sybil – Efficient Constrained Based Modelling in R

Gabriel Gelius-Dietrich

October 10, 2012

Contents

1 Introduction

The R-package *SyBiL* is a Systems Biology Library for R, implementing algorithms for constraint based analysis of metabolic networks. Among other functions, *SyBiL* currently provides efficient methods for flux-balance analysis (FBA), minimization of metabolic adjustment (MOMA) and regulatory on/off minimization (ROOM).

The package *SyBiL* makes use of the sparse matrix implementation in the R-package *Matrix* available from CRAN[1], or, from R version 2.14.0 on, is already included in your R installation.

2 Installation

The package *SyBiL* itself depends on an existing installation of the package *Matrix*. In order to run optimizations, at least one of the following additional R-packages and the corresponding libraries are required: *glpkAPI*, *cplexAPI*, *clpAPI* or *lpSolveAPI*. These packages are also available from CRAN[1].

3 Input files

Input files for *SyBiL* are text files containing a description of the metabolic model to analyse. These descriptions are basically lists of reactions. Two fundamentally different types of text files are supported: i) in tabular form (section 3.1), or ii) in SBML format (section 3.2).

3.1 Tabular form

Models in tabular form can be read using the function `readTSVmod` and written using the function `modelorg2tsv`. Each metabolic model description consits of three tables:

1. A model description, containing a model name, the compartments of the model and so on (section 3.1.2).

[1] `http://www.r-project.org/`

2. A list of all metabolites (section 3.1.3).

3. A list of all reactions (section 3.1.4).

A model must contain at least a list of all reactions. All other tables are optional. The tables contain columns storing the required data. Some of these columns are optional, but if a certain table exists, there must be a minimal set of columns. The column names (the first line in each file) are used as keywords and cannot be changed.

3.1.1 Field and entry delimiter

There are two important variables in connection with text based tables: The fields (columns) of the tables are separated by a variable `fielddelim`. If a single entry of a field contains a list of entries, they are separated by a variable `entrydelim`. The default values are given in the table below.

`fielddelim`	`\t`
`entrydelim`	`,␣`

The default behavior is, that the columns of each table are separated by a single `tab` character. If a column entry helds more than one entry, they are separated by a comma folowed by a single whitespace ␣ (not a tab!).

3.1.2 Model description

Every column in this table can have at most one entry, meaning each entry will be a single character string. If a model description file is used, there should be at least the two columns `name` and `id`. If they are missing—or if no model description file is used—they will be set to the file name of the reaction list, which must be there (any file name extension and the string `_react` at the end of the file name, will be removed).

name A single character string giving the model name. If this field is empty, the filename of the reaction list is used.

id A single character string giving the model id. If this field is empty, the filename of the reaction list is used.

description A single character string giving a model description (optional).

compartment A single character string containing the compartment names. The names must be separated by `fielddelim` (optional).

abbreviation A single character string containing the compartment abbreviations. The abbreviations must be in square brackets and separated by `fielddelim` as mentioned above (optional).

Nmetabolites A single integer giving the number of metabolites in the model (optional).

Nreactions A single integer giving the number of reactions in the model (optional).

Ngenes A single integer giving the number of independent genes in the model (optional).

Nnnz A single integer giving the number of non-zero elements in the stoichiometric matrix of the model (optional).

The file `Ec_core_desc.tsv` (in `extdata/`) contains an exemplarily table for the core energy metabolism of *E. coli* [Palsson, 2006, Orth et al., 2010a].

3.1.3 Metabolite list

This table is used in order to match metabolite id's given in the list of reactions to long metabolite names. This table is optional, but if it is used, the columns `abbreviation` and `name` should not be empty.

abbreviation A list of single character strings containing the metabolite abbreviations.

name A list of single character strings containing the metabolite names.

compartment A list of character strings containing the metabolite compartment names. Each entry can contain more than one compartment name, separated by `fielddelim` (optional, currently unused).

The file `Ec_core_met.tsv` (in `extdata/`) contains an exemplarily table for the core energy metabolism of *E. coli* [Palsson, 2006, Orth et al., 2010a].

3.1.4 Reaction list

This table contains the reaction equations used in the metabolic network.

abbreviation A list of single character strings containing the reaction abbreviations (optional, if empty, a warning will be produced). Entries in the field abbreviation are used as reaction id's, so they must be unique. If they are missing, they will be set to `vi`, $i \in \{1, \ldots, n\}$ $\forall i$ with n beeing the total number of reactions).

name A list of single character strings containing the reaction names (optional, if empty, the reaction id's (abbreviations) are used as reaction names.

equation A list of single character strings containing the reaction equation. See section 3.1.5 for a description of reaction equation strings.

reversible A list of single character strings making a particular reaction reversible or not. If the entry is `Reversible` or `TRUE`, the reaction is considered as reversible, otherwise not. If this column is not used, the arrow symbol of the reaction string is used (optional).

compartment A list of character strings containing the compartment names in which the current reaction takes place. Each entry can contain more than one name, separated by `fielddelim` (optional, currently unused).

lowbnd A list of numeric values containing the lower bounds of the reaction rates. If not set, zero is used for an irreversible reaction and the value of `def_bnd * -1` for a reversible reaction. See documentation of the function `readTSVmod` for the argument `def_bnd` (optional).

uppbnd A list of numeric values containing the upper bounds of the reaction rates. If not set, the value of `def_bnd` is used. See documentation of the function `readTSVmod` for the argument `def_bnd` (optional).

obj_coef A list of numeric values containing objective values for each reaction (optional, if missing, zero is used).

rule A list of single character strings containing the gene to reaction associations (optional).

subsystem A list of character strings containing the reaction subsystems. Each reaction can belong to more than one subsystem. The entries are separated by `fielddelim` (optional).

The file `Ec_core_react.tsv` (in `extdata/`) contains an exemplarily table for the core energy metabolism of *E. coli* [Palsson, 2006, Orth et al., 2010a].

3.1.5 How to write a reaction equation string

Any reaction string can be written without whitespaces. They are not required but showed here, in order to make the string more human readable.

Compartment Flag Each reaction string can start with a compartment flag in square brackets followed by a colon. The compartment flag here gives the location of all metabolites appearing in the reaction.

```
[c] :
```

The compartment flag can consist of more than one letter and—if used—must be an element of the field `abbreviation` in the model description. The letter `b` is reserved for boundary metabolites, which can be transported inside the system (those metabolites are only used in closed systems and will be removed during file parsing).

If the reaction string does not start with a compartment flag, the flag can be appended (without whitespace) to each metabolite id (e. g. for transport reactions):

```
h2o[e] <==> h2o[c]
```

If no compartment flag is found, it is set to `[unknown]`.

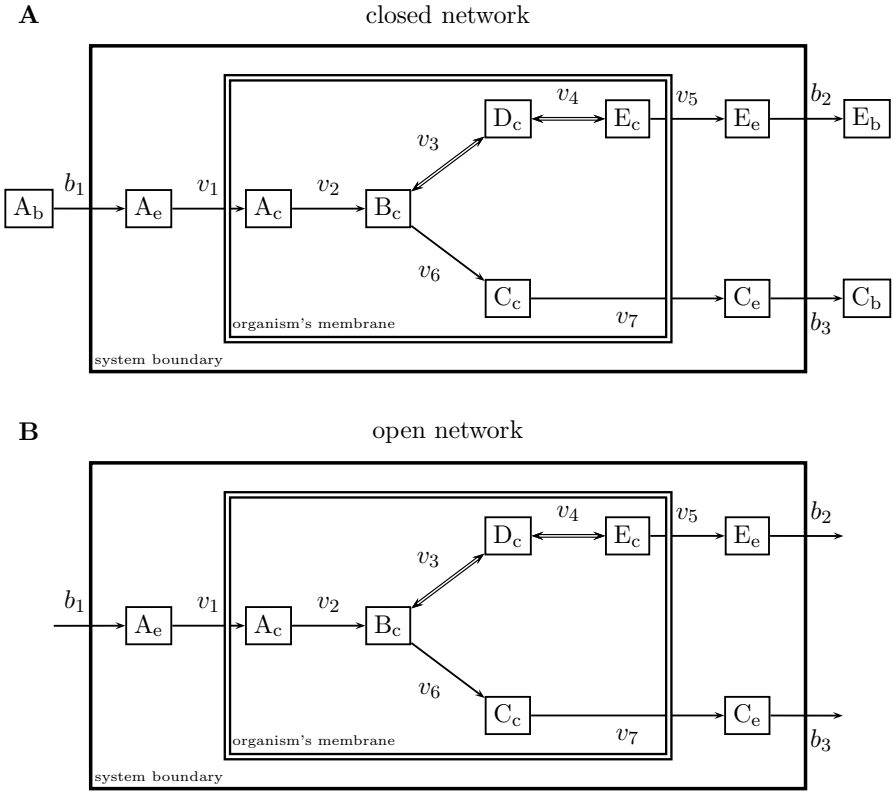


Figure 1: Simple example network. A) showing a closed network, B) an open network. Capital letters are used as metabolite id's, lower case letters are used as compartment id's: b: boundary metabolite, c: cytosol and e: external metabolite. Internal reactions are named $v_{1:7}$, transport reactions $b_{1:3}$. Reactions v_3 and v_4 are reversible, all others are irreversible.

Reaction Arrow All reactions must be written in the direction educt to product, so that all metabolites left of the reaction arrow are considered as educts, all metabolites on the right of the reaction arrow are products.

The reaction arrow itself consits of one or more `=` or `-` symbols. The last symbol must be a `>`. If a reaction arrow starts with `<`, it is taken as reversible, if the field `reversible` in the reaction list is empty. Each reaction must contain exactly one reaction arrow.

Stoichiometric Coefficients Stoichiometric coefficients must be in round brackets in front of the corresponding metabolite:

```
(2) h[c] + (0.5) o2[c] + q8h2[c] --> h2o[c] + (2) h[e] + q8[c]
```

Putting the stoichiometric coefficient in brackets makes it possible for the metabolite id to start with a number.

Examples A minimal reaction list without compartment flags for figure 1B (open network):

```
equation
A  --> B
B <==> D
D <==> E
B  --> C
   --> A
C  -->
E  -->
```

The same as above including compartment flags and external metabolites and all transport reactions for figure 1A (closed network). The reactions which take place in only one compartment (do not include a transport of metabolites accross membranes) have their compartment flag at the beginning of the line (`[c]` in this example). For transport reactions all metabolites have their own compartment flag, e. g. in line 5 metabolite `A` is transported from compartment `[e]` (external) to compartment `[c]` (cytosol):

```
equation
[c]: A  --> B
[c]: B <==> D
[c]: D <==> E
[c]: B  --> C
A[e] --> A[c]
C[c] --> C[e]
E[c] --> E[e]
A[b] --> A[e]
C[e] --> C[b]
E[e] --> E[b]
```

The same as above including reaction id's for figure 1 (fields are separated by tabulators):

```
abbreviation    equation
v2    [c]: A  --> B
v3    [c]: B <==> D
v4    [c]: D <==> E
v6    [c]: B  --> C
v1    A[e] --> A[c]
v7    C[c] --> C[e]
v5    E[c] --> E[e]
b1    A[b] --> A[e]
b3    C[e] --> C[b]
b2    E[e] --> E[b]
```

3.2 SBML

In order to read model files written in SBML, the package *sybilSBML* is required (which is also available from available from the *SyBiL* homepage[2].

4 Usage

Load *SyBiL* in a running R session:

```
> library(sybil)
```

4.1 Documentation

Get a list of all functions provided with *SyBiL*:

```
> library(help = "sybil")
```

Get details of the usage of a particular function in *SyBiL* (e. g. `doubleGeneDel()`):

```
> help(doubleGeneDel)
```

Search through help files for a specific topic (e. g. "flux variablilty analysis"):

```
> help.search("flux variablilty analysis")
```

Open this vignette:

```
> vignette("sybil")
```

In the following, it is assumed, that package *glpkAPI* is installed additionally to *SyBiL*, thus GLPK is used as optimization software.

4.2 Reading a model in tabular form

The package *SyBiL* can read metabolic network models written in tabular form as described in section 3.1. A reconstruction of the central metabolism of *E. coli* [Orth et al., 2010a, Palsson, 2006] is included as an example dataset. The example dataset consists of three files:

1. `Ec_core_desc.tsv` containing the model description,
2. `Ec_core_met.tsv` containing the metabolite list and
3. `Ec_core_react.tsv` containing the reaction list.

These files are located in the directory `extdata/` in the package *SyBiL*. The exact location of the files can be retrieved with the `system.file()` command:

[2] `http://www.cs.uni-duesseldorf.de/AG/BI/Software/SyBiL`

```
> mp  <- system.file(package = "sybil", "extdata")
```

Now the model files und can be read in by using the comand `readTSVmod`:

```
> mod <- readTSVmod(prefix = "Ec_core", fpath = mp, quoteChar = "\"")
> mod

model name:             Ecoli_core_model
number of compartments  2
                        C_c
                        C_e
number of reactions:    95
number of metabolites:  72
number of unique genes: 137
objective function:     1 * Biomass_Ecoli_core_w_GAM
```

If the fields in the input files for `readTSVmod` are quoted, use the argument `quoteChar` The value of `quoteChar` is passed to the argument `quote` of the function `read.table`. Models (instances of class `modelorg`, see section 5.1) can be converted to files in tabular form with the command `modelorg2tsv`:

```
> modelorg2tsv(mod, prefix = "Ec_core")
```

Load the example dataset included in *SyBiL*.

```
> data(Ec_core)
```

The example model is a 'ready to use' model, it contains a biomass objective function and an uptake of glucose [Orth et al., 2010a, Palsson, 2006]. It is the same model as used in the text files before.

4.3 Flux-balance analysis

Perform flux-balance analysis (FBA).

```
> optimizeProb(Ec_core, alg = "fba")

$ok
[1] 0

$obj
[1] 0.8739215

$stat
[1] 5

$fluxes
```

```
 [1]  0.000000e+00  0.000000e+00  0.000000e+00  6.007250e+00  6.007250e+00
 [6]  0.000000e+00  0.000000e+00  5.064376e+00  0.000000e+00  0.000000e+00
[11]  8.390000e+00  4.551401e+01  8.739215e-01 -2.280983e+01  6.007250e+00
[16]  4.359899e+01  0.000000e+00  1.471614e+01  0.000000e+00  0.000000e+00
[21]  0.000000e+00  0.000000e+00  2.280983e+01  0.000000e+00  0.000000e+00
[26]  0.000000e+00  0.000000e+00 -1.000000e+01  0.000000e+00  0.000000e+00
[31]  1.753087e+01  2.917583e+01  0.000000e+00  0.000000e+00 -4.765319e+00
[36] -2.179949e+01 -3.214895e+00  0.000000e+00  0.000000e+00  7.477382e+00
[41]  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
[46]  5.064376e+00  0.000000e+00  4.959985e+00  1.602353e+01  1.000000e+01
[51]  2.234617e-01  0.000000e+00 -4.541857e+00  0.000000e+00  0.000000e+00
[56]  0.000000e+00  4.959985e+00 -2.917583e+01  6.007250e+00  0.000000e+00
[61]  0.000000e+00  0.000000e+00  0.000000e+00  5.064376e+00  0.000000e+00
[66]  0.000000e+00  3.853461e+01  0.000000e+00  4.765319e+00  2.179949e+01
[71]  9.282533e+00  7.477382e+00  0.000000e+00  4.860861e+00 -1.602353e+01
[76]  4.959985e+00 -1.471614e+01  3.214895e+00  2.504309e+00  0.000000e+00
[81]  0.000000e+00  0.000000e+00  1.758177e+00 -1.419950e-29  2.678482e+00
[86] -2.281503e+00  0.000000e+00  0.000000e+00  5.064376e+00 -5.064376e+00
[91]  1.496984e+00  0.000000e+00  1.496984e+00  1.181498e+00  7.477382e+00

$preP
[1] NA

$postP
[1] NA

$fldind
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
[51] 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
[76] 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
```

The function `optimizeProb` performs flux-balance analysis [Edwards et al., 2002, Orth et al., 2010b]. It returns a list containing the return value of the optimization process (`"ok"`), the solution status (`"stat"`), the value of the objective function after optimization (`"obj"`), the resulting flux distribution—the phenotype of the metabolic network—(`"fluxes"`; argument `fld` has to be `TRUE`)—and results of pre- and postprocessing commands if indicated (`"preP"` and `$postP`). Additionally, a vector of integers is returned (`"fldind"`). The flux value `fluxes[fldind[i]]` is the flux value of reaction `i` in the model (see section 4.4).

Perform FBA, return an object of class `optsol_optimizeProb` (extends class `optsol`, see section 5.2).

```
> (opt <- optimizeProb(Ec_core, alg = "fba", retOptSol = TRUE))
```

```
solver:                                     glpkAPI
method:                                     simplex
algorithm:                                  fba
number of variables:                        95
number of constraints:                      72
number of problems to solve:                1
number of successful solution processes:    1
```

The variable `opt` contains an object of class `optsol_optimizeProb`, a data structure storing all results of the optimization and providing methods to access the data. Retrieve the value of the objective function after optimization.

```
> lp_obj(opt)

[1] 0.8739215
```

Translate the return and status codes of the optimization software into human readable strings.

```
> checkOptSol(opt)

Return code:

Code    #       meaning
0       1       solution process was successful


Solution status:

Code    #       meaning
5       1       solution is optimal
```

Retrieve reduced costs after optimization.

```
> optimizeProb(Ec_core, MoreArgs = list(poCmd = list("getRedCosts")))

$ok
[1] 0

$obj
[1] 0.8739215

$stat
[1] 5

$fluxes
```

```
 [1]  0.000000e+00  0.000000e+00  0.000000e+00  6.007250e+00  6.007250e+00
 [6]  0.000000e+00  0.000000e+00  5.064376e+00  0.000000e+00  0.000000e+00
[11]  8.390000e+00  4.551401e+01  8.739215e-01 -2.280983e+01  6.007250e+00
[16]  4.359899e+01  0.000000e+00  1.471614e+01  0.000000e+00  0.000000e+00
[21]  0.000000e+00  0.000000e+00  2.280983e+01  0.000000e+00  0.000000e+00
[26]  0.000000e+00  0.000000e+00 -1.000000e+01  0.000000e+00  0.000000e+00
[31]  1.753087e+01  2.917583e+01  0.000000e+00  0.000000e+00 -4.765319e+00
[36] -2.179949e+01 -3.214895e+00  0.000000e+00  0.000000e+00  7.477382e+00
[41]  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
[46]  5.064376e+00  0.000000e+00  4.959985e+00  1.602353e+01  1.000000e+01
[51]  2.234617e-01  0.000000e+00 -4.541857e+00  0.000000e+00  0.000000e+00
[56]  0.000000e+00  4.959985e+00 -2.917583e+01  6.007250e+00  0.000000e+00
[61]  0.000000e+00  0.000000e+00  0.000000e+00  5.064376e+00  0.000000e+00
[66]  0.000000e+00  3.853461e+01  0.000000e+00  4.765319e+00  2.179949e+01
[71]  9.282533e+00  7.477382e+00  0.000000e+00  4.860861e+00 -1.602353e+01
[76]  4.959985e+00 -1.471614e+01  3.214895e+00  2.504309e+00  0.000000e+00
[81]  0.000000e+00  0.000000e+00  1.758177e+00 -1.419950e-29  2.678482e+00
[86] -2.281503e+00  0.000000e+00  0.000000e+00  5.064376e+00 -5.064376e+00
[91]  1.496984e+00  0.000000e+00  1.496984e+00  1.181498e+00  7.477382e+00

$preP
[1] NA

$postP
An object of class "ppProc"
Slot "cmd":
[[1]]
[1] "getRedCosts(LP_PROB)"


Slot "pa":
[[1]]
 [1]  0.000000000  0.000000000  0.000000000  0.000000000  0.000000000
 [6]  0.000000000  0.000000000  0.000000000  0.000000000  0.000000000
[11] -0.005092486  0.000000000  0.000000000  0.000000000  0.000000000
[16]  0.000000000  0.000000000  0.000000000  0.000000000 -0.022916187
[21] -0.034374280 -0.061109831  0.000000000 -0.039466766  0.000000000
[26] -0.091664746 -0.045832373 -0.091664746 -0.070021681 -0.068748560
[31]  0.000000000  0.000000000 -0.040739887 -0.045832373  0.000000000
[36]  0.000000000  0.000000000 -0.034374280 -0.048378616  0.000000000
[41] -0.005092486 -0.001273121  0.000000000  0.000000000  0.000000000
[46]  0.000000000  0.000000000  0.000000000  0.000000000  0.000000000
[51]  0.000000000  0.000000000  0.000000000 -0.005092486 -0.005092486
[56]  0.000000000  0.000000000  0.000000000  0.000000000  0.000000000
```

```
[61]  0.000000000 -0.001273121  0.000000000  0.000000000 -0.005092486
[66] -0.003819364  0.000000000 -0.001273121  0.000000000  0.000000000
[71]  0.000000000  0.000000000 -0.007638729  0.000000000  0.000000000
[76]  0.000000000  0.000000000  0.000000000  0.000000000 -0.005092486
[81] -0.005092486  0.000000000  0.000000000  0.000000000  0.000000000
[86]  0.000000000  0.000000000 -0.003819364  0.000000000  0.000000000
[91]  0.000000000 -0.001273121  0.000000000  0.000000000  0.000000000


Slot "ind":
integer(0)


$fldind
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
[51] 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
[76] 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
```

4.4 Minimize total flux

Usually, a FBA solution is one solution out of many equivalent solutions. One to choose one is to compute the flux distribution minimizing the absolute flux (MTF) but still supporting the objective value of the FBA solution. At first, a "wild type" solution is required, which is the FBA solution:

```
> fba <- optimizeProb(Ec_core, alg = "fba")
```

Now, the objective value of the solution in `fba` is used to compute a flux distribution with a minimized absolute flux:

```
> mtf <- optimizeProb(Ec_core, alg = "mtf", wtobj = fba$obj)
```

The value of the objective function now is

```
> mtf$obj
```

```
[1] 518.4221
```

which is the minimized sum of all absolute flux values. The flux distribution of the MTF soliution is much longer than the one of the FBA solution

```
> length(fba$fluxes)
```

```
[1] 95
```

```
> length(mtf$fluxes)
```

```
[1] 285
```

which is due to the different formulations of the two optimization problems. Consult the documentation of class `sysBiolAlg` (section 5.4) for more detailed information.

```
> help("sysBiolAlg_fba-class")
> help("sysBiolAlg_mtf-class")
```

The element `fldind` can now be used to get the flux values for the reactions in the model.

```
> mtf$fluxes[mtf$fldind]
```

```
 [1]  0.000000e+00  0.000000e+00 -1.259623e-15  6.007250e+00  6.007250e+00
 [6] -1.259623e-15  0.000000e+00  5.064376e+00  0.000000e+00  0.000000e+00
[11]  8.390000e+00  4.551401e+01  8.739215e-01 -2.280983e+01  6.007250e+00
[16]  4.359899e+01  0.000000e+00  1.471614e+01  0.000000e+00  1.259623e-15
[21]  0.000000e+00  0.000000e+00  2.280983e+01  0.000000e+00  0.000000e+00
[26]  0.000000e+00  0.000000e+00 -1.000000e+01  0.000000e+00  0.000000e+00
[31]  1.753087e+01  2.917583e+01  0.000000e+00  0.000000e+00 -4.765319e+00
[36] -2.179949e+01 -3.214895e+00  0.000000e+00  0.000000e+00  7.477382e+00
[41]  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
[46]  5.064376e+00  0.000000e+00  4.959985e+00  1.602353e+01  1.000000e+01
[51]  2.234617e-01  0.000000e+00 -4.541857e+00  0.000000e+00  0.000000e+00
[56]  0.000000e+00  4.959985e+00 -2.917583e+01  6.007250e+00  0.000000e+00
[61]  0.000000e+00  0.000000e+00  0.000000e+00  5.064376e+00  0.000000e+00
[66]  0.000000e+00  3.853461e+01  0.000000e+00  4.765319e+00  2.179949e+01
[71]  9.282533e+00  7.477382e+00  0.000000e+00  4.860861e+00 -1.602353e+01
[76]  4.959985e+00 -1.471614e+01  3.214895e+00  2.504309e+00  0.000000e+00
[81]  0.000000e+00  1.259623e-15  1.758177e+00 -2.524355e-29  2.678482e+00
[86] -2.281503e+00  0.000000e+00  0.000000e+00  5.064376e+00 -5.064376e+00
[91]  1.496984e+00  0.000000e+00  1.496984e+00  1.181498e+00  7.477382e+00
```

4.5 Genetic perturbations

In order to compute the metabolic phenotype of *in silico* knock-out mutants, the function `oneGeneDel` can be used.

```
> opt <- oneGeneDel(Ec_core)
```

```
|            :            |            :            | 100 %
|=========================================================| :-)
```

```
> checkOptSol(opt)
```

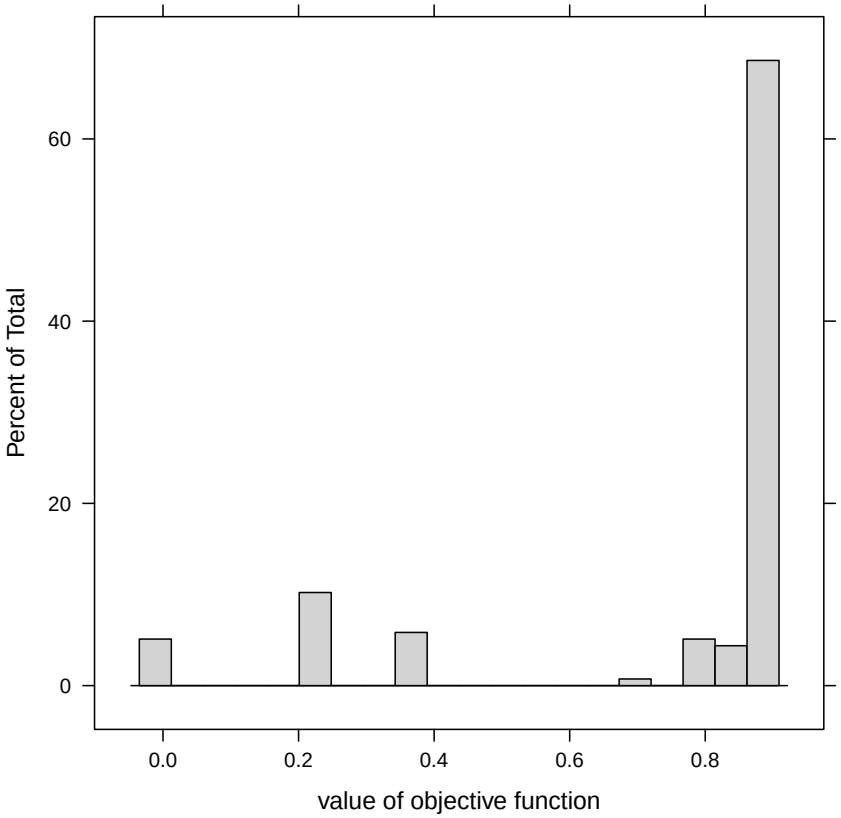
```
Return code:
```

```
Code    #       meaning
0       137     solution process was successful


Solution status:

Code    #       meaning
4       2       no feasible solution exists
5       135     solution is optimal

137 optimizations were performed.
```

The function `oneGeneDel` gets an argument `geneList`, a character vector containing the gene id's to knock out. If `geneList` is missing, all genes are taken into account. The example model contains 137 independent genes. The first optimization is for the wild type—no gene is knocked out—followed by one optimization for each gene.

The result in `opt` is an object of class `optsol_geneDel`, extending class `optsol_optimizeProb`. Class `optsol` contains a method `histogram`, plotting the vaules of the objective function (section 5.2).

```
> histogram(opt, col = "lightgray", nint = 20)
```

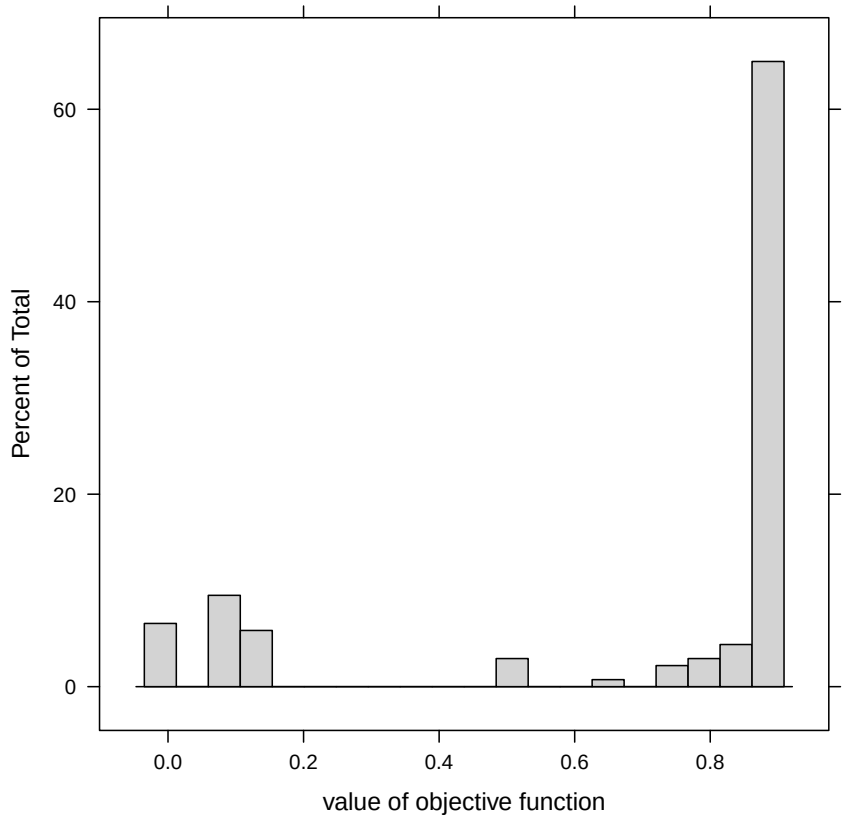


The default algorithm used is FBA [Edwards et al., 2002, Orth et al., 2010b], with the assumption, that the phenotype of the mutant metabolic network is independent of the wild-type phenotype. An alternative is the MOMA algorithm described in Segrè et al. [2002] minimizing the hammiltonian distance of the wild-type phenotype and the mutant phenotype (argument `alg = "lmoma"` computes a linearized version of the MOMA algorithm; `alg = "moma"` runs the quadratic formulation).

```
> opt <- oneGeneDel(Ec_core, alg = "lmoma")

|            :            |            :            | 100 %
|==========================================================| :-)

> checkOptSol(opt)

Return code:

Code    #       meaning
0       137     solution process was successful
```

```
Solution status:

Code    #       meaning
4       2       no feasible solution exists
5       135     solution is optimal

137 optimizations were performed.
```

```
> histogram(opt, col = "lightgray", nint = 20)
```



Another alternative is the ROOM algorithm (regulatory on/off minimization) described in Shlomi et al. [2005]. Set argument `alg` to `room` in order to run ROOM.
In order to perform all possible double-knock-out mutants, or n-knock-out mutants, the function `geneDeletion` can be used. Perform single gene deletions (in principle the same as before with `oneGeneDel`).

```
> opt <- geneDeletion(Ec_core)
```

```
|            :            |            :            | 100 %
|=============================================|  :-)
```

Compute all double-knock-out mutants and all triple-knock-out mutants

```
> opt2 <- geneDeletion(Ec_core, combinations = 2)
> opt3 <- geneDeletion(Ec_core, combinations = 3)
```

which will result in 9317 optimizations for double-knock-outs and 419 221 Optimizations for triple-knock-outs using the metabolic model of the core energy metabolism of *E. coli*. This model contains 137 genes.

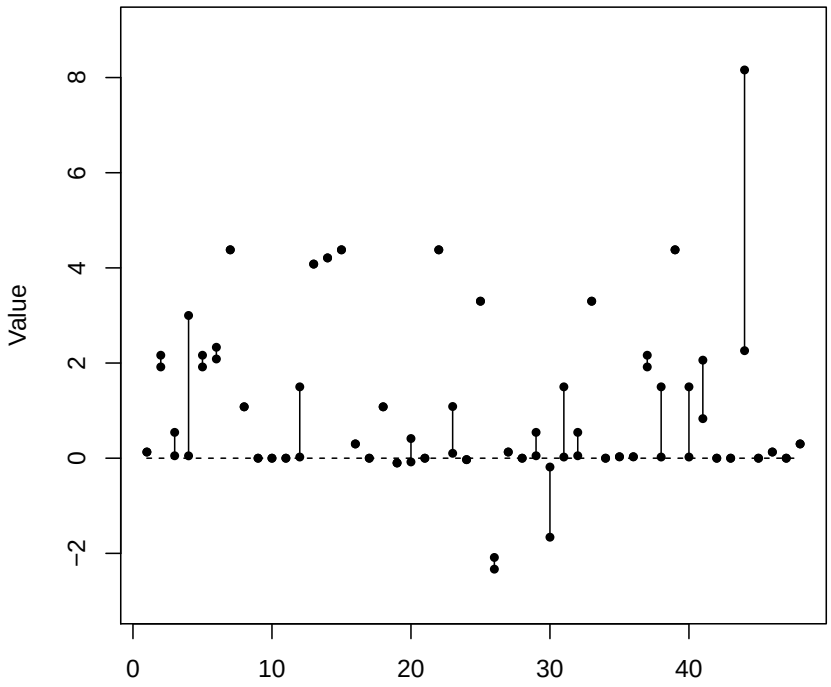
4.6 Flux variablility analysis

The function `fluxVar` performs a flux variability analysis with a given model [Mahadevan and Schilling, 2003]. The minimum and maximum flux values for each reaction in the model are calculated, which still support a given optimal functional state Z_{opt}. The example below is based upon the metabolic model of the human red blood cell by Palsson [2006] and Price et al. [2004].

```
> rbc <- readTSVmod(reactList = "rbc.tsv", fpath = mp, quoteChar = "\"")
```

Perform flux variablilty analysis.

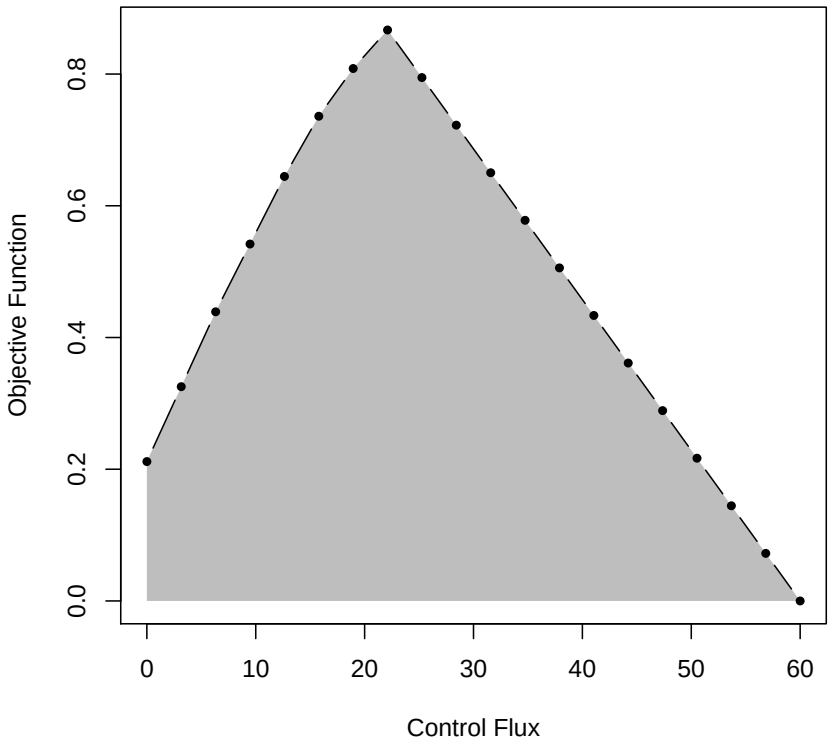
```
> opt <- fluxVar(rbc, verboseMode = 1)
> plot(opt)
```



4.7 Robustness analysis

The function `robAna` performs a robustness analysis with a given model. The flux of a control reaction will be varied stepwise between the maximum and minimum value the flux of the control reaction can reach [Palsson, 2006]. The example below shows a flux variability ananlyis based upon the metabolic model of the core energy metabolism of *E. coli* using the exchange flux of Oxygen as control reaction.

```
> opt <- robAna(Ec_core, "Ex_o2", verboseMode = 1)
> plot(opt)
```



4.8 Parallel computing

The package *SyBiL* provides basic support for the R-package *parallel* (*multicore* for R version prior 2.14.0) in the function `multidel`. The following example shows the computation of all possible triple-knock-out mutants using the model of the core energy metabolism of *E. coli*. The set of genes included in the analysis will be reduced to genes, which are not lethal. A gene i is considered as "lethal", if in a single-gene-knockout the deletion of gene i results in a maximum groth ratio of zero.

```
> ref    <- optimizeProb(Ec_core)
> opt    <- oneGeneDel(Ec_core, geneList = allGenes(Ec_core))

|            :            |            :            | 100 %
|=================================================| :-)

> let    <- lethal(opt, wt = ref$obj)
> nletid <- c(1:length(allGenes(Ec_core)))[! let]
```

At first, a wild-type maximum groth value is computed. Then, all single-gene-knock-outs are computed. The variable `let` contains pointers to the gene id's of genes, who's deletion

is lethal. The variable `nletid` contains pointers to the gene id's of all genes, except for the lethal ones.

```
> gmat <- combn(nletid, 3)
```

The variable `gmat` now contains a matrix with three rows, each column is one combination of three values in `nletid`; one set of genes to knock-out in one step.

```
> opt <- multiDel(Ec_core, nProc = 4, todo = "geneDeletion", del1 = gmat)
```

The function `multiDel` performs a `geneDeletion` with the model `Ec_core` on four CPU's (argument `nProc`) on a shared memory machine. Argument `del1` is the matrix containing the sets of genes to delete. This matrix will be split up in smaller submatrices all having about the same number of columns and three rows. The submatrices are passed to `geneDeletion` and are processed on separate cores in parallel. The resulting variable `opt` now contains a list of four objects of class `optsol_genedel`.

```
> mapply(checkOptSol, opt)
```

4.9 Optimization software

For optimizations, GLPK[3], IBM ILOG CPLEX[4], COIN-OR Clp[5] or lp_solve[6] can be used. All functions performing optimizations, get the arguments `solver` and `method`. The first setting the desired solver and the latter setting the desired optimization algorithm. Possible values for the argument `solver` are:

- `"glpkAPI"`, which is the default,
- `"cplexAPI"`,
- `"clpAPI"` or
- `"lpSolveAPI"`.

Perform FBA, using GLPK as solver and "simplex exact" as algorithm.

```
> optimizeProb(Ec_core, method = "exact")
```

Perform FBA, using IBM ILOG CPLEX as solver and "dualopt" as algorithm.

```
> optimizeProb(Ec_core, solver = "cplexAPI", method = "dualopt")
```

The R-packages *glpkAPI*, *clpAPI* and *cplexAPI* provide access to the C-API of the corresponding optimization software. They are also available from CRAN[1].

[3] Andrew Makhorin: GNU Linear Programming Kit, version 4.42 or higher `http://www.gnu.org/software/glpk/glpk.html`

[4] IBM ILOG CPLEX version 12.2 (or higher) from the IBM Academic Initiative `https://www.ibm.com/developerworks/university/academicinitiative/`

[5] COIN-OR linear programming version 1.12.0 or higher `https://projects.coin-or.org/Clp`

[6] lp_solve via R-package *lpSolveAPI* version 5.5.2.0-5 or higer `http://lpsolve.sourceforge.net/5.5/index.htm`

4.10 Setting parameters to the optimization software

All functions performing optimizations can handle the argument `solverParm` getting a list or data frame containing parameters used by the optimization software.

4.10.1 GLPK

For available parameters used by GLPK, see the GLPK and the *glpkAPI* documentation.

```
> opt <- oneGeneDel(Ec_core,
+                   solverParm = list(TM_LIM = 1000,
+                                     PRESOLVE = GLP_ON))
```

The above command performs a one gene deletion experiment, sets the time limit for each optimization to one second and does presolving in each optimization.

4.10.2 IBM ILOG CPLEX

For available parameters used by IBM ILOG CPLEX, see the IBM ILOG CPLEX and the *cplexAPI* documentation.

```
> opt <- optimizeProb(Ec_core,
+                     solverParm = list(CPX_PARAM_SCRIND = CPX_ON,
+                                       CPX_PARAM_EPRHS = 1E-09),
+                     solver = "cplexAPI")
```

The above command performs FBA, sets the messages to screen switch to "on" and sets the feasibility tolerance to 10^{-9}.

4.10.3 COIN-OR Clp

At the time of writing, it is not possible to set any parameters when using COIN-OR Clp.

4.10.4 lpSolveAPI

See the *lpSolveAPI* documentation for parameters for `lp_solve`.

```
> opt <- optimizeProb(Ec_core,
+                     solverParm = list(verbose = "full",
+                                       timeout = 10),
+                     solver = "lpSolveAPI")
```

The above command performs FBA, sets the verbose mode to "full" and sets the timeout to ten seconds.

parameter name	default value
SOLVER	glpkAPI
METHOD	simplex
SOLVER_CTRL_PARM	as.data.frame(NA)
ALGORITHM	fba
TOLERANCE	1E-6
MAXIMUM	1000
OPT_DIRECTION	max
PATH_TO_MODEL	.

Table 1: Available parameters in *SyBiL* and their default values.

4.11 Setting parameters in sybil

Parameters to *SyBiL* can be set using the function `SYBIL_SETTINGS`. Parameter names and their default values are shown in table 1, all possible values are described in the `SYBIL_SETTINGS` documentation.

```
> help(SYBIL_SETTINGS)
```

The function `SYBIL_SETTINGS` gets at most two arguments:

```
> SYBIL_SETTINGS("parameter name", value)
```

the first one giving the name of the parameter to set (as character string) and the second one giving the desired value. If `SYBIL_SETTINGS` is called with only one argument

```
> SYBIL_SETTINGS("parameter name")
```

the current setting of `"parameter name"` will be returned. All parameters and their values can be achieved by calling `SYBIL_SETTINGS` without any argument.

```
> SYBIL_SETTINGS()
```

4.11.1 Solver software specific

The two parameters `SOLVER` and `METHOD` depend on each other, e. g. the method called `simplex` is only available when `glpkAPI` is used as solver software. Each solver has its own specific set of methods available in order to solve optimization problems. If one changes the parameter `SOLVER` to, let's say `cplexAPI`, the parameter `METHOD` will automatically be adjusted to the default method used by `cplexAPI`. Set parameter solver to IBM ILOG CPLEX for every optimization:

```
> SYBIL_SETTINGS("SOLVER", "cplexAPI", loadPackage = FALSE)
```

Now, IBM ILOG CPLEX is used as default solver e. g. in `optimizeProb` or `oneGeneDel`, and parameter `METHOD` has changed to the default method in *cplexAPI*. Setting argument `loadPackage` to `FALSE` prevents loading the API package. Get the current setting for `Method`:

```
> SYBIL_SETTINGS("METHOD")

[1] "lpopt"
```

Reset the solver to `glpkAPI`:

```
> SYBIL_SETTINGS("SOLVER", "glpkAPI")
```

Now, the default method again is `simplex`

```
> SYBIL_SETTINGS("METHOD")

[1] "simplex"
```

It is not possible to set a wrong method for a given solver. If the desired method is not available, allways the default method is used. Parameters to the solver software (parameter `SOLVER_CTRL_PARM`) must be set as `list` or `data.frame` as described in section 4.10.

4.11.2 Analysis specific

The parameter `ALGORITHM` controls the way gene deletion anslysis will be performed. The default setting `"fba"` will use flux-balance analysis (FBA) as described in Edwards et al. [2002] and Orth et al. [2010b]. Setting this parameter to `"lmoma"`, results in a linearized version of the MOMA algorithm described in Segrè et al. [2002] ("moma" will run the original version). The linearized version of MOMA, like it is implemented in the COBRA Toolbox [Becker et al., 2007, Schellenberger et al., 2011], can be used in functions like `oneGeneDel()` via the boolean argument `COBRAflag`. Setting the parameter `"ALGORITHM"` to `"room"` will run a regulatory on/off minimization as described in Shlomi et al. [2005]. See also section 4.5 for details on gene deletion analysis.

5 Central data structures

5.1 Class modelorg

The class `modelorg` is the core datastructure to represent a metabolic network, in particular the stoichiometric matrix S. An example (E. coli core flux by [Palsson, 2006]) is shipped within *SyBiL* and can be loaded this way:

```
> data(Ec_core)
> Ec_core

model name:             Ecoli_core_model
number of compartments  2
                        C_c
                        C_e
number of reactions:    95
```

```
number of metabolites:  72
number of unique genes: 137
objective function:     1 * Biomass_Ecoli_core_w_GAM
```

The generic method `show` displays a short summary of the network. See

```
> help("modelorg")
```

for the list of available methods. All slots of an object of class `modelorg` are accessible via setter and getter methods having the same name as the slot. For example, slot `react_num` contains the number of reactions in the model (equals the number of columns in S). Access the number of reactions in the *E. coli* model.

```
> react_num(Ec_core)
```

```
[1] 95
```

Get all reaction is's:

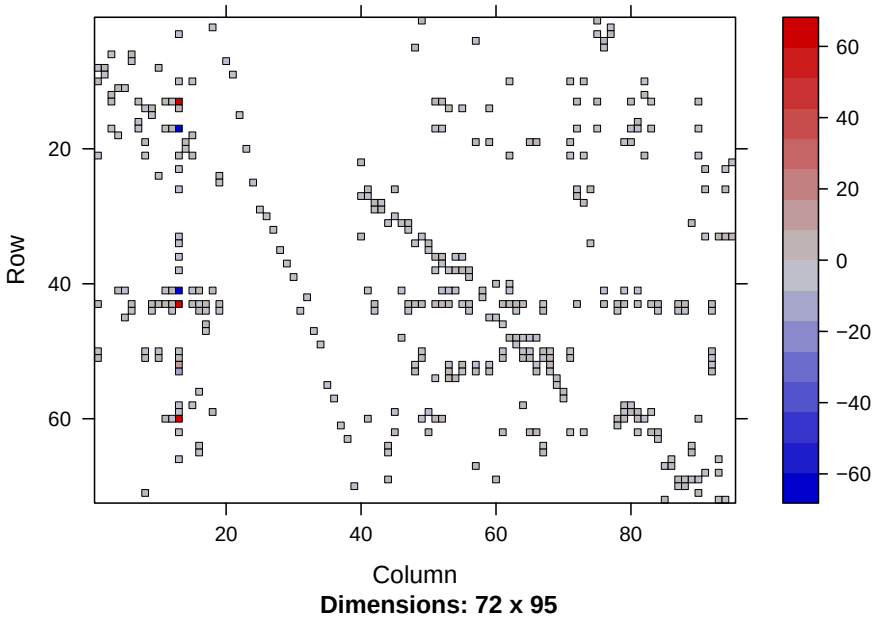
```
> id <- react_id(Ec_core)
```

Change a reaction id:

```
> react_id(Ec_core)[13] <- "biomass"
```

Plot an image of the stoichiometric matrix S:

```
> image(S(Ec_core))
```



Matrices in objects of class `modelorg` are stored in formats provided by the *Matrix*-package.

Objects of class `modelorg` can easily be created. Sources are common file formats like tab delimited files from the BiGG database [Schellenberger et al., 2010][7] or SBML files (with package *sybilSBML*[2]). See section 3 on page 2 about supported file formats and their description. Read a reaction list generated from the BiGG database:

```
> mod <- readTSVmod(reactList = "reactionList.txt")
```

Here, `"reactionList.txt"` is an from BiGG database exported reaction list. Usually, these files do neither contain an objective function, nor upper and lower bounds on the reaction rates. They need to be added to the returned object of class `modelorg` using the methods `obj_coef<-`, `lowbnd<-` and `uppbnd<-`, or by adding the columns `obj_coef`, `lowbnd` and `uppbnd` to the input file.

5.2 Class optsol

The derived classes of class `optsol` (optimization solution) are used to store information and results from various optimisation problems and their biological relation. See

[7] `http://bigg.ucsd.edu`

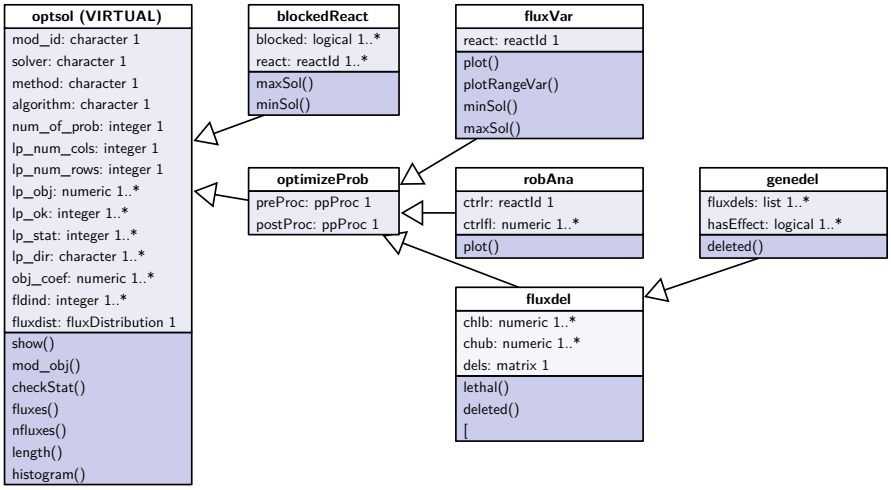


Figure 2: UML representation of class `optsol`.

```
> help("optsol")
```

for the list of available methods to access data (figure 2). A simple demonstration would be:

```
> data(Ec_core)
> os <- optimizeProb(Ec_core, retOptSol = TRUE)
> os
```

```
solver:                                    glpkAPI
method:                                    simplex
algorithm:                                 fba
number of variables:                       95
number of constraints:                     72
number of problems to solve:               1
number of successful solution processes:   1
```

```
> class(os)
```

```
[1] "optsol_optimizeProb"
attr(,"package")
[1] "sybil"
```

Retrieve objective value.

```
> lp_obj(os)
```

```
[1] 0.8739215
```

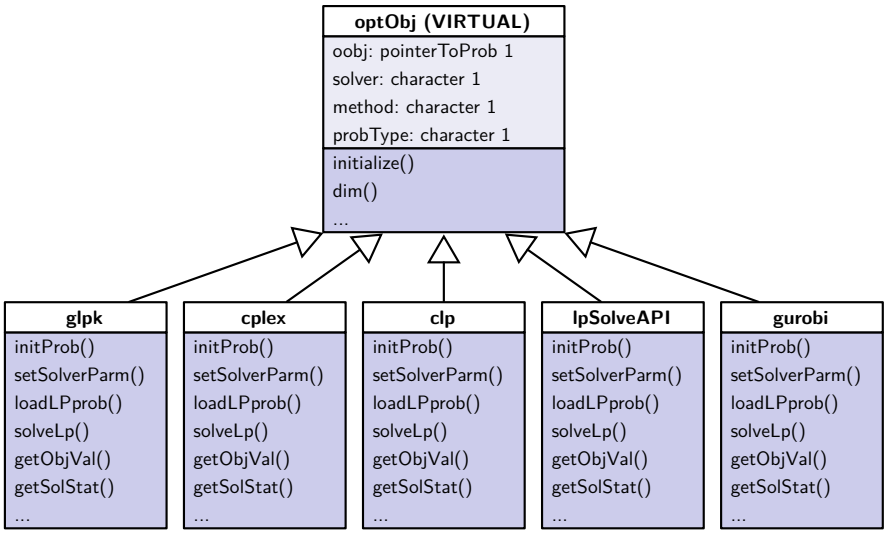


Figure 3: UML representation of class `optObj`.

5.3 Class optObj

The class `optObj` is *SyBiL*'s internal representation of a linear programming problem (figure 3). Objects of this class harbor four slots: a pointer to the C-structure of the problem, the name of the solver, the name of the optimization method used by the solver and a single character string, describing the problemtype (e. g. `lp` [linear programming]).

The package *SyBiL* provides several functions to alter the linear programming model. Each function takes care of the special needs of every supported solver for you. The following example should illustrate the purpose of class `optObj`. Consider a linear programming problem, here written in lp file format:

```
Maximize
  obj: + x_1 + x_2
Subject To
  r_1: + 0.5 x_1 + x_2 <= 4.5
  r_2: + 2   x_1 + x_2 <= 9
Bounds
  0 <= x_1 <= 1000
  0 <= x_2 <= 1000
```

In order to solve this lp problem with *SyBiL*, an onject of class `optObj` has to be created. The constructor function has the same name as the class it builds.

```
> lp <- optObj(solver = "glpkAPI", method = "exact")
> lp

linear programming problem object
solver: glpkAPI
```

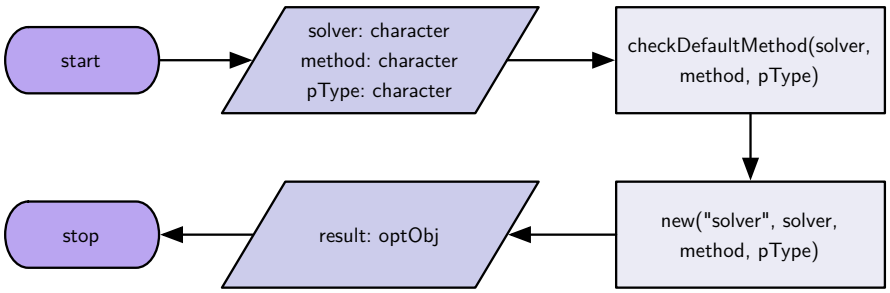


Figure 4: Work flow of the constructor function `optObj()`.

```
method: exact
problem is not initialized
```

The first argument is the used solver software, in this case it is GLPK. The second optional argument gives the method, how the solver software has to solve the problem. Here, it is the simplex exact algorithm of GLPK. The constructor functon `optObj()` (figure 4) returns an object of class `optObj_glpkAPI` in this case. This class enables *SyBiL* to communicate with the GLPK software. The constructor function `optObj()` calls the function `checkDefaultMethod()` which tries to load the specified solver package and also checks if all other arguments (`method` and `pType`) are valid arguments. Each solver package has its own set of methods for specific types of optimization (e. g. linear programming or quadratic programming) and is thus availabe maybe not for all problem types.

Initialize the new problem object. Each solver software needs to create specific data structures to hold the problem and solution data.

```
> lp <- initProb(lp)
> lp

linear programming problem object
solver: glpkAPI
method: exact
problem is currently empty
```

Slot `oobj` holds a pointer to the problem object of GLPK. Now, we need to allocate space for the problem data and load the data into the problem object.

```
> cm <- Matrix(c(0.5, 2, 1, 1), nrow = 2)
> loadLPprob(lp, nCols = 2, nRows = 2, mat = cm,
+            lb = c(0, 0), ub = rep(1000, 2), obj = c(1, 1),
+            rlb = c(0, 0), rub = c(4.5, 9), rtype = c("U", "U"),
+            lpdir = "max")
```

The first command generates the constraint matrix in sparse format (see also documentation in package *Matrix*). The second command loads the problem data into the problem object.

```
> lp
```

```
linear programming problem object
solver: glpkAPI
method: exact
problem has 2 variables and 2 constraints
```

All data are now set in the problem object, so it can be solved.

```
> status <- solveLp(lp)
> status
```

```
[1] 0
```

Translate the status code in a text string.

```
> getMeanReturn(code = status, solver = solver(lp))
```

```
[1] "solution process was successful"
```

Check the solution status.

```
> status <- getSolStat(lp)
> getMeanStatus(code = status, solver = solver(lp))
```

```
[1] "solution is optimal"
```

Retrieve the value of the objective function and the values of the variables after optimization.

```
> getObjVal(lp)
```

```
[1] 6
```

```
> getFluxDist(lp)
```

```
[1] 3 3
```

Get the reduced costs.

```
> getRedCosts(lp)
```

```
[1] 0 0
```

Delete problem object and free all memory allocated by the solver software.

```
> delProb(lp)
> lp
```

```
linear programming problem object
solver: glpkAPI
method: exact
problem is not initialized
```

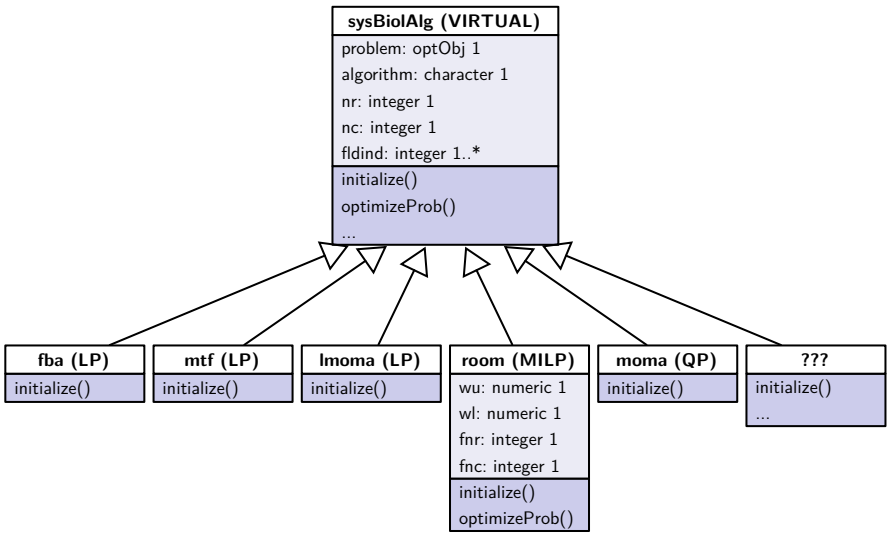


Figure 5: UML representation of class `sysBiolAlg`.

5.4 Class sysBiolAlg

The class `sysBiolAlg` holds objects of class `optObj` which are prepared for a particular kind of algorithm, e. g. FBA, MOMA or ROOM (figure 5). Class `optObj` takes care of the communication between *SyBiL* and the solver software. Class `sysBiolAlg` instead is responsible for the algorithm used to analyse a metabolic network. The constructor function `sysBiolalg()` (figure 6) gets at least two arguments: 1. an object of class modelorg (section 5.1) and 2. a single character string indicating the name of the desired algorithm. Further arguments are passed through argument `...` to the corresponding constructor of the class extending class `sysBiolAlg`. The base class `sysBiolAlg` is virtual, no objects can be created from that class directly. The constructor function builds an instance of a class extending the base class:

```
> data(Ec_core)
> ec <- sysBiolAlg(Ec_core, alg = "fba")
> is(ec)

[1] "sysBiolAlg_fba" "sysBiolAlg"
```

Now, the variable `ec` contains an object of class `sysBiolAlg_fba`. Slot `problem` of that object is of class `optObj` and is prepared for FBA. The optimization can be performed with method `optimizeProb()`:

```
> opt <- optimizeProb(ec)
```

The return value of `optimizeProb()` is discussed in section 4.3. In order to run a ROOM analysis create an object of class `sysBiolAlg_room`:

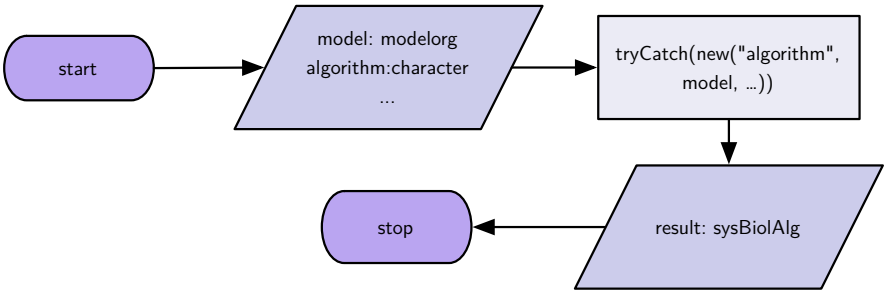


Figure 6: Work flow of the constructor function `sysBiolAlg()`.

```
> ecr <- sysBiolAlg(Ec_core, alg = "room", wtflux = opt$fluxes)
> is(ecr)
```

```
[1] "sysBiolAlg_room" "sysBiolAlg"
```

Argument `wtflux` gets the optimal flux distribution computed via FBA earlier. It is used by the constructor method of class `sysBiolAlg_room`.

5.4.1 Constructor methods

The base class `sysBiolAlg` contains a constructor method `initialize` which is called by the constructor methods of the subclasses via `callNextMethod()` (figure 7). Every subclass has its own constructor method prepareing all necessary data structures in order to call the constructor of the base class. For example, for the ROOM algorithm, a "wild type" flux distribution is required (argument `wtflux` in the example above). The constructor of `sysBiolAlg_room` generates all data structures to build the optimization problem, e. g. the constraint matrix, objective coefficients, right hand side, ... It passes all these data to the constructor of `sysBiolAlg` via a call to `callNextMethod()`. This constructor generates the object of class `optObj` while taking care on solver software specific details.

5.4.2 New algorithms

In order to extend the functionality of *SyBiL* with new algorithms, a new class describing that algorithm is required. The function `promptSysBiolAlg()` generates a skeletal structure of a new class definition and a corresponding constructor method. To implement an algorithm named "foo", run

```
> promptSysBiolAlg(algorithm = "foo")
```

which generates a file `sysBiolAlg_fooClass.R` containing the new class definition. The class `sysBiolAlg_foo` will extend class `sysBiolAlg` directly and will not add any slots to the class. Additionally, an unfinished method `initialize` is included. Here it is necessary to generate the data structures required by the new algorithm. There are comments in the skeletal structure guiding through the process.

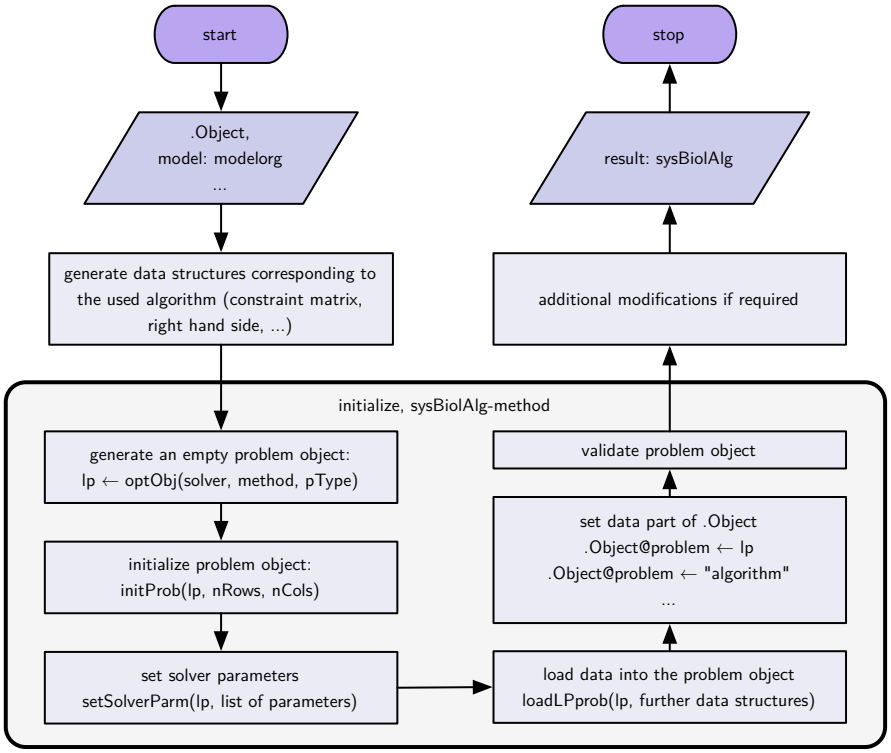


Figure 7: Work flow of the constructor methods of classes extending class `sysBiolAlg`. The gray shaded part is done by the constructor method or the base class.

References

S. A. Becker et al. Quantitative prediction of cellular metabolism with constraint-based models: the COBRA Toolbox. *Nat Protoc*, 2(3):727–738, 2007. doi: 10.1038/nprot.2007.99.

J. S. Edwards, M. Covert, and B. Ø. Palsson. Metabolic modelling of microbes: the flux-balance approach. *Environ Microbiol*, 4(3):133–140, Mar 2002.

R. Mahadevan and C. H. Schilling. The effects of alternate optimal solutions in constraint-based genome-scale metabolic models. *Metab Eng*, 5(4):264–276, Oct 2003.

J. D. Orth, R. M. T. Fleming, and B. Ø. Palsson. Reconstruction and use of microbial metabolic networks: the core *Escherichia coli* metabolic model as an educational guide. EcoSal Chapter 10.2.1, 2010a.

J. D. Orth, I. Thiele, and B. Ø. Palsson. What is flux balance analysis? *Nat Biotechnol*, 28(3):245–248, Mar 2010b. doi: 10.1038/nbt.1614.

B. Ø. Palsson. *Systems Biology: Properties of Recontructed Networks*. Cambridge University Press, 2006.

N. D. Price, J. Schellenberger, and B. Ø. Palsson. Uniform sampling of steady-state flux spaces: means to design experiments and to interpret enzymopathies. *Biophys J*, 87 (4):2172–2186, Oct 2004. doi: 10.1529/biophysj.104.043000.

J. Schellenberger, J. O. Park, T. M. Conrad, and B. Ø. Palsson. BiGG: a biochemical genetic and genomic knowledgebase of large scale metabolic reconstructions. *BMC Bioinformatics*, 11:213, 2010. doi: 10.1186/1471-2105-11-213.

J. Schellenberger, R. Que, R. M. T. Fleming, I. Thiele, J. D. Orth, A. M. Feist, D. C. Zielinski, A. Bordbar, N. E. Lewis, S. Rahmanian, J. Kang, D. R. Hyduke, and B. Ø. Palsson. Quantitative prediction of cellular metabolism with constraint-based models: the COBRA Toolbox v2.0. *Nat Protoc*, 6(9):1290–1307, 2011. doi: 10.1038/nprot.2011.308.

D. Segrè et al. Analysis of optimality in natural and perturbed metabolic networks. *Proc Natl Acad Sci U S A*, 99(23):15112–15117, Nov 2002. doi: 10.1073/pnas.232349399.

T. Shlomi, O. Berkman, and E. Ruppin. Regulatory on/off minimization of metabolic flux changes after genetic perturbations. *Proc Natl Acad Sci U S A*, 102(21):7695–7700, May 2005. doi: 10.1073/pnas.0406346102.