous. Henceforth, use *cobra.util.solver.linear_reaction_coefficients* to get a dictionary of reactions with their linear coefficients (empty if there are none)

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

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

property objective_direction (*self*)

Get or set the objective direction.

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

summary(*self*, *solution=None*, *threshold=0.01*, *fva=None*, *names=False*, *float_format='{:.3g}'.format*)
Create a summary of the exchange fluxes of the model.

Parameters

- **solution** (*cobra.Solution, optional*) – A previous model solution to use for generating the summary. If None, the summary method will resolve the model. Note that the solution object must match the model, i.e., changes to the model such as changed bounds, added or removed reactions are not taken into account by this method (default None).
- **threshold** (*float, optional*) – Threshold below which fluxes are not reported. May not be smaller than the model tolerance (default 0.01).
- **fva** (*pandas.DataFrame or float, optional*) – Whether or not to include flux variability analysis in the output. If given, fva should either be a previous FVA solution matching the model or a float between 0 and 1 representing the fraction of the optimum objective to be searched (default None).
- **names** (*bool, optional*) – Emit reaction and metabolite names rather than identifiers (default False).
- **float_format** (*callable, optional*) – Format string for floats (default `'{:3G}'.format`).

Returns

Return type cobra.ModelSummary

See also:

Reaction.summary(), *Metabolite.summary()*

__enter__(*self*)
Record all future changes to the model, undoing them when a call to __exit__ is received

__exit__(*self*, *type*, *value*, *traceback*)
Pop the top context manager and trigger the undo functions

merge(*self*, *right*, *prefix_existing=None*, *inplace=True*, *objective='left'*)
Merge two models to create a model with the reactions from both models.

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

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

prefix_existing [string] Prefix the reaction identifier in the right that already exist in the left model with this string.

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

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

_repr_html_(*self*)

class cobra.**Object**(*id=None*, *name=''*)
Bases: object

Defines common behavior of object in cobra.core

property id(*self*)

_set_id_with_model(*self*, *value*)

__getstate__(*self*)
To prevent excessive replication during deepcopy.

__repr__(*self*)
Return repr(self).

__str__(*self*)
Return str(self).

class cobra.**Reaction**(*id=None*, *name=''*, *subsystem=''*, *lower_bound=0.0*, *upper_bound=None*)
Bases: *cobra.core.object.Object*

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

Reactions are by default irreversible with bounds *(0.0, cobra.Configuration().upper_bound)* if no bounds are provided on creation. To create an irreversible reaction use *lower_bound=None*, resulting in reaction bounds of *(cobra.Configuration().lower_bound, cobra.Configuration().upper_bound)*.

Parameters

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

__radd__

_set_id_with_model(*self*, *value*)

property reverse_id(*self*)
Generate the id of reverse_variable from the reaction's id.

property flux_expression(*self*)
Forward flux expression

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

Return type sympy expression

property forward_variable(*self*)
An optlang variable representing the forward flux

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

Return type optlang.interface.Variable

property reverse_variable(*self*)
An optlang variable representing the reverse flux

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

Return type optlang.interface.Variable

property objective_coefficient(*self*)
Get the coefficient for this reaction in a linear objective (float)

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

__copy__(*self*)

__deepcopy__(*self*, *memo*)

static _check_bounds(*lb*, *ub*)

update_variable_bounds(*self*)

property lower_bound(*self*)
Get or set the lower bound

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

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

property upper_bound(*self*)
Get or set the upper bound

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

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

property bounds(*self*)
Get or set the bounds directly from a tuple

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

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

property flux(*self*)
The flux value in the most recent solution.

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

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

Raises

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

Examples

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

property reduced_cost (*self*)

The reduced cost in the most recent solution.

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

Warning:

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

Raises

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

Examples

```
>>> import cobra.test
>>> model = cobra.test.create_test_model("textbook")
>>> solution = model.optimize()
>>> model.reactions.PFK.reduced_cost
-8.673617379884035e-18
>>> solution.reduced_costs.PFK
-8.6736173798840355e-18
```

property metabolites (*self*)

property genes (*self*)

property gene_reaction_rule (*self*)

property gene_name_reaction_rule (*self*)

Display gene_reaction_rule with names intead.

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

property functional (*self*)

All required enzymes for reaction are functional.

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

Return type bool

property x(*self*)
The flux through the reaction in the most recent solution.

Flux values are computed from the primal values of the variables in the solution.

property y(*self*)
The reduced cost of the reaction in the most recent solution.

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

property reversibility(*self*)
Whether the reaction can proceed in both directions (reversible)

This is computed from the current upper and lower bounds.

property boundary(*self*)
Whether or not this reaction is an exchange reaction.

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

property model(*self*)
returns the model the reaction is a part of

_update_awareness(*self*)
Make sure all metabolites and genes that are associated with this reaction are aware of it.

remove_from_model(*self*, *remove_orphans=False*)
Removes the reaction from a model.

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

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

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

delete(*self*, *remove_orphans=False*)
Removes the reaction from a model.

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

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

Deprecated, use *reaction.remove_from_model* instead.

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

__setstate__(*self*, *state*)
Probably not necessary to set _model as the cobra.Model that contains self sets the _model attribute for all metabolites and genes in the reaction.

However, to increase performance speed we do want to let the metabolite and gene know that they are employed in this reaction

copy(*self*)
Copy a reaction

The referenced metabolites and genes are also copied.

__add__(*self*, *other*)
Add two reactions

The stoichiometry will be the combined stoichiometry of the two reactions, and the gene reaction rule will be both rules combined by an and. All other attributes (i.e. reaction bounds) will match those of the first reaction

__iadd__(*self*, *other*)

__sub__(*self*, *other*)

__isub__(*self*, *other*)

__imul__ (*self*, *coefficient*)
Scale coefficients in a reaction by a given value

E.g. A -> B becomes 2A -> 2B.

If coefficient is less than zero, the reaction is reversed and the bounds are swapped.

__mul__ (*self*, *coefficient*)

property reactants (*self*)
Return a list of reactants for the reaction.

property products (*self*)
Return a list of products for the reaction

get_coefficient (*self*, *metabolite_id*)
Return the stoichiometric coefficient of a metabolite.

Parameters **metabolite_id** (*str or cobra.Metabolite*) –

get_coefficients (*self*, *metabolite_ids*)
Return the stoichiometric coefficients for a list of metabolites.

Parameters **metabolite_ids** (*iterable*) – Containing str or ``cobra.Metabolite``s.

add_metabolites (*self*, *metabolites_to_add*, *combine=True*, *reversibly=True*)
Add metabolites and stoichiometric coefficients to the reaction. If the final coefficient for a metabolite is 0 then it is removed from the reaction.

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

Parameters

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

subtract_metabolites (*self*, *metabolites*, *combine=True*, *reversibly=True*)
Subtract metabolites from a reaction.

That means add the metabolites with -1*coefficient. If the final coefficient for a metabolite is 0 then the metabolite is removed from the reaction.

Notes

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

Parameters

- **metabolites** (*dict*) – Dictionary where the keys are of class Metabolite and the values are the coefficients. These metabolites will be added to the reaction.
- **combine** (*bool*) – Describes behavior a metabolite already exists in the reaction. True causes the coefficients to be added. False causes the coefficient to be replaced.

- **reversibly** (*bool*) – Whether to add the change to the context to make the change reversibly or not (primarily intended for internal use).

property reaction (*self*)
Human readable reaction string

build_reaction_string (*self*, *use_metabolite_names=False*)
Generate a human readable reaction string

check_mass_balance (*self*)
Compute mass and charge balance for the reaction

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

property compartments (*self*)
lists compartments the metabolites are in

get_compartments (*self*)
lists compartments the metabolites are in

_associate_gene (*self*, *cobra_gene*)
Associates a cobra.Gene object with a cobra.Reaction.

Parameters **cobra_gene** (*cobra.core.Gene.Gene*) –

_dissociate_gene (*self*, *cobra_gene*)
Dissociates a cobra.Gene object with a cobra.Reaction.

Parameters **cobra_gene** (*cobra.core.Gene.Gene*) –

knock_out (*self*)
Knockout reaction by setting its bounds to zero.

build_reaction_from_string (*self*, *reaction_str*, *verbose=True*, *fwd_arrow=None*, *rev_arrow=None*, *reversible_arrow=None*, *term_split='+'*)
Builds reaction from reaction equation reaction_str using parser

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

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

Parameters

- **reaction_str** (*string*) – a string containing a reaction formula (equation)
- **verbose** (*bool*) – setting verbosity of function
- **fwd_arrow** (*re.compile*) – for forward irreversible reaction arrows
- **rev_arrow** (*re.compile*) – for backward irreversible reaction arrows
- **reversible_arrow** (*re.compile*) – for reversible reaction arrows
- **term_split** (*string*) – dividing individual metabolite entries

summary (*self*, *solution=None*, *threshold=0.01*, *fva=None*, *names=False*, *float_format='{:.3g}'.format*)
Create a summary of the producing and consuming fluxes of the reaction.

Parameters

- **solution** (*cobra.Solution, optional*) – A previous model solution to use for generating the summary. If None, the summary method will resolve the model. Note that the solution object must match the model, i.e., changes to the model such as changed bounds, added or removed reactions are not taken into account by this method (default None).

- **threshold** (*float, optional*) – Threshold below which fluxes are not reported. May not be smaller than the model tolerance (default 0.01).
- **fva** (*pandas.DataFrame or float, optional*) – Whether or not to include flux variability analysis in the output. If given, fva should either be a previous FVA solution matching the model or a float between 0 and 1 representing the fraction of the optimum objective to be searched (default None).
- **names** (*bool, optional*) – Emit reaction and metabolite names rather than identifiers (default False).
- **float_format** (*callable, optional*) – Format string for floats (default '{:3G}'.format).

Returns

Return type cobra.ReactionSummary

See also:

Metabolite.summary(), *Model.summary()*

__str__(*self*)
Return str(self).

_repr_html_(*self*)

class cobra.**Solution**(*objective_value*, *status*, *fluxes*, *reduced_costs=None*, *shadow_prices=None*, ***kwargs*)
Bases: object

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

Notes

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

objective_value
The (optimal) value for the objective function.

Type float

status
The solver status related to the solution.

Type str

fluxes
Contains the reaction fluxes (primal values of variables).

Type pandas.Series

reduced_costs
Contains reaction reduced costs (dual values of variables).

Type pandas.Series

shadow_prices
Contains metabolite shadow prices (dual values of constraints).

Type pandas.Series

get_primal_by_id

__repr__(*self*)
String representation of the solution instance.

_repr_html_(*self*)

__getitem__(*self*, *reaction_id*)
Return the flux of a reaction.

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

to_frame(*self*)
Return the fluxes and reduced costs as a data frame

class cobra.**Species**(*id=None*, *name=None*)
Bases: *cobra.core.object.Object*

Species is a class for holding information regarding a chemical Species

Parameters

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

property reactions(*self*)

__getstate__(*self*)
Remove the references to container reactions when serializing to avoid problems associated with recursion.

copy(*self*)
When copying a reaction, it is necessary to deepcopy the components so the list references aren't carried over.

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

This should be fixed with self.__deepcopy__ if possible

property model(*self*)

cobra.**show_versions**()
Print dependency information.

cobra.**__version__ = 0.17.1**

17.2 test_room

Test functionalities of ROOM.

17.2.1 Module Contents

Functions

test_room_sanity(model, all_solvers)	Test optimization criterion and optimality for ROOM.
test_linear_room_sanity(model, all_solvers)	Test optimization criterion and optimality for linear ROOM.

test_room.**test_room_sanity**(*model*, *all_solvers*)
Test optimization criterion and optimality for ROOM.

test_room.**test_linear_room_sanity**(*model*, *all_solvers*)
Test optimization criterion and optimality for linear ROOM.

17.3 test_geometric

Test functionalities of Geometric FBA.

17.3.1 Module Contents

Functions

geometric_fba_model()	Generate geometric FBA model as described in[1]
test_geometric_fba_benchmark(model, benchmark, all_solvers)	Benchmark geometric_fba.
test_geometric_fba(geometric_fba_model, all_solvers)	Test geometric_fba.

test_geometric.**geometric_fba_model**()
Generate geometric FBA model as described in[1]

References

test_geometric.**test_geometric_fba_benchmark**(*model*, *benchmark*, *all_solvers*)
Benchmark geometric_fba.

test_geometric.**test_geometric_fba**(*geometric_fba_model*, *all_solvers*)
Test geometric_fba.

17.4 test_parsimonious

Test functionalities of pFBA.

17.4.1 Module Contents

Functions

test_pfba_benchmark(large_model, benchmark, all_solvers)	Benchmark pFBA functionality.
test_pfba(model, all_solvers)	Test pFBA functionality.

test_parsimonious.**test_pfba_benchmark**(*large_model*, *benchmark*, *all_solvers*)
Benchmark pFBA functionality.

test_parsimonious.**test_pfba**(*model*, *all_solvers*)
Test pFBA functionality.

[1] Smallbone, Kieran & Simeonidis, Vangelis. (2009). Flux balance analysis: A geometric perspective. Journal of theoretical biology.258. 311-5. 10.1016/j.jtbi.2009.01.027.

17.5 test_reaction

Test _assess functions in reaction.py

17.5.1 Module Contents

Functions

test_assess(model, all_solvers)	Test assess functions.

test_reaction.**test_assess**(*model*, *all_solvers*)
Test assess functions.

17.6 test_gapfilling

Test functionalities of gapfilling.

17.6.1 Module Contents

Functions

test_gapfilling(salmonella)	Test Gapfilling.

test_gapfilling.**test_gapfilling**(*salmonella*)
Test Gapfilling.

17.7 test_variability

Test functionalities of Flux Variability Analysis.

17.7.1 Module Contents

Functions

test_flux_variability_benchmark(large_model, benchmark, all_solvers)	Benchmark FVA.
test_pfba_flux_variability(model, pfba_fva_results, fva_results, all_solvers)	Test FVA using pFBA.
test_loopless_pfba_fva(model)	
test_flux_variability(model, fva_results, all_solvers)	Test FVA.
test_parallel_flux_variability(model, fva_results, all_solvers)	Test parallel FVA.
test_flux_variability_loopless_benchmark(model, benchmark, all_solvers)	Benchmark loopless FVA.
test_flux_variability_loopless(model, all_solvers)	Test loopless FVA.

Continued on next page

Table 100 – continued from previous page

test_fva_data_frame(model)	Test DataFrame obtained from FVA.
test_fva_infeasible(model)	Test FVA infeasibility.
test_fva_minimization(model)	Test minimization using FVA.
test_find_blocked_reactions_solver_none(model)	Test find_blocked_reactions() [no specific solver].
test_essential_genes(model)	Test find_essential_genes().
test_essential_reactions(model)	Test find_blocked_reactions().
test_find_blocked_reactions(model, all_solvers)	Test find_blocked_reactions().

test_variability.**test_flux_variability_benchmark**(*large_model*, *benchmark*, *all_solvers*)
Benchmark FVA.

test_variability.**test_pfba_flux_variability**(*model*, *pfba_fva_results*, *fva_results*, *all_solvers*)
Test FVA using pFBA.

test_variability.**test_loopless_pfba_fva**(*model*)

test_variability.**test_flux_variability**(*model*, *fva_results*, *all_solvers*)
Test FVA.

test_variability.**test_parallel_flux_variability**(*model*, *fva_results*, *all_solvers*)
Test parallel FVA.

test_variability.**test_flux_variability_loopless_benchmark**(*model*, *benchmark*, *all_solvers*)
Benchmark loopless FVA.

test_variability.**test_flux_variability_loopless**(*model*, *all_solvers*)
Test loopless FVA.

test_variability.**test_fva_data_frame**(*model*)
Test DataFrame obtained from FVA.

test_variability.**test_fva_infeasible**(*model*)
Test FVA infeasibility.

test_variability.**test_fva_minimization**(*model*)
Test minimization using FVA.

test_variability.**test_find_blocked_reactions_solver_none**(*model*)
Test find_blocked_reactions() [no specific solver].

test_variability.**test_essential_genes**(*model*)
Test find_essential_genes().

test_variability.**test_essential_reactions**(*model*)
Test find_blocked_reactions().

test_variability.**test_find_blocked_reactions**(*model*, *all_solvers*)
Test find_blocked_reactions().

17.8 test_fastcc

Test functionalities of FASTCC.

17.8.1 Module Contents

Functions

figure1_model()	Generate a toy model as described in[1] figure 1.
opposing_model()	Generate a toy model with opposing reversible reactions.
test_fastcc_benchmark(model, benchmark, all_solvers)	Benchmark fastcc.
test_figure1(figure1_model, all_solvers)	Test fastcc.
test_opposing(opposing_model, all_solvers)	Test fastcc.
test_fastcc_against_fva_nonblocked_rxns(model, all_solvers)	Test non-blocked reactions obtained by FASTCC against FVA.

test_fastcc.**figure1_model**()
Generate a toy model as described in[1] figure 1.

References

test_fastcc.**opposing_model**()
Generate a toy model with opposing reversible reactions.

This toy model ensures that two opposing reversible reactions do not appear as blocked.

test_fastcc.**test_fastcc_benchmark**(*model*, *benchmark*, *all_solvers*)
Benchmark fastcc.

test_fastcc.**test_figure1**(*figure1_model*, *all_solvers*)
Test fastcc.

test_fastcc.**test_opposing**(*opposing_model*, *all_solvers*)
Test fastcc.

test_fastcc.**test_fastcc_against_fva_nonblocked_rxns**(*model*, *all_solvers*)
Test non-blocked reactions obtained by FASTCC against FVA.

17.9 test_moma

Test functionalities of MOMA.

[1] Vlassis N, Pacheco MP, Sauter T (2014) Fast Reconstruction of Compact Context-Specific Metabolic Network Models. PLoS Comput Biol 10(1): e1003424. doi:10.1371/journal.pcbi.1003424

17.9.1 Module Contents

Functions

test_moma_sanity(model, qp_solvers)	Test optimization criterion and optimality for MOMA.
test_linear_moma_sanity(model, all_solvers)	Test optimization criterion and optimality for linear MOMA.

test_moma.**test_moma_sanity**(*model*, *qp_solvers*)
Test optimization criterion and optimality for MOMA.

test_moma.**test_linear_moma_sanity**(*model*, *all_solvers*)
Test optimization criterion and optimality for linear MOMA.

17.10 conftest

Define module level fixtures.

17.10.1 Module Contents

Functions

achr(model)	Return ACHRSampler instance for tests.

conftest.**achr**(*model*)
Return ACHRSampler instance for tests.

17.11 test_loopless

Test functionalities of loopless.py

17.11.1 Module Contents

Functions

construct_ll_test_model()	Construct test model.
ll_test_model(request)	Return test model set with different solvers.
test_loopless_benchmark_before(benchmark)	Benchmark initial condition.
test_loopless_benchmark_after(benchmark)	Benchmark final condition.
test_loopless_solution(ll_test_model)	Test loopless_solution().
test_loopless_solution_fluxes(model)	Test fluxes of loopless_solution()
test_add_loopless(ll_test_model)	Test add_loopless().

test_loopless.**construct_ll_test_model**()
Construct test model.

test_loopless.**ll_test_model**(*request*)
Return test model set with different solvers.

test_loopless.**test_loopless_benchmark_before**(*benchmark*)
Benchmark initial condition.

test_loopless.**test_loopless_benchmark_after**(*benchmark*)
Benchmark final condition.

test_loopless.**test_loopless_solution**(*ll_test_model*)
Test loopless_solution().

test_loopless.**test_loopless_solution_fluxes**(*model*)
Test fluxes of loopless_solution()

test_loopless.**test_add_loopless**(*ll_test_model*)
Test add_loopless().

17.12 test_deletion

Test functionalities of reaction and gene deletions.

17.12.1 Module Contents

Functions

test_single_gene_deletion_fba_benchmark(model, benchmark, all_solvers)	Benchmark single gene deletion using FBA.
test_single_gene_deletion_fba(model, all_solvers)	Test single gene deletion using FBA.
test_single_gene_deletion_moma_benchmark(model, benchmark, qp_solvers)	Benchmark single gene deletion using MOMA.
test_single_gene_deletion_moma(model, qp_solvers)	Test single gene deletion using MOMA.
test_single_gene_deletion_moma_reference(model, qp_solvers)	Test single gene deletion using MOMA (reference solution).
test_single_gene_deletion_linear_moma_benchmark(model, benchmark, all_solvers)	Benchmark single gene deletion using linear MOMA.
test_single_gene_deletion_linear_moma(model, all_solvers)	Test single gene deletion using linear MOMA (reference solution).
test_single_gene_deletion_room_benchmark(model, benchmark, all_solvers)	Benchmark single gene deletion using ROOM.
test_single_gene_deletion_linear_room_benchmark(model, benchmark, all_solvers)	Benchmark single gene deletion using linear ROOM.
test_single_reaction_deletion_benchmark(model, benchmark, all_solvers)	Benchmark single reaction deletion.
test_single_reaction_deletion(model, all_solvers)	Test single reaction deletion.
test_single_reaction_deletion_room(room_model, room_solution, all_solvers)	Test single reaction deletion using ROOM.
test_single_reaction_deletion_linear_room(room_model, room_solution, all_solvers)	Test single reaction deletion using linear ROOM.
test_double_gene_deletion_benchmark(large_model, benchmark)	Benchmark double gene deletion.
test_double_gene_deletion(model)	Test double gene deletion.
test_double_reaction_deletion_benchmark(large_model, benchmark)	Benchmark double reaction deletion.
test_double_reaction_deletion(model)	Test double reaction deletion.

test_deletion.**test_single_gene_deletion_fba_benchmark**(*model*, *benchmark*, *all_solvers*)
Benchmark single gene deletion using FBA.

test_deletion.**test_single_gene_deletion_fba**(*model*, *all_solvers*)
Test single gene deletion using FBA.

test_deletion.**test_single_gene_deletion_moma_benchmark**(*model*, *benchmark*, *qp_solvers*)
Benchmark single gene deletion using MOMA.

test_deletion.**test_single_gene_deletion_moma**(*model*, *qp_solvers*)
Test single gene deletion using MOMA.

test_deletion.**test_single_gene_deletion_moma_reference**(*model*, *qp_solvers*)
Test single gene deletion using MOMA (reference solution).

test_deletion.**test_single_gene_deletion_linear_moma_benchmark**(*model*, *benchmark*, *all_solvers*)
Benchmark single gene deletion using linear MOMA.

test_deletion.**test_single_gene_deletion_linear_moma**(*model*, *all_solvers*)
Test single gene deletion using linear MOMA (reference solution).

test_deletion.**test_single_gene_deletion_room_benchmark**(*model*, *benchmark*, *all_solvers*)
Benchmark single gene deletion using ROOM.

test_deletion.**test_single_gene_deletion_linear_room_benchmark**(*model*, *benchmark*, *all_solvers*)
Benchmark single gene deletion using linear ROOM.

test_deletion.**test_single_reaction_deletion_benchmark**(*model*, *benchmark*, *all_solvers*)
Benchmark single reaction deletion.

test_deletion.**test_single_reaction_deletion**(*model*, *all_solvers*)
Test single reaction deletion.

test_deletion.**test_single_reaction_deletion_room**(*room_model*, *room_solution*, *all_solvers*)
Test single reaction deletion using ROOM.

test_deletion.**test_single_reaction_deletion_linear_room**(*room_model*, *room_solution*, *all_solvers*)
Test single reaction deletion using linear ROOM.

test_deletion.**test_double_gene_deletion_benchmark**(*large_model*, *benchmark*)
Benchmark double gene deletion.

test_deletion.**test_double_gene_deletion**(*model*)
Test double gene deletion.

test_deletion.**test_double_reaction_deletion_benchmark**(*large_model*, *benchmark*)
Benchmark double reaction deletion.

test_deletion.**test_double_reaction_deletion**(*model*)
Test double reaction deletion.

17.13 test_phenotype_phase_plane

Test functionalities of Phenotype Phase Plane Analysis.

17.13.1 Module Contents

Functions

test_envelope_one(model)	Test flux of production envelope.
test_envelope_multi_reaction_objective(model)	Test production of multiple objectives.
test_multi_variable_envelope(model, variables, num)	Test production of envelope (multiple variable).
test_envelope_two(model)	Test production of envelope.

test_phenotype_phase_plane.**test_envelope_one**(*model*)
Test flux of production envelope.

test_phenotype_phase_plane.**test_envelope_multi_reaction_objective**(*model*)
Test production of multiple objectives.

test_phenotype_phase_plane.**test_multi_variable_envelope**(*model*, *variables*, *num*)
Test production of envelope (multiple variable).

test_phenotype_phase_plane.**test_envelope_two**(*model*)
Test production of envelope.

17.14 update_pickles

17.14.1 Module Contents

update_pickles.**config**

update_pickles.**solver = glpk**

update_pickles.**ecoli_model**

update_pickles.**salmonella**

update_pickles.**gene_names**

update_pickles.**name**

update_pickles.**media_compositions**

update_pickles.**textbook**

update_pickles.**mini**

update_pickles.**compartments**

update_pickles.**upper_bound**

update_pickles.**objective = ['PFK', 'ATPM']**

update_pickles.**r**

update_pickles.**gene_reaction_rule**

update_pickles.**reaction**

update_pickles.**gene_reaction_rule**

update_pickles.**upper_bound**

update_pickles.**lower_bound**

update_pickles.**tg**

update_pickles.**raven**

update_pickles.**fva_result**

update_pickles.**clean_result**

update_pickles.**fva_result**

update_pickles.**clean_result**

update_pickles.**solution**

17.15 test_util

Test functions of util.py

17.15.1 Module Contents

Functions

test_show_versions(capsys)	

test_util.**test_show_versions**(*capsys*)

17.16 test_array

Test functions of array.py

17.16.1 Module Contents

Functions

test_dense_matrix(model)	
test_sparse_matrix(model)	

test_array.**scipy**

test_array.**test_dense_matrix**(*model*)

test_array.**test_sparse_matrix**(*model*)

17.17 test_solver

Test functions of solver.py

17.17.1 Module Contents

Functions

test_solver_list()	
test_interface_str()	
test_solver_name()	
test_choose_solver(model)	
test_linear_reaction_coefficients(model)	
test_fail_non_linear_reaction_coefficients(model, solver)	
test_add_remove(model)	
test_add_remove_in_context(model)	
test_absolute_expression(model)	
test_fix_objective_as_constraint(solver, model)	
test_fix_objective_as_constraint_minimize(model, solver)	
test_add_lp_feasibility(model, solver)	
test_add_lexicographic_constraints(model, solver)	
test_time_limit(large_model)	

test_solver.**stable_optlang = ['glpk', 'cplex', 'gurobi']**

test_solver.**optlang_solvers**

test_solver.**test_solver_list**()

test_solver.**test_interface_str**()

test_solver.**test_solver_name**()

test_solver.**test_choose_solver**(*model*)

test_solver.**test_linear_reaction_coefficients**(*model*)

test_solver.**test_fail_non_linear_reaction_coefficients**(*model*, *solver*)

test_solver.**test_add_remove**(*model*)

test_solver.**test_add_remove_in_context**(*model*)

test_solver.**test_absolute_expression**(*model*)

test_solver.**test_fix_objective_as_constraint**(*solver*, *model*)

test_solver.**test_fix_objective_as_constraint_minimize**(*model*, *solver*)

test_solver.**test_add_lp_feasibility**(*model*, *solver*)

test_solver.**test_add_lexicographic_constraints**(*model*, *solver*)

test_solver.**test_time_limit**(*large_model*)

17.18 test_optgp

Test functionalities of OptGPSampler.

17.18.1 Module Contents

Functions

optgp(model)	Return OptGPSampler instance for tests.
test_optgp_init_benchmark(model, benchmark)	Benchmark inital OptGP sampling.
test_optgp_sample_benchmark(optgp, benchmark)	Benchmark OptGP sampling.
test_sampling(optgp)	Test sampling.
test_batch_sampling(optgp)	Test batch sampling.
test_variables_samples(achr, optgp)	Test variable samples.
test_reproject(optgp)	Test reprojection of sampling.

test_optgp.**optgp**(*model*)
Return OptGPSampler instance for tests.

test_optgp.**test_optgp_init_benchmark**(*model*, *benchmark*)
Benchmark inital OptGP sampling.

test_optgp.**test_optgp_sample_benchmark**(*optgp*, *benchmark*)
Benchmark OptGP sampling.

test_optgp.**test_sampling**(*optgp*)
Test sampling.

test_optgp.**test_batch_sampling**(*optgp*)
Test batch sampling.

test_optgp.**test_variables_samples**(*achr*, *optgp*)
Test variable samples.

test_optgp.**test_reproject**(*optgp*)
Test reprojection of sampling.

17.19 test_achr

Test functionalities of ACHRSampler.

17.19.1 Module Contents

Functions

test_achr_init_benchmark(model, benchmark)	Benchmark inital ACHR sampling.
test_achr_sample_benchmark(achr, benchmark)	Benchmark ACHR sampling.
test_validate_wrong_sample(achr, model)	Test sample correctness.
test_sampling(achr)	Test sampling.
test_batch_sampling(achr)	Test batch sampling.

Continued on next page

Table 111 – continued from previous page

test_variables_samples(achr)	Test variable samples.

test_achr.**test_achr_init_benchmark**(*model*, *benchmark*)
Benchmark inital ACHR sampling.

test_achr.**test_achr_sample_benchmark**(*achr*, *benchmark*)
Benchmark ACHR sampling.

test_achr.**test_validate_wrong_sample**(*achr*, *model*)
Test sample correctness.

test_achr.**test_sampling**(*achr*)
Test sampling.

test_achr.**test_batch_sampling**(*achr*)
Test batch sampling.

test_achr.**test_variables_samples**(*achr*)
Test variable samples.

17.20 test_sampling

Test functionalities of flux sampling methods.

17.20.1 Module Contents

Functions

test_single_achr(model)	Test ACHR sampling (one sample).
test_single_optgp(model)	Test OptGP sampling (one sample).
test_multi_optgp(model)	Test OptGP sampling (multi sample).
test_wrong_method(model)	Test method intake sanity.
test_fixed_seed(model)	Test result of fixed seed for sampling.
test_equality_constraint(model)	Test equality constraint.
test_inequality_constraint(model)	Test inequality constraint.
test_inhomogeneous_sanity(model)	Test whether inhomogeneous sampling gives approximately the same
test_complicated_model()	Test a complicated model.
test_single_point_space(model)	Test the reduction of the sampling space to one point.

test_sampling.**test_single_achr**(*model*)
Test ACHR sampling (one sample).

test_sampling.**test_single_optgp**(*model*)
Test OptGP sampling (one sample).

test_sampling.**test_multi_optgp**(*model*)
Test OptGP sampling (multi sample).

test_sampling.**test_wrong_method**(*model*)
Test method intake sanity.

test_sampling.**test_fixed_seed**(*model*)
Test result of fixed seed for sampling.

test_sampling.**test_equality_constraint**(*model*)
Test equality constraint.

test_sampling.**test_inequality_constraint**(*model*)
Test inequality constraint.

test_sampling.**test_inhomogeneous_sanity**(*model*)
Test whether inhomogeneous sampling gives approximately the same standard deviation as a homogeneous version.

test_sampling.**test_complicated_model**()
Test a complicated model.

Difficult model since the online mean calculation is numerically unstable so many samples weakly violate the equality constraints.

test_sampling.**test_single_point_space**(*model*)
Test the reduction of the sampling space to one point.

CHAPTER

EIGHTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

t

u

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z