CMPL

<Coliop|Coin> Mathematical Programming Language

Version 1.10.0

June 2014

Manual

M. Steglich, T. Schleiff

Table of contents

1 About CMPL....6
2 CMPL Language reference manual....7
2.1 CMPL elements....7
2.1.1 General structure of a CMPL model....7
2.1.2 Keywords and other syntactic elements....8
2.1.3 Objects....9
2.1.3.1 Parameters....9
2.1.3.2 Variables....11
2.1.3.3 Indices and sets....13
2.1.3.4 Line names....15
2.1.4 CMPL header....16
2.2 Parameter Expressions....19
2.2.1 Overview....19
2.2.2 Array functions....19
2.2.3 Set operations and functions....20
2.2.4 Mathematical functions....22
2.2.5 Type casts....24
2.2.6 String operations....24
2.3 Input and output operations....26
2.3.1 Error and user messages....26
2.3.2 cmplData files....27
2.3.3 Readcsv and readstdin....31
2.3.4 Include....32
2.4 Statements....32
2.4.1 parameters and variables section....33
2.4.2 objectives and constraints section....33
2.5 Control structure....34
2.5.1 Overview....34
2.5.2 Control header....35
2.5.2.1 Iteration headers....35
2.5.2.2 Condition headers....35
2.5.2.3 Local assignments....36
2.5.3 Alternative bodies....37
2.5.4 Control statements....37
2.5.5 Specific control structures....38
2.5.5.1 For loop....38
2.5.5.2 If-then clause....39
2.5.5.3 Switch clause....40
2.5.5.4 While loop....41
2.5.6 Set and sum control structure as expression....41
2.6 Matrix-Vector notations....43
2.7 Automatic model reformulations....46

2.7.1 Overview....46
2.7.2 Matrix reductions....46
2.7.3 Equivalent transformations of Variable Products....46
2.7.3.1 Variable Products with at least one binary variable....47
2.7.3.2 Variable Product with at least one integer variable....47
2.8 Examples....48
2.8.1 Selected decision problems....48
2.8.1.1 The diet problem....48
2.8.1.2 Production mix....51
2.8.1.3 Production mix including thresholds and step-wise fixed costs....54
2.8.1.4 The knapsack problem....55
2.8.1.5 Transportation problem using 1-tuple sets....58
2.8.1.6 Transportation problem using multidimensional sets (2-tuple sets)....61
2.8.1.7 Quadratic assignment problem....63
2.8.1.8 Quadratic assignment problem using the solutionPool option....66
2.8.1.9 Generic travelling salesman problem....67
2.8.2 Other selected examples....69
2.8.2.1 Solving the knapsack problem....69
2.8.2.2 Finding the maximum of a concave function using the bisection method....71
3 CMPL software package....72
3.1 CMPL software package in a glance....72
3.2 Installation....73
3.3 CMPL....73
3.3.1 Running CMPL....73
3.3.2 Usage of the CMPL command line tool....74
3.3.3 Syntax checks....76
3.3.4 Using CMPL with several solvers....77
3.3.4.1 CBC....77
3.3.4.2 GLPK....78
3.3.4.3 Gurobi....78
3.3.4.4 SCIP....79
3.3.4.5 CPLEX....80
3.3.4.6 Other solvers....81
3.3.5 Coliop....81
3.4 CMPLServer....83
3.4.1 Single server mode....85
3.4.2 Grid mode....87
3.4.3 Reliability and failover....91
3.5 pyCMPL....94
3.6 jCMPL....95
3.7 Input and output file formats....95
3.7.1 Overview....95
3.7.2 CMPL and CmplData....96
3.7.3 Free - MPS....97

3.7.4 CmplInstance....97
3.7.5 ASCII or CSV result files....100
3.7.6 CmplSolutions....101
3.7.7 CmplMessages....104
3.7.8 CmplInfo....106
4 CMPL's APIs....107
4.1 Creating Python and Java applications with a local CMPL installation....108
4.1.1 pyCMPL....109
4.1.2 jCMPL....112
4.2 Creating Python and Java applications using CMPLServer....116
4.2.1 pyCMPL....117
4.2.2 jCMPL....118
4.3 pyCMPL reference manual....119
4.3.1 CmplSets....119
4.3.2 CmplParameters....121
4.3.3 Cmpl....123
4.3.3.1 Establishing models....123
4.3.3.2 Manipulating models....125
4.3.3.3 Solving models....126
4.3.3.4 Reading solutions....130
4.3.3.5 Reading CMPL messages....137
4.3.4 CmplExceptions....138
4.4 jCMPL reference manual....138
4.4.1 CmplSets....138
4.4.2 CmplParameters....141
4.4.3 Cmpl....144
4.4.3.1 Establishing models....144
4.4.3.2 Manipulating models....145
4.4.3.3 Solving models....147
4.4.3.4 Reading solutions....151
4.4.3.5 Reading CMPL messages....156
4.4.4 CmplExceptions....157
4.5 Examples....158
4.5.1 The diet problem....158
4.5.1.1 Problem description and CMPL model....158
4.5.1.2 pyCMPL....158
4.5.1.3 jCmpl....159
4.5.2 Transportation problem....161
4.5.2.1 Problem description and CMPL model....161
4.5.2.2 pyCMPL....161
4.5.2.3 jCMPL....164
4.5.3 The shortest path problem....166
4.5.3.1 Problem description and CMPL model....166
4.5.3.2 pyCMPL....167

4.5.3.3 jCMPL....168
4.5.4 Solving randomized shortest path problems in parallel....170
4.5.4.1 Problem description....170
4.5.4.2 pyCMPL....170
4.5.4.3 jCMPL....172
4.5.5 Column generation for a cutting stock problem....175
4.5.5.1 Problem description and CMPL model....175
4.5.5.2 jCMPL....176
4.5.5.3 jCMPL....180
5 Authors and Contact....185
6 Appendix....186
6.1 Selected CBC parameters....186
6.2 Selected GLPK parameters....199

1 About CMPL

CMPL (<Coliop|Coin> Mathematical Programming Language) is a mathematical programming language and a system for mathematical programming and optimisation of linear optimisation problems.

The CMPL syntax is similar in formulation to the original mathematical model but also includes syntactic elements from modern programming languages. CMPL is intended to combine the clarity of mathematical models with the flexibility of programming languages.

CMPL executes CBC, GLPK, Gurobi, SCIP or CPLEX directly to solve the generated model instance. Because it is also possible to transform the mathematical problem into MPS, Free-MPS or OSiL files, alternative solvers can be used.

CMPL is an open source project licensed under GPL. It is written in C++ and is available for most of the relevant operating systems (Windows, OS X and Linux).

The CMPL distribution contains **Coliop** which is an (simple) IDE (Integrated Development Environment) for CMPL. Coliop is an open source project licensed under GPL. It is written in Java and is as an integral part of the CMPL distribution.

The CMPL package also contains pyCMPL, jCMPL and CMPLServer.

pyCMPL is the CMPL application programming interface (API) for Python and an interactive shell and **jCMPL** is CMPL's Java API. The main idea of this APIs is to define sets and parameters within the user application, to start and control the solving process and to read the solution(s) into the application if the problem is feasible. All variables, objective functions and constraints are defined in CMPL. These functionalities can be used with a local CMPL installation or a CMPLServer.

CMPLServer is an XML-RPC-based web service for distributed and grid optimisation that can be used with CMPL, pyCMPL and jCMPL. It is reasonable to solve large models remotely on the CMPLServer that is installed on a high performance system. CMPL provides four XML-based file formats for the communication between a CMPLServer and its clients. (CmplInstance, CmplSolutions, CmplMessages, CmplInfo).

pyCMPL, jCMPL and CMPLServer are licensed under LGPLv3.

CMPL, Coliop, pyCMPL, jCMPL and CMPLServer are COIN-OR projects initiated by the Technical University of Applied Sciences Wildau and the Institute for Operations Research and Business Management at the Martin Luther University Halle-Wittenberg.

2 CMPL Language reference manual

2.1 CMPL elements

2.1.1 General structure of a CMPL model

The structure of a CMPL model follows the standard model of linear programming (LP), which is defined by a linear objective function and linear constraints. Apart from the variable decision vector *x* all other components are constant.

$$c^T \cdot x \rightarrow max\,!$$
$$s.t.$$
$$A \cdot x \leq b$$
$$x \geq 0$$

A CMPL model consists of four sections, the `parameters` section, the `variables` section, the `objectives` section and the `constraints` section, which can be inserted several times and mixed in a different order. Each sector can contain one or more lines with user-defined expressions.

```
parameters:
    # definition of the parameters
variables:
    # definition of the variables
objectives:
    # definition of the objective(s)
constraints:
    # definition of the constraints
```

A typical LP problem is the production mix problem. The aim is to find an optimal quantity for the products, depending on given capacities. The objective function is defined by the profit contributions per unit c and the variable quantity of the products x. The constraints consist of the use of the capacities and the ranges for the decision variables. The use of the capacities is given by the product of the coefficient matrix A and the vector of the decision variables x and restricted by the vector of the available capacities b.
The simple example:

$$1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 \rightarrow max\,!$$
$$s.t.$$
$$5.6 \cdot x_1 + 7.7 \cdot x_2 + 10.5 \cdot x_3 \leq 15$$
$$9.8 \cdot x_1 + 4.2 \cdot x_2 + 11.1 \cdot x_3 \leq 20$$
$$0 \leq x_n \;\; ; n = 1(1)3$$

can be formulated in CMPL as follows:

```
parameters:
    c[] := ( 1, 2, 3 );
    b[] := ( 15, 20 );

    A[,] := (( 5.6, 7.7, 10.5 ),
             ( 9.8, 4.2, 11.1 ));
```

```
variables:
    x[1..count(c[])]: real;

objectives:
    c[]T * x[] -> max;

constraints:
    A[,] * x[] <= b[];
    x[] >= 0;
```

2.1.2 Keywords and other syntactic elements

Keywords

`parameters, variables, objectives, constraints`	section markers
`real, integer, binary`	types of variable
`real, integer, binary, string, set`	types only used for type casts
`max, min`	objective senses
`set, in, len, defset`	key words for sets
`max, min, count, format, type`	functions for parameter expressions
`sqrt, exp, ln, lg, ld, srand, rand, sin, cos, tan, acos, asin, atan, sinh, cosh, tanh, abs, ceil, floor, round`	mathematical functions that can be used for parameter expressions
`include`	include of a CMPL file
`readcsv, readstdin`	data import from a CSV file or from user input
`error, echo`	error and user message
`sum`	summation
`continue, break, default, repeat`	key words for control structures

Arithmetic operators

`+ -`	signs for parameters or addition/subtraction
`^`	to the power of
`* /`	multiplication and division
`div mod`	integer division and remainder on division
`:=`	assignment operator

Condition operators

`= <= >=`	conditions for constraints, while-loops and if-then clauses
`== < > != <>`	additional conditions in while-loops and if-then clauses

`&& || !`	logical operations (and, or, not)
`<<`	element operator for checking whether an index is an element of a set

Other syntactic elements

`( )`	- arithmetical bracketing in constant expressions - lists for initialising vectors of constants - parameters for constant functions - increment in an algorithmic set
`[ ]`	- indexing of vectors - range specification in variable definitions
`{ }`	- control structures
`..`	- algorithmic set (e.g. range for indices or loop counters) - range specification in variable definitions
`,`	- element separation in an initialisation list for constant vectors and enumeration sets - separation of function parameters - separation of indices - separation of loop heads in a loop
`:`	- mark indicating beginning of sections - definition of variables - definition of parameter type - separation of loop header from loop body - separation of line names
`|`	- separation of alternative blocks in a control structure
`;`	- mark indicating end of a statement - every statement is to be closed by a semicolon
`#` `/* */`	- comment (up to end of line) - comment (between /* and */)

2.1.3 Objects

2.1.3.1 Parameters

A `parameters` section consists of parameter definitions and assignments to parameters. A parameter can only be defined within the `parameters` section using an assignment or through using a cmplData file (see 2.3.2 cmplData files) and a corresponding CMPL header option.

Note that a parameter can be used as a constant in a linear optimisation model as coefficients in objectives and constraints. Otherwise parameters can be used like variables in programming languages. Parameters are

usable in expressions, for instance in the calculation and definition of other parameters. A user can assign a value to a parameter and can then subsequently change the value with a new assignment.

A parameter is identified by name and, if necessary, by indices. A parameter can be a scalar or an array of parameter values (e.g. vector, matrix or another multidimensional construct). A parameter is defined by an assignment with the assignment operator `:=`.

Usage:

```
name := scalarExpression;
name[index] := scalarExpression;
name[[set]] := arrayExpression;
```

name	Name of the parameter
index	Indexing expression that defines a position in an array of parameters. Described in 2.1.3.3 Indices and sets
scalarExpression	A scalar parameter or a single part of an array of parameters is assigned a single integer or real number, a single string, the scalar result of a mathematical function.
set	(Optional) set expression (list of indices) for the definition of the dimension of the array If more than one set is used then the sets have to be separated by commas. Described in 2.1.3.3 Indices and sets
arrayExpression	A non-scalar expression consists of a list of *scalarExpressions* or *arrayExpression*. The elements of the list are separated by commas and imbedded in brackets. The elements of the list can also be sets. But it is not possible to mix set and non-set expressions. If an array contains only one element, then it is necessary to include an additional comma behind the element. Otherwise the expression is interpreted as an arithmetical bracket.

Examples:

`k := 10;`	parameter `k` with value 10
`k[1..5] := ( 0.5, 1, 2, 3.3, 5.5);` `k[] := ( 0.5, 1, 2, 3.3, 5.5);`	vector of parameters with 5 elements
`A[]:= (16, 45.4);`	definition of a vector with two integer values `a[1]`=16 and `a[2]`=45.

`a[, ] := (( 5.6, 7.7, 10.5 ),` `( 9.8, 4.2, 11.1 ));`	dense matrix with 2 rows and 3 columns
`b[] := ( 22 ,);`	definition of the vector `b` with only one element.
`b[] := ( 22 );`	causes an error: `array dimensions don't match`, since `(22)` is not interpreted as an array but as an assignment of a scalar expression.
`products := set( "bike1", "bike2");` `machineHours[products ]:= (5.4, 10);`	defines a vector for machine hours based on the set `products`
`myString := "this is a string";`	string parameter
`q := 3;`	parameter `q` with value 3
`g[1..q] := ( 1, 2, 3);`	usage of `q` for the definition of the parameter `g`

`x := 1(1)2;`
`y := 1(1)2;`
`z := 1(1)2;`
 	
`cube[x,y,z]:= ( ( (1,2),(3,4) ) ,`
`( (5,6),(7,8 ) ) );`
	definition of a parameter cube that is based on the sets `x`, `y` and `z`.

`a := set( [1,1],[1,2],[2,2],[3,2] );` `b[a] := ( 10, 20, 30 , 40);`	definition of a sparse matrix `b` that is based on the 2-tuple set `a`.

If a name is used for a parameter the name cannot be used for a variable.
A special kind of parameter is local parameters, which can only be defined within the head of a control structure. A local parameter is only valid in the body of the control structure and can be used like any other parameter. Only scalar parameters are permitted as local parameters. Local parameters are mainly used as loop counters that are to be iterated over a set.

2.1.3.2 Variables

The `variables` section is intended to declare the variables of a decision model, which are necessary for the definition of objectives and constraints in the decision model. A model variable is identified by name and, if necessary, by an index. A type must be specified. A model variable can be a scalar or a part of a vector, a matrix or another array of variables. A variable cannot be assigned a value.

Usage:

```
variables:
  name : type [[lowerBound]..[upperBound]];
  name[index] : type [[lowerBound]..[upperBound]];
  name[set] : type [[lowerBound]..[upperBound]];
```

name	name of model variable
type	type of model variable. Possible types are `real, integer, binary`.
[*lowerBound..upperBound*]	optional parameter for limits of model variable *lowerBound* and *upperBound* must be a real or integer expression. For the type `binary` it is not possible to specify bounds.
index	Indexing expression that defines a position in an array of variables. Described in 2.1.3.3 Indices and sets
set	(Optional) set expression (list of indices) for the definition of the dimension of the array If more than one set is used then the sets have to be separated by commas. Described in 2.1.3.3 Indices and sets

Examples:

`x: real;`	`x` is a real model variable with no ranges
`x: real[0..100];`	`x` is a real model variable, $0 \le x \le 100$
`x[1..5]: integer[10..20];`	vector with 5 elements, $10 \le x_n \le 20; n = 1(1)5$
`x[1..5,1..5,1..5]: real[0..];`	a three-dimensional array of real model variables with 125 elements identified by indices, $x_{i,j,k} \ge 0; i,j,k = 1(1)5$

`parameters:`
`  prod := set( "bike1", "bike2");`
`variables:`
`  x[prod]: real[0.. ];`
	defines a vector of non-negative real model variables based on the set `prod`

`y: binary;`	`x` is a binary model variable $y \in \{0,1\}$

`parameters:`
`  a:=set([1,1],[1,2],[2,2],[3,2]);`
`variables:`
`  x[a]: real[0..];`
	defines a sparse matrix of non-negative real model variables based on the set `a` of 2-tupels.

Different indices may cause model variables to have different types. (e.g. the following is permissible: variables: x[1]: real; x[2]: integer;)

If a name is used for a model variable definition, different usages of this name with indices can only refer to model variables and not to parameters.

2.1.3.3 Indices and sets

Sets are used for the definitions of arrays of parameters or model variables and for iterations in loops. Indices are necessary to identify an element of an array like a vector or matrix of parameters or variables.

An index is always an *n*-tuple (pair of *n* entries), where *n* is the count of dimensions of an array. Entries are single integers or strings. If *n*>1 then the entries have to be separated by commas.

Usage:

```
[ entry-1 [, entry-2, … , entry-n]  ]    # n-Tuple
```

A set is a collection of indices. Sets can be defined by an enumeration of elements or by algorithms within the `parameters` section. A set can be stored in a scalar parameter or in an element of an array of parameters. A set can also be defined by using a cmplData file and a corresponding CMPL header option.

Usage of set definitions:

```
startNumber(in/decrementor)endNumber          #algorithmic 1-tuple set
[startNumber]..[endNumber]                    #algorithmic 1-tuple set

.integer.                                     #algorithmic 1-tuple set
.string.                                      #algorithmic 1-tuple set

set(entry-1 [, entry-2, … , entry-n])         #enumeration 1-tuple set
set(n-tuple-1 [, n-tuple-2, … ,n-tuple-n])    #enumeration (n>1)-tuple set
```

startNumber`(`*in/decrementor*`)`*endNumber*	1-tuple set of integers based on an algorithm The set starts at the *startNumber*, is changed by an *incrementer* or *decrementer* at every iteration and ends at the *endNumber*.
startNumber`..`*endNumber*	1-tuple set of integers based on an algorithm The set starts at the *startNumber*, is changed by the number one at every iteration and ends at the *endNumber*. `startNumber` and `endNumber` are optional elements.
startNumber`..`	infinite 1-tuple set with all integers greater than or equal to *startNumber*
`..`*endNumber*	infinite 1-tuple set with all integers less than or equal to *endNumber*
`..`	infinite 1-tuple set with all integers and strings
`.integer.`	infinite 1-tuple set with all integers

.string.	infinite 1-tuple set with all strings
set(*entry-1 [, entry-2, … , entry-n]****)***	definition of a 1-tuple enumeration set An enumeration 1-tuple set consists of one or more integer expressions or string expressions separated by commas and embedded in brackets, and is described by the key word `set`.
set(*n-tuple-1[,n-tuple-2,…]****)***	definition of an *n*-tuple enumeration set with *n*>1 An enumeration (*n*>1)-tuple set consists of one or more tuples separated by commas and embedded in brackets, and is described by the key word `set`.

Examples:

`s:=..;`	`s` is assigned an infinite 1-tuple set of all integers and strings
`s:=..6;`	`s` is assigned $s \in (\ldots, 4, 5, 6)$
`s:=6..;`	`s` is assigned $s \in (6, 7, 8, \ldots)$
`s:=0..6;` `s:=0(1)6;`	`s` is assigned $s \in (0, 1, \ldots, 6)$
`s:=10(-2)4;`	`s` is assigned $s \in (10, 8, 6, 4)$
`prod := set( "bike1", "bike2");`	enumeration 1-tuple set of strings

`a:= set( 1, "a", 3, "b", 5, "c" );`
 	
`x[a]:=(10,20,30,40,50,60);`
 	
`echo x[1];`
`echo x["a"];`
`{i in a: echo x[i];}`
	enumeration 1-tuple set of strings and integers
	
	vector `x` identified by the set `a` is assigned an integer vector
	
	The following user messages are displayed:
	`10`
	`20`
	`10 20 30 40 50 60`

`a:=[1,2];`	`a` is assigned a 2-tuple of integers
`b:=["p1","p2"];`	`b` is assigned a 2-tuple of strings
`routes := set([1,1],[1,2],[1,4],` `[2,2],[2,3],[2,4],[3,1],[3,3] );`	`routes` is assigned a 2-tuple set of integers
`c[routes]:=(3, 2, 6, 5, 2, 3, 2, 4);`	The parameter array is defined over routes and is assigned `3, 2, 6, 5, 2, 3, 2, 4.`

`{ [i,j] in routes:`
`  echo "["+i+","+j+"]: " + c[i,j];`
`}`
	The following user messages are displayed:
	`[1,1]: 3`
	`[1,2]: 2`
	`[1,4]: 6`
	`[2,2]: 5`
	`[2,3]: 2`
	`[2,4]: 3`
	`[3,1]: 2`
	`[3,3]: 4`

2.1.3.4 Line names

Line names are useful in huge models to provide a better overview of the model. In CMPL a line name can be defined by characters, numbers and the underscore character _ followed by a colon. Names that are used for parameters or model variables cannot be used for a line name. Within a control structure a line name can include the current value of local parameters. This is especially useful for local parameters which are used as a loop counter.

Usage:

```
lineName:

lineName$k$:

lineName$1$:
lineName$2$:

loopName { controlStructure }
```

lineName:	Defines a line name for a single row of the model. If more than one row is to be generated by CMPL, then the line names are extended by numbers in natural order.
$*k*$	`$k$` is replaced by the value of the local parameter k.
1	`$1$` is replaced by the number of the current line of the matrix.
2	In an implicit loop `$2$` is replaced by the specific value of the free index.
loopName{*controlStructure*}	Defines a line name subject to the following control structure. The values of loop counters in the control structure are appended automatically.

Examples:

`parameters:`
`  A[1..2,1..3] :=((1,2,3),(4,5,6));`
`  b[1..2] := (100,100);`
`  c[1..3] := (20,10,10);`
 	
`variables:`
`  x[1..3]: real[0..];`
 	
`objectives:`
`  profit: c[]T *x[] ->max;`
	generates a line `profit`

`constraints:` `  restriction: A[,] * x[] <=b[];`	generates 2 lines `restriction_1` `restriction_2`

`{ i:=1(1)2:`
`  restriction_$i$: A[,]*x[]<=b[];`
`}`
	generates 2 lines	`restriction_1`
		`restriction_2`

`restriction { i:=1(1)3: A[,]*x[]<=b[]; }`	generates 2 lines	`restriction_1` `restriction_2`

`parameters:`
`  products:=set("P1", "P2", "P3");`
`  machines:=set("M1","M2");`
`  A[machines,products] :=( (1,2,3),`
`  (4,5,6));`
`  b[machines] := (100,100);`
`  c[products] := (20,10,10);`
`variables:`
`  x[products]: real[0..];`
`objectives:`
`  profit: c[]T *x[] ->max;`
`constraints:`
`  capa_$2$: A[,] * x[] <=b[];`
	generates 3 lines	`profit`
		`capa_M1`
		`capa_M2`

2.1.4 CMPL header

A CMPL header is intended to define CMPL options, solver options and display options for the specific CMPL model. An additional intention of the CMPL header is to specify external data files which are to be connected to the CMPL model. The elements of the CMPL header are not part of the CMPL model and are processed before the CMPL model is interpreted.

Usage CMPL header for CMPL options, solver options and display options:

```
%arg optionName [optionValue]                          #CMPL options

%opt solverName solverOption [solverOptionValue]       #Solver options

%display var|con name[*] [name1[*]] ...                #Display options
%display nonZeros                                      #Display option
%display solutionPool                                  #Display option
```

optionName [*optionValue*]	All CMPL command line arguments excluding a new definition of the input file. Please see subchapter 3.3.2.
solverName	In this version are only solver options for `cbc`, `glpk` and `gurobi` supported.
solverOption [*solverOptionValue*]	Please see to the solver specific parameters subchapter 6 Appendix.

var|con *name*[*****] [*name1*[*****]] — Sets variable name(s) or constraint name(s) that are to be displayed in one of the solution reports. Different names are to be separated by spaces.

If `name` is combined with the asterix `*` then all variables or constraints with names that start with `name` are selected.

nonZeros — Only variables and constraints with nonzero activities are shown in the solution report.

solutionPool — Gurobi and Cplex are able to generate and store multiple solutions to a mixed integer programming (MIP) problem. With the display option **`solutionPool`** feasible integer solutions found during a MIP optimisation can be shown in the solution report. It is recommended to control the behaviour of the solution pool by setting the particular Gurobi or Cplex solver options.

Examples:

`%arg -solver glpk`	GLPK is used as the solver.
`%arg -solutionAscii`	CMPL writes the optimisation results in an ASCII file.

`%arg -solver cbc`
`%arg -cmplUrl ↲`
`  http://194.95.44.187:8008`
	CBC is to be executed on a CMPLServer located at 194.95.44.187 .

`%opt glpk nopresol`	If GLPK is used then the presolver is switched off.
`%display var x`	Only the variable `x` is to be displayed in the solution report.
`%display con x* y*`	All constraints with names that start with `x` or `y` are shown in the solution report.

If an external cmplData file is to be read into the CMPL model then a user can specify the file name and the needed parameters and sets within the CMPL header. All definitions of the parameters and sets can be mixed with another. The syntax of a cmplData is described in subchapter 2.3.2 cmplData files.

Usage CMPL header for defining external data:

```
%data [filename] : [set1 set[[rank]]] [, set2 set[[rank]] , … ]
%data [filename] : [param1] [, param2 , … ]
%data [filename] : [paramarray1[set]] [, paramarray2[set] , … ]
```

filename — file name of the cmplData file

If the file name contains white spaces the name can be enclosed in double quotes.

	If *filename* is not specified a generic name `modelname.cdat` will be used.
[set1 **set***[***[***rank***]]][,set2* **set***[***[***rank***]], …]*	specifies a set with the name *set1* and the rank *rank* The `rank` defines the number *n* of the entries in the *n*-tuples that are contained in the set. For 1-tuple sets is the definition of the `rank` optional. For more than one set the sets are to be separated by commas.
[param1] [, param2 , …]	specifies a scalar parameter If more than one parameters are to be specified then the parameters are to be separated by commas.
*[paramarray1***[***set***]][,***paramarray2***[***set***],…]***	specifies a parameter array and the set over which the array is defined For more than one parameter array the entries are to be separated by commas.

The easiest form to specify external data is `%data`. In this case a generic filename `modelname.cdat` will be used and all sets and parameters that are defined in `modelname.cdat` will be read.

Examples:

`%data myProblem.cdat : n set,  a[n]`	reads the 1-tuple set `n` and the vector `a` which is defined over the set `n` from the file `myProblem.cdat`
`%data myProblem.cdat`	reads all parameters and sets that are defined in the file `myProblem.cdat`
`%data : n set[1],  a[n]`	reads (assuming a CMPL model name `myproblem2.cmpl`) the 1-tuple set `n` and the vector `a` which is defined over `n` from `myProblem2.cdat`.
`%data`	Assuming a CMPL model name `myproblem2.cmpl` all sets and parameters are to be read from `myProblem2.cdat`.
`%data : routes set[2], costs[routes]`	Assuming a CMPL model name `myproblem.cmpl` the 2-tuple set `routes` and the matrix `costs` defined over `routes` are to be read from `myProblem.cdat`.

2.2 Parameter Expressions

2.2.1 Overview

Parameter expressions are rules for computing a value during the run-time of a CMPL programme. Therefore a parameter expression generally cannot include a model variable. Exceptions to this include special functions whose value depends solely on the definition of a certain model variable. Parameter expressions are a part of an assignment to a parameter or are usable within the echo function. Assignments to a parameter are only permitted within the `parameters` section or within a control structure. An expression can be a single number or string, a function, a set or a tupel. Therefore only real, integer, binary, string, set or tupel expressions are possible in CMPL. A parameter expression can contain the normal arithmetic operations.

2.2.2 Array functions

With the following functions a user may identify specific characteristics of an array or a single parameter or model variable.

Usage:

```
max(expressions)#returns the numerically largest of a list of values
min(expressions)#returns the numerically smallest of a list of values

count(parameter|variable|parameterArray|variableArray)
                      #returns the count of elements or 0 if the parameter
                      #or the variable does not exist
```

expressions can be a list of numerical expressions separated by commas or can be a multi-dimensional array of parameters

Examples:

`a[]:= (1,2,5);`
`echo max( a[] );`
`echo min( a[] );`
	
	returns user message `5`
	returns user message `1`

`echo count( a[] );`	returns user message `3`

`echo count( a[1] );`
`echo count( a[5] );`
`echo count( a[] );`
	returns user message `1`
	returns user message `0`
	returns user message `3`

`b[,]:=((1,2,3,4), (2,3,4,5));`
`echo count(b[1,]);`
`echo count(b[,1]);`
`echo count(b[,]);`
`echo count(b[1,35]);`
	user messages:
	`4` - 4 elements in the first row
	`2` - 2 elements in the first column
	`8` - 4 x 2 elements in the entire matrix
	`0` - parameter does not exist

```
a := set( [1,1],[1,2],[2,2],[3,2] );
b[a] := ( 10, 20, 30 , 40);
echo "count : " + count(b[a]);
echo "min   : " + min(b[a]);
echo "max   : " + max(b[a]);
```

user messages:

```
count : 4
min   : 10
max   : 40
```

2.2.3 Set operations and functions

Set operations and functions can be used to manipulate sets, to create sets or to analyse the characteristics of a set.

Usage:

```
set{ controlHeader : bodyExpressions  };
                      #condition set (only for 1-tupel sets)

set1 + set2;          #union set (only for 1-tupel sets)


set1 * set2;          #intersection set

len(set)              #count of the elements of the set - returns an integer

defset(array)         #returns the set of the first free index of the array
                      #only useful for dense arrays

index << set          #returns 1 - if the index is an element of the set
                      #returns 0 - otherwise

set *> [tuppelPattern]      #Set pattern matching
                            /*Returns an n-tuple set consisting of unique
                            elements of the set set which match tuplePattern
                            in the order of their first appearance. */
```

set{*controlHeader*:*bodyExpressions***}**	Constructs a set consisting of the *bodyexpressions* that satisfy the conditions in the *controlHeader.*
set1 **+** *set2*	union of *set1* and *set2*
set1 ***** *set2*	intersection of *set1* and *set2*
array	array of parameters or model variables with at least one free index.
set ***>** **[***tuppelPattern***]**	Returns an *n*-tuple set consisting of unique elements of the set *set* which match *tuplePattern* in order of their first appearance.

A *tuplePattern* have to be formulated in the form of a tuple and has to have the same rank as the original *set*.

The following entries are allowed and to be separated by commas.

`*` all elements at the position of the indexing entry

`/` ignore all elements at the position of the indexing entry

`string` or `integer`

A `string` or `integer` fixes the indexing entry at its specific position. The fixed indexing entry will not be returned by the set pattern matching expression. It is also possible to use a parameter which is assigned a `string` or `integer`.

`*string` or `*integer`

Fixes also the indexing entry at the specific position, but returns the fixed indexing entry too.

Can also be understood as an intersection of two sets followed by a rank reduction controlled by `*` or `/`.

Examples:

`s1 := set( "a","b","c","d");`
`s2 := set( "a","e","c","f");`
 	
`s3 := s1 + s2;`
`s4 := s1 * s2;`
	
	`s3` is assigned `("a","b","c","d","e","f" )`
	`s4` is assigned `( "a", "c" )`

`s5 := set{i in 1..10, i mod 2 = 0: i};`	`s5` is assigned `(2,4,6,8,10)`
`s6 := set{i in s1, !(i << s2): i};`	`s6` is assigned `( "b", "d" )`

`a := set( [1,1],[1,2],[2,2],[3,2] );`
`b := set( [1,1],[4,4],[2,2],[3,7] );`
 	
`c := a * b;`
	
	`c` is assigned the set `( [1, 1], [2, 2] )`

`a:= set( 1, "a", 3, "b", 5, "c" );`
 	
`echo "length of the set: "+ len(a);`
	returns the user message
	
	`length of the set: 6`

`A[,] := ( ( 1,2,3,4,5),`
`(1,2,3,4,5,6,7));`
`row := defset(A[,]);`
`col := defset(A[1,]);`
	
	`row` is assigned the set `1..2`
	`col` is assigned the set `1..5`

`a:= set( 1, "a", 3, "b", 5, "c" );`
 	
`echo "a" << a;`
`echo 5 << a;`
`echo "bb" << a;`
	
	returns the user message `1`
	returns the user message `1`
	returns the user message `0`

`a := set( [1,1],[1,2],[2,2],[3,2] );`
 	
`echo len(a);`
`echo [1,1] << a;`
`echo [1,7] << a;`
	
	returns the user message `4`
	returns the user message `1`
	returns the user message `0`

`a := set( [1,1],[1,2],[2,2],[3,2] );`
`b := [1..2, 1..2];`
 	
`echo a * b;`
	
	returns the user message
	`set([1, 1], [1, 2], [2, 2])`

`a := set( [1,1],[1,2],[2,2],[3,2] );`
`b := [.. , 2];`
 	
`echo a * b;`
	
	returns the user message
	`set([1, 2], [2, 2], [3, 2])`

`b:=set([1,1],[1,2],[1,4],[2,2],[2,3],`
`[2,4],[3,1],[3,3]);`
`echo b *> [*,/];`
`echo b *> [/,*];`
`echo b *> [*1,*];`
`echo b *> [1,*];`
	
	displays the user messages:
	`[1..3]`
	`[set( 1, 2, 4, 3 )]`
	`set([1, 1], [1, 2], [1, 4])`
	`[set( 1, 2, 4 )]`

`c:=set([1,1,1],[1,2,2],[2,2,5],[3,2,2]);`
`echo  c *> [1,*,*];`
`p:=2;`
`echo  c *> [*,p,*];`
`echo  c *> [*,*p,*];`
`echo  c *> [1,/,*];`
	displays the user messages:
	`set([1, 1], [2, 2])`
	
	`set([1, 2], [2, 5], [3, 2])`
	`set([1, 2, 2], [2, 2, 5], [3, 2, 2])`
	`[1..2]`

2.2.4 Mathematical functions

In CMPL there are the following mathematical functions which can be used in expressions. Excluding `div` and `mod` all these functions return a real value.

Usage:

```
p div q        #integer division
p mod q        #remainder on division
sqrt( x )      #sqrt function
exp( x )       #exp function
ln( x )        #natural logarithm
lg( x )        #common logarithm
ld( x )        #logarithm to the basis 2
```

```
srand( x )     #Initialisation of a pseudo-random number generator using the
               argument x. Returns the value of the argument x.
rand( x )      #returns an integer random number in the range 0<= rand <= x
sin( x )       #sine measured in radians
cos( x )       #cosine measured in radians
tan( x )       #tangent measured in radians
acos( x )      #arc cosine measured in radians
asin( x )      #arc sine measured in radians
atan( x )      #arc tangent measured in radians
sinh( x )      #hyperbolic sine
cosh( x )      #hyperbolic cosine
tanh( x )      #hyperbolic tangent
abs( x )       #absolute value
ceil( x )      #smallest integer value greater than or equal to a given value
floor( x )     #largest integer value less than or equal to a given value
round( x )     #simple round
```

p, q	integer expression
x	real or integer expression

Examples:

```
c[1] := sqrt( 36 );
c[2] := exp(10);
c[3] := ln(10);
c[4] := lg(10000);
c[5] := ld(8);
c[6] := rand(10);
c[7] := sin(2.5);
c[8] := cos(7.7);
c[9] := tan(10.1);
c[10] := acos(0.1);
c[11] := asin(0.4);
c[12] := atan(1.1);
c[13] := sinh(10);
c[14] := cosh(3);
c[15] := tanh(15);
c[16] := abs(-12.55);
c[17] := ceil(12.55);
c[18] := floor(-12.55);
c[19] := round(12.4);
c[20] := 35 div 4;
c[21] := 35 mod 4;
```

```
value is:
6.000000
22026.465795
2.302585
4.000000
3.000000
7.000000    (random number)
0.598472
0.153374
0.800789
1.470629
0.411517
0.832981
11013.232875
10.067662
1.000000
12.550000
13.000000
-13.000000
12.000000
8
3
```

2.2.5 Type casts

It is useful in some situations to change the type of an expression into another type. A set expression can only be converted to a string. A string can only be converted to a numerical type if it contains a valid numerical string. Every expression can be converted to a string.

Usage:

```
type(expression)      #type cast
```

type	Possible types are: `real, integer, binary, string`.
expression	expression

Examples:

	returns the user messages:

`a := 6.666;`
`echo integer(a);`
`echo binary(a);`
`a:=0;`
`echo binary(a);`
`a := 6.6666;`
`echo string(a);`
	
	`7`
	`1`
	
	`0`
	
	`6.666600`

`b := 100;`
`echo real(b);`
`echo binary(b);`
`b := 0;`
`echo binary(b);`
`b:= 100;`
`echo string(b);`
	
	`100.000000`
	`1`
	
	`0`
	
	`100`

`c :=1;`
`echo real(c);`
`echo integer(c);`
`echo string(c);`
	
	`1.000000`
	`1`
	`1`

`e := "1.888";`
`echo real(e);`
`echo integer(e);`
`echo binary(e);`
`e := "";`
`echo binary(e);`
	
	`1.888000`
	`1`
	`1`
	
	`0`

2.2.6 String operations

Especially for displaying strings or numbers with the echo function there are string operations to concatenate and format strings.

Usage:

```
expression + expression                 #concat strings if one expression
                                        #has the type string

format( formatString, expression )      #converts a number into a
                                        #string using a format string

len( stringExpression )                 #length of a string


type( expression )                      #returns the type of the expression
                                        #as a string
```

expression	expression which is converted to string Cannot be a set expression. Such an expression must be converted to a string expression by a type cast
formatString	a string expression containing format parameters CMPL uses the format parameters of the programming language C++. For further information please consult a C++ manual.

Usage format parameters:

```
%<flags><width><.precision>specifier
```

specifier	
d	integer
f	real
s	string

flags	
-	left-justify
+	Forces the result to be preceded by a plus or minus sign (+ or -) even for positive numbers. By default only negative numbers are preceded with a - sign.
width	
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The *width* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	
.number	For integer specifiers `d`: precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of `0` means that no character is written for the value `0`. For `f`: this is the number of digits to be printed after the decimal point. For `s`: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. When no precision is specified, the default is `1`. If the period is specified without an explicit value for precision, `0` is assumed.
*.**	The *precision* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

Examples:

```
a:=66.77777;
echo type(a)+ " " + a + " to string
" + format("%10.2f", a);
```

returns the user message

```
real 66.777770 to string 66.78
```

If you would like to display an entire set concatenating with a string, then you have to use a string cast of your set.

Example:

```
s:= set( 7, "qwe", 6, "fe", 5, 8 );
echo "set is " + string(s);
```

returns the user message

```
set is set( 7, "qwe", 6, "fe", 5, 8 )
```

2.3 Input and output operations

The CMPL input and output operations can be separated into message function, a function that reads the external data and the include statement that reads external CMPL code.

2.3.1 Error and user messages

Both kinds of message functions display a string as a message. In contrast to the echo function an error message terminates the CMPL programme after displaying the message.

Usage:

```
error expression;      #error message - terminates the CMPL programme

echo expression;       #user message
```

expression A message that is to be displayed. If the expression is not a string it will be automatically converted to string.

Examples:

`{a<0: error "negative value"; }`	If `a` is negative an error message is displayed and the CMPL programme will be terminated.
`echo "constant definitions finished";`	A user message is displayed.

`{ i:=1(1)3: echo "value:" + i;}`	The following user messages are displayed:
	`value: 1`
	`value: 2`
	`value: 3`

2.3.2 cmplData files

A cmplData file is a plain text file that contains the definition of parameters and sets with their values in a specific syntax. The parameters and sets can be read into a CMPL model by using the CMPL header argument `%data`.

Usage:

```
%name < numberOrString >                    # scalar parameter

%name set[[rank]] < setExpression >         # set definition

%name [set] [= default] [indices] < listOfNumbersOrStrings >
                                            # parameter array
#text                                       # comments
```

Excluding comments each cmplData definition starts with `%`.

%name **<** *numberOrString* **>**

a scalar parameter `name` is assigned a single string or number

%name **set***[***[***rank***]***]* **<** *setExpression* **>**

definition of an *n*-tuple set

A set definition starts with the *name* followed by the keyword **set**. For *n*-tuple sets with *n*>1 the rank of the set is to be specified enclosed by square brackets.

For enumeration sets the entries of the sets are separated by white spaces and imbedded in angle brackets. It is also possible to define algorithmic sets in normal CMPL syntax.

%name ***[****set****]*** *[***=** *default]* *[****indices****]*
< *listOfNumbersOrStrings* **>**

definition of a parameter array

The specification of a parameter array starts with the `name` followed by one or more sets, over which the array is defined. If more than one set is used then the sets have to be separated by commas.

The set or sets have to be defined before the parameter definition.

If the data entries are specified by their indices (keyword ***indices***) then a default value can be defined.

The data entries can be strings or numbers and have to be separated by white spaces and imbedded in angle brackets.

If the data entries are specified by their indices then each data entry has to start with the indices followed by the value and separated by white spaces.

If not so then the order of the elements are given by the natural order of the set or sets.

Examples:

`%a < 10 >`	Defines a scalar parameter `a` and assigns the number 10.
`%s set < 0..6 >` `%s set < 0..6 >`	`s` is assigned $s \in (0,1,\ldots,6)$
`%s set < 10(-2)4 >`	`s` is assigned $s \in (10,8,6,4)$
`%prod set < bike1 bike2 >` `%prod set < "bike 1"  "bike 2" >`	1-tuple enumeration set of strings
`%a set< 1 a  3  b 5 c >` `%x[a] < 10 20 30 40 50 60 >`	1-tuple enumeration set of strings and integers vector `x` identified by the set `a` is assigned an integer vector

`%data : a set, x[a]`
 	
`parameters:`
`  echo x[1];`
`  echo x["a"];`
`  {i in a: echo x[i];}`
	reads the set `a` and the vector `x` into a CMPL model
	The following user messages are displayed:
	`10`
	`20`
	`10 20 30 40 50 60`

`%n set < 1..3 >`
`%m set < 1..3 >`
 	
`%a[n,m] = 0 indices <  1 1 1`
`2 2 1`
`3 3 1 >`
	defines a 3x3 identity matrix

`%x set < 1..2 >`
`%y set < 1..2 >`
`%z set < 1..2 >`
	definition of a data cube with the dimension `x,y,z`

```
%cube[x,y,z] < 1 2 3 4 5 6 7 8 >
```

x	y	z	*value*
1	1	1	1
1	1	2	2
1	2	1	3
1	2	2	4
2	1	1	5
2	1	2	6
2	2	1	7
2	2	2	8

```
%data : x set, y set, z set, cube[x,y,z]


parameters:
  {i in x, j in y, k in z:
      echo i+","+j+","+k+":"+cube[i,j,k];
  }
```

reads the sets `x,y,z` and the `cube` into a CMPL model

The following user messages are displayed:

```
1,1,1:1
1,1,2:2
1,2,1:3
1,2,2:4
2,1,1:5
2,1,2:6
2,2,1:7
2,2,2:8
```

```
%cube[x,y,z] = 0 indices < 1 1 1 1
                           2 2 2 1 >
```

defines the following data cube

x	y	z	*value*
1	1	1	1
1	1	2	0
1	2	1	0
1	2	2	0
2	1	1	0
2	1	2	0
2	2	1	0
2	2	2	1

```
%x set[3] <  1 1 1
             1 1 2
             1 2 1
             1 2 2
             2 1 1
             2 1 2
             2 2 1
             2 2 2 >

%cube[x] < 1 2 3 4 5 6 7 8 >
```

cube defined over a 3-tuple set

x	y	z	*value*
1	1	1	1
1	1	2	2
1	2	1	3
1	2	2	4
2	1	1	5
2	1	2	6
2	2	1	7
2	2	2	8

```
%data : x set[3], cube[x]

parameters:
   {i in x:  echo i +":"+cube[i]; }
```

reads the 3-tuple set `x` and `cube`

The following user messages are displayed:

```
[1, 1, 1]:1
[1, 1, 2]:2
[1, 2, 1]:3
[1, 2, 2]:4
[2, 1, 1]:5
[2, 1, 2]:6
[2, 2, 1]:7
[2, 2, 2]:8
```

```
%x set[3] <  1 1 1   1 1 2   1 2 1   1 2 2
             2 1 1   2 1 2   2 2 1   2 2 2 >

%cube[x] = 0 indices <      1 1 1 1
                            2 2 2 1 >
```

data `cube` defined over `x`

x	*y*	*z*	*value*
1	1	1	1
1	1	2	0
1	2	1	0
1	2	2	0
2	1	1	0
2	1	2	0
2	2	1	0
2	2	2	1

```
%routes set[2] <   p1 c1
                   p1 c2
                   p1 c4
                   p2 c2
                   p2 c3
                   p2 c4
                   p3 c1
                   p3 c3  >
%c[routes]  < 3  2  6  5  2  3  2  4 >
```

defines a 2-tuple set `routes` and a matrix `c` that is defined over `routes`

```
%data : routes set[2], c[routes]

parameters:
  {i in routes: echo i + " : "+ c[i];}
```

reads the 2-tuple set `routes` and the matrix `c` into a CMPL model

The following user messages are displayed:

```
["p1", "c1"] : 3
["p1", "c2"] : 2
["p1", "c4"] : 6
["p2", "c2"] : 5
["p2", "c3"] : 2
["p2", "c4"] : 3
["p3", "c1"] : 2
["p3", "c3"] : 4
```

2.3.3 Readcsv and readstdin

CMPL has two additional functions that enable a user to read external data. The function `readstdin` is designed to read a user's numerical input and assign it to a parameter. The function `readcsv` reads numerical data from a CSV file and assigns it to a vector or matrix of parameters. For a vector with a length *n* to be read into a CMPL model the data in the CSV file can be organized as one row with *n* elements or *n* rows with one element. But in CMPL this vector is always a column vector.

Usage:

```
readstdin(message);                 #returns a user numerical input

readcsv(fileName);                  #reads numerical data from a csv file
                                    #for assigning these data to an array
```

message string expression for the message that is to be displayed

fileName string expression for the file name of the CSV file (relative to the directory in which the current CMPL file resides)
In CMPL CSV files that use a comma or semicolon to separate values are permitted.

Example:

`a := readstdin("give me a number");`	reads a value from `stdin` to be used as value for `a`. Only recommended when using CMPL as a command line interpreter.

The following example uses three CSV files:

1 2 3	c.csv
5.6;7.7;10.5 9.8;4.2;11.1	a.csv
15;20	b.csv

parameters:
  c[] := readcsv("c.csv");
  b[] := readcsv("b.csv");
  A[,] := readcsv("a.csv");
variables:
  x[defset(c[])]: real[0..];
objectives:
  c[]T * x[]->max;
constraints:
  A[,] * x[] <= b[];
	Using `readcsv` CMPL generates the following model:
	$1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 \rightarrow max\ !$
	$s.t.$
	$5.6 \cdot x_1 + 7.7 \cdot x_2 + 10.5 \cdot x_3 \leq 15$
	$9.8 \cdot x_1 + 4.2 \cdot x_2 + 11.1 \cdot x_3 \leq 20$
	$x_j \geq 0\ ;\ j = 1(1)3$

2.3.4 Include

Using the `include` directive it is possible to read external CMPL code in a CMPL programme. The CMPL code in the external CMPL file can be used by several CMPL programmes. This makes sense for sharing basic data in a couple of CMPL programmes or for the multiple use of specific CMPL statements in several CMPL programmes. The `include` directive can stand in any position in a CMPL file. The content of the included file is inserted at this position before parsing the CMPL code. Because `include` is not a statement it is not closed with a semicolon.

Usage:

```
include "fileName"          #include external CMPL code
```

fileName — file name of the CMPL file (relative to the directory in which the current CMPL file resides)

Note that *fileName* can only be a literal string value. It cannot be a string expression or a string parameter.

The following CMPL file "const-def.gen" is used for the definition of a couple of parameters:

```
c[] := ( 1, 2, 3 );
b[] := ( 15, 20 );
A[,] := (   ( 5.6, 7.7, 10.5 ),
            ( 9.8,4.2,11.1) );
```

const-def.gen

```
parameters:
    include "const-def.cmpl"

variables:
    x[defset(c[])]: real[0..];

objectives:
    c[]T * x[] -> max;
constraints:
    A[,] * x[] <= b[];
```

Using the `include` statement CMPL generates the following model:

$$1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 \rightarrow max\ !$$
$$s.t.$$
$$5.6 \cdot x_1 + 7.7 \cdot x_2 + 10.5 \cdot x_3 \leq 15$$
$$9.8 \cdot x_1 + 4.2 \cdot x_2 + 11.1 \cdot x_3 \leq 20$$
$$x_j \geq 0;\ j = 1(1)3$$

2.4 Statements

As mentioned earlier, every CMPL programme consists of at least one of the following sections: `parameters:`, `variables:`, `objectives:` and `constraints:`. Each section can be inserted several times and mixed in a different order. Every section can contain special statements. Every statement finishes with a semicolon.

2.4.1 `parameters` and `variables` section

Statements in the `parameters` section are assignments to parameters. These assignments define parameters or reassign a new value to already defined parameters. Statements in the `variables` sections are definitions of model variables.

All the syntactic and semantic requirements are described in the chapters above.

2.4.2 `objectives` and `constraints` section

In the `objectives` and `constraints` sections a user has to define the content of the decision model in linear terms. In general, an objective function of a linear optimisation model has the form:

$$c_1 \cdot x_1 + c_2 \cdot x_2 + \ldots + c_n \cdot x_n \;\rightarrow\; max\,! \quad (\text{or } min\,!)$$

with the objective function coefficient c_j and model variables x_j . Constraints in general have the form:

$$\begin{aligned} k_{11} \cdot x_1 + k_{12} \cdot x_2 + \ldots + k_{1n} \cdot x_n &\le b_1 \\ k_{21} \cdot x_1 + k_{22} \cdot x_2 + \ldots + k_{2n} \cdot x_n &\le b_2 \\ &\vdots \\ k_{m1} \cdot x_1 + k_{m2} \cdot x_2 + \ldots + k_{mn} \cdot x_n &\le b_m \end{aligned}$$

with constraint coefficients k_{ij} and model variables x_j .

An objective or constraint definition in CMPL must use exactly this form or a sum loop that expresses this form. A coefficient can be an arbitrary numerical expression, but the model variables cannot stand in expressions that are different from the general form formulated. The rule that model variables cannot stand in bracketed expressions serves to enforce this.

Please note, it is not permissible to put model variables in brackets!

The example (`a` and `b` are parameters, `x` and `y` model variables)

```
a*x + a*y + b*x + b*y
```

can be written alternatively (with parameters in brackets) as:

```
(a + b)*x + (a + b)*y
```

but not (with model variables in brackets) as:

```
a*(x + y) + b*(x + y)
```

For the definition of the objective sense in the `objectives` section the syntactic elements `->max` or `->min` are used. A line name is permitted and the definition of the objective function has to have a linear form.

Usage of an objective function:

```
objectives:
          [lineName:] linearTerm ->max|->min;
```

lineName	optional element description of objective
linearTerm	definition of linear objective function

The definition of a constraint has to consist of a linear definition of the use of the constraint and one or two relative comparisons. Line names are permitted.

Usage of a constraint:

```
constraints:
          [lineName:] linearTerm <=|>=|= linearTerm  [<=|>=|= linearTerm];
```

lineName	optional element description of objective
linearTerm	linear definition of the left-hand side or the right-hand side of a constraint

2.5 Control structure

2.5.1 Overview

A control structure is imbedded in `{ }` and defined by a header followed by a body separated off by `:`.

General usage of a control structure:

```
[controlName]|[sum|set] { controlHeader : controlBody }
```

A control structure can be started with an optional name for the control structure. In the `objectives` and in the `constraints` section this name is also used as the line name.

It is possible to define different kinds of control structures based on different headers, control statements and special syntactical elements. Thus the control structure can used for for loops, while loops, if-then-else clauses and switch clauses. Control structures can be used in all sections.

A control structure can be used for the definition of statements. In this case the control body contains one or more statements which are permissible in this section.

It is also possible to use control structures for `sum` and `set` as expressions. Then the body contains a single expression. A control structure as an expression cannot have a name because this place is taken by the keyword `sum` or `set`. Moreover a control structure as an expression cannot use control statements because the body is an expression and not a statement.

2.5.2 Control header

A control header consists of one or more control headers. Where there is more than one header, the headers must be separated by commas. Control headers can be divided into iteration headers, condition headers, local assignments and empty headers.

2.5.2.1 Iteration headers

Iteration headers define how many repeats are to be executed in the control body. Iteration headers are based on sets.

Usage:

```
localParam in set     # iteration over a set
```

localParam	name of the local parameter
set	The defined local parameter iterates over the elements of the set and the body is executed for every element in the set.

Examples:

`s1 := set( "a","b","c","d");` `{k in s1:  … }`	`k` is iterated over all elements of the set `s1`
`s2 := 1(1)10;` `{k in s2:  … }`	`k` is iterated over the set $k \in \{1,2,\ldots,10\}$
`s3 := 2..6;` `{k := s3:  …  }`	`k` is iterated over the set $k \in \{2,3,\ldots,6\}$

`a := set( [1,1],[1,2],[2,2],[3,2] );`
`{ k in a : … }`
`a := set( [1,1],[1,2],[2,2],[3,2] );`
`{ [i,j] in a : … }`
	`k` is iterated over the 2-tuple set `a`
	
	2-tuple index `[i,j` is iterated over the 2-tuple set `a`

2.5.2.2 Condition headers

A condition returns `1(True)` or `0(False)` subject to the result of a comparison or the properties of a parameter or a set. If the condition returns `1(True)` the body is executed once or else the body is skipped.

Comparison operators for parameters:

=, ==	equality
<>, !=	inequality
<	less than
>	greater than
<=	equal to or less than
>=	equal to or greater than

Comparison operators for sets:

=	equality
==	tests whether the iteration order of two sets is equal
<>	inequality
!=	tests whether the iteration order of two sets is not equal
<	subset or not equal (only for 1-tuple sets)
>	greater than (only for 1-tuple sets)
<=	subset or equal (only for 1-tuple sets)
>=	equal to or greater than (only for 1-tuple sets)

Logical operators:

&&	AND
||	OR
!	NOT

If a real or integer parameter is assigned 0, the condition returns `0 (false)`. Alternatively if the parameter is assigned `1` the condition returns `1 (true)`.

Examples:

i:=1;	
j:=2;	
{i>j : … }	condition is false
{!(i>j) : … }	condition is true
{!i || j=2 : … }	condition is true
{!i && j=2 : … }	condition is false (`!i` is false, because `i` is not 0)

2.5.2.3 Local assignments

A local assignment as control header is useful if a user wishes to make several calculations in a local environment. Assigning expression to a parameter within the `constraints` section is generally not allowed with the exception of a local assignment within a control structure. The body will be executed once.

Usage:

```
localParam := expression     # assignment to a local parameter
```

localParam	Defines a local parameter with this name.
expression	Expression which is assigned to the local parameter.

Examples:

`constraints:` `{ k:=1 : … }`	`k` is assigned `1` and used as local parameter within the control structure.

2.5.3 Alternative bodies

If a control header consists of at least one condition, it is possible to define alternative bodies. Structures like that make sense e.g. if a user wishes to combine a for loop with an if-then clause.

The first defined body after the headers is the main body of the control structure. Subsequent bodies must be separated by the syntactic element `|`. Alternative bodies are only executed if the main body is skipped.

Usage:

```
{ controlHeader: mainBody [ | condition1: alternativeBody1 ]
                    [ | ... ] [ | default: alternativeDefaultBody ] }
```

controlHeader	header of the control structure including at least one condition The alternative bodies belong to last header of control header. This header cannot be an assignment of a local parameter, because in this case the main body is never skipped.
mainBody	main body of control structure
condition1	Will be evaluated if alternative body is executed.
alternativeBody1	The first alternative body with a condition that evaluates to true is executed. The remaining alternative bodies are skipped without checking the conditions.
alternativeDefaultBody	If no condition evaluates to true then the alternative default body is executed. If the control structure has no alternative default body, then no body is executed.

2.5.4 Control statements

It is possible to change or interrupt the execution of a control structure using the keywords `continue`, `break` and `repeat`. A `continue` stops the execution of the specified loop, jumps to the loop header and executes the next iteration. A `break` only interrupts the execution of the specified loop. The keyword `repeat` starts the execution again with the referenced header.

Every control statement references one control header. If no reference is given, it references the innermost header. Possible references are the name of the local parameter which is defined in this head, or the name of the control structure. The name of the control structure belongs to the first head in this control structure.

Usage:

```
continue [reference];
break [reference];
repeat [reference];
```

reference — a reference to a control header specified by a name or a local parameter

break *[reference]* — The execution of the body of the referenced head is cancelled. Remaining statements are skipped.

If the referenced header contains iteration over a set, the execution for the remaining elements of the set is skipped.

continue *[reference]* — The execution of the body of the referenced head is cancelled. Remaining statements are skipped.

If the referenced header contains iteration over a set, the execution is continued with the next element of the set. For other kinds of headers `continue` is equivalent to `break`.

repeat *[reference]* — The execution of the body of the referenced header is cancelled. Remaining statements are skipped.

The execution starts again with the referenced header. The expression in this header is to be evaluated again. If the header contains iteration over a set, the execution starts with the first element. If this header is an assignment to a local parameter, the assignment is executed again. If the header is a condition, the expression is to be checked prior to execution or skipping the body.

2.5.5 Specific control structures

2.5.5.1 For loop

A for loop is imbedded in `{ }` and defined by at least one iteration header followed by a loop body separated off by `:`. The loop body contains user-defined instructions which are repeatedly carried out. The number of repeats is based on the iteration header definition.

Usage:

```
{ iterationHeader [, iterationHeader1] [, ...]  : controlBody }
```

iterationHeader *iterationHeader1*	defined iteration headers
controlBody	CMPL statements that are executed in every iteration

Examples:

`{ i in 1(1)3 : ... }`	loop counter `i` with a start value of 1, an increment of 1 and an end condition of 3
`{ i in 1..3 : ... }`	alternative definition of a loop counter; loop counter `i` with a start value of 1 and an end condition of 3. (The increment is automatically defined as 1)

`products:= set( "p1", "p2", "p3");`
`hours[products]:=(20,55,10);`
`{i in products:`
`  echo "hours of product " +`
`  i + " : "+ hours[i];`
`}`
	for loop using the set `products` returns
	user messages `hours of product: p1 : 20`
	`hours of product: p2 : 55`
	`hours of product: p3 : 10`

`{i in 1(1)2:`
`  {j in 2(2)4:  A[i,j] := i + j; }`
`}`
	defines `A[1,2] = 3`, `A[1,4] = 5`, `A[2,2] = 4` and `A[2,4] = 6`

`a := set([1,1],[1,2],[2,2],[3,2]);`
`b[a] := ( 10, 20, 30 , 40);`
 	
`{ k in a : echo k + ":"+ b[k] ;}`
	`k` is iterated over the 2-tuple set `a`
	
	The following user messages are displayed:
	`[1, 1]:10`
	`[1, 2]:20`
	`[2, 2]:30`
	`[3, 2]:40`

Several loop heads can be combined. The above example can thus be abbreviated to:

`{i in 1(1)2, j in 2(2)4:`
`  A[i,j] := i + j;`
`}`
	defines `A[1,2] = 3, A[1,4] = 5, A[2,2] = 4` and `A[2,4] = 6`

`{i in 1(1)5, j in 1(1)i:`
`  A[i,j] := i + j;`
`}`
	definition of a triangular matrix

2.5.5.2 If-then clause

An if-then consists of one condition as control header and user-defined expressions which are executed if the if condition or conditions are fulfilled. Using an alternative default body the if-then clause can be extended to an if-then-else clause.

Usage:

```
{ condition: thenBody [| default: elseBody ]}
```

condition	If the evaluated condition is true, the code within the body is executed.
thenBody	This body is executed if the *condition* is true.
elseBody	This body is executed if the *condition* is false.

Examples:

`{i := 1(1)5, j := 1(1)5:`
`  {i = j:  A[i,j] := 1; }`
`  {i != j: A[i,j] := 0; }`
`}`
`{i := 1(1)5, j := 1(1)5:`
`  {i = j:  A[i,j] := 1; |`
`  default: A[i,j] := 0; }`
`}`
	definition of the identity matrix with combined loops and two if-then clauses
	
	same example, but with one if-then-else clause

`i:=10;`
`{ i<10: echo "i less than 10";`
`  | default: echo "i greater than 9";`
`}`
	example of an if-then-else clause
	returns user message `i greater than 9`

`sum{ i = j : 1 | default: 2 }`	conditional expression, evaluates to 1 if `i = j`, otherwise to 2

2.5.5.3 Switch clause

Using more than one alternative body the if-then clause can be extended to a switch clause.

Usage:

```
{ condition1: body1 [| condition2: body2>] [| ... ] [| default: defaultBody ]}
```

If the first condition returns TRUE, only *body1* will be executed. Otherwise the next condition *condition1* will be verified. *body2* is executed if all of the previous conditions are not fulfilled. If no condition returns true, then the *defaultBody* is executed.

Example:

`i:=2;`
`{  i=1: echo "i equals 1";`
`  | i=2: echo "i equals 2";`
`  | i=3: echo "i equals 3";`
`  | default: echo "any other value";`
`}`
	example of a switch clause
	returns user message `i equals 2`

2.5.5.4 While loop

A while loop is imbedded in `{ }` and defined by a condition header followed by a loop body separated off by `:` and finished by the keyword `repeat`. The loop body contains user-defined instructions which are repeatedly carried out until the condition in the loop header is false.

Usage:

```
{ condition : statements repeat; }
```

condition	If the evaluated condition is true, the code within the body is executed. This repeats until the condition becomes false.
statements	one or more user-defined CMPL instructions To prevent an infinite loop the statements in the control body must have an impact on the *condition*.

Examples:

`i:=2;`
`{i<=4:`
`  A[i] := i;`
`  i := i+1;`
`  repeat;`
`}`
	while loop with a global parameter
	Can only be used in the `parameters` section, because the assignment to a global parameter is not permitted in other sections.
	defines `A[2] = 2`, `A[3] = 3` and `A[4] = 4`

`{a := 1, a < 5:`
`  echo a;`
`  a := a + 1;`
`  repeat;`
`}`
	while loop using a local parameter
	returns user messages `1`
	`2`
	`3`
	`4`

`{a:=1:`
`  xx {:`
`  echo a;`
`  a := a + 1;`
`  {a>=4: break xx;}`
`  repeat;`
`  }`
`}`
	Alternative formulation:
	The outer control structure defines the local parameter `a`. This control structure is used as a loop with a defined name and an empty header. The name is necessary, because it is needed as reference for the `break` statement in the inner control structure. (Without this reference the `break` statement would refer to the condition a>=4)

2.5.6 Set and sum control structure as expression

Starting with the keyword `sum` or the keyword `set` a control structure returns an expression. Only expressions are permitted in the body of the control structure. Control statements are not allowed, because the body cannot contain a statement. It is possible to define alternative bodies.

Usage:

```
sum { controlHeader : bodyExpressions }
set { controlHeader : bodyExpressions }
```

controlHeader	header of the control structure The header of a sum or a set control structure is usually an iteration header, but all kinds of control header can be used.
bodyExpressions	user-defined expressions

A `sum` expression repeatedly summarises the user-defined expressions in the *bodyExpressions*. If the body is never executed, it evaluates to 0. A `set` expression returns a set subject to the *controlHeader* and the *bodyExpressions*. The element type included in *bodyExpressions* must be integer or string. Please note that the `set` expression only works for 1-tuple sets.

Examples:

`x[1..3] := (2, 4, 6);` `a := sum{i := 1(1)3 : x[i] };`	`a` is assigned `12`

`products:= set( "p1", "p2", "p3");`
`hours[products]:=(20,55,10);`
`totalHours:= sum{i in products: hours[i] };`
	`totalHours` is assigned `85`

`x[1..3,1..2]:=((1,2),(3,4),(5,6));` `b:= sum{i := 1(1)3, j := 1(1)2: x[i,j] };`	`sum` with more than one control header `b` is assigned `21`.
`s:=set();` `d:= sum{i in s: i |default: -1 };`	sums up all elements in the set `s`. Since `s` is an empty set, `d` is assigned to the alternative default value `-1`.

`a := set( [1,1],[1,2],[2,2],[3,2] );`
`b[a] := ( 10, 20, 30 , 40);`
 	
`c := sum{ k in a : b[k]};`
	calculates a sum over all elements in b which is defined over the 2-tuple set `a`.
	`c` is assigned `100`.

`e:= set{i:= 1..10: i^2 };`	e is assigned the set `(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)`
`f:= set{i:= 1..100, round(sqrt(i))^2 = i: i };`	f is assigned the set `(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)`

The `sum` expression can also be used in linear terms for the definition of objectives and constraints. In this case the body of the control structure can contain model variables.

Examples:

```
parameters:
  a[1..2,1..3] :=((1,2,3),(4,5,6));
  b[1..2] := (100,100);
  c[1..3] := (20,10,10);

variables:
  x[1..3]: real[0..];

objectives:
  sum{j:=1..3: c[j] *x[j]}->max;
constraints:
  { i:=1..2:
    sum{j:=1..3: a[i,j] * x[j]}<= b[i];
  }
```

objective definition using a sum

$$20 \cdot x_1 + 10 \cdot x_2 + 10 \cdot x_3 \rightarrow max!$$

constraints definition using a sum

$$1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 \leq 100$$
$$4 \cdot x_1 + 5 \cdot x_2 + 6 \cdot x_3 \leq 100$$

2.6 Matrix-Vector notations

CMPL allows users to define objectives and constraints in a matrix-vector notation (e.g. matrix vector multiplication). CMPL generates all required rows and columns automatically by implicit loops.

Implicit loops are formed by matrices and vectors, which are defined by the use of free indices. A free index is an index which is not specified by a position in an array. It can be specified by an entire set or without any specification. But the separating commas between indices must in any case be specified. A multidimensional array with one free index is always treated as a column vector, regardless of where the free index stands. A column vector can be transposed to a row vector with `T`. A multidimensional array with two free indices is always treated as a matrix. The first free index is the row, the second the column. Implicit loops are only possible in the `objectives` section and the `constraints` section.

Please note that matrix-vector notations only works for arrays which are defined over 1-tuple sets.

Usage:

```
vector[[set]]                    #column vector
vector[[set]]T                   #transpose of column vector  - row vector

matrix[index, [set]]             #column vector
matrix[[set], index]             #also column vector

matrix[index, [set]]T            #transpose of column vector  - row vector
matrix[[set], index]T            #transpose of column vector  - row vector

matrix[[set1], [set2]]           #matrix
```

vector, matrix	name of a vector or matrix
index	a certain index value
[set]	optional specification of a set for the free index

Examples:

`x[]`	vector with free index across the entire defined area
`x[2..5]`	vector with free index in the range 2 – 5
`A[,]`	matrix with two free indices
`A[1,]`	matrix with one fixed and one free index; this is a column vector.
`A[,1]`	matrix with one fixed and one free index; this is also a column vector.

The most important ways to define objectives and constraints with implicit loops are vector-vector multiplication and matrix-vector multiplication. A vector-vector multiplication defines a row of the model (e.g. an objective or one constraint). A matrix-vector multiplication can be used for the formulation of more than one row of the model.

Usage of multiplication using implicit loops :

```
paramVector[[set]]T * varVector[[set]]  #vector-vector multiplication
varVector[[set]]T * paramVector[[set]]  #vector-vector multiplication

paramMatrix[[set1],[set2]] * varVector[[set2]]
                                        #matrix-vector multiplication
varVector[[set1]]T * paramMatrix[[set1],[set2]]
                                        #matrix-vector multiplication
```

paramVector	name of a vector of parameters
varVector	name of a vector of model variables
paramMatrix	name of a matrix of parameters
`T`	syntactic element for transposing a vector

Examples:

```
parameters:
  a[1..2,1..3] :=( (1,2,3),
  (4,5,6));
  b[1..2] := (100,100);
  c[1..3] := (20,10,10);
variables:
  x[1..3]: real[0..];
objectives:
  c[]T * x[] ->max;
```

objective definition using implicit loops

$$20 \cdot x_1 + 10 \cdot x_2 + 10 \cdot x_3 \rightarrow max\ !$$

`constraints:` `  a[, ] * x[] <=b[];`	constraint definition using implicit loops $1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 \leq 100$ $4 \cdot x_1 + 5 \cdot x_2 + 6 \cdot x_3 \leq 100$

Aside from vector-vector multiplication and matrix-vector multiplication vector subtractions or additions are also useful for the definition of constraints. The addition or subtraction of a variable vector adds new columns to the constraints. The addition or subtraction of a constant vector changes the right side of the constraints.

Usage of additions or subtractions using implicit loops:

```
  linearTerms + varVector[set]          #variable vector addition
  linearTerms - varVector[set]          #variable vector subtraction

  linearTerms + paramVector[set]        #parameter vector addition
  linearTerms - paramVector[set]        #parameter vector subtraction
```

linearTerms other linear terms in an objective or constraint

Examples:

`parameters:`
`  a[1..2,1..3] :=( (1,2,3),`
`  (4,5,6));`
`  b[1..2] := (100,100);`
`  d[1..2] := (10,10);`
`  c[1..3] := (20,10,10);`
`variables:`
`  x[1..3]: real[0..];`	

`objectives:` `  c[]T * x[] ->max;`	
`constraints:` `  a[,] * x[] + d[] <=b[];`	constraints definition using implicit loops $1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 \leq 90$ $4 \cdot x_1 + 5 \cdot x_2 + 6 \cdot x_3 \leq 90$ equivalent to `a[,] * x[]  <=b[] - d[];`
`0 <= x[1..3]+y[1..3]+z[2]<= b[1..3];`	implicit loops for a column vector

`0 <= x[1] + y[1] + z[2] <= b[1];`
`0 <= x[2] + y[2] + z[2] <= b[2];`
`0 <= x[3] + y[3] + z[2] <= b[3];`
	equivalent formulation

`parameters:`
`  a[1..2,1..3] :=( (1,2,3),`
`  (4,5,6));`
`  b[1..2] := (100,100);`
`  d[1..2] := (10,10);`
`  c[1..3] := (20,10,10);`	

`variables:`
`  x[1..3]: real[0..];`
`  z[1..2]: real[0..];`	

`objectives:` `  c[]T * x[] ->max;`	
`constraints:` `  a[,] * x[] + z[] <=b[];`	constraints definition using implicit loops $1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 + z_1 \leq 90$ $4 \cdot x_1 + 5 \cdot x_2 + 6 \cdot x_3 + z_2 \leq 90$

2.7 Automatic model reformulations

2.7.1 Overview

CMPL includes two types of automatic code generation which release the user from additional modelling work. CMPL automatically optimizes the generated model by means of matrix reductions. The second type of automatic code reformulations is the equivalent transformation of variable products.

2.7.2 Matrix reductions

Matrix reductions are subject to constraints of a specific form.

a) If a constraint contains only one variable or only one of the variables with a coefficient not equal to 0, then the constraint is taken as a lower or upper bound.

   For the following summation (`x[]` is a variable vector)

   ```
   sum{i:=1(1)2: (i-1) * x[i]} <= 10;
   ```

   no matrix line is generated; rather `x[2]` has an upper bound of 10.

b) If there is a constraint in the coefficients of all variables proportional to another constraint, only the more strongly limiting constraint is retained.

   Only the second of the two constraints (`x[]` is a variable vector)

   ```
   2*x[1] + 3*x[2] <= 20;
   10*x[1] + 15*x[2] <= 50;
   ```

   is used in generating a model line.

Matrix reductions are switched off by default, but can be enabled by the command line argument `-gn`.

2.7.3 Equivalent transformations of Variable Products

A product of variables cannot be a part of an LP or MIP model, because such a variable product is a non-linear term. But if one factor of the product is an integer variable then it is possible to formulate an equivalent transformation using a set of specific linear inequations. [cf. Rogge/Steglich (2007)] The automatic generation of an equivalent transformation of a variable product is a unique characteristic of CMPL.

2.7.3.1 Variable Products with at least one binary variable

A product of variables with at least one binary variable can be transformed equivalently in a system of linear inequations as follows (Rogge and Steglich 2007, p. 25ff.) :

$w := u \cdot v$, $\underline{u} \le u \le \overline{u}$ (u real or integer), $v \in \{0,1\}$
is equivalent to

u real or integer, $v \in \{0,1\}$ and
$\underline{u} \cdot v \le w \le \overline{u} \cdot v$
$\underline{u} \cdot (1-v) \le u - w \le \overline{u} \cdot (1-v)$

CMPL is able to perform these transformations automatically. For the following given variables

```
variables:  x: binary;
     y: real[YU..YO];
```

each occurrence of the term `x*y` in the CMPL model description is replaced by an implicit newly-defined variable `x_y`, and the following additional statements are generated automatically:

```
constraints:
  min(YU, 0) <= x_y <= max(YO, 0);
  {YU < 0: x_y - YU*x >= 0; }
  {YO > 0: x_y - YO*x <= 0; }
  y - x_y + YU*x >= YU;
  y - x_y + YO*x <= YO;
```

2.7.3.2 Variable Product with at least one integer variable

It is also possible to formulate an equivalent system of linear in-equation for products of variables with at least one integer variable (Rogge and Steglich 2007, p. 28ff.):

$w := u \cdot v$,
$\underline{u} \le u \le \overline{u}$, ($u$ real or integer, if u integer then $\underline{v} - \overline{v} \le \underline{u} - \overline{u}$),
$\underline{v} \le v \le \overline{v}$ (v integer)
is equivalent to

u real or integer and
$v = \underline{v} + \sum_{j=0}^{d} 2^j \cdot y_j$, $v \le \overline{v}$, with $d = \lceil ld(\overline{v} - \underline{v} + 1) \rceil - 1$
$w = u \cdot \underline{v} + \sum_{j=0}^{d} 2^j \cdot w_j$
$\underline{u} \cdot y_j \le w_j \le \overline{u} \cdot y_j$
$\underline{u} \cdot (1 - y_j) \le u - w_j \le \overline{u} \cdot (1 - y_j)$

$y_j \in \{0,1\}$, $j = 0(1)d$

CMPL is able to perform these transformations automatically as described above. For the following given variables

```
variables:  x: integer[XU..XO];
     y: real[YU..YO];
```

each occurrence of the term `x*y` in the CMPL model description is replaced by an implicit newly-defined variable `x_y`, and the following additional statements are generated automatically (here `d` stands for the number of binary positions needed for `XO-XU+1`):

```
variables:
  _x[1..d]: binary;
  _x_y[1..d]: real;

constraints:
  min(XU*YU,XU*YO,XO*YU,XO*YO) <= x_y <= max(XU*YU,XU*YO,XO*YU,XO*YO);

  x = XU + sum{i=1(1)d: (2^(i-1))*_x[i]};
  x_y = XU*y + sum{i=1(1)d: (2^(i-1))*_x_y[i]};

  {i = 1(1)d:
    min(YU, 0) <= _x_y[i] <= max(YO, 0);
    {YU < 0: _x_y[i] - YU*_x[i] >= 0; }
    {YO > 0: _x_y[i] - YO*_x[i] <= 0; }
    y - _x_y[i] + YU*_x[i] >= YU;
    y - _x_y[i] + YO*_x[i] <= YO;
  }
```

2.8 Examples

2.8.1 Selected decision problems

2.8.1.1 The diet problem

The goal of the diet problem is to find the cheapest combination of foods that will satisfy all the daily nutritional requirements of a person for a week.

The following data is given (example cf. Fourer/Gay/Kernigham 2003, p. 27ff.) :

food	cost per package	provision of daily vitamin requirements in percentages			
		A	B1	B2	C
BEEF	3.19	60	20	10	15
CHK	2.59	8	2	20	520
FISH	2.29	8	10	15	10
HAM	2.89	40	40	35	10
MCH	1.89	15	35	15	15

MTL	1.99	70	30	15	15
SPG	1.99	25	50	25	15
TUR	2.49	60	20	15	10

The decision is to be made for one week. Therefore the combination of foods has to provide at least 700% of daily vitamin requirements. To promote variety, the weekly food plan must contain between 2 and 10 packages of each food.

The mathematical model can be formulated as follows:

$$3.19 \cdot x_{BEEF} + 2.59 \cdot x_{CHK} + 2.29 \cdot x_{FISH} + 2.89 \cdot x_{HAM} + 1.89 \cdot x_{MCH} + 1.99 \cdot x_{MTL} + 1.99\, x_{SPG} + 2.49 \cdot x_{TUR} \rightarrow min\,!$$

$$s.t.$$

$$60 \cdot x_{BEEF} + 8 \cdot x_{CHK} + 8 \cdot x_{FISH} + 40 \cdot x_{HAM} + 15 \cdot x_{MCH} + 70 \cdot x_{MTL} + 25\, x_{SPG} + 60 \cdot x_{TUR} \leq 700$$

$$20 \cdot x_{BEEF} + 0 \cdot x_{CHK} + 10 \cdot x_{FISH} + 40 \cdot x_{HAM} + 35 \cdot x_{MCH} + 30 \cdot x_{MTL} + 50\, x_{SPG} + 20 \cdot x_{TUR} \leq 700$$

$$10 \cdot x_{BEEF} + 20 \cdot x_{CHK} + 15 \cdot x_{FISH} + 35 \cdot x_{HAM} + 15 \cdot x_{MCH} + 15 \cdot x_{MTL} + 25\, x_{SPG} + 15 \cdot x_{TUR} \leq 700$$

$$15 \cdot x_{BEEF} + 20 \cdot x_{CHK} + 10 \cdot x_{FISH} + 10 \cdot x_{HAM} + 15 \cdot x_{MCH} + 15 \cdot x_{MTL} + 15\, x_{SPG} + 10 \cdot x_{TUR} \leq 700$$

$$x_j \in \{2,3,\ldots,10\} \quad ; j \in \{BEEF, CHK, DISH, HAM, MCH, MTL, SPG, TUR\}$$

The CMPL model `diet.cmpl` can be formulated as follows:

```
parameters:
  NUTR := set("A","B1", "B2", "C");
  FOOD := set("BEEF", "CHK", "FISH", "HAM", "MCH", "MTL", "SPG", "TUR");

  #cost per package
  costs[FOOD] := ( 3.19, 2.59, 2.29, 2.89, 1.89, 1.99, 1.99, 2.49 );
  #provision of the daily requirements for vitamins in percentages
  vitamin[NUTR, FOOD] := (  (60, 8, 8, 40, 15, 70, 25, 60) ,
                            (20, 0, 10, 40, 35, 30, 50, 20) ,
                            (10, 20, 15, 35, 15, 15, 25, 15),
                            (15, 20, 10, 10, 15, 15, 15, 10)
                         );

  #weekly vitamin requirements
  vitMin[NUTR]:= (700,700,700,700);

variables:
  x[FOOD]: integer[2..10];

objectives:
  cost: costs[]T * x[]->min;

constraints:
  # capacity restriction
  $2$: vitamin[,] * x[] >= vitMin[];
```

An alternative formulation is based on the cmplData file `diet-data.cdat` that is formulated as follows:

```
%NUTR set < A B1  B2  C >
%FOOD set < BEEF CHK FISH HAM MCH MTL SPG TUR   >

#cost per package
%costs[FOOD] < 3.19 2.59 2.29 2.89 1.89 1.99 1.99 2.49 >

#provision of the daily requirements for vitamins in percentages
%vitamin[NUTR,FOOD] <   60  8  8  40  15  70  25  60
                        20  0  10  40  35  30  50  20
                        10  20  15  35  15  15  25  15
                        15  20  10  10  15  15  15  10 >

#weekly vitamin requirements
%vitMin[NUTR]  < 700 700 700 700 >
```

Assuming that the corresponding CMPL file `diet-data.cmpl` is in the same working directory the model can be formulated as follows:

```
%data diet-data.cdat: FOOD set, NUTR set, costs[FOOD], vitamin[NUTR,FOOD],
vitMin[NUTR]

variables:
  x[FOOD]: integer[2..10];

objectives:
  cost: costs[]T * x[]->min;

constraints:
  # capacity restriction
  $2$: vitamin[,] * x[] >= vitMin[];
```

Solving this CMPL model through using the command:

```
cmpl diet-data.cmpl
```

leads to the same solution as for the first formulation:

```
-----------------------------------------------------------------------------------------------------
Problem             diet.cmpl
Nr. of variables    8
Nr. of constraints  4
Objective name      cost
Solver name         CBC
-----------------------------------------------------------------------------------------------------

Objective status    optimal
Objective value     101.14 (min!)
```

```
Variables
Name                 Type            Activity        Lower bound        Upper bound            Marginal
-------------------------------------------------------------------------------------------------------
x[BEEF]                 I                   2                  2                 10                   -
x[CHK]                  I                   8                  2                 10                   -
x[FISH]                 I                   2                  2                 10                   -
x[HAM]                  I                   2                  2                 10                   -
x[MCH]                  I                  10                  2                 10                   -
x[MTL]                  I                  10                  2                 10                   -
x[SPG]                  I                  10                  2                 10                   -
x[TUR]                  I                   2                  2                 10                   -
-------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity        Lower bound        Upper bound            Marginal
-------------------------------------------------------------------------------------------------------
A                       G                1500                700           Infinity                   -
B1                      G                1330                700           Infinity                   -
B2                      G                 860                700           Infinity                   -
C                       G                 700                700           Infinity                   -
-------------------------------------------------------------------------------------------------------
```

2.8.1.2 Production mix

This model calculates the production mix that maximizes profit subject to available resources. It will identify the mix (number) of each product to produce and any remaining resource.

The example involves three products which are to be produced with two machines. The following data is given:

		P1	P2	P3	upper bounds [h]
upper bound of a product	[units]	250	240	250	
selling price per unit	[€/unit]	500	600	450	
direct costs per unit	[€/unit]	425	520	400	
profit contribution per unit	[€/unit]	75	80	50	
machine hours required per unit					
machine 1	[h/unit]	8	15	12	1,000
machine 2	[h/unit]	15	10	8	1,000

The mathematical model can be formulated as follows:

$$75 \cdot x_1 + 80 \cdot x_2 + 50 \cdot x_3 \rightarrow max\ !$$
$$s.t.$$
$$8 \cdot x_1 + 15 \cdot x_2 + 12 \cdot x_3 \leq 1{,}000$$
$$15 \cdot x_1 + 10 \cdot x_2 + 8 \cdot x_3 \leq 1{,}000$$

$$x_1 \in \{0,1,\ldots,250\}$$
$$x_2 \in \{0,1,\ldots,240\}$$
$$x_3 \in \{0,1,\ldots,250\}$$

The CMPL model `production-mix.cmpl` is formulated as follows:

```
%arg -solver glpk
parameters:
  products := 1..3;
  machines := 1..2;

  price[products] := (500, 600, 450 );
  costs[products] := (425, 520, 400);

  #machine hours required per unit
  a[machines,products] := ((8, 15, 12), (15, 10, 8));

  #upper bounds of the machines
  b[machines] := (1000, 1000);

  #profit contribution per unit
  {j in products:  c[j] := price[j]-costs[j]; }

  #upper bound of the products
  xMax[products] := (250, 240, 250 );

variables:
  x[products]: integer;

objectives:
  profit: c[]T  * x[] ->max;

constraints:
  res: a[,] * x[] <= b[];
  0<=x[]<=xMax[];
```

The model can be formulated alternatively by using the cmplData `prodmix-data.cdat` file.

```
%products set < 1..3 >
%machines set < 1..2 >

%price[products] <500 600 450 >
%costs[products] <425 520 400 >

#machine hours required per unit
%a[machines,products] < 8  15  12  15 10 8 >

#upper bounds of the machines
%b[machines] < 1000 1000 >
```

```
#lower and upper bound of the products
%xMax[products] < 250  240  250>
%xMin[products] < 45 45 45 >

#fixed setup costs
%FC[products] < 500 400 500>
```

The parameter arrays `xMin` and `FC` are not necessary for the given problem and therefore not specified within the `%data` options in the following CMPL file `prodmix-data.cdat`:

```
%arg -solver glpk
%data : products set, machines set, price[products], costs[products]
%data : a[machines,products], b[machines], xMax[products]

parameters:
  #profit contribution per unit
  {j in products:  c[j] := price[j]-costs[j]; }

variables:
  x[products]: integer;

objectives:
  profit: c[]T  * x[] ->max;

constraints:
  res: a[,] * x[] <= b[];
  0<=x[]<=xMax[];
```

The CMPL command

```
cmpl production-mix-data.cmpl
```

leads to the following Solution:

```
-------------------------------------------------------------------------------------------------------------
Problem              production-mix.cmpl
Nr. of variables     3
Nr. of constraints   2
Objective name       profit
Solver name          GLPK
-------------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      6395 (max!)

Variables
Name                 Type           Activity        Lower bound        Upper bound           Marginal
-------------------------------------------------------------------------------------------------------------
x[1]                    I                 33                  0                250                  -
x[2]                    I                 49                  0                240                  -
x[3]                    I                  0                  0                250                  -
-------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type           Activity        Lower bound        Upper bound           Marginal
-------------------------------------------------------------------------------------------------------------
res_1                   L                999          -Infinity               1000                  -
res_2                   L                985          -Infinity               1000                  -
-------------------------------------------------------------------------------------------------------------
```

2.8.1.3 Production mix including thresholds and step-wise fixed costs

This model calculates the production mix that maximizes profit subject to available resources. When a product is produced, there are fixed set-up costs. There is also a threshold for each product. The quantity of a product is zero or greater than the threshold.

The example involves three products which are to be produced with two machines. The following data is given:

		P1	P2	P3	upper bounds [h]
production minimum of a product	[units]	45	45	45	
upper bound of a product	[units]	250	240	250	
selling price per unit	[€/unit]	500	600	450	
direct costs per unit	[€/unit]	425	520	400	
profit contribution per unit	[€/unit]	75	80	50	
set-up costs	[€]	500	400	500	
machine hours required per unit					
machine 1	[h/unit]	8	15	12	1,000
machine 2	[h/unit]	15	10	8	1,000

The mathematical model can be formulated as follows:

$$75 \cdot x_1 + 80 \cdot x_2 + 50 \cdot x_3 - 500 \cdot y_1 - 400 \cdot y_2 - 500 \cdot y_3 \rightarrow max!$$

$$s.t.$$

$$8 \cdot x_1 + 15 \cdot x_2 + 12 \cdot x_3 \leq 1{,}000$$

$$15 \cdot x_1 + 10 \cdot x_2 + 8 \cdot x_3 \leq 1{,}000$$

$$45 \cdot y_1 \leq x_1 \leq 250 \cdot y_1$$

$$45 \cdot y_2 \leq x_2 \leq 240 \cdot y_2$$

$$45 \cdot y_3 \leq x_3 \leq 250 \cdot y_3$$

$$x_1 \in \{0,1,\ldots,250\}$$

$$x_2 \in \{0,1,\ldots,240\}$$

$$x_3 \in \{0,1,\ldots,250\}$$

$$y_j \in \{0,1\} \quad ; j = 1(1)3$$

The CMPL model `production-mix-fixed-costs.cmpl` is formulated as follows:

```
%data production-mix-data.cdat

parameters:
  #profit contribution per unit
  {j in products:  c[j] := price[j]-costs[j]; }
```

```
variables:
  {j in products : x[j]: integer[0..xMax[j]]; }
  y[products] : binary;

objectives:
  profit: c[]T  * x[] - FC[]T * y[] ->max;

constraints:
  res: a[,] * x[] <= b[];
  bounds {j in products: xMin[j] * y[j] <= x[j] <= xMax[j] * y[j]; }
```

CMPL command:

```
cmpl production-mix-fixed-costs.cmpl
```

Solution:

```
-------------------------------------------------------------------------------------------------------------
Problem              production-mix-fixed-costs.cmpl
Nr. of variables     6
Nr. of constraints   8
Objective name       profit
Solver name          CBC
-------------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      4880 (max!)

Variables
Name                 Type            Activity        Lower bound        Upper bound            Marginal
-------------------------------------------------------------------------------------------------------------
x[1]                    I                   0                  0                250                   -
x[2]                    I                  66                  0                240                   -
x[3]                    I                   0                  0                250                   -
y[1]                    B                   0                  0                  1                   -
y[2]                    B                   1                  0                  1                   -
y[3]                    B                   0                  0                  1                   -
-------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity        Lower bound        Upper bound            Marginal
-------------------------------------------------------------------------------------------------------------
res_1                   L                 990          -Infinity               1000                   -
res_2                   L                 660          -Infinity               1000                   -
bounds_1_1              L                   0          -Infinity                  0                   -
bounds_1_2              L                   0          -Infinity                  0                   -
bounds_2_1              L                 -21          -Infinity                  0                   -
bounds_2_2              L                -174          -Infinity                  0                   -
bounds_3_1              L                   0          -Infinity                  0                   -
bounds_3_2              L                   0          -Infinity                  0                   -
-------------------------------------------------------------------------------------------------------------
```

2.8.1.4 The knapsack problem

Given a set of items with specified weights and values, the problem is to find a combination of items that fills a knapsack (container, room, …) to maximize the value of the knapsack subject to its restricted capacity or to minimize the weight of items in the knapsack subject to a predefined minimum value.

In this example there are 10 boxes, which can be sold on the market at a defined price.

box number	weight [pounds]	price [€/box]
1	100	10
2	80	5
3	50	8
4	150	11
5	55	12
6	20	4
7	40	6
8	50	9
9	200	10
10	100	11

1. What is the optimal combination of boxes if you are seeking to maximize the total sales and are able to carry a maximum of 60 pounds?
2. What is the optimal combination of boxes if you are seeking to minimize the weight of the transported boxes bearing in mind that the minimum total sales must be at least €600 ?

Model 1: maximize the total sales

The mathematical model can be formulated as follows:

$$100 \cdot x_1 + 80 \cdot x_2 + 50 \cdot x_3 + 150 \cdot x_4 + 55 \cdot x_5 + 20 \cdot x_6 + 40 \cdot x_7 + 50 \cdot x_8 + 200 \cdot x_9 + 100 \cdot x_{10} \rightarrow max\,!$$

$$s.t.$$

$$10 \cdot x_1 + 5 \cdot x_2 + 8 \cdot x_3 + 11 \cdot x_4 + 12 \cdot x_5 + 4 \cdot x_6 + 6 \cdot x_7 + 9 \cdot x_8 + 10 \cdot x_9 + 11 \cdot x_{10} \leq 60$$

$$x_j \in \{0,1\} \quad ; j = 1(1)10$$

The basic data is saved in the CMPL file `knapsack-data.cdat`:

```
%boxes set < 1(1)10 >

#weight of the boxes
%w[boxes] < 10 5 8 11 12 4 6 9 10 11 >

#price per box
%p[boxes] <100 80 50 150 55 20 40 50 200 100 >

#max capacity
%maxWeight <60>

#min sales
%minSales <600>
```

A simple CMPL model `knapsack-max-basic.cmpl` can be formulated as follows:

```
%data knapsack-data.cdat : boxes set, w[boxes], p[boxes], maxWeight, minSales
%display nonZeros

variables:
  x[boxes] : binary;
objectives:
  sales: p[]T * x[] ->max;
constraints:
  weight: w[]T * x[] <= maxWeight;
```

CMPL command:

```
cmpl  knapsack-max-basic.cmpl
```

Solution:

```
-----------------------------------------------------------------------------------------------------------
Problem             knapsack-max-basic.cmpl
Nr. of variables    10
Nr. of constraints  1
Objective name      sales
Solver name         CBC
-----------------------------------------------------------------------------------------------------------

Objective status    optimal
Objective value     700 (max!)

Nonzero variables
Name                Type            Activity        Lower bound        Upper bound            Marginal
-----------------------------------------------------------------------------------------------------------
x[1]                   B                   1                  0                  1                   -
x[2]                   B                   1                  0                  1                   -
x[4]                   B                   1                  0                  1                   -
x[6]                   B                   1                  0                  1                   -
x[8]                   B                   1                  0                  1                   -
x[9]                   B                   1                  0                  1                   -
x[10]                  B                   1                  0                  1                   -
-----------------------------------------------------------------------------------------------------------

Nonzero constraints
Name                Type            Activity        Lower bound        Upper bound            Marginal
-----------------------------------------------------------------------------------------------------------
weight                 L                  60          -Infinity                 60                   -
-----------------------------------------------------------------------------------------------------------
```

Model 2: minimize the weight

The mathematical model can be formulated as follows:

$$10 \cdot x_1 + 5 \cdot x_2 + 8 \cdot x_3 + 11 \cdot x_4 + 12 \cdot x_5 + 4 \cdot x_6 + 6 \cdot x_7 + 9 \cdot x_8 + 10 \cdot x_9 + 11 \cdot x_{10} \rightarrow min!$$

$$s.t.$$

$$100 \cdot x_1 + 80 \cdot x_2 + 50 \cdot x_3 + 150 \cdot x_4 + 55 \cdot x_5 + 20 \cdot x_6 + 40 \cdot x_7 + 50 \cdot x_8 + 200 \cdot x_9 + 100 \cdot x_{10} \geq 600$$

$$x_j \in \{0,1\} \quad ; j = 1(1)10$$

A simple CMPL model `knapsack-min-basic.cmpl` can be formulated as follows:

```
%data knapsack-data.cdat
%display nonZeros

variables:
  x[boxes] : binary;
objectives:
  weight: w[]T * x[] ->min;
constraints:
  sales: p[]T * x[] >= minSales;
```

CMPL command:

```
cmpl  knapsack-min-basic.cmpl
```

Solution:

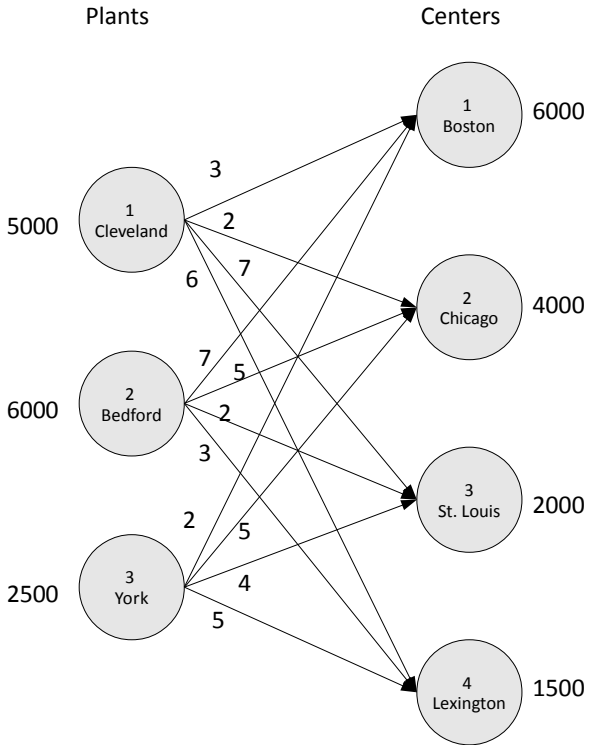
```
-----------------------------------------------------------------------------------------------------------
Problem              knapsack-min-basic.cmpl
Nr. of variables     10
Nr. of constraints   1
Objective name       weight
Solver name          CBC
-----------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      47 (min!)

Nonzero variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-----------------------------------------------------------------------------------------------------------
x[1]                    B                   1                   0                   1                   -
x[2]                    B                   1                   0                   1                   -
x[4]                    B                   1                   0                   1                   -
x[9]                    B                   1                   0                   1                   -
x[10]                   B                   1                   0                   1                   -
-----------------------------------------------------------------------------------------------------------

Nonzero constraints
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-----------------------------------------------------------------------------------------------------------
sales                   G                 630                 600            Infinity                   -
-----------------------------------------------------------------------------------------------------------
```

2.8.1.5 Transportation problem using 1-tuple sets

A transportation problem is a special kind of linear programming problem which seeks to minimize the total shipping costs of transporting goods from several supply locations (origins or sources) to several demand locations (destinations).

The following example is taken from (Anderson et.al. 2011, p. 261ff). This problem involves the transportation of a product from three plants to four distribution centres. Foster Generators operates plants in Cleveland, Ohio; Bedford, Indiana; and York, Pennsylvania. The supplies are defined by the production capacities over the next three-month planning period for one particular type of generator.

The company distributes its generators through four regional distribution centres located in Boston, Chicago, St. Louis, and Lexington. It is to decide how much of its products should be shipped from each plant to each distribution centre. The objective is to minimize the transportation costs.



The problem can be formulated in the form of the general linear programme below

$$\sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij} \cdot x_{ij} \quad \rightarrow \min!$$

$$s.t.$$

$$\sum_{j=1}^{n} x_{ij} = s_i \qquad ; i = 1(1)m$$

$$\sum_{i=1}^{m} x_{ij} = d_j \qquad ; j = 1(1)n$$

$$x_{ij} \geq 0 \qquad ; i = 1(1)m \, , \, j = 1(1)n$$

x_{ij} – number of units shipped from plant i to center j
c_{ij} – cost per unit of shipping from plant i to center j
s_i – supply in units at plant i
d_j – demand in units at desitination j

The CMPL model `transportation.cmpl` can be formulated as follows:

```
%display nonZeros

parameters:
  plants  := 1(1)3;
  centres := 1(1)4;
  s[plants]  := (5000,6000,2500);
```

```
  d[centres] := (6000,4000,2000,1500);
  c[plants,centres] := ( (3,2,7,6), (7,5,2,3),(2,5,4,5) );

variables:
  x[plants,centres]: real[0..];

objectives:
  costs: sum{i in plants, j in centres : c[i,j] * x[i,j] } ->min;

constraints:
  supplies {i in plants  : sum{j in centres: x[i,j]} = s[i];}
  demands  {j in centres : sum{i in plants : x[i,j]} =  d[j];}
```

or by using an additional cmplData file `transportation-data.cdat`

```
%plants set < 1..3 >
%centres set < 1..4 >

%s[plants] <  5000 6000 2500 >
%d[centres] < 6000 4000 2000 1500 >

%c[plants, centres]  <  3 2 7 6
                        7 5 2 3
                        2 5 4 5 >
```

and the corresponding CMPL model:

```
%data transportation-data.cdat
%display nonZeros

variables:
  x[plants,centres]: real[0..];

objectives:
  costs: sum{i in plants, j in centres : c[i,j] * x[i,j] } ->min;

constraints:
  supplies {i in plants  : sum{j in centres: x[i,j]} = s[i];}
  demands  {j in centres : sum{i in plants : x[i,j]} =  d[j];}
```

CMPL command:

```
cmpl  transportation.cmpl
```

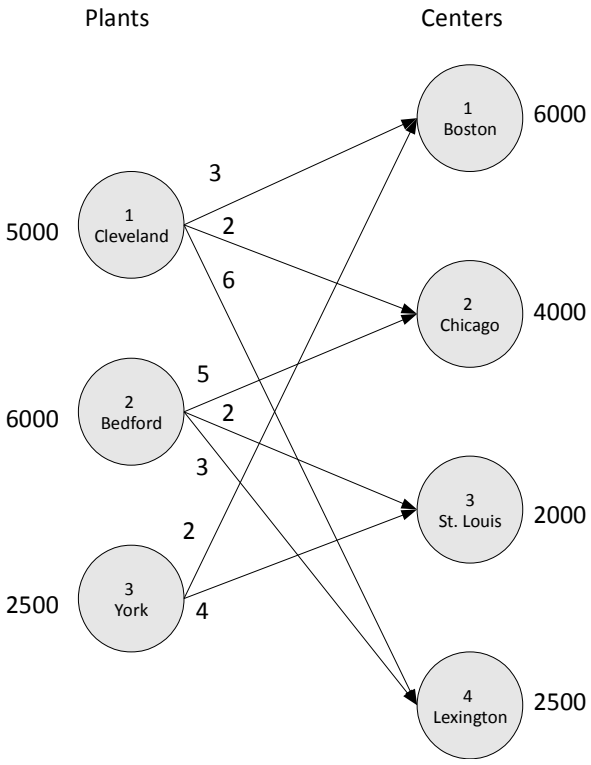
Solution:

```
-----------------------------------------------------------------------------------------------------
Problem             transportation.cmpl
Nr. of variables    12
Nr. of constraints  7
Objective name      costs
Solver name         CBC
```

```
------------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      39500 (min!)

Nonzero variables
Name                 Type            Activity        Lower bound        Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
x[1,1]                  C                3500                  0           Infinity                   0
x[1,2]                  C                1500                  0           Infinity                   0
x[2,2]                  C                2500                  0           Infinity                   0
x[2,3]                  C                2000                  0           Infinity                   0
x[2,4]                  C                1500                  0           Infinity                   0
x[3,1]                  C                2500                  0           Infinity                   0
------------------------------------------------------------------------------------------------------------

Nonzero constraints
Name                 Type            Activity        Lower bound        Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
supplies_1              E                5000               5000               5000                   1
supplies_2              E                6000               6000               6000                   4
supplies_3              E                2500               2500               2500                   -
demands_1               E                6000               6000               6000                   2
demands_2               E                4000               4000               4000                   1
demands_3               E                2000               2000               2000                  -2
demands_4               E                1500               1500               1500                  -1
------------------------------------------------------------------------------------------------------------
```

2.8.1.6 Transportation problem using multidimensional sets (2-tuple sets)

In the case that not all of the connections are possible for technological or commercial reasons (e.g. as in the picture below) then an alternative model to the model above has to be formulated. Additionally is assumed that the total demand is greater than the supplies.



The mathematical model is based on the 2-tuple set routes that contains only the valid connections between the plants and the centres.

$$\sum_{(i,j)\in routes} c_{ij}\cdot x_{ij} \qquad \rightarrow \min!$$

$$s.t.$$

$$\sum_{\substack{(k,j)\in routes \\ k=i}} x_{kj} = s_i \qquad ; i=1(1)m$$

$$\sum_{\substack{(i,l)\in routes \\ l=j}} x_{il} \le d_j \qquad ; j=1(1)n$$

$$x_{ij} \ge 0 \qquad ;(i,j)\in routes$$

Die sets and parameters are specified in `transportation-tuple-data.cdat`

```
%routes  set[2]  <   1 1
                     1 2
                     1 4
                     2 2 2 3 2 4
                     3 1 3 3  >
%plants  set < 1(1)3 >
%centres  set < 1..4 >

%s[plants] <  5000 6000 2500 >
%d[centres] < 6000 4000 2000 2500 >
%c[routes]  < 3  2  6  5  2  3  2  4 >
```

that is connected to the CMPL model `transportation-tuple-data.cmpl`:

```
%data : plants set, centres set[1], routes set[2]
%data : c[routes] , s[plants] , d[centres]
%display nonZeros

variables:
   x[routes]: real[0..];
objectives:
   costs: sum{ [i,j] in routes : c[i,j]*x[i,j] } ->min;
constraints:
   supplies {i in plants : sum{j in routes *> [i,*] : x[i,j]} = s[i];}
   demands  {j in centres: sum{i in routes *> [*,j] : x[i,j]} <= d[j];}
```

Solution:

```
-----------------------------------------------------------------------------------------------------
Problem             transportation-tuple-data.cmpl
Nr. of variables    8
Nr. of constraints  7
Objective name      costs
Solver name         CBC
-----------------------------------------------------------------------------------------------------
```

```
Objective status     optimal
Objective value      36500 (min!)

Nonzero variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
x[1,1]                  C                2500                   0            Infinity                   0
x[1,2]                  C                2500                   0            Infinity                   0
x[2,2]                  C                1500                   0            Infinity                   0
x[2,3]                  C                2000                   0            Infinity                   0
x[2,4]                  C                2500                   0            Infinity                   0
x[3,1]                  C                2500                   0            Infinity                   0
------------------------------------------------------------------------------------------------------------

Nonzero constraints
Name                 Type            Activity         Lower bound         Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
supplies_1              E                5000                5000                5000                   3
supplies_2              E                6000                6000                6000                   6
supplies_3              E                2500                2500                2500                   2
demands_1               L                5000           -Infinity                6000                   -
demands_2               L                4000           -Infinity                4000                  -1
demands_3               L                2000           -Infinity                2000                  -4
demands_4               L                2500           -Infinity                2500                  -3
------------------------------------------------------------------------------------------------------------
```

2.8.1.7 Quadratic assignment problem

Assignment problems are special types of linear programming problems which assign assignees to tasks or locations. The goal of this quadratic assignment problem is to find the cheapest assignments of *n* machines to *n* locations. The transport costs are influenced by

- the distance d_{jk} between location *j* and location *k* and
- the quantity t_{hi} between machine h and machine i, which is to be transported.

The assignment of a machine *h* to a location *j* can be formulated with the Boolean variables

$$x_{hj} = \begin{cases} 1 \text{ , if machine } h \text{ is assigned to location } j \\ 0 \text{ , if not} \end{cases}$$

The general model can be formulated as follows:

$$\sum_{h=1}^{n} \sum_{\substack{i=1 \\ i \neq h}}^{n} \sum_{j=1}^{n} \sum_{\substack{k=1 \\ k \neq j}}^{n} t_{hi} \cdot d_{jk} \cdot x_{hj} \cdot x_{ik} \rightarrow min!$$

$$s.t.$$

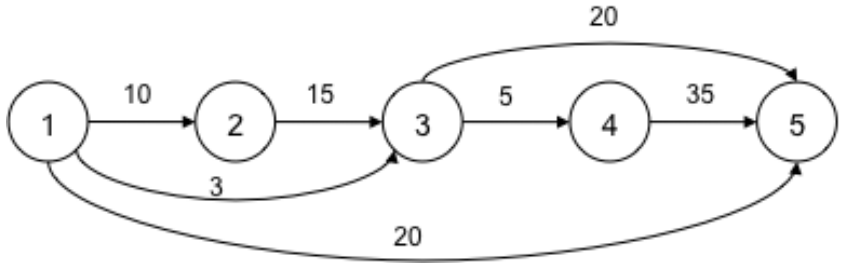
$$\sum_{j=1}^{n} x_{hj} = 1 \quad ; h = 1(1)n$$

$$\sum_{h=1}^{n} x_{hj} = 1 \quad ; j = 1(1)n$$

$$x_{hj} \in \{0,1\} \quad ; h = 1(1)n \, , \, j = 1(1)n$$

Because of the product $x_{hj} \cdot x_{ik}$ in the objective function the model is not a linear model. But it is possible to use a set of inequations to make an equivalent transformation of such multiplications of variables. This transformation is implemented in CMPL and the set of inequations will be generated automatically.

Consider the following case: There are 5 machines and 5 locations in the given factory. The quantities of goods which are to be transported between the machines are indicated in the figure below.



As shown in the picture below the machines are not fully connected. Therefor it makes sense to formulate the objective function with a sum over a 2-tuple set with the name *routes* for the valid combinations between the machines.

$$\sum_{(h,i) \in routes} \sum_{\substack{j=1 \\ k \neq j}}^{n} \sum_{k=1}^{n} t_{hi} \cdot d_{jk} \cdot x_{hj} \cdot x_{ik} \rightarrow min!$$

The distances between the locations are given in the following table:

from/to	1	2	3	4	5
1	M	1	2	3	4
2	2	M	1	2	3
3	3	1	M	1	2
4	2	3	1	M	1
5	5	3	2	1	M

The CMPL model `quadratic-assignment.cmpl` can be formulated as follows:

```
%display nonZeros
%display var x[*
%display ignoreCons

parameters:
  n:=5;
  d[,]:= (  ( 0, 1, 2, 3, 4),
          ( 2, 0, 1, 2, 3),
          ( 3, 1, 0, 1, 2),
          ( 2, 3, 1, 0, 1),
          ( 5, 3, 1, 1, 0) );

  routes := set ( [1,2] , [1,3], [1,5], [2,3] , [3,4] , [3,5] , [4,5]);
  t[routes] := (10,3,20,15,5,20,35);
```

```
variables:
  x[1..n,1..n]: binary;
  #dummy variables to store the products x_hj * x_ik
  { [h,i] in routes, j:=1(1)n, k:=1(1)n , k<>j: w[h,j,i,k]: real[0..1]; }

objectives:
  costs: sum{    [h,i] in routes,  j:=1(1)n, k:=1(1)n , k<>j:
                 t[h,i]*d[j,k]*w[h,j,i,k] } ->min;
constraints:
  { [h,i] in routes, j:=1(1)n, k:=1(1)n, k<>j: w[h,j,i,k]=x[h,j]*x[i,k];}

  sos1 { h:=1(1)n: sum{ j:=1(1)n: x[h,j] } = 1; }
  sos2 { j:=1(1)n: sum{ h:=1(1)n: x[h,j] } = 1; }
```

CMPL command:

```
cmpl  quadratic-assignment.cmpl
```

Solution:

```
------------------------------------------------------------------------------------------------------------
Problem              quadratic-assignment.cmpl
Nr. of variables     305
Nr. of constraints   570
Objective name       costs
Solver name          CBC
Display variables    nonzero variables (x[*)
Display constraints  ignore all constraints
------------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      134 (min!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
x[1,4]                  B                   1                   0                   1                   -
x[2,1]                  B                   1                   0                   1                   -
x[3,2]                  B                   1                   0                   1                   -
x[4,5]                  B                   1                   0                   1                   -
x[5,3]                  B                   1                   0                   1                   -
------------------------------------------------------------------------------------------------------------
```

The optimal assignments of machines to locations are given in the the table below:

machines \ locations	1	2	3	4	5
1				x	
2	x				
3		x			
4					x
5			x		

2.8.1.8 Quadratic assignment problem using the solutionPool option

It is for several reasons interesting to catch the feasible integer solutions found during a MIP optimisation. Gurobi and Cplex are able to generate and store multiple solutions to a mixed integer programming (MIP) problem. With the display option `solutionPool` these feasible integer solutions can be shown in the solution report. It is recommended to control the behaviour of the solution pool by setting the particular Gurobi or Cplex solver options.

If the CMPL model for quadratic assignment problem above is extended by the following CMPL header entries, then all feasible integer solutions found by Cplex.

```
%arg -solver cplex
%display solutionPool
```

Solution:

```
---------------------------------------------------------------------------------------------------------
Problem              quadratic-assignment.cmpl
Nr. of variables     305
Nr. of constraints   570
Objective name       costs
Nr. of solutions     5
Solver name          CPLEX
Display variables    nonzero variables (x[*)
Display constraints  ignore all constraints
---------------------------------------------------------------------------------------------------------

Solution nr.         1
Objective status     integer optimal solution
Objective value      134 (min!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
---------------------------------------------------------------------------------------------------------
x[1,4]                  B                   1                   0                   1                   -
x[2,1]                  B                   1                   0                   1                   -
x[3,2]                  B                   1                   0                   1                   -
x[4,5]                  B                   1                   0                   1                   -
x[5,3]                  B                   1                   0                   1                   -
---------------------------------------------------------------------------------------------------------

Solution nr.         2
Objective status     integer feasible solution
Objective value      134 (min!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
---------------------------------------------------------------------------------------------------------
x[1,4]                  B                   1                   0                   1                   -
x[2,1]                  B                   1                   0                   1                   -
x[3,2]                  B                   1                   0                   1                   -
x[4,5]                  B                   1                   0                   1                   -
x[5,3]                  B                   1                   0                   1                   -
---------------------------------------------------------------------------------------------------------

Solution nr.         3
Objective status     integer feasible solution
Objective value      188 (min!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
---------------------------------------------------------------------------------------------------------
x[1,1]                  B                   1                   0                   1                   -
```

```
x[2,3]                  B                   1                   0                   1                   -
x[3,2]                  B                   1                   0                   1                   -
x[4,5]                  B                   1                   0                   1                   -
x[5,4]                  B                   1                   0                   1                   -
------------------------------------------------------------------------------------------------------------

Solution nr.         4
Objective status     integer feasible solution
Objective value      177 (min!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
x[1,1]                  B                   1                   0                   1                   -
x[2,4]                  B                   1                   0                   1                   -
x[3,5]                  B                   1                   0                   1                   -
x[4,3]                  B                   1                   0                   1                   -
x[5,2]                  B                   1                   0                   1                   -
------------------------------------------------------------------------------------------------------------

Solution nr.         5
Objective status     integer feasible solution
Objective value      191 (min!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
x[1,1]                  B                   1                   0                   1                   -
x[2,2]                  B                   1                   0                   1                   -
x[3,3]                  B                   1                   0                   1                   -
x[4,4]                  B                   1                   0                   1                   -
x[5,5]                  B                   1                   0                   1                   -
------------------------------------------------------------------------------------------------------------
```

2.8.1.9 Generic travelling salesman problem

The travelling salesman problem is well known and often described. In the following CMPL model the (x,y) coordinates of the cities are defined by random numbers and the distances are calculated by the euclidian distance of the (x.y) coordinates. The CMPL model `tsp.cmpl` can be formulated as follows:

```
%arg -solver cbc
%arg -ignoreZeros
%display var x*

parameters:
  seed:=srand(100);
  M:=10000;

  nrOfCities:=10;
  cities:=1..nrOfCities;

  {i in cities:
      xp[i]:=rand(100);
      yp[i]:=rand(100);
  }
```

```
  {i in cities, j in cities:
      {i==j:
         dist[i,j]:=M; |
      default:
         dist[i,j]:= sqrt( (xp[i]-xp[j])^2 + (yp[i]-yp[j])^2 );
         dist[j,i]:= dist[i,j]+rand(10)-rand(10);
      }
  }

variables:
  x[cities,cities]: binary;
  u[cities]: real[0..];

objectives:
  distance: sum{i in cities, j in cities: dist[i,j]* x[i,j]} ->min;

constraints:
  sos_i {j in cities:  sum{i in cities: x[i,j]}=1; }
  sos_j {i in cities:  sum{j in cities: x[i,j]}=1; }
  noSubs {i:=2..nrOfCities, j:=2..nrOfCities, i<>j: u[i] - u[j] +
                                nrOfCities * x[i,j] <= nrOfCities-1; }
```

CMPL command:

```
cmpl  tsp.cmpl
```

Solution:

```
-----------------------------------------------------------------------------------------------------------
Problem              tsp.cmpl
Nr. of variables     109
Nr. of constraints   92
Objective name       distance
Solver name          CBC
-----------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      321.319 (min!)

Nonzero variables (x*)
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-----------------------------------------------------------------------------------------------------------
x[1,4]                  B                   1                   0                   1                   -
x[2,1]                  B                   1                   0                   1                   -
x[3,6]                  B                   1                   0                   1                   -
x[4,10]                 B                   1                   0                   1                   -
x[5,8]                  B                   1                   0                   1                   -
x[6,9]                  B                   1                   0                   1                   -
x[7,2]                  B                   1                   0                   1                   -
x[8,7]                  B                   1                   0                   1                   -
x[9,5]                  B                   1                   0                   1                   -
x[10,3]                 B                   1                   0                   1                   -
-----------------------------------------------------------------------------------------------------------
```

The tour is optimal as follows:
1→4→10→3→6→9→5→8→7→2→1

2.8.2 Other selected examples

CMPL can be used as a pre-solver or simple solver. In this way it is possible to find a preliminary solution of a problem as a basis for the model which is to be generated.

2.8.2.1 Solving the knapsack problem

The knapsack problem is a very simple problem that does not necessarily have to be solved by an MIP solver. CMPL can be used as a simple solver for knapsack problems to approximate the optimal solution.

The idea of the following models is to evaluate each item using the relation between the value per item and weight per item. The knapsack will be filled with the items sorted in descending order until the capacity limit or the minimum value is reached. Using the data from the examples in section 2.8.1.4 a CMPL model to maximize the total sales relative to capacity can be formulated as follows.

Model 1: maximize the total sales `knapsack-max-presolved.cmpl`

```
include "knapsack-data.cmpl"

#calculating the relative value of each box
{j in boxes: val[j]:= p[j]/w[j]; }
sumSales:=0;
sumWeight:=0;
#initial solution
x[]:=(0,0,0,0,0,0,0,0,0,0);

{ i in boxes:
  maxVal:=max(val[]);
  {j in boxes:
    { maxVal=val[j] :
        { sumWeight+w[j] <= maxWeight:
            x[j]:=1;
            sumSales:=sumSales + p[j];
            sumWeight:=sumWeight + w[j];
        }
        val[j]:=0;
        break j;
    }
  }
}
echo "Solution found";
echo "Optimal total sales: "+ sumSales;
echo "Total weight : " + sumWeight;
{j in boxes: echo "x_"+ j + ": " + x[j]; }
```

CMPL command:

```
cmpl knapsack-max-presolved.cmpl -noOutput -cd
```

Solution:

```
Solution found
Optimal total sales: 690
Total weight : 57
x_1: 1
x_2: 1
x_3: 0
x_4: 1
x_5: 0
x_6: 1
x_7: 1
x_8: 0
x_9: 1
x_10: 1
```

This solution is not identical to the optimal solution in section 2.8.1.4 but good enough as an approximate solution.

Model 2: minimize the total weight `knapsack-min-presolved.cmpl`

```
include "knapsack-data.cmpl"
#calculating the relative value of each box
{j in boxes: val[j]:= w[j]/p[j]; }

M:=10000;
sumSales:=0;
sumWeight:=0;
#initial solution
x[]:=(0,0,0,0,0,0,0,0,0,0);
{sumSales < minSales:
  maxVal:=min(val[]);
  {j in boxes:
    { maxVal=val[j] :
        { sumSales < minSales:
            x[j]:=1;
            sumSales:=sumSales + p[j];
            sumWeight:=sumWeight + w[j];
        }
        val[j]:=M;
        break j;
    }
  }
  repeat;
}
echo "Solution found";
echo "Optimal total weight : " + sumWeight;
echo "Total sales: "+ sumSales;
{j in boxes: echo "x_"+ j + ": " + x[j]; }
```

CMPL command:

```
cmpl knapsack-min-presolved.cmpl -noOutput -cd
```

Solution:

```
Optimal total weight : 47
Total sales: 630
x_1: 1
x_2: 1
x_3: 0
x_4: 1
x_5: 0
x_6: 0
x_7: 0
x_8: 0
x_9: 1
x_10: 1
```

This solution is identical to the optimal solution in section 2.8.1.4.

2.8.2.2 Finding the maximum of a concave function using the bisection method

One of the alternative methods for finding the maximum of a negative convex function is the bisection method. (Hillier and Liebermann 2010, p. 554f.) A CMPL programme to find the maximum of $f(x)=12\cdot x-3\cdot x^4-2\cdot x^6$ can be formulated as follows (`bisection.cmpl`):

```
parameters:
  #distance epsilon
  e:=0.02;
  #initial solution
  xl:= 0;
  xo:= 2;
  xn:= (xl+xo)/2;
  { (xo-xl) > e :

    fd:= 12 - 12 * xn^3 - 12 * xn^5;
    { fd >= 0 : xl:=xn; |
      fd <= 0 : xo:=xn ;}

    xn:= (xl+xo)/2;

    fx := 12 * xn -3 * xn^4 - 2 * xn^6;

    echo   "f'(xn): " + format("%10.4f",fd) + " xl: " +
           format("%6.4f",xl) +  " xo: " + format("%6.4f",xo) + " xn: " +
           format("%6.4f",xn) +  " f(xn): " + format("%6.4f",fx);
```

```
    repeat;
  }
  echo "Optimal solution found";
  x:=xn;
  echo "x: "+ format("%2.3f",x);
  echo "function value: " + (12 * x -3 * x^4 - 2 * x^6);
```

CMPL command:

```
cmpl bisection.cmpl -noOutput -cd
```

Solution:

```
f'(xn):   -12.0000 xl: 0.0000 xo: 1.0000 xn: 0.5000 f(xn): 5.7812
f'(xn):    10.1250 xl: 0.5000 xo: 1.0000 xn: 0.7500 f(xn): 7.6948
f'(xn):     4.0898 xl: 0.7500 xo: 1.0000 xn: 0.8750 f(xn): 7.8439
f'(xn):    -2.1940 xl: 0.7500 xo: 0.8750 xn: 0.8125 f(xn): 7.8672
f'(xn):     1.3144 xl: 0.8125 xo: 0.8750 xn: 0.8438 f(xn): 7.8829
f'(xn):    -0.3397 xl: 0.8125 xo: 0.8438 xn: 0.8281 f(xn): 7.8815
f'(xn):     0.5113 xl: 0.8281 xo: 0.8438 xn: 0.8359 f(xn): 7.8839
Optimal solution found
x: 0.836
function value: 7.883868
```

3 CMPL software package

3.1 CMPL software package in a glance

CMPL (<Coliop|Coin> Mathematical Programming Language) is a mathematical programming language and a system for mathematical programming and optimisation of linear optimisation problems.

CMPL executes CBC, GLPK, Gurobi, SCIP or CPLEX directly to solve the generated model instance. Because it is also possible to transform the mathematical problem into MPS, Free-MPS or OSiL files, alternative solvers can be used.

The CMPL distribution contains **Coliop** which is an (simple) IDE (Integrated Development Environment) for CMPL and also pyCMPL, jCMPL and CMPLServer.

pyCMPL is the CMPL application programming interface (API) for Python and an interactive shell and **jCMPL** is CMPL's Java API. The main idea of this APIs is to define sets and parameters within the user application, to start and control the solving process and to read the solution(s) into the application if the problem is feasible. All variables, objective functions and constraints are defined in CMPL. These functionalities can be used with a local CMPL installation or a CMPLServer.

CMPLServer is an XML-RPC-based web service for distributed and grid optimisation that can be used with CMPL, pyCMPL and jCMPL. It is reasonable to solve large models remotely on the CMPLServer that is installed on a high performance system. CMPL provides four XML-based file formats for the communication between a CMPLServer and its clients. (CmplInstance, CmplSolutions, CmplMessages, CmplInfo).

3.2 Installation

1. Download the relevant ZIP file from http://www.coliop.org and unzip the CMPL package
2. Installation
   - Windows 32
     To install the CMPL package you only have to copy the folder `Cmpl` into
     `c:\program files\Cmpl`
   - Windows 64
     To install the CMPL package copy the folder `Cmpl` into
     `c:\program files (x86)\Cmpl`
   - Linux
     To install the CMPL package you should type the command: `./install`
     This command installs the CMPL distribution package in the following folders:
     programme folder -> `/usr/share/Cmpl`
     scripts (cmpl, coliop, cmplServer, pyCmpl) -> `/usr/bin/cmpl`
   - Mac - OS X
     To install the CMPL package please use the following command: `./install`
     This command installs the CMPL distribution package in the following folders:
     programme folder -> `/Applications/Cmpl`
     scripts (cmpl, coliop, cmplServer, pyCmpl) -> `/usr/bin/cmpl`
3. Installation Prerequisites
   - Java
     Since Coliop is written in Java you have to download and install the java runtime environment http://java.com/download/index.jsp.
   - Python 2.7
     Under Linux and OS X a user has only to ensure that Python 2.7 is installed. (Usually by default).
     Under Windows pyCMPL should work out of the box because the CMPL binary package contains pypy as Python environment.

3.3 CMPL

3.3.1 Running CMPL

- Windows
  CMPL - Please start the cmplShell `c:\program files[ (x86)]\Cmpl\cmplShell.bat` and execute CMPL with the command `cmpl <problemname>.cmpl` .
  Coliop - Double-click `coliop.bat` in `c:\program files[ (x86)]\Cmpl`

- Linux
  CMPL - Please start the terminal and execute CMPL with the command
  `cmpl <problemname>.cmpl` .
  Coliop - Please start the terminal und execute Coliop with the command `coliop`
- Mac - OS X
  CMPL - Please start the terminal and execute CMPL with the command
  `cmpl <problemname>.cmpl`.
  Coliop - Double-click the Application `coliop` in `/Applications/Cmpl`

3.3.2 Usage of the CMPL command line tool

The CMPL command line tool can be used in two modes. Using the solver mode, an LP or MIP can be formulated, solved and analysed. In this mode, OSSolverService, GLPK or Gurobi is invoked. In the model mode it is possible to transform the mathematical problem into MPS, Free-MPS or OSiL files that can be used by certain alternative LP or MIP solvers.

```
cmpl <cmplFile> [<options>]
```

Usage: cmpl <cmplFile.cmpl> [options]

Model mode:

-i <cmplFile>	: input file
-m [<File>]	: exports model in MPS format in a file or stdout
-fm [<File>]	: exports model in Free-MPS format in a file or stdout
-x [<File>]	: exports model in OSiL XML format in a file or stdout
-syntax	: checks the syntax of the CMPL model w/o generating of a MPS or OSiL file
-noOutput	: no generating of an MPS or OSiL file

Solver mode:

-solver <solver>	: name of the solver you want to use possible options: glpk, cbc, scip, gurobi, cplex (default cbc)
-cmplUrl <url>	: URL of a CmplServer
-maxServerTries <x>	: maximum number of tries of failed CmplServer calls
-maxQueuingTime <x>	: maximum time in <x> seconds that a problem waits in a CmplServer queue
-solution [<File>]	: optimisation results in CmplSolution XML format
-solutionCsv [<File>]	: optimisation results in CSV format
-solutionAscii [<File>]	: optimisation results in ASCII format

-obj <objName>	: name of the objective function
-objSense <max/min>	: objective sense
-maxDecimals <x>	: maximal number of decimals in the solution report (max 12)
-zeroPrecision <x>	: precision of zero values in the solution report (default 1e-9)
-ignoreZeros	: display only variables and constraints with non-zero values in the solution report
-dontRemoveTmpFiles	: don't remove temporary files (mps,osil,osrl,gsol)
-alias <alias>	: uses an alias name for the cmpl model

General options:

-data <cmplDataFile>	: reads a cmplData file
-e [<File>]	: output for error messages and warnings -e simple output to stderr (default) -e <File> output in CmplMessage XML format to file
-matrix [<File>]	: writes the generated matrix in a file or on stdout
-l [<File>]	: output for replacements for products of variables
-s [<File>]	: short statistic info
-p [<File>]	: output for protocol
-silent	: suppresses CMPL and solver messages
-integerRelaxation	: all integer variables are changed to continuous variables
-gn	: matrix reductions
-gf	: generated constraints for products of variables are included at the original position of the product
-cd	: warning at multiple parameter definition
-ci <x>	: mode for integer expressions (0 - 3), (default 1) If the result of an integer operation is outside the range of a long integer then the type of result will change from integer to real. This flag defines the integer range check behaviour. -ci 0 no range check -ci 1 default, range check with a type change if necessary -ci 2 range check with error message if necessary -ci 3 each numerical operation returns a real result
-f% <format>	: format option for MPS or OSiL files (C++ style - default %f)
-h	: get this help
-v	: version

Examples - solver mode:

`cmpl test.cmpl`	solves the problem `test.cmpl` locally with the default solver and displays a standard solution report
`cmpl test.cmpl -solver glpk test.cmpl`	solves the problem `test.cmpl` locally using GLPK and displays a standard solution report
`cmpl test.cmpl ↲` `  -cmplUrl http://194.95.44.187:8080`	solves the problem `test.cmpl` remotely with the defined CMPLServer and displays a standard solution report
`cmpl test.cmpl -solutionCsv`	solves the problem `test.cmpl` locally with the default solver writes the solution in the CSV-file `test.csv` and displays a standard solution report
`cmpl "/Users/test/Documents/ ↲` `  Projects/Project 1/test.cmpl"`	If the file name or the path contains blanks then one can enclose the entire file name in double quotes.

Examples - model mode:

`cmpl test.cmpl -m test.mps`	reads the file `test.cmpl` and generates the MPS-file `test.mps`.
`cmpl test.cmpl -fm test.mps`	reads the file `test.cmpl` and generates the Free-MPS-file `test.mps`.
`cmpl test.cmpl -x test.osil`	reads the file `test.cmpl` and generates the OSiL-file `test.osil`.

3.3.3 Syntax checks

Syntax checks can be carried out with or without data.

If the parameters and sets are specified within the `parameter` section it is only necessary to use the command line argument `-syntax` or the CMPL header option `%arg -syntax`. The following CMPL model `test.cmpl`:

```
%arg -syntax
parameters:
  n := 1..2;
  m := 1..3;
  c[m] := ( 1, 2, 3 );
  b[n] := ( 15, 20 )
  A[m,n] := (( 5.6, 7.7, 10.5 ),( 9.8, 4.2, 11.1 ));
variables:
  x[m]: real[0..];
objectives:
  profit: c[]T * x[] -> max;
constraints:
  machine: A[,] * x[] <= b[];
```

causes the error message

```
CMPL model syntax check - running
error (compiler): file test.cmpl line 7: syntax error, unexpected SYMBOL_UNDEF, expecting ';'
error (compiler): file test.cmpl line 13: syntax error, unexpected SYMBOL_UNDEF
CMPL syntax check has finished with 2 error(s).
```

because the statement `b[n] := ( 15, 20 )` in line 6 has to be closed by a semicolon.

If a user wants to execute a syntax check without data then a CMPL header entry `%data` has to be defined including a complete specification of the sets and parameters that are necessary for the model. Please note the CMPL header option `%arg -syntax` has to be specified before the `%data` entry.

The following CMPL model:

```
%arg -syntax
%data datafile.cdat : n set, m set, c[m], b[n], A[m,n]
variables:
     x[m]: real[0..]
objectives:
     profit: c[]T * x[] -> max;
constraints:
     machine: A[,] * x[] <= b[];
```

causes the error message

```
CMPL model syntax check - running
error (compiler): file .cmpl line 5: syntax error, unexpected SECT_OBJ, expecting ';'
CMPL syntax check has finished with 1 error(s).
```

because the statement `x[m]: real[0..]` in line 4 has to be closed by a semicolon.

3.3.4 Using CMPL with several solvers

There are two ways to interact with several solvers. It is recommended to use one of the solvers which are directly supported and executed by CMPL. The CMPL installation routine installs CBC and GLPK, where CBC is the default solver. If you have installed Gurobi, CPLEX or SCIP then you can also use these solvers directly. Because CMPL transforms a CMPL model into an MPS, a Free-MPS or an OSiL file, the generated model instance can be solved by using most of the free or commercial solvers.

3.3.4.1 CBC

Cbc (Coin-or branch and cut) is an open-source mixed integer programming solver written in C++. It can be used as a callable library or stand-alone solver. The CMPL distribution contains the CBC binary. For more information please visit https://projects.coin-or.org/Cbc.

Since CBC is the default solver CBC doesn't need not to be specified:

```
cmpl <problem>.cmpl     #Solves the problem locally with CBC
```

It is possible to use most of the CBC solver options within the CMPL header. Please see Appendix 6.1 for a list of useful CBC parameters.

Usage of CBC parameters within the CMPL header:

```
%opt cbc solverOption [solverOptionValue]
```

3.3.4.2 GLPK

The GLPK (GNU Linear Programming Kit) package is intended for solving large-scale linear programming (LP), mixed integer programming (MIP), and other related problems. "The GLPK package includes the program glpsol, which is a stand-alone LP/MIP solver. This program can be invoked from the command line ... to read LP/MIP problem data in any format supported by GLPK, solve the problem, and write the problem solution obtained to an output text file." (GLPK 2014, p. 166.). For more information please visit the GLPK project website: http://www.gnu.org/software/glpk.

The CMPL package contains GLPK and it can be used by the following command:

```
cmpl <problem>.cmpl -solver glpk
```

or by the CMPL header flag:

```
%arg -solver glpk
```

Most of the GLPK solver options can be used by defining solver options within the CMPL header. Please see Appendix 6.2 for a list of useful GLPK parameters.

Usage of GLPK parameters within the CMPL header:

```
%opt glpk solverOption [solverOptionValue]
```

3.3.4.3 Gurobi

"The Gurobi Optimizer is a state-of-the-art solver for linear programming (LP), quadratic programming (QP) and mixed-integer programming (MIP including MILP and MIQP). It was designed from the ground up to exploit modern multi-core processors. For solving LP and QP models, the Gurobi Optimizer includes high-performance implementations of the primal simplex method, the dual simplex method, and a parallel barrier solver. For MILP and MIQP models, the Gurobi Optimizer incorporates the latest methods including cutting planes and powerful solution heuristics." (www.gurobi.com)

If Gurobi is installed on the same computer as CMPL then Gurobi can be executed directly only by using the command

```
cmpl <problem>.cmpl -solver gurobi
```

or by the CMPL header flag:

```
%arg -solver gurobi
```

All Gurobi parameters (excluding NodefileDir, LogFile and ResultFile) described in the Gurobi manual can be used in the CMPL header.

Usage of Gurobi parameters within the CMPL header:

```
%opt gurobi solverOption [solverOptionValue]
```

3.3.4.4 SCIP

SCIP is a project of the Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB). "SCIP is a framework for Constraint Integer Programming oriented towards the needs of Mathematical Programming experts who want to have total control of the solution process and access detailed information down to the guts of the solver. SCIP can also be used as a pure MIP solver or as a framework for branch-cut-and-price. SCIP is implemented as C callable library and provides C++ wrapper classes for user plugins. It can also be used as a standalone program to solve mixed integer programs."
[http://scip.zib.de/whatis.shtml](Achterberg 2009)

SCIP can be used only for mixed integer programming (MIP) problems. If SCIP is chosen as solver and the problem is an LP then CBC is executed as solver.

If SCIP is installed on the same computer as CMPL then SCIP can be connected to CMPL by changing the entry `ScipFileName` in the file `<cmplhome>/bin/cmpl.opt`.

Examples:

`ScipFileName = /Applications/Scip/scip`	The binary scip is located in the folder /Applications/Scip
`ScipFileName = /Program Files/Scip/scip.exe`	Example for a Windows system. Please keep in mind to use a slash as a path separator.

If this entry is correct then you can execute SCIP directly by using the command

```
cmpl <problem>.cmpl -solver scip
```

or by the CMPL header flag:

```
%arg -solver scip
```

All SCIP parameters described in the SCIP Doxygen Documentation can be used in the CMPL header.
Please see: http://scip.zib.de/doc/html/PARAMETERS.shtml

Usage SCIP parameters within the CMPL header:

```
%opt scip solverOption solverOptionValue
```

Please keep in mind, that in contrast to the SCIP Doxygen Documentation you do not have to use = as assignment operator between the *solverOption* and the *solverOptionValue*.

Examples:

`%opt scip branching/scorefunc p`	CMPL solver parameter description for `the parameter branching score function` which is described in the SCIP Doxygen Documentation as follows:
	`# branching score function ('s'um, 'p'roduct)`
	`# [type: char, range: {sp}, default: p]`
	`branching/scorefunc = p`

`%opt scip lp/checkfeas TRUE`	`# should LP solutions be checked, resolving LP when numerical troubles occur?`
	`# [type: bool, range: {TRUE,FALSE}, default: TRUE]`
	`lp/checkfeas = TRUE`

`%opt scip lp/fastmip 1`	`# which FASTMIP setting of LP solver should be used? 0: off, 1: low`
	`# [type: int, range: [0,1], default: 1]`
	`lp/fastmip = 1`

3.3.4.5 CPLEX

CPLEX is a part of the IBM ILOG CPLEX Optimization Studio and includes simplex, barrier, and mixed integer optimizers. "IBM ILOG CPLEX Optimization Studio provides the fastest way to build efficient optimization models and state-of-the-art applications for the full range of planning and scheduling problems. With its integrated development environment, descriptive modelling language and built-in tools, it supports the entire model development process." (IBM ILOG CPLEX Optimization Studio manual)

If CPLEX is installed on the same computer as CMPL then CPLEX can be connected to CMPL by changing the entry `CplexFileName` in the file `<cmplhome>/bin/cmpl.opt`.

Example:

`CplexFileName = /Applications/IBM/ILOG/ ↲`
`  CPLEX_Studio125/cplex/bin/ ↲`
`  x86-64_darwin/cplex/`
	The cplex binary is located in the specified folder

Please note that for Windows installations you also have to use slashes as a path separators (instead of the usual backslashes). If this entry is correct then you can execute CPLEX directly by using the command

```
cmpl <problem>.cmpl -solver cplex
```

or by the CMPL header flag:

```
%arg -solver cplex
```

All CPLEX parameters described in the CPLEX manual (Parameters of CPLEX → Parameters Reference Manual) can be used in the CMPL header.

Usage CPLEX parameters within the CMPL header:

```
%opt cplex solverOption solverOptionValue
```

You have to use the parameters for the Interactive Optimizer. The names of sub-parameters of hierarchical parameters are to be separated by slashes.

Examples:

`%opt cplex threads 2`	Sets the default number of parallel threads that will be invoked.
`%opt cplex mip/limits/aggforcut 4`	Limits the number of constraints that can be aggregated for generating flow cover and mixed integer rounding (MIR) cuts to 4.
`%opt cplex ↲ simplex/tolerances/optimality ↲ 1e-8`	Sets the reduced-cost tolerance for optimality to 1e-8.

3.3.4.6 Other solvers

Since CMPL transforms a CMPL model into an MPS, a Free-MPS or an OSiL file, the model can be solved using most free or commercial solvers. To create MPS, Free-MPS or OSiL files please use the following commands:

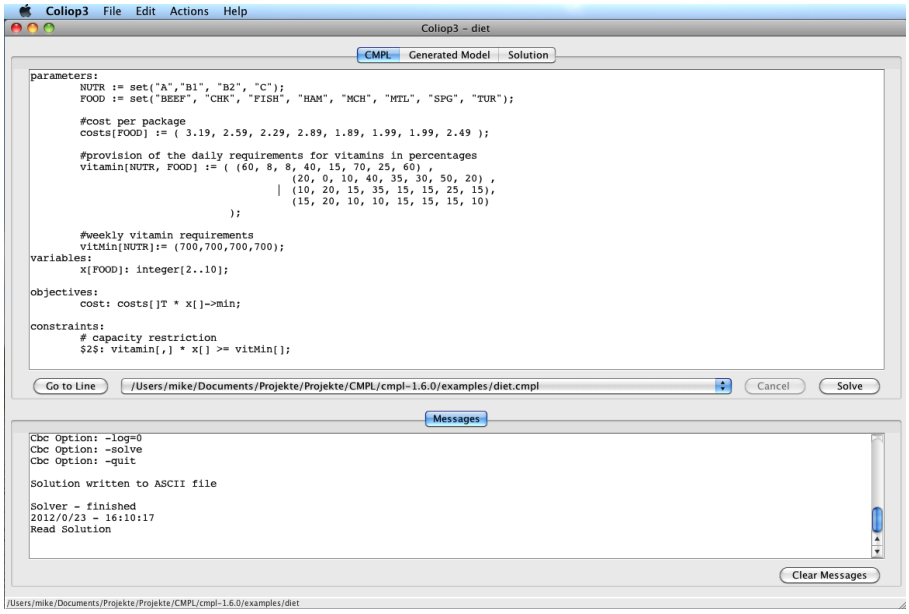
```
cmpl <problemname>.cmpl -m <problemname>.mps        #MPS export
cmpl <problemname>.cmpl -fm <problemname>.mps       #Free-MPS export
cmpl <problemname>.cmpl -x <problemname>.osil       #OSiL export
```

3.3.5 Coliop

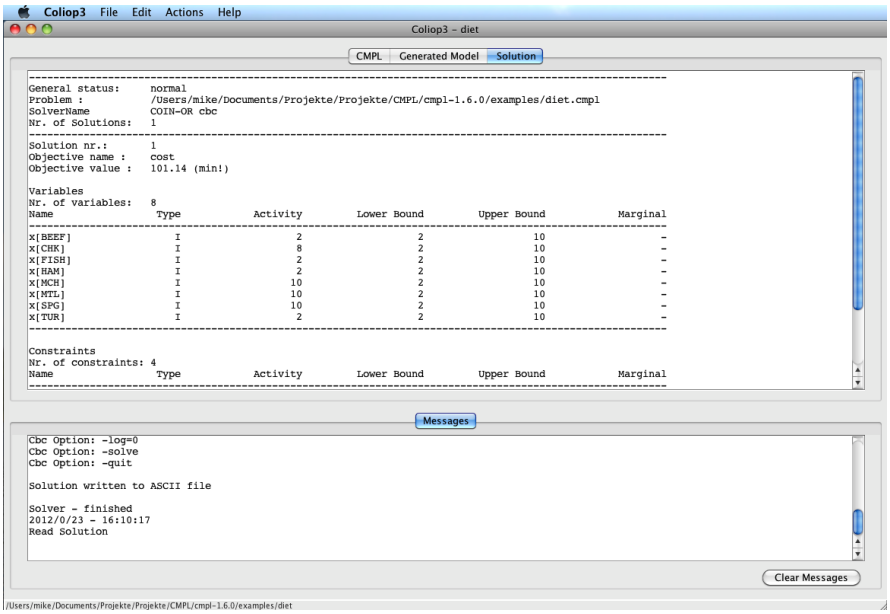
Coliop is an (simple) IDE (Integrated Development Environment) for CMPL intended to solve linear programming (LP) problems and mixed integer programming (MIP) problems. Coliop is a project of the Technical Uni-

versity of Applied Sciences Wildau and the Institute for Operations Research and Business Management at the Martin Luther University Halle-Wittenberg. Coliop is an open source project licensed under GPL. It is written in Java and is as an integral part of the CMPL distribution available for most of the relevant operating systems (OS X, Linux and Windows). Because Coliop is written in Java it is necessary to install the java runtime environment. Please visit to install java: http://java.com/de/download/index.jsp.

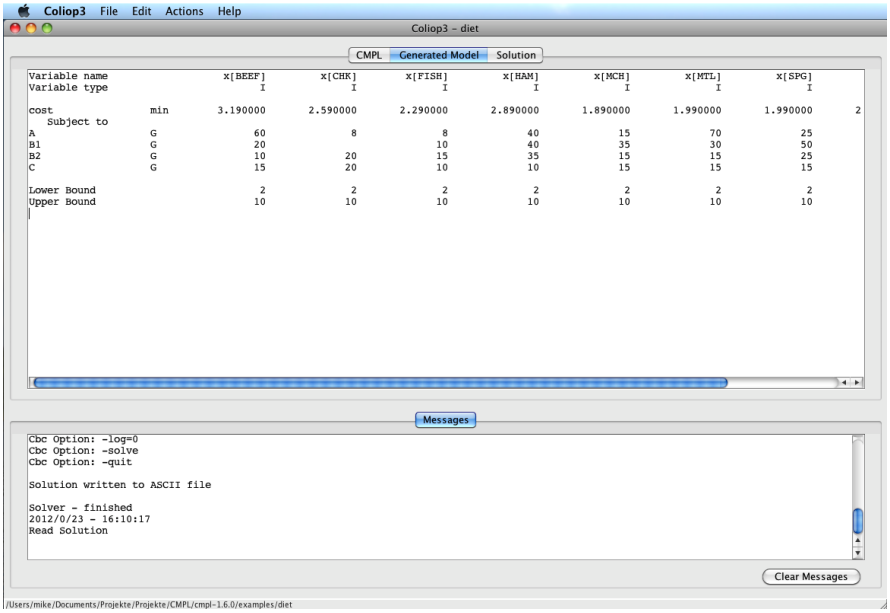
The first working step is to create or to open a CMPL model. By pushing the button <Solve> the model can be solved.



If a syntax error occurs then a user can analyse it by checking the CMPL messages. If the CMPL model is well formulated and the problem is feasible the standard solution report appears.



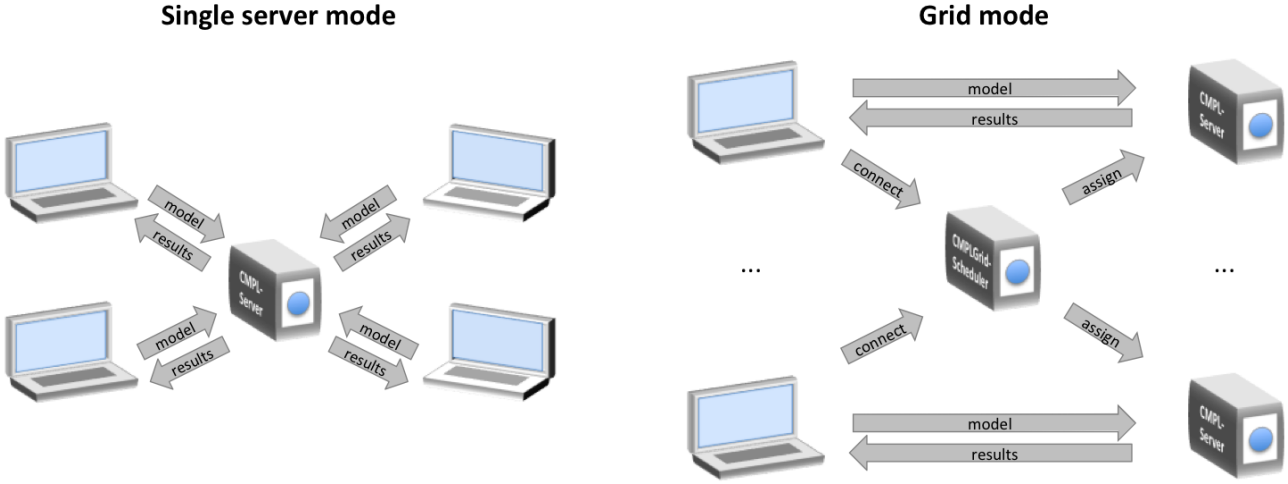
It is also possible to check the generated model matrix in the tab <Generated Model>.



3.4 CMPLServer

The CMPLServer is an XML-RPC-based web service for distributed and grid optimisation. XML-RPC provides XML based procedures for Remote Procedure Calls (RPC), which are transmitted between a client and a server via HTTP. (St. Laurent et al. 2001, p. 1.) XML-RPC has been chosen since this it is less resource consuming than other protocols like SOAP or REST due to its simpler functionalities.

A CMPLServer can be used in a single server mode or in a grid mode:



Both modes can be understood as distributed systems "in which hardware and software components located at networks computers communicate and coordinate their actions only by passing messages". (Coulouris et al, 2012, p. 17) Distributed optimisation is in this meaning interpretable as a distributed system that can be used for solving optimisation problems. (cf. Kshemkalyani & Singhal, 2008, p. 1; Fourer et.al., 2010)

CMPL provides four XML-based file formats for the communication between a CMPLServer and its clients in both modes (`CmplInstance`, `CmplSolutions`, `CmplMessages`, `CmplInfo`). A `CmplInstance` file contains an optimisation problem formulated in CMPL, the corresponding sets and parameters in the `Cm-`

`plData` file format as well all CMPL and solver options that belong to the CMPL model. If the model is feasible and a solution is found then a `CmplSolutions` file contains the solution(s) and the status of the invoked solver. If the model is not feasible then only the solver's status and the solver messages are given in the solution file. The `CmplMessages` file is intended to provide the CMPL status and (if existing) the CMPL messages. A `CmplInfo` file is an XML file that contains (if requested) several statistics and the generated matrix of the CMPL model.

In the single server mode only one CMPLServer that can be accessed synchronously or asynchronously by the clients exists in the network. A model can be solved synchronously by executing the CMPL binary with the command line argument `-cmplUrl <url>` or by running a pyCMPL or jCMPL programme by using the methods `Cmpl.connect(url)` for connecting the server and `Cmpl.solve()` for solving the model remotely.[1] The client sends the model to the CMPLServer and then waits for the results. If the model is feasible and an optimal solution is found the solution(s) can be received. If the model contains syntax or other errors or if the model is not feasible the CMPL and solver messages can be obtained. Whereby in the synchronous mode the client has to wait after sending the problem for the results and Messages in one process, a model can also be solved asynchronously with pyCMPL and jCMPL by using the methods `Cmpl.send()`, `Cmpl.knock()` and `Cmpl.retrieve()` in several steps. After sending the model to the CMPLServer via `Cmpl.send()` the server status can be obtained with `Cmpl.knock()`. If the CMPLServer is finished the solution, the CMPL and the solver states and messages can be received by `Cmpl.retrieve()`. It is reasonable to use the single server mode if a large model is formulated on a thin client in order to solve it remotely on the CMPLServer that is installed on a high performance system.

All these distributed optimisation procedures require a one-to-one connection between a CMPLServer and the client. The grid mode extends this approach by coupling CMPLServers from several locations and at least one coordinating CMPLGridScheduler to one "virtual CMPLServer" as a grid computing system that can be defined "as a system that coordinates distributed resources using standard, open, general-purpose protocols and interfaces to deliver non-trivial qualities of service." (Forster & Kesselmann 2003, pos. 722) For the client there does not appear any difference whether there is a connection to a single CMPLServer or to a CMPLGrid. The client's model is to be connected with the same functionalities as for a single CMPLServer to a CMPLGridScheduler which is responsible for the load balancing within the CMPLGrid and the assignment of the model to one of the connected CMPLServers. After this step the client is automatically connected to the chosen CMPLServer and the model can be solved synchronously or asynchronously. A CMPLGrid should be used for handling a huge amount of large scale optimisation problems. An example can be a simulation in which each agent has to solve its own optimisation problem at several times. An additional example for such a CMPLGrid application is an optimisation web portal that provides a huge amount of optimisation problems.

Both modes can be controlled by the `cmplServer` script that can be started on Linux and Mac in the terminal or under Windows in the `CmplShell` (`c:\program files[ (x86)]\Cmpl\cmplShell.bat`).

cmplServer <*command*> [<*port*>] [**-showLog**]

command:

`-start`	starts as single CMPLServer
`-startInGrid`	starts CMPLServer and connects to CMPLGrid
`-startScheduler`	starts as CMPLGridScheduler

1 Please take a look at the pyCMPL and jCMPL descriptions in chapter 4.

`-stop`	stops CMPLServer or CMPLGridScheduler
`-status`	returns the status of the CMPLServer or CMPLGridScheduler
port	defines CMPLServer's or CMPLGridScheduler's port
`-showLog`	shows the CMPLServer or CMPLGridScheduler log file

□

3.4.1 Single server mode

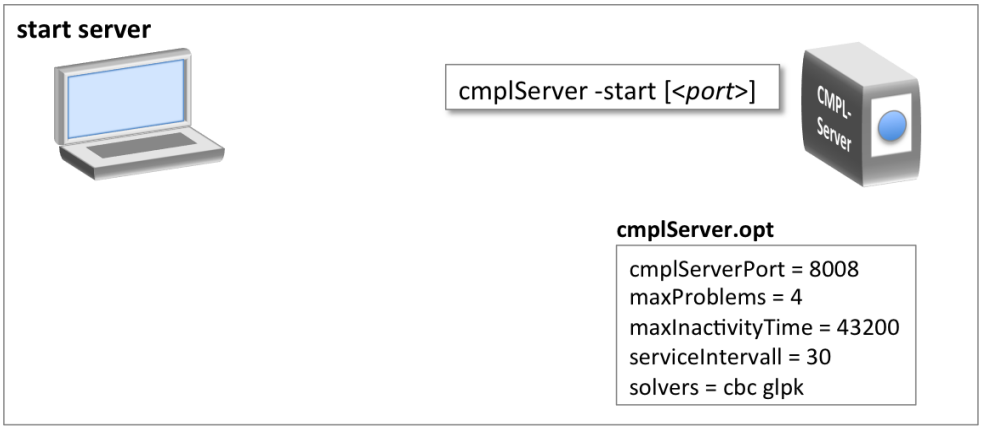
The first step to establish the single server mode is to start the CMPLServer by typing the command:

```
cmplServer -start [<port>]
```

Optionally a port can be specified as second argument. The behaviour of a CMPLServer can be influenced by editing the file `cmplServer.opt` that is located on Mac OS X in `/Applications/Cmpl/cmplServer`, on Linux in `/usr/share/Cmpl/cmplServer` and on Windows in `c:\program files[ (x86)]\Cmpl\cmplServer`. The example below shows the default values in this file.

```
cmplServerPort = 8008
maxProblems = 4
maxInactivityTime = 43200
serviceIntervall = 30
solvers = cbc glpk
```

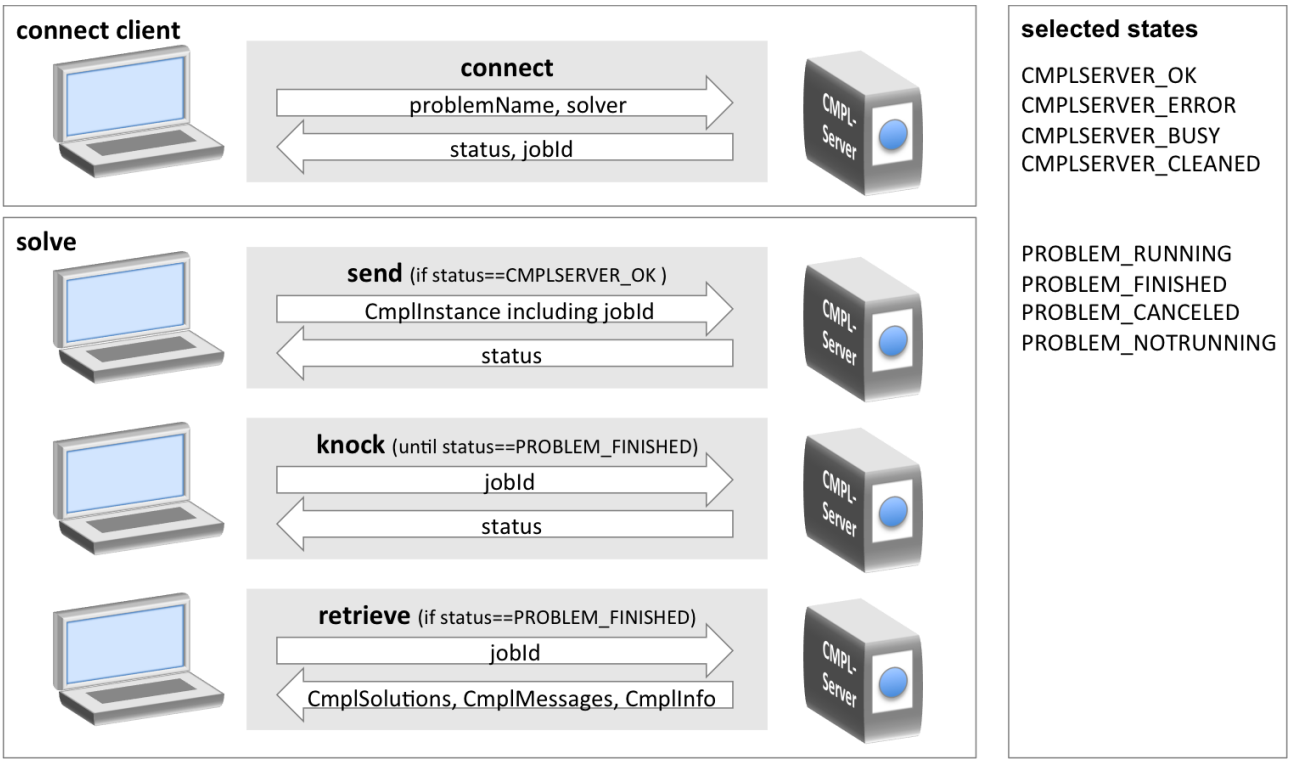
The default port of the CMPLServer can be specified with the parameter `port`. The parameter `maxProblems` defines how many problems can be carried out simultaneously. If more problems than `maxProblems` are connected with the CMPLServer the supernumerary problems are assigned to the problem waiting queue and automatically started if a running problem is finished or cancelled. If a problem is longer inactive than defined by the parameter `maxInactivityTime` it is cancelled and deleted automatically by the CMPLServer. This procedure as well as the problem waiting queue handling are performed by a service thread that works perpetual after a couple of seconds defined by the parameter `serviceIntervall`. With the parameter `solvers it` can be specified which solvers in the set of the installed solvers can be provided by the CMPLServer.



A running CMPLServer can be accessed by the CMPL binary or via CMPL's Python and Java APIs that contain CMPLServer clients. One can execute a CMPL model remotely on a CMPLServer by using the command line argument `-cmplUrl`.

```
cmpl <problem>.cmpl -cmplUrl http://<ip-adress-or-Domain>:<port>
```

For CMPL it is only possible to use the CMPLServer synchronously. That means CMPL waits right after sending the problem for the results and messages in one process. In pyCMPL and jCMPL a CMPLServer can be connected by using the method `Cmpl.connect()`. Executing a model can be done synchronously by executing the method `Cmpl.solve()` or asynchronously by using the methods `Cmpl.send()`, `Cmpl.knock()` and `Cmpl.retrieve()`. These main functionalities are illustrated in the following picture.



In the first step the client connects the CMPLServer, hands over its problem name and the solver with which the problem is to be solved. Then the client receives the status of the CMPLServer and if the status is `CMPLSERVER_OK` also the `jobId` is also sent. The status is `CMPLSERVER_ERROR` if the demanded solver is not supported or a CMPLServer occurs.

The synchronous method `Cmpl.solve()` is a bundle of the asynchronous methods `Cmpl.send()`, `Cmpl.knock()` and `Cmpl.retrieve()`.

`Cmpl.send()` sends a `CmplInstance` XML string that contains all relevant information about a CMPL model including the `jobId`, the CMPL and the solver options as well as the model itself and its data files to the CMPLServer. If the number of running problems including the model sent is greater than `maxProblems` the model is moved to the problem waiting queue and the CMPLServer returns the status `CMPLSERVER_BUSY`. If not the CMPLServer starts the solving process automatically if the CmplInstance string is completely received and the model and data files are written to the hard disc. In this case the status is set to `PROBLEM_RUNNING`.

A CMPLServer uses the home path of the user who is running it and saves all relevant data in `$HOME/CmplServer` (Mac and Linux) or `%HOMEPATH%\CmplServer` (Windows). The activities of the server can be

obtained in the file `CmplServer.log`. Each problem is stored in an own folder specified by the `jobId` which is deleted automatically after disconnecting the problem.

In the next step the client asks the CMPLServer whether solving the problem is finished or not via `Cmpl.knock()` whereby the `jobId` identifies the problem and the CMPLServer returns the current status. The client has to knock until the status is `PROBLEM_RUNNING` (or `CMPLSERVER_ERROR`). If the status is `CMPLSERVER_BUSY` the problem is put into the problem waiting queue until an empty solving slot is available or the maximum queuing time (defined with the CMPL option `-maxQueuingTime` or by default 300 seconds) is reached. The procedure then stops automatically.

If the status is equal to `PROBLEM_RUNNING` the solution, the CMPL and the solver messages and if requested some statistics can be received by using `Cmpl.retrieve()`. The client sends its `jobId` and then retrieves the `CmplSolution`, `CmplMesages` and `CmplInfo` XML strings. If `Cmpl.knock()` returns `CMPLSERVER_ERROR` the process is stopped.

The CMPLServer can be stopped by typing the command:

```
cmplServer -stop [<port>]
```

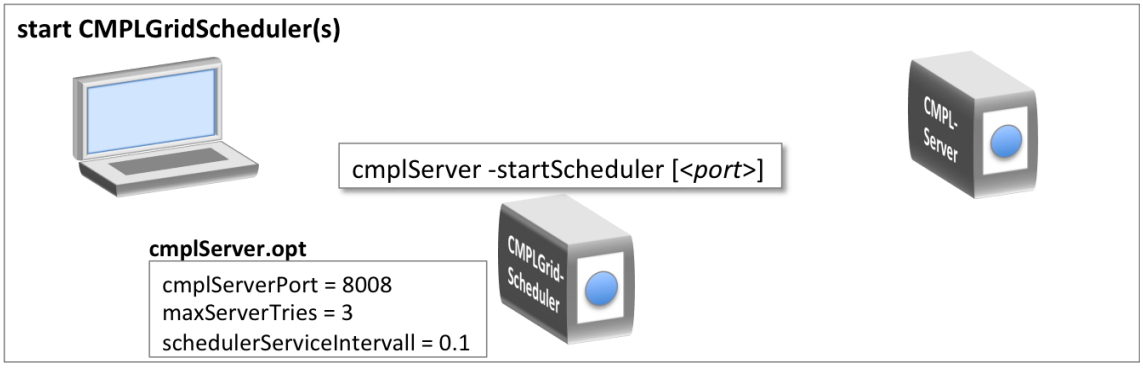
3.4.2 Grid mode

A CMPLGrid consists at least of one CMPLGridScheduler and usually a couple of CMPLServers that are connected to at least one scheduler. A CMPLGridScheduler is the gateway to the CMPLGrid for the clients and has to coordinate the traffic in the grid, that means it is responsible for the load balancing within the CMPLGrid and the assignment of the models to the connected CMPLServers. After receiving a model from a CMPLGridScheduler a CMPLServer has to communicate directly with the client to receive the model, to solve it and to send (if the problem is feasible) the solution(s), the CMPL and solver messages and if requested some information to the client. After these steps the client is disconnected automatically and the CMPLServers is waiting for the next problem from a CMPLGridScheduler.

The first step to start a CMPLGrid is to execute one or more CMPLGridScheduler by typing the command:

```
cmplServer -startScheduler [<port>]
```

As for the CMPLServers the parameter of a CMPLGridScheduler can be edited in the file `cmplServer.opt`.



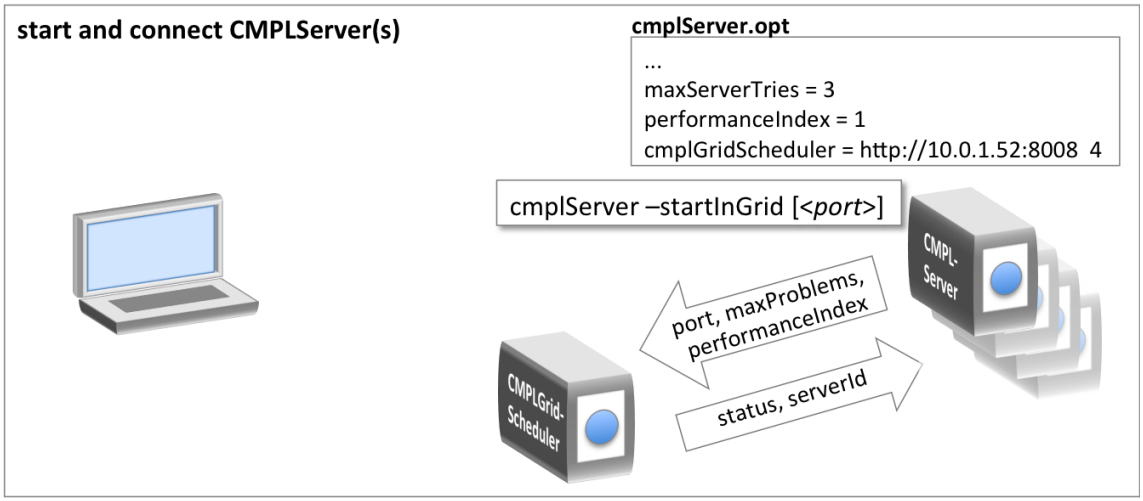
The relevant parameters in `cmplServer.opt` for a CMPLGridScheduler with there default values are shown below.

```
cmplServerPort = 8008
maxServerTries = 3
schedulerServiceIntervall = 0.1
```

The default port of the CMPLGridScheduler can be specified by the parameter `port`. If one wants to run a CMPLServer on the same computer as the CMPLGridScheduler then the server needs to be started with a different port via command line argument. Since the CMPLGridScheduler has to call functions provided by connected CMPLServers and additionally has to ensure a high availability and failover, the CMPLGridScheduler repeats failed CMPLServer calls whereby the number of tries are specified by the parameter `maxServerTries`. There is also a service thread that works permanently after a couple of seconds defined by the parameter `serviceIntervall`. Because this service thread is among others responsible for the problem waiting queue handling on the CMPLGridScheduler it makes sense to choose very short service intervals.

After running one or more CMPLGridSchedulers the involved CMPLServers can be started by typing the command:

```
cmplServer -startInGrid [<port>]
```



In addition to the described parameters in `cmplServer.opt` the following parameters are necessary for running a CMPLServer in a CMPLGrid.

```
...
maxServerTries = 3
performanceIndex = 1
cmplGridScheduler = http://10.0.1.52:8008  4
```

A CMPLServer in a CMPLGrid also has to call functions provided by a CMPLGridScheduler. Due to maximum availability and failover the maximum number of tries of failed CMPLGridScheduler calls are to be specified with the parameter `maxServerTries`. Assuming heterogeneous hardware for the CMPLServers in a CMPLGrid it is necessary for a reasonable load balancing to identify several performance levels of the invoked CMPLServers. This can be done by the parameter `performanceIndex` that influences the load balancing function directly. The involved operators of the CMPLServers and the CMPLGridScheduler(s) should specify standardised performance classes used within the entire CMPLGrid with the simple rule: the higher the performance class, the higher the `performanceIndex`. The parameter `cmplGridScheduler` is intended to specify the CMPLGridScheduler to which the CMPLServer is to be connected. The first argument is the URL of

the scheduler. The second parameter defines the maximum number of problems that the CMPLServer provides to this CMPLGridScheduler. If a CMPLServer should be connected to more than one scheduler one entry per CMPLGridScheduler is required. In the following example the CMPLServer will be connected to two CMPLGridSchedulers with maximally two problems per scheduler.

```
...
cmplGridScheduler = http://10.0.1.52:8008  2
cmplGridScheduler = http://10.0.1.53:8008  2
```

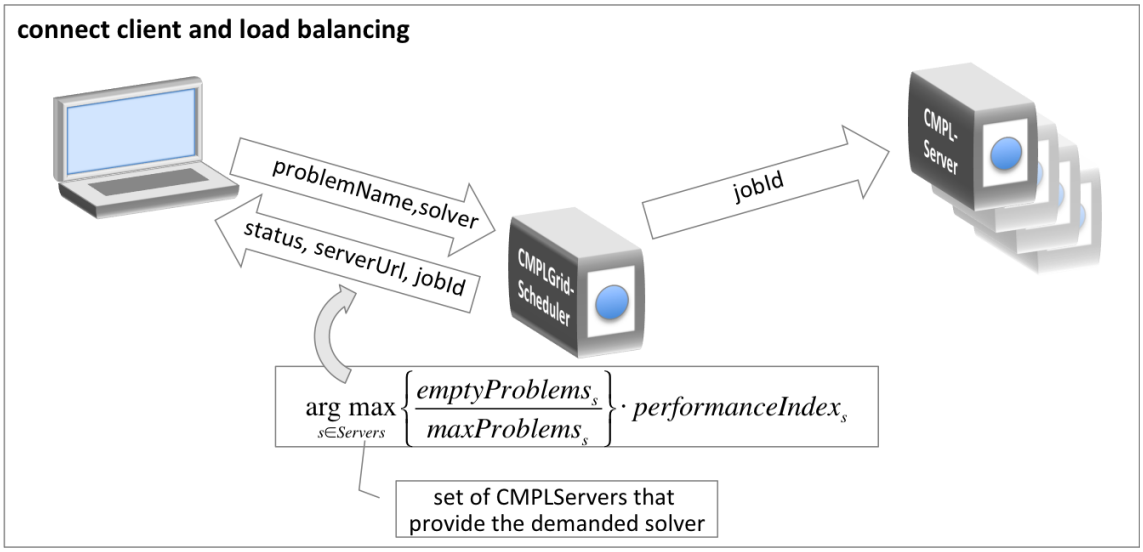
While connecting the CMPLGridScheduler the CMPLServer sends its port, the maximum number of provided problems and its performance index. It receives the status of the CMPLGridScheduler and a `serverId`. Possible states for connecting a CMPLServer are `CMPLGRID_SCHEDULER_OK` or `CMPLGRID_SCHEDULER _ERROR`.

Now a client can connect the CMPLGrid in the same way as a client connects a single CMPLServer either by using the CMPL binary

```
cmpl <problem>.cmpl -cmplUrl http://<ip-adress-or-Domain>:<port>
```

or in pyCmpl and jCMPL programmes through the method `Cmpl.connect()`.

The client sends automatically the name of the problem and the name of the solver with which the problem should be solved to the CMPLGridScheduler.



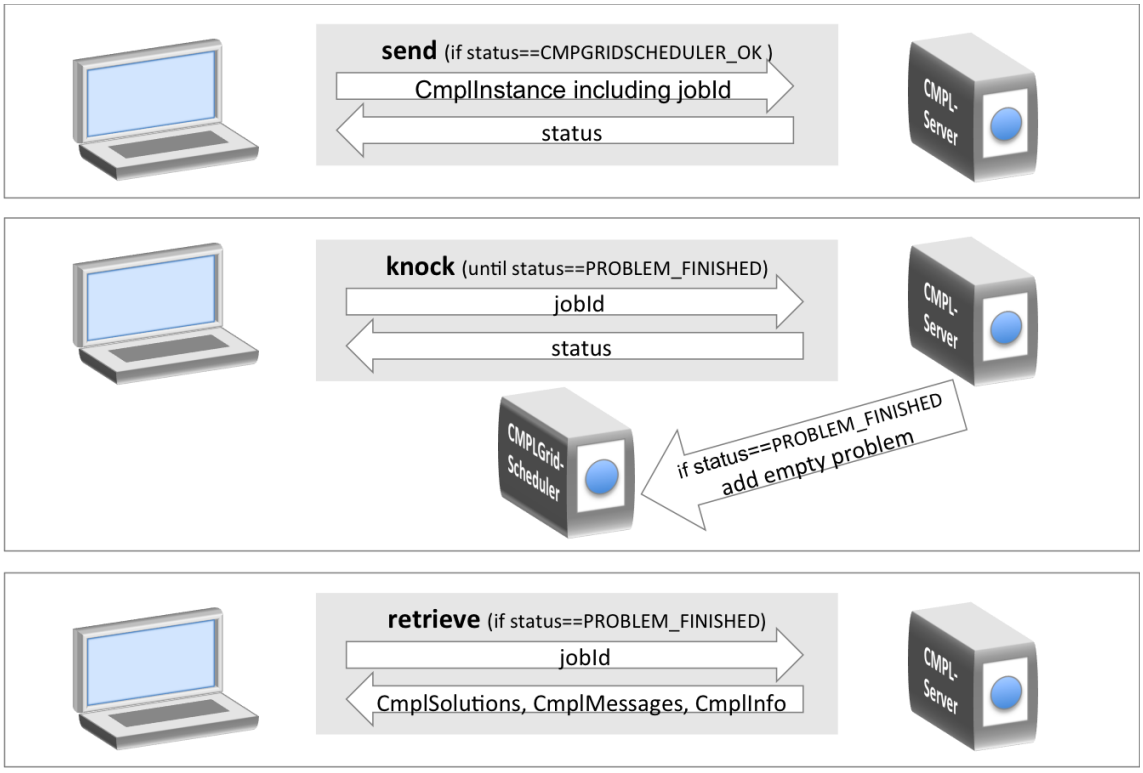
If the name of the solver is unknown or this solver is not available in the CMPLGrid the CMPLGridScheduler returns `CMPLSERVER_ERROR`. In case the problem waiting queue is not empty the problem is then assigned to the problem waiting queue and the status is `CMPLGRID_SCHEDULER_BUSY`.

Otherwise the CMPLGridScheduler returns the status `CMPLGRID_SCHEDULER_OK`, the `serverUrl` of the CMPLServer on which the problem will be solved and the `jobId` of the problem. This CMPLServer is determined on the basis of the load balancing function that is shown in the picture below. Per server that is providing the solver the current capacity factor is to be calculated by the relationship between the current empty problems of this server and the maximum number of provided problems. The number of empty problems is controlled by the CMPLGridScheduler with a lower bound of zero and an upper bound equal to the maximum number of provided problems. This parameter is decreased if the CMPLServer is taking over a problem and it

is increased when the CMPLServer has finished the problem or the problem is cancelled. The idea is to send problems tendentiously to those CMPLServer with the highest empty capacity. To include the different performance levels of the invoked CMPLServers in the load balancing decision, the current capacity factor is to be multiplied by the performance index. The result is the load balancing factor and the CMPLServer with the highest load balancing factor is assigned to the client to solve the problem. This CMPLServer then gets the `jobId` of the CMPL problem by the CMPLGridServer in order to take over all relevant processes to solve this problem. Afterwards the client is automatically connected to this CMPLServer.

The problem waiting queue handling is organised by the CMPLGrid Scheduler service thread that assigns the waiting problems automatically to CMPLServers by using the same functionalities as described above. The waiting clients either ask automatically in the synchronous mode or manually in the asynchronous mode both through `Cmpl.knock()` until the received status is not equal to `CMPGRID_SCHEDULER_BUSY`.

The next steps to solve the problem synchronously or asynchronously on the CMPLServer are similar to the procedures in the single server mode as shown in the following figure.



The methods `Cmpl.send()`, `Cmpl.knock()` and `Cmpl.retrieve()` are used to send the problem to the CMPLServer, to knock for the current status, to retrieve the solution and the CMPL and the solver messages and if requested some statistics. The main differences to the single server mode are that the CMPLServer calls the CMPLServerGrid to add an empty problem slot after finishing solving the problem and that the client is disconnected automatically from the CMPLServer after retrieving the solution, messages and statistics.

The CmplGridScheduler and the CmplServers can be stopped by typing the command:

```
cmplServer -stop [<port>]
```

3.4.3 Reliability and failover

A distributed optimisation system or a grid optimisation system is usually implemented in a heterogeneous environment. The network notes can be installed on different hardware as well as on different operating systems. This fact could cause some disturbances within the optimisation network that should be either avoided or reduced in their negative impact of the optimisation processes.

Beside ensuring a good performance, maximum reliability and failover are therefore important targets of the CMPLServer and the CMPLGrid implementations. They are ensured by:

(a) the problem queue handling on the CMPLGridScheduler and the CMPLServer,
(b) multiple executions of failed server calls and
(c) re-connections of problems to the CMPLGridScheduler if an assigned CMPLServer fails.

(a) Problem queue handling

If a problem is connected to a CMPLServer or a CMPLGridScheduler and the number of running problems including the model sent is greater than `maxProblems`, it neither makes sense to cancel the problem nor to interrupt the solving process. Especially in case of an iterating solving process with a couple of depending problems it is the better way to refer the supernumerary problems automatically to the problem waiting queue.

For the single server mode the problem queue handling is organised by the CMPLServer whilst in the grid mode the CMPLGridScheduler(s) are responsible for it. In both modes a problem stored in the problem waiting queue has to wait until an empty solving slot is available or the maximum queuing time is reached.

In the single server mode the number of problems that can be executed simultaneously on the particular CMPLServer are defined by the parameter `maxproblems` in `cmplServer.opt`. With this parameter it should be avoided to overwhelm the server and to avoid the super-proportional effort for managing a huge amount of parallel problems. The first empty solving slot that appears when a running problem is finished or cancelled, is taking over a waiting problem by using the FIFO approach.

The number of simultaneously running problems in a CMPLGrid is defined by the sum over all connected CMPLServer of the maximum number of problems provided by the servers. This parameter is to be defined per CMPLServer in `cmplServer.opt` as second argument in the entry `cmplGridScheduler = <`*`url`*`> <`*`maxProblems`*`>`. The CMPLGridScheduler counts the number of running problems per CMPLServer in relation to its maximum number of provided problems. If it is not possible to find a connected CMPLServer with an empty solving slot then the problem is put to the problem waiting queue. In contrast to the single server mode the problem which has been waiting longest is not executed by the first appearing free CMPLServer but it is organised by the described load balancing function over the set of CMPLServers that stated an empty solving slot during two iterations of the CMPLGridScheduler service thread.

The client's maximum queuing time in seconds can be specified with the CMPL command line argument `-maxQueuingTime <`*`sec`*`>`. This argument can also be set as CMPL header entry `%arg -maxQueuingTime <`*`sec`*`>` or in pyCMPL and jCMPL with the method `Cmpl.setOption("%arg -maxQueuingTime <`*`sec`*`>")`. The default value is 300 seconds.

(b) Multiple executions of failed server calls

To avoid that a single execution of a server method, which fails due to network problems like socket errors or others, cancels the entire process, all failed server calls can be executed again several times. The maximum number of executions of failed server calls can be specified for the clients by the CMPL command line argument `-maxServerTries <tries>`. It can also be used in a CMPL header entry `%arg -maxServerTries <tries>` or in pyCMPL and jCMPL by using `Cmpl.setOption("%arg -maxServerTries <tries>")`. The default value is 10. The number of maximum executions of failed server calls in the communication between the CMPLGridScheduler and CMPLServers is defined in `cmplServer.opt` with the entry `maxServerTries = <tries>`.

An exemplary and simplified implementation of this behaviour is shown in the pseudo code listing below:

```
1   serverTries=0
2   while True do
3       try
4         callServerMethod()
5       except
6         serverTries+=1
7         if serverTries>maxServerTries then
8              status=CMPLSERVER_ERROR
9              raise CmplException("calling CmplServer function … failed")
10        end if
11      end try
12      break
13  end while
```

In a first step the variable `serverTries` is assigned zero. The call of the server method (line 4) is imbedded in an infinite loop (lines 2-13) and in a try-except-block for the exception handling (lines 3-11). If no exception occurs then the loop is finished by the break command in line 12. Otherwise `serverTries` is incremented by 1. If the maximum number is not exceeded (line 7) the server method is called again (line 4). If `serverTries` is greater than `maxServerTries` then the class variable `Cmpl.status` is set to `CMPLSERVER_ERROR` and a `CmplException` is raised that have to be handled in the code in which the listing below is imbedded (lines 7-9).

(c) Re-connections of failed problems to the CMPLGridScheduler

Multiple server calls are mainly intended to prevent network problems. But it could be also possible that other problems caused by CMPLServers connected to a CMPLGridScheduler (e.g. a failed execution of a solver, file handling problems at a CMPLServer or the unpredictable shutdown of a CMPLServer) occur. The idea to handle such problems is that if the assigned CMPLServer fails the particular problem is then reconnected to the CMPLGridScheduler and is taken over by another CMPLServer automatically.

The following pseudo code listing describes a simplified implementation of `Cmpl.solve()` only for the grid mode to illustrate this approach:

```
1   serverTries=0
2   while True do
3       try
4         if status==CMPLSERVER_ERROR then
5              CmplGridScheduler.connect()
6         end if
7
8         if status==CMPLGRID_SCHEDULER_BUSY then
9              while status<>CMPLGRID_SCHEDULER_OK do
10                  CmplGridScheduler.knock()
11                  if waitingTime()>=maxQueuingTime then
12                       raise CmplException("max. queuing time is exceeded.")
13                  end if
14             end while
15        end if
16        connectedToServer=True
17
18        CmplServer.send()
19
20        while status<>PROBLEM_FINISHED do
21             CmplServer.knock()
22        end while
23
24        CmplServer.retrieve()
25        break
26
27      except CmplException
28        serverTries+=1
29        if status==CMPL_ERROR and connectedToServer==True then
30             CmplGridScheduler.cmplServerFailed()
31        end if
32        if serverTries>maxServerTries or status==CMPLGRID_SCHEDULER_BUSY then
33             ExceptionHandling()
34             exit
35        end if
36      end try
37  end while
```

As in the listing of the multiple server calls the variable `serverTries` is assigned zero (line 1). The entire method is also imbedded in an infinite loop (lines 2-37) and the exception handling is organised as try-except-block (lines 3-36).

Before `Cmpl.solve()` is called the client has to execute `Cmpl.connect()` successfully. Therefore the class variable `Cmpl.status` has to be unequal to `CMPLSERVER_ERROR` and an additional `Cmpl.connect()` is not necessary in the first run of `Cmpl.solve()`(lines 4-6).It is possible that the entire CMPLGrid is busy, the status equals `CMPLGRID_SCHEDULER_BUSY` and the problem is moved to the CMPLGridScheduler problem waiting queue (line 8). In this case the problem has to wait for the next empty

solving slot via `Cmpl.knock()` (line 10) until the CMPLGridScheduler returns the status `CMPLGRID_SCHEDULER_OK` (line 9) or the waiting time exceeds the maximum queuing time and a `CmplException` is raised (lines 11-13).

After this loop the problem is automatically connected to a CMPLServer within the CMPLGrid. The class variable `Cmpl.connectedToServer` is assigned `True` (line 16) and the problem is sent to this server through `Cmpl.send()` (line 18). The problem then has to wait until the problem status is `PROBLEM_FINISHED` (lines 20-22). As soon as the problem is finished, the solution(s), the CMPL and the solver messages as well as (if requested) some statistics can be retrieved via `Cmpl.retrieve()` (line 24). If no `CmplException` or another exception appeared during this procedures the infinite loop is left by the break command in line 25.

Otherwise the `CmplException` or other exceptions have to be handled in the except block in the lines 27-36. The first step is to increase the number of failed server call tries (line 28). If while executing `Cmpl.connect()`, `Cmpl.send()`, `Cmpl.knock()` or `Cmpl.retrieve()` an exception is raised and the problem is connected to a CMPLServer then the client calls the CMPLGridScheduler method `cmplServerFailed()` in order to report that this CMPLServer failed and to set the status of this server to `inactive` on the CMPLGridScheduler (line 30). This CMPLServer is then excluded from the CMPLGridScheduler load balancing until CMPLGridScheduler's service thread recognises that this CMPLServer is able to take over problems again.

If the number of failed server calls exceeds the maximum number of tries or the status is `CMPLGRID_SCHEDULER_BUSY` because the maximum queuing time is exceeded (line 32), the entire procedure stops by doing the necessary exception handling and by exiting the programme (lines 33-34).

Otherwise the problem has to pass the loop again. That means that the problem is reconnected to the CMPLGrid via `CMPLGridScheduler.connect()` (lines 4-6) and the solving process starts again.

3.5 pyCMPL

pyCMPL is the CMPL API for Python and an interactive shell. The main idea of this API is to define sets and parameters within the user application, to start and control the solving process and to read the solution(s) into the application if the problem is feasible. All variables, objective functions and constraints are defined in CMPL. These functionalities can be used with a local CMPL installation or a CMPLServer.

- Running pyCMPL
  - Windows
    Please start the cmplShell `c:\program files[ (x86)]\Cmpl\cmplShell.bat` and execute a pyCMPL script with the command `pyCmpl <problemname>.py`.

  - Linux and Apple - OS X
    Please start the terminal and execute a pyCMPL script with the command
    `pyCmpl <problemname>.py`.

3.6 jCMPL

jCMPL is the CMPL API for Java. The main idea of this API is similar to pyCMPL to define sets and parameters within the user application, to start and control the solving process and to read the solution(s) into the application if the problem is feasible. All variables, objective functions and constraints are defined in CMPL.

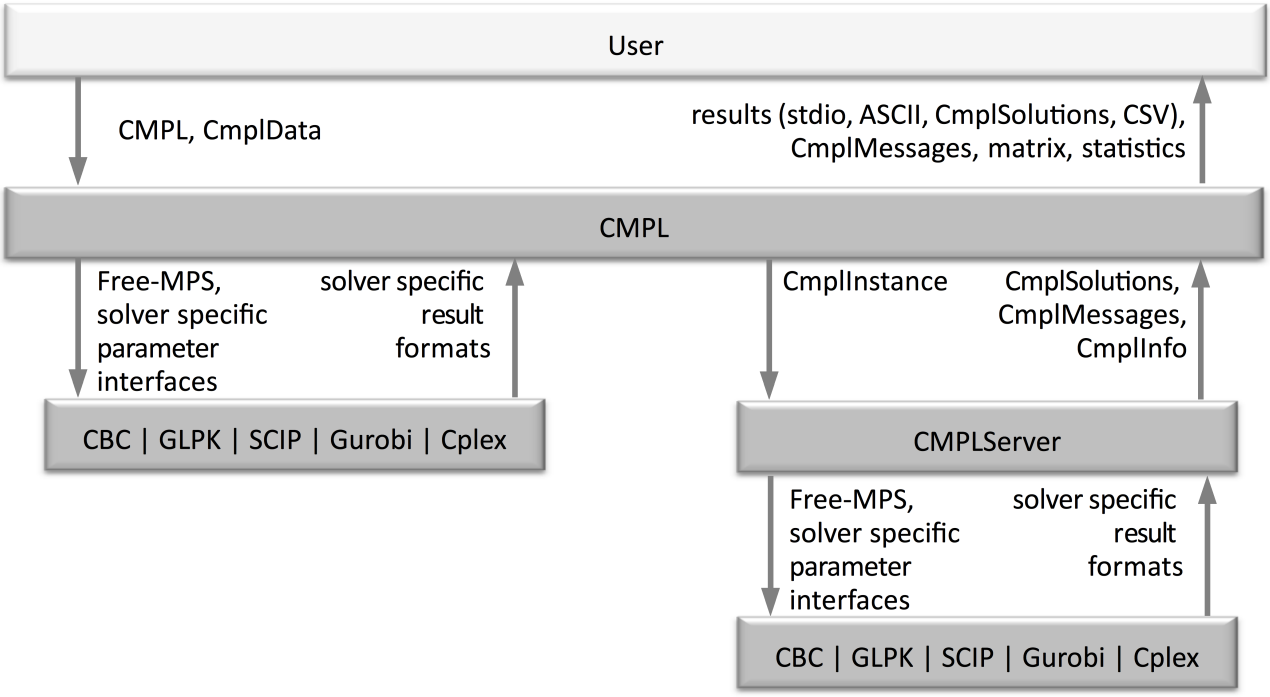
These functionalities can be used with a local CMPL installation or a CMPLServer.

To use the jCMPL functionalities a Java programme has to import jCMPL by `import jCMPL.*;` and to link your application against `jCmpl.jar` and the following jar files, that you can find in the CMPL application folder in `jCmpl/Libs` : `commons-lang3`, `ws-commons-util`, `xmplrpc-client`, `xmlrpc-commons`.

3.7 Input and output file formats

3.7.1 Overview

As shown in the picture below CMPL uses several ASCII files for the communication with the user, the solvers and a CMPLServer.



CMPL	input file for CMPL - syntax as described above
CmplData	data file format for CMPL - syntax as described above
Free-MPS	output file for the generated model in Free-MPS format
CmplInstance	XML file that contains all relevant information about a CMPL model sent to a CMPLServer
Result files	solutions of a CMPL model can be obtained in the form of an ASCII, CSV or CmplSolutions file

CmplSolutions	solutions can be solved in CMPL's XML based solution file format
CmplMessages	XML file that contains the status and messages of a CMPL model
CmplInfo	XML file that contains (if requested) several statistics and the generated matrix of the CMPL model

To describe the several file types it is necessary to distinguish between the local and the remote mode.

In the local mode a CMPL model and (if existing) the corresponding CmplData files are parsed and translated into a Free-MPS file (If no syntax or other error occur). If there are some errors in the CMPL model the CMPL messages are shown automatically or can be saved in a CmplMessages file. The Free-MPS file is together with solver specific parameter handed over to the chosen solver that is executed directly by CMPL. If the problem is feasible and an optimal solution is found CMPL reads the solution in form of the solver specific result format. A CMPL user can now obtain the standard solution report or can save the solution(s) as ASCII or CSV file or as CmplSolutions file. It is also possible to obtain the generated matrix and some statistics on the screen or in a plain text file.

A user can also process his or her CMPL model remotely on a CMPLServer. In the first step CMPL writes automatically all model relevant information (CMPL and CmplData files, CMPL and solver options) in a CmplInstance file and sends it to the connected CMPLServer. After solving the model CMPL receives three XML-based file formats (CmplSolutions, CmplMessages, CmplInfo) and the user can obtain (if a optimal solution is found) the standard solution report or can save the solution(s) and also can get the generated matrix and some statistics. If the CMPL model contains errors then the user can retrieve the CMPL messages.

3.7.2 CMPL and CmplData

A CMPL file is an ASCII file that includes the user-defined CMPL code with a syntax as described in this manual.

The example

$$15 \cdot x_1 + 18 \cdot x_2 + 22 \cdot x_3 \rightarrow max\ !$$
$$s.t.$$
$$5 \cdot x_1 + 10 \cdot x_2 + 15 \cdot x_3 \leq 175$$
$$10 \cdot x_1 + 5 \cdot x_2 + 10 \cdot x_3 \leq 200$$
$$0 \leq x_n \quad ; n = 1(1)3$$

can be formulated with the CmplData file `test.cdat`

```
%n set <1..2>
%m set <1..3>

%c[m] < 15 18 22 >
%b[n] < 175 200 >
%A[n,m] <   5 10 15
           10 5 10 >
```

and the CMPL file `test.cmpl`

```
%data test.cdat
%arg -solver glpk
```

```
variables:
     x[m]: real[0..];
objectives:
     profit: c[]T * x[] -> max;
constraints:
     res: A[,] * x[] <= b[];
```

3.7.3 Free - MPS

The Free-MPS-format is internally used for the communication between CMPL and all local installed solvers.

The Free-MPS format is an improved version of the MPS format. There is no standard for this format but it is widely accepted. The structure of a Free-MPS file is the same as an MPS file. But most of the restricted MPS format requirements are eliminated, e.g. there are no requirements for the position or length of a field. For more information please visit the project website of the lp_solve project. [http://lpsolve.sourceforge.net]

The Free-MPS file for the given CMP example is generated as follows:

```
* CMPL - Free-MPS - Export
NAME          test.cmpl
ROWS
 N profit
 L res[1]
 L res[2]
COLUMNS
 x[1] profit 15 res[1] 5
 x[1] res[2] 10
 x[2] profit 18 res[1] 10
 x[2] res[2] 5
 x[3] profit 22 res[1] 15
 x[3] res[2] 10
RHS
 RHS res[1] 175 res[2] 200
RANGES
BOUNDS
 PL BOUND x[1]
 PL BOUND x[2]
 PL BOUND x[3]
ENDATA
```

3.7.4 CmplInstance

CmplInstance is an XML-based format that contains all relevant information about a CMPL model (CMPL and CmplData files, CMPL and solver options) to be sent to a CMPLServer.

A CmplInstance file consists of three major sections. The `<general>` section contains the name of the problem and the jobId that is received automatically during connecting the CMPLServer. The `<options>` section consists of the CMPL and solver options that a user has specified on the command line. The `<prob-`

`lemFiles>` section is indented to store the CMPL file and all corresponding CmplData files. All CmplData files no matter whether they are specified within the CMPL model or as command line argument are automatically included in the CmplInstance file. To avoid some misinterpretation of some special characters while reading the CmplInstance on the CMPLServer the content of the CMPL model and the CmplData files are automatically unescaped by CMPL.

The XSD (XML Schema Definition) of CmplInstance is defined as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:element name="CmplInstance">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="general" minOccurs="1" maxOccurs="1" />
                <xs:element ref="options" minOccurs="0" maxOccurs="1"/>
                <xs:element ref="problemFiles" minOccurs="1" maxOccurs="1"/>
            </xs:sequence>
            <xs:attribute name="version" type="xs:decimal" use="required"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="general">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="name"  type="xs:string" minOccurs="1" maxOccurs="1"/>
                <xs:element  name="jobId"  type="xs:string" minOccurs="1" maxOccurs="1"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="options">
        <xs:complexType>
            <xs:sequence>
                <xs:element  name="opt" type="xs:string" minOccurs="0" maxOccurs="unbounded" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="problemFiles">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="file" minOccurs="1" maxOccurs="unbounded" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="file">
        <xs:complexType>
              <xs:simpleContent>
```

```
                    <xs:extension base="xs:string">
                    <xs:attribute name="name" type="xs:string" use="required"/>
                    <xs:attribute name="type" type="fileType" use="required"/>
                </xs:extension>
                </xs:simpleContent>
        </xs:complexType>
    </xs:element>

    <xs:simpleType name="fileType">
                <xs:restriction base="xs:string">
                    <xs:enumeration value="cmplMain"/>
                    <xs:enumeration value="cmplData"/>
                </xs:restriction>
        </xs:simpleType>

</xs:schema>
```

For the given example the CmplInstance file `test.cinst` is created automatically by CMPL through by using the command line arguments `-matrix "test.mat" -s "test.stat"` and the model is executed remotely on a CMPLServer.

```
<?xml version = "1.0" encoding="UTF-8" standalone="yes"?>
<CmplInstance version="1.0">
<general>
<name>test.cmpl</name>
<jobId>10.0.1.2-2014-01-05-17-05-23-496795</jobId>
</general>
<options>
<opt >%arg -matrix "test.mat"</opt>
<opt>%arg -s "test.stat"</opt>
</options>
<problemFiles>
<file name="test.cmpl" type="cmplMain">
%data test.cdat
%arg -solver glpk
%arg -cmplUrl http://10.0.1.2:8008

variables:
    x[m]: real[0..];
objectives:
    profit: c[]T * x[] -&gt; max;
constraints:
    res: A[,] * x[] &lt;= b[];
</file>
<file name="test.cdat" type="cmplData">
%n set &lt;1..2&gt;
%m set &lt;1..3&gt;
```

```
%c[m] &lt; 15 18 22 &gt;
%b[n] &lt; 175 200 &gt;
%A[n,m] &lt; 5 10 15
      10 5 10 &gt;
</file>
</problemFiles>
</CmplInstance>
```

3.7.5 ASCII or CSV result files

If the problem is feasible and an optimal solution is found a user can obtain this optimal solution in the form of an ASCI or CSV file by using the command line arguments `-solutionAscii [<file>]` or `-solutionCsv [<file>]`. This files can additionally contain all integer feasible solutions if Cplex or Gurobi are used and the the CMPL header option `%display solutionPool` is defined.

The ASCII result file `test.sol` for the given CMPL example is generated as follows:

```
---------------------------------------------------------------------------------------------------------
Problem             test.cmpl
Nr. of variables    3
Nr. of constraints  2
Objective name      profit
Solver name         GLPK
Display variables   (all)
Display constraints (all)
---------------------------------------------------------------------------------------------------------

Objective status    optimal
Objective value     405 (max!)

Variables
Name                Type            Activity        Lower bound        Upper bound           Marginal
---------------------------------------------------------------------------------------------------------
x[1]                   C                  15                  0           Infinity                  0
x[2]                   C                  10                  0           Infinity                  0
x[3]                   C                   0                  0           Infinity                 -7
---------------------------------------------------------------------------------------------------------

Constraints
Name                Type            Activity        Lower bound        Upper bound           Marginal
---------------------------------------------------------------------------------------------------------
res[1]                 L                 175          -Infinity                175                 1.4
res[2]                 L                 200          -Infinity                200                 0.8
---------------------------------------------------------------------------------------------------------
```

The corresponding CSV result file `test.csv` is generated as follows:

```
CMPL csv export

Problem;test.cmpl
Nr. of variables;3
Nr. of constraints;2
Objective name;profit
Objective sense;max
Solver;GLPK
Display variables;(all)
```

```
Display constraints;(all)

Objective status;optimal
Objective value;405
Variables;
Name;Type;Activity;LowerBound;UpperBound;Marginal
x[1];C;15;0;Infinity;0
x[2];C;10;0;Infinity;0
x[3];C;0;0;Infinity;-7
Constraints;
Name;Type;Activity;LowerBound;UpperBound;Marginal
res[1];L;175;-Infinity;175;1.4
res[2];L;200;-Infinity;200;0.8
```

3.7.6 CmplSolutions

CmplSolutions is an XML-based format for representing the general status and the solution(s) if the problem is feasible and one or more solutions are found. A user can save it by using the command line argument `-solution [<File>]`. It is also internally used for receiving solution(s) from a CMPLServer.

As shown in the corresponding XSD below A CmplSolutions file contains a `<general>` block for general information about the solved problem and a `<solutions>` block for the results of all solutions found including the variables and constraints.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

  <xs:element name="CmplSolutions">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="general" minOccurs="1" maxOccurs="1"/>
        <xs:element ref="solution" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="version" use="required" type="xs:decimal"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="general">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="instanceName" type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="nrOfVariables" type="xs:nonNegativeInteger" minOccurs="1" maxOccurs="1"/>
        <xs:element name="nrOfConstraints" type="xs:nonNegativeInteger" minOccurs="1" maxOccurs="1"
/>
        <xs:element name="objectiveName" type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="objectiveSense" type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="nrOfSolutions" type="xs:nonNegativeInteger" minOccurs="1" maxOccurs="1"/>
        <xs:element name="solverName" type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="variablesDisplayOptions" type="xs:string" minOccurs="1" maxOccurs="1" />
```

```
      <xs:element name="constraintsDisplayOptions" type="xs:string" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="solution">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="variables" minOccurs="1" maxOccurs="1"/>
      <xs:element ref="linearConstraints"  minOccurs="1" maxOccurs="1" />
    </xs:sequence>
    <xs:attribute name="idx" use="required" type="xs:nonNegativeInteger"/>
    <xs:attribute name="status" use="required" type="xs:string"/>
    <xs:attribute name="value" use="required" type="xs:decimal"/>
  </xs:complexType>
</xs:element>

<xs:element name="variables">
  <xs:complexType>
    <xs:sequence>
      <xs:element  minOccurs="1" maxOccurs="unbounded" ref="variable"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="linearConstraints">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="unbounded" ref="constraint"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="variable">
  <xs:complexType>
    <xs:attribute name="idx" use="required" type="xs:nonNegativeInteger"/>
    <xs:attribute name="name" use="required" type="xs:string"/>
    <xs:attribute name="type" use="required" type="varType"/>
       <xs:attribute name="activity" use="required" type="xs:double"/>
    <xs:attribute name="lowerBound" use="required" type="xs:double"/>
    <xs:attribute name="upperBound" use="required" type="xs:double"/>
    <xs:attribute name="marginal" use="required" type="xs:double"/>
  </xs:complexType>
</xs:element>

<xs:element name="constraint">
  <xs:complexType>
    <xs:attribute name="idx" use="required" type="xs:nonNegativeInteger"/>
    <xs:attribute name="name" use="required" type="xs:string"/>
```

```
      <xs:attribute name="type" use="required" type="conType"/>
         <xs:attribute name="activity" use="required" type="xs:double"/>
      <xs:attribute name="lowerBound" use="required" type="xs:double"/>
      <xs:attribute name="upperBound" use="required" type="xs:double"/>
      <xs:attribute name="marginal" use="required" type="xs:double"/>
    </xs:complexType>
  </xs:element>

  <xs:simpleType name="varType">
       <xs:restriction base="xs:string">
              <xs:enumeration value="C"/>
              <xs:enumeration value="I"/>
              <xs:enumeration value="B"/>
       </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="conType">
       <xs:restriction base="xs:string">
              <xs:enumeration value="L"/>
              <xs:enumeration value="E"/>
              <xs:enumeration value="G"/>
       </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

The CmplSolutions file `test.csol` for the given CMPL example is generated as follows:

```
<?xml version = "1.1" encoding="UTF-8" standalone="yes"?>
<CmplSolutions version="1.0">
   <general>
       <instanceName>test.cmpl</instanceName>
       <nrOfVariables>3</nrOfVariables>
       <nrOfConstraints>2</nrOfConstraints>
       <objectiveName>profit</objectiveName>
       <objectiveSense>max</objectiveSense>
       <nrOfSolutions>1</nrOfSolutions>
       <solverName>GLPK</solverName>
       <variablesDisplayOptions>(all)</variablesDisplayOptions>
       <constraintsDisplayOptions>(all)</constraintsDisplayOptions>
   </general>
   <solution idx="0" status="optimal" value="405">
       <variables>
           <variable idx="0" name="x[1]" type="C" activity="15" lowerBound="0"
              upperBound="Infinity" marginal="0"/>
           <variable idx="1" name="x[2]" type="C" activity="10" lowerBound="0"
              upperBound="Infinity" marginal="0"/>
           <variable idx="2" name="x[3]" type="C" activity="0" lowerBound="0"
```

```
            upperBound="Infinity" marginal="-7"/>
      </variables>
      <linearConstraints>
          <constraint idx="0" name="res[1]" type="L" activity="175"
            lowerBound="-INF" upperBound="175" marginal="1.4"/>
          <constraint idx="1" name="res[2]" type="L" activity="200"
            lowerBound="-INF" upperBound="200" marginal="0.8"/>
      </linearConstraints>
   </solution>
</CmplSolutions>
```

3.7.7 CmplMessages

CmplMessages is an XML-based format for representing the general status and/or errors of the transformation of a CMPL model in one of the described output files. CmplMessages is intended for communication with other software that uses CMPL for modelling linear optimisation problems and can be obtained by the command line argument `-e [<file>]`.

It is also internally used for receiving CMPL messages from a CMPLServer.

An CmplMessages file consists of two major sections. The `<general>` section describes the general status and the name of the model and a general message after the transformation. The `<messages>` section consists of one or more messages about specific lines in the CMPL model.

The XSD is defined as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

  <xs:element name="CmplMessages">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="general" minOccurs="1" maxOccurs="1"/>
        <xs:element ref="messages" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="version" use="required" type="xs:decimal"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="general">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="generalStatus"  type="xs:string" minOccurs="1" maxOccurs="1"/>
        <xs:element name="instanceName" type="xs:string" minOccurs="1" maxOccurs="1"/>
        <xs:element name="message" type="xs:string" minOccurs="0" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

```
  <xs:element name="messages">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="message" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="numberOfMessages" use="required" type="xs:nonNegativeInteger"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="message">
        <xs:complexType>
                <xs:attribute name="type" type="msgType" use="required"/>
                <xs:attribute name="file" type="xs:string" use="required"/>
                <xs:attribute name="line" type="xs:nonNegativeInteger" use="required"/>
                <xs:attribute name="description" type="xs:string" use="required"/>
        </xs:complexType>
  </xs:element>

  <xs:simpleType name="msgType">
        <xs:restriction base="xs:string">
                <xs:enumeration value="error"/>
                <xs:enumeration value="warning"/>
        </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

After excecuting the given CMPL model, CMPL will finish without errors. The general status is represented in the following CmplMesages file `test.cmsg`.

```
<?xml version="1.0" encoding="UTF-8"?>
<CmplMessages version="1.1">
    <general>
      <generalStatus>normal</generalStatus>
      <instanceName>test.cmpl</instanceName>
      <message>cmpl finished normal</message>
    </general>
</CmplMessages>
```

If a wrong symbol name for the matrix `A[,]` (e.g. `a[,]`) is used in line 11, CMPL would be finish with errors represented in CmplMesages file `test.cmsg`.

```
<?xml version="1.0" encoding="UTF-8"?>
<CmplMessages version="1.1">
    <general>
      <generalStatus>error</generalStatus>
      <instanceName>test.cmpl</instanceName>
      <message>cmpl finished with errors</message>
    </general>
```

```
    <messages numberOfMessages="1">
      <message type ="error" file="test.cmpl" line="11"
           description="syntax error, unexpected SYMBOL_UNDEF"/>
    </messages>
</CmplMessages>
```

3.7.8 CmplInfo

CmplInfo is a simple XML file that contains as shown in the XSD below (if requested) several statistics and the generated matrix of the CMPL model.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

  <xs:element name="CmplInfo">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="general" minOccurs="1" maxOccurs="1"/>
        <xs:element ref="statistics" minOccurs="0" maxOccurs="1"/>
        <xs:element ref="matrix" minOccurs="0" maxOccurs="1"/>
      </xs:sequence>
      <xs:attribute name="version" use="required" type="xs:decimal"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="general">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="instancename"  type="xs:string" minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="statistics">
    <xs:complexType>
           <xs:simpleContent>
              <xs:extension base="xs:string">
                     <xs:attribute name="file" use="required" type="xs:string"/>
              </xs:extension>
           </xs:simpleContent>
    </xs:complexType>
  </xs:element>

  <xs:element name="matrix">
    <xs:complexType>
           <xs:simpleContent>
              <xs:extension base="xs:string">
                     <xs:attribute name="file" use="required" type="xs:string"/>
              </xs:extension>
```

```
            </xs:simpleContent>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

For the example the CmplInstance file `test.cinfo` is created automatically by CMPL by using the command line arguments `-matrix "test.mat" -s "test.stat"` and the model is executed remotely on a CMPLServer.

```
<?xml version="1.0" encoding="UTF-8"?>
<CmplInfo version="1.0">
     <general>
        <instancename>test.cmpl</instancename>
     <general>
     <statistics file="test.stat">


File: /Users/mike/CmplServer/10.0.1.2-2014-01-05-17-05-23-496795/test.cmpl
3 Columns (variables), 3 Rows (constraints + objective function)
6 (100%) of 6 coefficients of the constraints are non-zero.


     </statistics>
     <matrix file="test.mat">
Variable name                      x[1]           x[2]           x[3]
Variable type                         C              C              C

profit              max               15             18             22
   Subject to                                                                RHS
res[1]              L                  5             10             15       175
res[2]              L                 10              5             10       200

Lower Bound                            0              0              0
Upper Bound

     </matrix>
</CmplInfo>
```

4 CMPL's APIs

CMPL provides two APIs: pyCMPL for Python and jCMPL for Java.

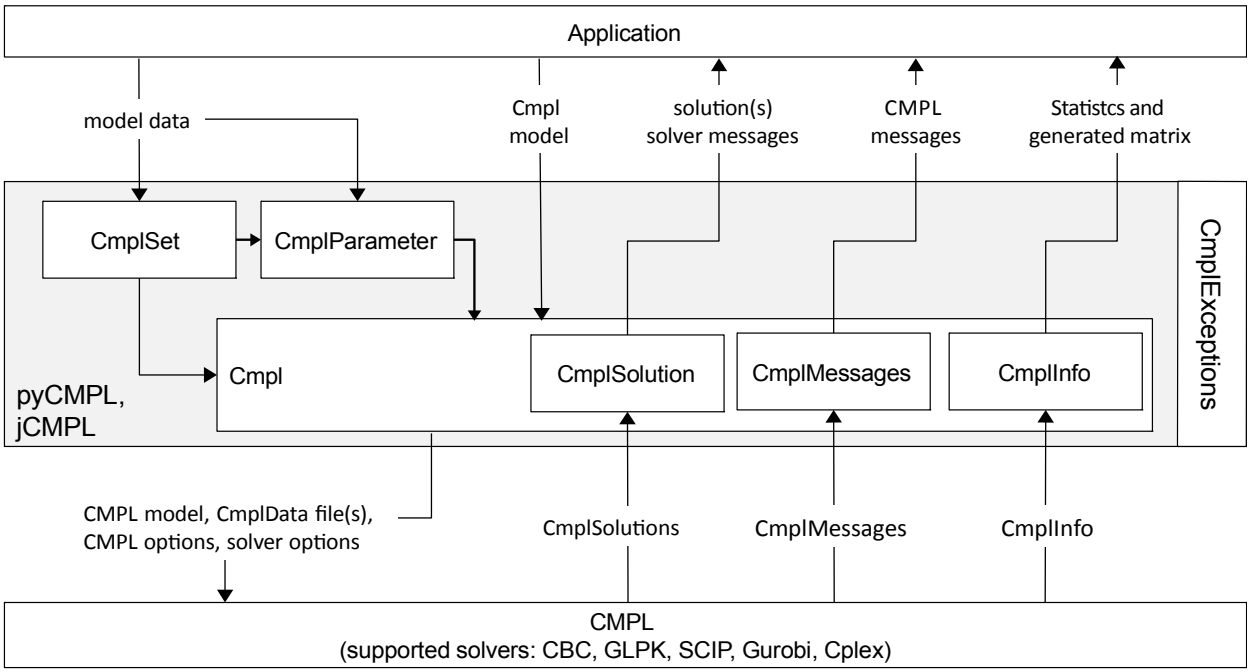
The main idea of this APIs is to define sets and parameters within the user application, to start and control the solving process and to read the solution(s) into the application if the problem is feasible. All variables, objective functions and constraints are defined in CMPL. These functionalities can be used with a local CMPL installation or a CMPLServer.

The structure and the classes including the methods and attributes are mostly identical or very similar in both APIs. The main difference are the attributes of a class that can be obtained in pyCmpl by r/o attributes and in jCMPL by getter methods.

4.1 Creating Python and Java applications with a local CMPL installation

pyCMPL and jCMPL contain a couple of classes to connect a Python or Java application with CMPL as shown in the figure below.

The classes `CmplSet` and `CmplParameter` are intended to define sets and parameters that can be used with several `Cmpl` objects. With the `Cmpl` class it is possible to define a CMPL model, to commit sets and parameters to this model, to start and control the solving process and to read the CMPL and solver messages and to have access to the solution(s) via `CmplMessages` and `CmplSolutions` objects.



To illustrate the formulation of a pyCmpl script and the corresponding java programme an example taken from (Hillier/Liebermann 2010, p. 334f.) is used. Consider a simple assignment problem that deals with the assignment of three machines to four possible locations. There is no work flow between the machines. The total material handling costs are to be minimised. The hourly material handling costs per machine and location are given in the following table.

		Locations			
		1	2	3	4
Machines	1	13	16	12	11
	2	15	-	13	20
	3	5	7	10	6

The mathematical model

$$\sum_{(i,j)\in A} c_{ij} \cdot x_{ij} \quad \rightarrow \min!$$

$$s.t.$$

$$\sum_{\substack{(k,j)\in A \\ k=i}} x_{kj} = 1 \quad ; i = 1(1)m$$

$$\sum_{\substack{(i,l)\in A \\ l=j}} x_{il} \leq 1 \quad ; j = 1(1)n$$

$$x_{ij} \in \{0,1\} \quad ; (i,j) \in A$$

with

Parameters

A	- set of the possible combination of machines and locations
m	- number of machines
n	- number of locations
c_{ij}	- hourly material handling costs of machine i at location j

Variables

x_{ij}	- assignment variable of machine i at location j

can be formulated in CMPL as follows:

```
%data : machines set, locations set, A set[2], c[A]

variables:
  x[A]: binary;

objectives:
  costs: sum{ [i,j] in A : c[i,j]*x[i,j] } -> min ;

constraints:
  sos_m { i in machines:  sum{ j in (A *> [i,*]) : x[i,j] } = 1; }
  sos_l { j in locations: sum{ i in (A *> [*,j]) : x[i,j] } <= 1; }
```

The interface for the sets and parameters provided by a pyCmpl script or jCMPL programme is the CMPL header entry `%data`.

4.1.1 pyCMPL

The first step to formulate this problem as a pyCmpl script after importing the pyCmpl package is to create a `Cmpl` object where the argument of the constructor is the name of the CMPL file.

```
#!/usr/bin/python
from pyCmpl import *

m = Cmpl("assignment.cmpl")
```

As in the `%data` entry two 1-tuple sets `machines` and `locations` and one 2-tuple set `A` are necessary for the CMPL model. To create a `CmplSet` a name and for *n*-tuple sets with *n*>1 the rank are needed as argu-

ments for the constructor. The name has to be identical to the corresponding name in the CMPL header entry `%data`. The set data is specified by the `CmplSet` method `setValues`. This is an overloaded method with different arguments for several types of sets.

```
locations = CmplSet("locations")
locations.setValues(1,4)

machines = CmplSet("machines")
machines.setValues(1,3)

combinations = CmplSet("A", 2)
combinations.setValues([  [1,1],[1,2],[1,3],[1,4], [2,1],[2,3],[2,4],\
                          [3,1],[3,2],[3,3],[3,4]])
```

As shown in the listing above the set `locations` is assigned `(1,2,..,4)` and the set `machines` consists of `(1,2,3)` because the first argument of `setValues` for this kind of sets is the starting value and the second argument is the end value while the increment is by default equal to one. The values of the 2-tuple set `combinations` are defined in the form of a list that consists of lists of valid combinations of machines and locations.

For the definition of a CMPL parameter a user has to create a `CmplParameter` object where the first argument of the constructor is the name of the parameter. If the parameter is an array it is also necessary to specify the set or sets through which the parameter array is defined. Therefore it is necessary to commit the `CmplSet combinations` (beside the name `"c"`) to create the CmplParameter array `c` .

```
c = CmplParameter("c",combinations)
c.setValues([13,16,12,11,15,13,20,5,7,10,6])
```

`CmplSet` objects and `CmplParameter` objects can be used in several CMPL models and have to be committed to a `Cmpl` model by the `Cmpl` methods `setSets` and `setParameters`. After this step the problem can be solved by using the `Cmpl` method `solve`.

```
m.setSets(machines,locations,combinations)
m.setParameters(c)

m.solve()
```

After solving the model the status of CMPL and the invoked solver can be analysed through the `Cmpl` attributes `solution.solverStatus` and `solution.cmplStatus`.

```
print "Objective value: " , m.solution.value
print "Objective status: " , m.solution.status
```

If the problem is feasible and a solution is found it is possible to read the names, the types, the activities, the lower and upper bounds and the marginal values of the variables and the constraints into the Python application. The `Cmpl` attributes `solution.variables` and `solution.constraints` contain a list of variable and constraint objects.

```
print "Variables:"
for v in m.solution.variables:
  print v.name, v.type, v.activity,v.lowerBound,v.upperBound

print "Constraints:"
for c in m.solution.constraints:
  print c.name, c.type, c.activity,c.lowerBound,c.upperBound
```

pyCmpl provides its own exception handling through the class `CmplException` that can be used in a `try` and `except` block.

```
try:
  ...
except CmplException, e:
  print e.msg
```

The entire pyCmpl script assignment.py shows as follows:

```
#!/usr/bin/python
from pyCmpl import *

try:
  m = Cmpl("assignment.cmpl")

  locations = CmplSet("locations")
  locations.setValues(1,4)

  machines = CmplSet("machines")
  machines.setValues(1,3)

  combinations = CmplSet("A", 2)
  combinations.setValues([  [1,1],[1,2],[1,3],[1,4],\
                            [2,1],[2,3],[2,4],[3,1],[3,2],[3,3],[3,4]])

  c = CmplParameter("c",combinations)
  c.setValues([13,16,12,11,15,13,20,5,7,10,6])

  m.setSets(machines,locations,combinations)
  m.setParameters(c)

  m.solve()

  print "Objective value: " , m.solution.value
  print "Objective status: " , m.solution.status

  print "Variables:"
  for v in m.solution.variables:
    print v.name, v.type, v.activity, v.lowerBound, v.upperBound
```

```
  print "Constraints:"
  for c in m.solution.constraints:
    print c.name, c.type, c.activity, c.lowerBound, c.upperBound

except CmplException, e:
  print e.msg
```

and can be executed by typing the command

```
pyCmpl assignment.py
```

under Linux and Mac in the terminal or under Windows in the CmplShell and prints the following solution to stdOut.

```
Objective value:  29.0
Objective status:  optimal

Variables:
x[1,1] B 0.0 0.0 1.0
x[1,2] B 0.0 0.0 1.0
x[1,3] B 0.0 0.0 1.0
x[1,4] B 1.0 0.0 1.0
x[2,1] B 0.0 0.0 1.0
x[2,3] B 1.0 0.0 1.0
x[2,4] B 0.0 0.0 1.0
x[3,1] B 1.0 0.0 1.0
x[3,2] B 0.0 0.0 1.0
x[3,3] B 0.0 0.0 1.0
x[3,4] B 0.0 0.0 1.0
Constraints:
sos_m[1] E 1.0 1.0 1.0
sos_m[2] E 1.0 1.0 1.0
sos_m[3] E 1.0 1.0 1.0
sos_l[1] L 1.0 -inf 1.0
sos_l[2] L 0.0 -inf 1.0
sos_l[3] L 1.0 -inf 1.0
sos_l[4] L 1.0 -inf 1.0
```

4.1.2 jCMPL

To use the jCMPL functionalities a Java programme has to import jCMPL by `import jCMPL.*;` and to link your application against `jCmpl.jar` and the following jar files, that you can find in the CMPL applicatiopn folder in `jCmpl/Libs` : `commons-lang3`, `ws-commons-util`, `xmplrpc-client`, `xmlrpc-commons`.

The first step to formulate this problem as a jCmpl programme after importing the jCmpl package is to create a `Cmpl` object where the argument of the constructor is the name of the CMPL file. Since jCMPL provides it own exception handling the main method has to throw `CmplExeptions`.

```
import jCMPL.*;

public class Assignment {
    public static void main(String[] args) throws CmplException {
        try {
            Cmpl m = new Cmpl("assignment.cmpl");
```

As in pyCMPL to create a `CmplSet` a name and for *n*-tuple sets with *n*>1 the rank are needed as arguments for the constructor whereby the name has to be identical to the corresponding name in the CMPL header entry `%data`. The set data is specified by the `CmplSet.setValues()`. This is an overloaded method with different arguments for several types of sets.

```
CmplSet locations = new CmplSet("locations");
locations.setValues(1, 4);

CmplSet machines = new CmplSet("machines");
machines.setValues(1, 3);

CmplSet combinations = new CmplSet("A", 2);
int[][] combiVals = {{1, 1}, {1, 2}, {1, 3}, {1, 4}, {2, 1},
                     {2, 3}, {2, 4}, {3, 1}, {3, 2}, {3, 3}, {3, 4}};
combinations.setValues(combiVals);
```

In the listing above the set `locations` is assigned `(1,2,..,4)` and the set `machines` consists of `(1,2,3)`. The first argument of `setValues` for this algorithmic sets is the starting value and the second argument is the end value while the increment is by default equal to one. The values of the 2-tuple set `combinations` are defined in the form of a matrix of integers that consists all valid combinations of machines and locations.

To create a CMPL parameter a user has to define a `CmplParameter` object whereby the first argument of the constructor is the name of the parameter. For parameter arrays it is also necessary to specify the set or sets through which the parameter array is defined. Therefore it is necessary to commit the `CmplSet combinations` (beside the name `"c"`) to create the CmplParameter array `c`.

```
CmplParameter costs = new CmplParameter("c", combinations);
int[] costVals = {13, 16, 12, 11, 15, 13, 20, 5, 7, 10, 6};
costs.setValues(costVals);
```

In the next step the sets and parameters have to be committed to a `Cmpl` model by the `Cmpl` methods `setSets` and `setParameters` and the problem can be solved by using the `Cmpl` method `solve`.

```
m.setSets(machines, locations, combinations);
m.setParameters(costs);

m.solve();
```

After solving the model the status of CMPL and the invoked solver can be analysed through the methods `Cmpl.solution().solverStatus()` and `Cmpl.solution().cmplStatus()`.

```
System.out.printf("Objective value:  %f %n", m.solution().value());
System.out.printf("Objective status: %s %n",  m.solution().status());
```

If the problem is feasible and a solution is found it is possible to read the names, the types, the activities, the lower and upper bounds and the marginal values of the variables and the constraints into the Python application. The methods `Cmpl.solution().variables()` and `Cmpl.solution().constraints()` return a list of variable and constraint objects.

```
System.out.println("Variables:");
for (CmplSolElement v :  m.solution().variables()) {
  System.out.printf("%10s %3s %10d %10.0f %10.0f%n", v.name(), v.type(),
         v.activity(), v.lowerBound(), v.upperBound());
}
System.out.println("Constraints:");
for (CmplSolElement c :  m.solution().constraints()) {
  System.out.printf("%10s %3s %10.0f %10.0f %10.0f%n", c.name(), c.type(),
       c.activity(), c.lowerBound(), c.upperBound());
}
```

The entire jCmpl programme `assignment.java` shows as follows:

```
import jCMPL.*;

public class Assignment1 {
  public static void main(String[] args) throws CmplException {
    try {
      Cmpl m = new Cmpl("assignment.cmpl");

      CmplSet locations = new CmplSet("locations");
      locations.setValues(1, 4);

      CmplSet machines = new CmplSet("machines");
      machines.setValues(1, 3);

      CmplSet combinations = new CmplSet("A", 2);
      int[][] combiVals = {{1, 1}, {1, 2}, {1, 3}, {1, 4},{2, 1}, {2, 3},
                           {2, 4},{3, 1}, {3, 2}, {3, 3}, {3, 4}};
      combinations.setValues(combiVals);

      CmplParameter costs = new CmplParameter("c", combinations);
      int[] costVals = {13, 16, 12, 11, 15, 13, 20, 5, 7, 10, 6};
      costs.setValues(costVals);

      m.setSets(machines, locations, combinations);
      m.setParameters(costs);

      m.solve();
```

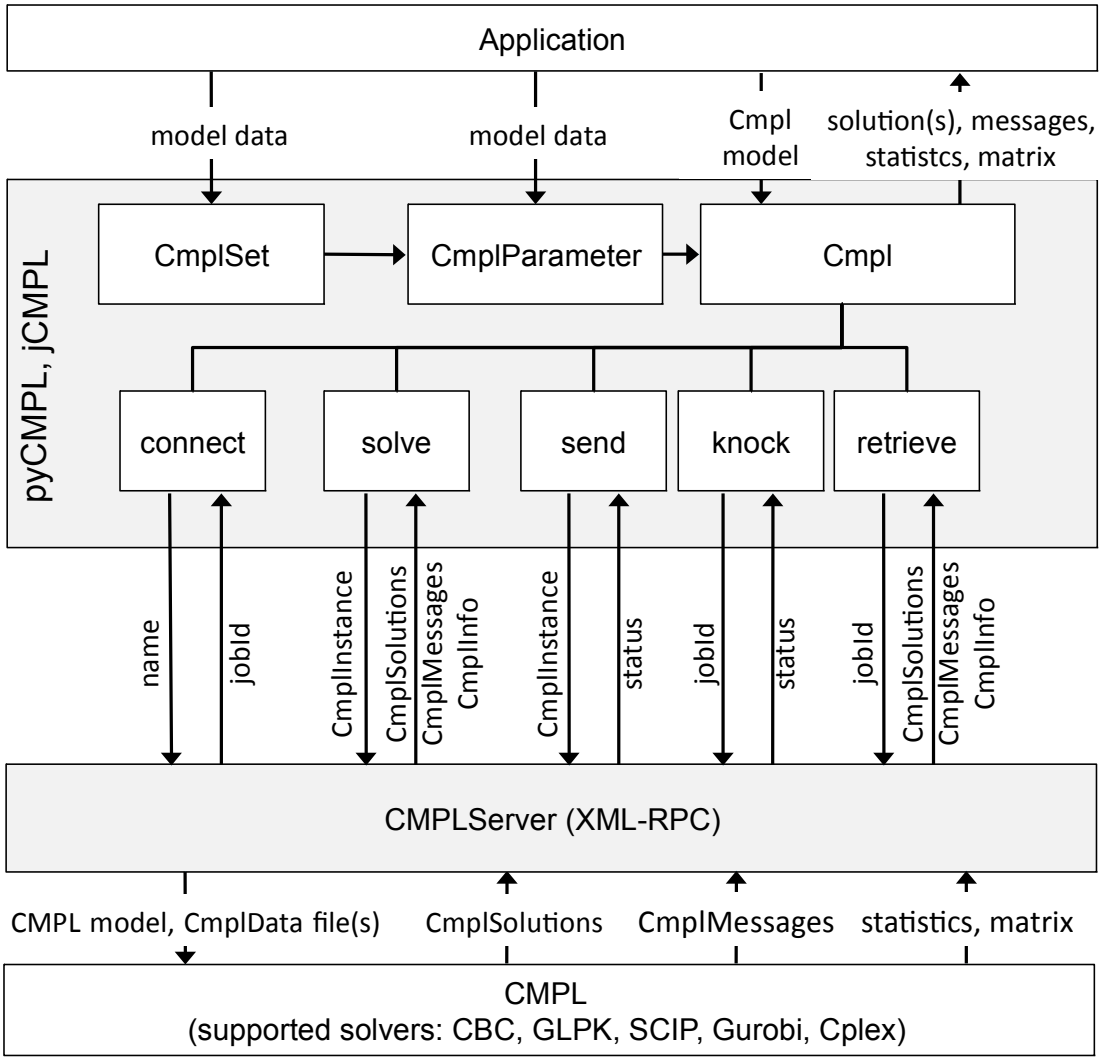
```
        System.out.printf("Objective value:  %f %n", m.solution().value());
        System.out.printf("Objective status: %s %n",  m.solution().status());

        System.out.println("Variables:");
        for (CmplSolElement v :  m.solution().variables()) {
                System.out.printf("%10s %3s %10d %10.0f %10.0f%n", v.name(),
                     v.type(), v.activity(), v.lowerBound(), v.upperBound());
        }
        System.out.println("Constraints:");
        for (CmplSolElement c :  m.solution().constraints()) {
          System.out.printf("%10s %3s %10.0f %10.0f %10.0f%n", c.name(),
                     c.type(), c.activity(), c.lowerBound(), c.upperBound());
        }
      } catch (CmplException e) {
        System.out.println(e);
      }
    }
}
```

and prints after starting the following solution to stdOut.

```
Objective value:  29.000000
Objective status: optimal
Variables:
    x[1,1]   B          0          0          1
    x[1,2]   B          0          0          1
    x[1,3]   B          0          0          1
    x[1,4]   B          1          0          1
    x[2,1]   B          0          0          1
    x[2,3]   B          1          0          1
    x[2,4]   B          0          0          1
    x[3,1]   B          1          0          1
    x[3,2]   B          0          0          1
    x[3,3]   B          0          0          1
    x[3,4]   B          0          0          1
Constraints:
  sos_m[1]   E          1          1          1
  sos_m[2]   E          1          1          1
  sos_m[3]   E          1          1          1
  sos_l[1]   L          1  -Infinity          1
  sos_l[2]   L          0  -Infinity          1
  sos_l[3]   L          1  -Infinity          1
  sos_l[4]   L          1  -Infinity          1
```

4.2 Creating Python and Java applications using CMPLServer

The class `Cmpl` also provides the functionality to communicate with a CMPLServer or a CMPLGridScheduler whereas it doesn't matter for the client whether it is connected to a single CMPLServer or to a CMPLGrid. As shown in the figure below the first step to communicate with a CMPLServer is the `Cmpl.connect` method that returns (if connected) a `jobId`. After connecting, a problem can be solved synchronously or asynchronously.



The `Cmpl` method `solve` sends a `CmplInstance` string to the connected CMPLServer and waits for the returning `CmplSolutions`, `CmplMessages` and if requested `CmplInfo` XML strings. After this synchronous process a user can access the solution(s) if the problem is feasible or if not it can be analysed, whether the CMPL formulations or the solver is the cause of the problem. To execute the solving process asynchronously the `Cmpl` methods `send`, `knock` and `retrieve` have to be used. `Cmpl.send` sends a `CmplInstance` string to the CMPLServer and starts the solving process remotely. `Cmpl.knock` asks for a CMPL model with a given `jobId` whether the solving process is finished or not. If the problem is finished the `CmplSolutions` and the `CmplMessages` strings can be read into the user application with `Cmpl.retrieve`.

4.2.1 pyCMPL

The first step to create a distributed optimisation application is to start the CMPLServer. Assuming that a CMPLServer is running on 194.95.45.70:8008 the assignment problem can be solved remotely only by including

```
m.connect("http://194.95.45.70:8008")
```

in the source code before `Cmpl.solve` is executed.

The pyCmpl script `assignment-remote.py` shows as follows:

```
#!/usr/bin/python
from pyCmpl import *

try:
  m = Cmpl("assignment.cmpl")

  locations = CmplSet("locations")
  locations.setValues(1,4)

  machines = CmplSet("machines")
  machines.setValues(1,3)

  combinations = CmplSet("A", 2)
  combinations.setValues([  [1,1],[1,2],[1,3],[1,4],\
                            [2,1],[2,3],[2,4],[3,1],[3,2],[3,3],[3,4]])

  c = CmplParameter("c",combinations)
  c.setValues([13,16,12,11,15,13,20,5,7,10,6])

  m.setSets(machines,locations,combinations)
  m.setParameters(c)

  m.connect("http://194.95.45.70:8008")
  m.solve()

  print "Objective value: " , m.solution.value
  print "Objective status: " , m.solution.status

  print "Variables:"
  for v in m.solution.variables:
    print v.name, v.type, v.activity,v.lowerBound,v.upperBound

  print "Constraints:"
  for c in m.solution.constraints:
    print c.name, c.type, c.activity,c.lowerBound,c.upperBound

except CmplException, e:
  print e.msg
```

4.2.2 jCMPL

The jCMPL programme `assignment-remote.java` shows as follows:

```
import jCMPL.*;

public class Assignment1 {
  public static void main(String[] args) throws CmplException {
    try {
      Cmpl m = new Cmpl("assignment.cmpl");

      CmplSet locations = new CmplSet("locations");
      locations.setValues(1, 4);

      CmplSet machines = new CmplSet("machines");
      machines.setValues(1, 3);

      CmplSet combinations = new CmplSet("A", 2);
      int[][] combiVals = {{1, 1}, {1, 2}, {1, 3}, {1, 4},{2, 1}, {2, 3},
                           {2, 4},{3, 1}, {3, 2}, {3, 3}, {3, 4}};
      combinations.setValues(combiVals);

      CmplParameter costs = new CmplParameter("c", combinations);
      int[] costVals = {13, 16, 12, 11, 15, 13, 20, 5, 7, 10, 6};
      costs.setValues(costVals);

      m.setSets(machines, locations, combinations);
      m.setParameters(costs);

      m.connect("http://194.95.45.70:8008");
      m.solve();

      System.out.printf("Objective value:  %f %n", m.solution().value());
      System.out.printf("Objective status: %s %n",  m.solution().status());


      System.out.println("Variables:");
      for (CmplSolElement v :  m.solution().variables()) {
               System.out.printf("%10s %3s %10d %10.0f %10.0f%n", v.name(),
                    v.type(), v.activity(), v.lowerBound(), v.upperBound());
      }
      System.out.println("Constraints:");
      for (CmplSolElement c :  m.solution().constraints()) {
        System.out.printf("%10s %3s %10.0f %10.0f %10.0f%n", c.name(),
                    c.type(), c.activity(), c.lowerBound(), c.upperBound());
      }
    } catch (CmplException e) {
```

```
        System.out.println(e);
      }
    }
}
```

4.3 pyCMPL reference manual

4.3.1 CmplSets

The class `CmplSet` is intended to define sets that can be used with several `Cmpl` objects.

Methods:

`CmplSet(`*`setName[,rank]`***`)`**

Description: Constructor

Parameter:
- `str` *`setName`* — name of the set, Has to be equal to the corresponding name in the CMPL model.
- `int` *`rank`* — optional - rank *n* for a *n*-tuple set (default 1)

Return: `CmplSet` object

`CmplSet``.`**`setValues(`***`setList`***`)`**

Description: Defines the values of an enumeration set

Parameter:
- `list` *`setList`* — for a set of *n*-tuples with *n*=1 - `list` of single indexing entries `int|long|str`
  for a set of *n*-tuples with *n*>1 - `list` of `list`(s) that contain `int|long|str` tuples

Return: -

`CmplSet``.`**`setValues(`***`startNumber,endNumber`***`)`**

Description: Defines the values of an algorithmic set
`(startNumber, startNumber+1, ..., endNumber)`

Parameter:
- `int` *`startNumber`* — start value of the set
- `int` *`endNumber`* — end value of the set

Return: -

`CmplSet``.`**`setValues(`***`startNumber,step,endNumber`***`)`**

Description: Defines the values of an algorithmic set
`(startNumber, startNumber+step, ..., endNumber)`

Parameter:
- `int` *`startNumber`* — start value of the set
- `int` *`step`* — positive value for increment
  negative value for decrement

	`Int` *endNumber*	end value of the set
Return:	-	

R/o attributes:

CmplSet.**values**

Description:	List of the indexing entries of the set	
Return:	`list`	of single indexing entries - for a set of *n*-tuples with *n*=1 of `tuple`(s) - for a set of *n*-tuples with *n*>1

CmplSet.**name**

Description:	Name of the set	
Return:	`str`	name of the CMPL set (not the name of the `CmplSet` object)

CmplSet.**rank**

Description:	Rank of the set	
Return:	`int`	number of *n* of a *n*-tuple set

CmplSet.**len**

Description:	Length of the set	
Return:	`int`	number of indexing entries

Examples:

`s = CmplSet("s")`
`s.setValues(0,4)`
 	
`print s.rank`
`print s.len`
`print s.name`
`print s.values`
	`s` is assigned $s \in (0,1,\ldots,4)$
	
	`1`
	`4`
	`s`
	`[0, 1, 2, 3, 4]`

`s = CmplSet("a")`
`s.setValues(10,-2,0)`
 	
`print s.rank`
`print s.len`
`print s.name`
`print s.values`
	`s` is assigned $s \in (10,8,\ldots,0)$
	
	`1`
	`6`
	`s`
	`[10, 8, 6, 4, 2, 0]`

`s = CmplSet("FOOD")`
`s.setValues(["BEEF","CHK","FISH"])`
 	
`print s.rank`
`print s.len`
	`s` is assigned $s \in ('BEEF','CHK','FISH')$
	
	`1`
	`3`

`print s.name` `print s.values`	`FOOD` `['BEEF', 'CHK', 'FISH']`

`s = CmplSet("c",3)`
`s.setValues([ [1,1,1], [1,1,2], \`
`  [1,2,1] ])`
 	
`print s.rank`
`print s.len`
`print s.name`
`print s.values`
	`s` is assigned a 3-tuple set of integers
	
	`3`
	`3`
	`c`
	`[(1, 1, 1), (1, 1, 2), (1, 2, 1)]`

4.3.2 CmplParameters

The class `CmplParameters` is intended to define parameters that can be used with several `Cmpl` objects.

Methods:

`CmplParameter(`*`paramName`*` [,`***`set1`***`,`***`set2`***`,...])`**

Description: Constructor

Parameter:

`str` *`paramName`*	name of the parameter Has to be equal to the corresponding name in the CMPL model.
`CmplSet` *`set1,set2,...`*	optional - set or sets through which the parameter array is defined (default `None`)

Return: `CmplParameter` object

`CmplParameter``.`**`setValues(`***`val`***`)`**

Description: Defines the values of a scalar parameter

Parameter:

`int|long|float|str` *`val`*	value of the scalar parameter

Return: -

`CmplParameter``.`**`setValues(`***`valList`***`)`**

Description: Defines the values of a parameter array

Parameter:

`list` *`valList`*	`list` of `int|long|float|str|list` - value list of the parameter array

Return: -

R/o attributes:

CmplParameter.**values**

Description: List of the values of a parameter

Return: `list` of `int|long|float|str|list` - value list of the parameter array

CmplParameter.**value**

Description: Value of a scalar parameter

Return: `int|long|float|str` - value of the scalar parameter

CmplParameter.**setList**

Description: List of sets through which the parameter array is defined

Return: `list` of `CmplSet` objects through which the parameter array is defined

CmplParameter.**name**

Description: Name of the parameter

Return: `str` - name of the CMPL parameter (not the name of the `CmplParameter` object)

CmplParameter.**rank**

Description: Rank of the parameter

Return: `int` - rank of the CMPL parameter

CmplParameter.**len**

Description: Length of the parameter array

Return: `long` number of elements in the parameter array

Examples:

`p = CmplParameter("p")` `p.setValues(2)`	`p` is assigned `2`

`print p.values`
`print p.value`
`print p.name`
`print p.rank`
`print p.len`
	`[2]`
	`2`
	`p`
	`0`
	`1`

`s = CmplSet("s")`
`s.setValues(0,4)`
`p = CmplParameter("p",s)`
`p.setValues([1,2,3,4,5])`
	`p` is assigned $(1,2,\ldots,5)$

`print p.values` `print p.name`	`[1, 2, 3, 4, 5]` `p`

`print p.rank` `print p.len`	`1` `5`

`products = CmplSet("products")`
`products.setValues(1,3)`
 	
`machines = CmplSet("machines")`
`machines.setValues(1,2)`
 	
`a=CmplParameter("a",machines, products)`
`a.setValues([[ 8,15,12],[15,10,8]])`
 	
`print a.values`
`print a.name`
`print a.rank`
`print a.len`
 	
`for e in a.setList:`
`  print e.values`
	`a` is assigned a `2X3` matrix of integers
	
	`[[8, 15, 12], [15, 10, 8]]`
	`a`
	`2`
	`6`
	
	`[1, 2]`
	`[1, 2, 3]`

`s = CmplSet("s",2)`
`s.setValues([[1,1],[2,2]])`
 	
`p = CmplParameter("p",s)`
`p.setValues([1,1])`
 	
`print p.values`
`print p.name`
`print p.rank`
`print p.len`
	`s` is assigned the indices of a matrix diagonal
	
	`s` is assigned a `2x2` identity matrix
	
	`[1, 1]`
	`p`
	`2`
	`2`

4.3.3 Cmpl

With the `Cmpl` class it is possible to define a CMPL model, to commit sets and parameters to this model, to start and control the solving process and to read the CMPL and solver messages and to have access to the solution(s) via `CmplMessages` and `CmplSolutions` objects.

4.3.3.1 Establishing models

Methods:

`Cmpl(`*`name`***`)`**

Description: Constructor
Parameter: `str` *`name`* filename of the CMPL model
Return: `Cmpl` object

Cmpl.**setSets(***set1[,set2,...]***)**

Description: Committing `CmplSet` objects to the `Cmpl` model

Parameter: `CmplSet` *set1[,set2,...]* `CmplSet` object(s)

Return: -

Cmpl.**setParameters(***par1[,par2,...]***)**

Description: Committing `CmplParameter` objects to the `Cmpl` model

Parameter: `CmplParameter` *par1[,par2,...]* `CmplParameter` object(s)

Return: -

Examples:

`m = Cmpl("prodmix.cmpl")`
 	
`products = CmplSet("products")`
`products.setValues(1,3)`
 	
`machines = CmplSet("machines")`
`machines.setValues(1,2)`
 	
`c = CmplParameter("c",products)`
`c.setValues([75,80,50])`
 	
`b = CmplParameter("b",machines)`
`b.setValues([1000,1000])`
 	
`a = CmplParameter("a",machines, products)`
`a.setValues([[ 8,15,12],[15,10,8]])`	

`m.setSets(products,machines)`	Commits the sets `products, machines` to the Cmpl object `m`
`m.setParameters(c,a,b)`	Commits the parameter `c,a,b` to the Cmpl object m

4.3.3.2 Manipulating models

Methods:

Cmpl.**setOption(***option***)**

Description: Sets a CMPL, display or solver option

Parameter: `str` *option* option in CmplHeader syntax

Return: `int` option id

Cmpl.**delOption(***optId***)**

Description: Deletes an option

Parameter: `int` *optId* option id

Return: -

Cmpl.**delOptions()**

Description: Deletes all options

Parameter: -

Return: -

Cmpl.**setOutput(***ok[,leadString]]***)**

Description: Turns the output of CMPL and the invoked solver on or off

Parameter: `bool` *ok* `True|False`

`str` *leadString* optional - Leading string for the output (default - model name)

Return: -

Cmpl.**setRefreshTime(***rTime***)**

Description: Refresh time for getting the output of CMPL and the invoked solver from a CMPLServer if the model is solved synchronously.

Parameter: `float` *rTime* refresh time in seconds (default 0.5)

Return: -

R/o attributes:

Cmpl.**refreshTime**

Description: Returns the refresh time for getting the output of CMPL and the invoked solver from a CMPLServer if the model is solved synchronously.

Return: `float` Refresh time

Examples:

`m = Cmpl("assignment.cmpl")`
 	
`c1=m.setOption("%display nonZeros")`
`m.setOption("%arg -solver cplex")`
`m.setOption("%display solutionPool")`
 	
`m.delOption(c1)`
`m.delOptions()`
	
	Setting some options
	
	Deletes the first option
	Deletes all options

`m = Cmpl("assignment.cmpl")`
 	
`m.setOutput(True)`
 	
`m.setOutput(True,"my special model")`
	
	The stdOut and stdErr of CMPL and the invoked solver are shown for the `Cmpl` object `m`.
	
	As above but the output starts with the leading string `"my special model>"`.

`m = Cmpl("assignment.cmpl")`
`m.connect("http://194.95.45.70:8008")`
`m.setOutput(True)`
`m.setRefreshTime(1)`
	The stdOut and stdErr of CMPL and the invoked solver located at the specified CMPLServer will be refreshed every second.

4.3.3.3 Solving models

Methods:

Cmpl.**solve()**

Description: Solves a `Cmpl` model either with a local installed CMPL or if the model is connected with a CMPLServer remotely.

Parameter: -

Return: - status of the model and the solver can be obtained by the attributes `cmplStatus`, `cmplStatusText`,`solverStatus` and `solverStatusText`

Cmpl.**start()**

Description: Solves a `Cmpl` model in a separate thread either with a local installed CMPL or if the model is connected with a CMPLServer remotely.

Parameter: -

Return: - status of the model and the solver can be obtained by the attributes `cmplStatus`, `cmplStatusText`,`solverStatus` and `solverStatusText`

Cmpl.**join()**

Description: Waits until the solving thread terminates.

Parameter: -

Return: - status of the model and the solver can be obtained by the attributes `cmplStatus`, `cmplStatusText`,`solverStatus` and `solverStatusText`

Cmpl.**isAlive()**

Description: Return whether the thread is alive

Parameter: -

Return: `bool` `True` or `False` - return whether the thread is alive or not

Cmpl.**connect(***cmplUrl***)**

Description: Connects a CMPLServer or CMPLGridScheduler under *`cmplUrl`* – first step of solving a model on a CMPLServer remotely

Parameter: `str` *`cmplUrl`* URL of the CMPLServer or CMPLGridScheduler

Return: –

Cmpl.**disconnect()**

Description: Disconnects the connected CMPLServer or CMPLGridScheduler

Parameter: –

Return: –

Cmpl.**send()**

Description: Sends the `Cmpl` model instance to the connected CMPLServer - first step of solving a model on a CMPLServer asynchronously (after connect())

Parameter: –

Return: – status of the model can be obtained by the attributes `cmplStatus` and `cmplStatusText`

Cmpl.**knock()**

Description: Knocks on the door of the connected CMPLServer or CMPLGridScheduler and asks whether the model is finished - second step of solving a model on a CMPLServer asynchronously

Parameter: –

Return: – status of the model can be obtained by the attributes `cmplStatus` and `cmplStatusText`

Cmpl.**retrieve()**

Description: Retrieves the `Cmpl` solution(s) if possible from the connected CMPLServer - last step of solving a model on a CMPLServer asynchronously

Parameter: –

Return: – status of the model and the solver can be obtained by the attributes `cmplStatus`, `cmplStatusText`,`solverStatus` and `solverStatusText`

Cmpl.**cancel()**

Description: Cancels the `Cmpl` solving process on the connected CMPLServer

Parameter: –

Return: – status of the model can be obtained by the attributes `cmplStatus` and `cmplStatusText`

R/o attributes:

Cmpl.**cmplStatus**

Description: Returns the CMPL related status of the `Cmpl` object

Return: `int`

```
CMPL_UNKNOWN = 0
CMPL_OK = 1
CMPL_WARNINGS = 2
CMPL_FAILED = 3
CMPLSERVER_OK = 6
CMPLSERVER_ERROR = 7
CMPLSERVER_BUSY = 8
CMPLSERVER_CLEANED = 9
CMPLSERVER_WARNING = 10
PROBLEM_RUNNING = 11
PROBLEM_FINISHED = 12
PROBLEM_CANCELED = 13
PROBLEM_NOTRUNNING = 14
CMPLGRID_SCHEDULER_UNKNOWN = 15
CMPLGRID_SCHEDULER_OK = 16
CMPLGRID_SCHEDULER_ERROR = 17
CMPLGRID_SCHEDULER_BUSY = 18
CMPLGRID_SCHEDULER_SOLVER_NOT_AVAILABLE = 19
CMPLGRID_SCHEDULER_WARNING = 20
CMPLGRID_SCHEDULER_PROBLEM_DELETED = 21
```

Cmpl.**cmplStatusText**

Description: Returns the CMPL related status text of the `Cmpl` object

Return: `str`

```
CMPL_UNKNOWN
CMPL_OK
CMPL_WARNINGS
CMPL_FAILED
CMPLSERVER_OK
CMPLSERVER_ERROR
CMPLSERVER_BUSY
CMPLSERVER_CLEANED
CMPLSERVER_WARNING
PROBLEM_RUNNING
PROBLEM_FINISHED
PROBLEM_CANCELED
PROBLEM_NOTRUNNING
CMPLGRID_SCHEDULER_UNKNOWN
CMPLGRID_SCHEDULER_OK
CMPLGRID_SCHEDULER_ERROR
CMPLGRID_SCHEDULER_BUSY
CMPLGRID_SCHEDULER_SOLVER_NOT_AVAILABLE
CMPLGRID_SCHEDULER_WARNING
CMPLGRID_SCHEDULER_PROBLEM_DELETED
```

Cmpl.**solverStatus**

Description: Returns the solver related status of the `Cmpl` object

Return: `int`

```
SOLVER_OK = 4
SOLVER_FAILED = 5
```

Cmpl.**solverStatusText**

Description: Returns the solver related status text of the `Cmpl` object

Return: `str`

```
SOLVER_OK
SOLVER_FAILED
```

Cmpl.**jobId**

Description: Returns the jobId of the `Cmpl` problem at the connected CMPLServer

Return: `str` string of the jobId

Cmpl.**output**

Description: Returns the output of CMPL and the invoked solver.
Intended to use if an application needs to parse the output.

Return: `str` string of output of CMPL and the invoked solver

Examples:

`m = Cmpl("assignment.cmpl")` `m.solve()`	Solves the `Cmpl` object `m` locally

`m = Cmpl("assignment.cmpl")`
`m.connect("http://194.95.45.70:8008")`
`m.solve()`
	Solves the `Cmpl` object `m` remotely and synchronously on the specified CMPLServer

`m = Cmpl("assignment.cmpl")`
`m.connect("http://194.95.45.70:8008")`
`m.send()`
`m.knock()`
`m.retrieve()`
	Solves the `Cmpl` object `m` remotely and asynchronously on the specified CMPLServer

`models= []`
`models.append(Cmpl("m1.cmpl"))`
`models.append(Cmpl("m2.cmpl"))`
`models.append(Cmpl("m3.cmpl"))`
 	
`for m in models:`
`  m.start()`
 	
`for m in models:`
`  m.join()`
	Starts all models in separate threads.
	
	Waits until the all solving threads are terminated.

`m = Cmpl("assignment.cmpl")`
`m.solve()`
`if m.solverstatus!=SOLVER_OK:`
`  m.solutionReport()`
	Displays the optimal solution if the solver didn't fail.

4.3.3.4 Reading solutions

Methods:

Cmpl.**solutionReport()**

Description: Writes a standard solution report to stdOut

Parameter: -

Return: -

Cmpl.**saveSolution(***[solFileName]***)**

Description: Saves the solution(s) as `CmplSolutions` file

Parameter: `str` *solFileName* optional file name (default `<modelname>.csol`)

Return: -

Cmpl.**saveSolutionAscii(***[solFileName]***)**

Description: Saves the solution(s) as ASCII file

Parameter: `str` *solFileName* optional file name (default `<modelname>.sol`)

Return: -

Cmpl.**saveSolutionCsv(***[solFileName]***)**

Description: Saves the solution(s) as CSV file

Parameter: `str` *solFileName* optional file name (default `<modelname>.csv`)

Return: -

Cmpl.**varByName(***[solIdx]***)**

Description: Enables a direct access to variables by their name

Parameter: `int` *solIdx* optional solution index (default `0`)

Return: -

Cmpl.**conByName(***[solIdx]***)**

Description: Enables a direct access to constraints by their name

Parameter: `int` *solIdx* optional solution index (default `0`)

Return: -

Cmpl.**getVarByName(***name, [solIdx]***)**

Description: Returns a `CmplSolElement` object or a list of `CmplSolElement` objects for the variable or variable array with the specified name

Parameter: `str name` name of the variable or variable array
`int` *solIdx* optional solution index (default `0`)

Return: `CmplSolElement` | for a single variable
`list of CmplSolElement` for a variable array

Cmpl.**getConByName(***[solIdx]***)**

Description: Returns a `CmplSolElement` object or a list of `CmplSolElement` objects for the constraint or constraint array with the specified name

Parameter: `str name` name of the constraint or constraint array
`int` *solIdx* optional solution index (default `0`)

Return: `CmplSolElement` | for a single constraint
`list of CmplSolElement` for a constraint array

R/o attributes:

Cmpl.**nrOfVariables**

Description: Returns the number of variables of the generated and solved CMPL model
Return: `int` number of variables

Cmpl.**nrOfConstraints**

Description: Returns the number of constraints of the generated and solved CMPL model
Return: `int` number of constraints

Cmpl.**objectiveName**

Description: Returns the name of the objective function of the generated and solved CMPL model
Return: `str` objective name

Cmpl.**objectiveSense**

Description: Returns the objective sense of the generated and solved CMPL model
Return: `str` objective sense

Cmpl.**nrOfSolutions**

Description: Returns the number of solutions of the generated and solved CMPL model
Return: `int` number of solutions

Cmpl.**solver**

Description: Returns the name of the invoked solver of the generated and solved CMPL model
Return: `str` invoked solver

Cmpl.**solverMessage**

Description: Returns the message of the invoked solver of the generated and solved CMPL model
Return: `str` message of the invoked solver

Cmpl.**varDisplayOptions**

Description: Returns a string with the display options for the variables of the generated and solved CMPL model
Return: `str` display options for the variables

Cmpl.**conDisplayOptions**

Description: Returns a string with the display options for the constraints of the generated and solved CMPL model
Return: `str` display options for the constraints

Cmpl.**solution**

Description: Returns the first (optimal) `CmplSolutions` object

Return: `CmplSolutions` first (optimal) solution

Cmpl.**solutionPool**

Description: Returns a list of `CmplSolutions` objects

Return: `list` of `CmplSolutions` objects — list of `CmplSolution` object for solutions found

CmplSolutions.**status**

Description: Returns a string with the status of the current solution provided by the invoked solver

Return: `str` solution status

CmplSolutions.**value**

Description: Returns the value of the objective function of the current solution

Return: `float` objective function value

CmplSolutions.**idx**

Description: Returns the index of the current solution

Return: `int` index of the current solution

CmplSolutions.**variables**

Description: Returns a list of `CmplSolElement` objects for the variables of the current solution

Return: `list` of *`CmplSolLine`* objects — list of variables

CmplSolutions.**constraints**

Description: Returns a list of `CmplSolElement` objects for the constraints of the current solution

Return: `list` of `CmplSolElement` objects — list of constraints

`CmplSolElement`.**idx**

Description: Index of the variable or constraint

Return: `int` index of the variable or constraint

`CmplSolElement`.**name**

Description: Name of the variable or constraint

Return: `str` name of the variable or constraint

`CmplSolElement.`**`type`**

Description: Type of the variable or constraint

Return: *str* type of the variable or constraint
`C|I|B` for variables
`L|E|G` for constraints

`CmplSolElement.`**`activity`**

Description: Activity of the variable or constraint

Return: *long|float* activity of the variable or constraint

`CmplSolElement.`**`lowerBound`**

Description: Lower bound of the variable or constraint

Return: `float` lower bound of the variable or constraint

`CmplSolElement.`**`upperBound`**

Description: Upper bound of the variable or constraint

Return: `float` upper bound of the variable or constraint

`CmplSolElement.`**`marginal`**

Description: Marginal value (shadow prices or reduced costs) bound of the variable or constraint

Return: `float` marginal value of the variable or constraint

Examples:

```
m = Cmpl("assignment.cmpl")
...

m.solve()

print m.solver
print m.solverMessage
print m.nrOfVariables
print m.nrOfConstraints

print m.varDisplayOptions
print m.conDisplayOptions
print m.objectiveName
print m.objectiveSense
print m.solution.value
print m.solution.status
print m.nrOfSolutions
print m.solution.idx
```

Solves the example from subchapter 4.1 and displays some information about the generated and solved model

```
CBC

11
7
(all)
(all)
costs
min
29.0
optimal
1
0
```

```
for v in m.solution.variables:
  print v.idx,v.name, v.type, \
    v.activity,v.lowerBound, v.upperBound
```

Displays all information about variables and constraints of the optimal solution

Variables:

```
0 x[1,1] B 0.0 0.0 1.0
1 x[1,2] B 0.0 0.0 1.0
2 x[1,3] B 0.0 0.0 1.0
3 x[1,4] B 1.0 0.0 1.0
4 x[2,1] B 0.0 0.0 1.0
5 x[2,3] B 1.0 0.0 1.0
6 x[2,4] B 0.0 0.0 1.0
7 x[3,1] B 1.0 0.0 1.0
8 x[3,2] B 0.0 0.0 1.0
9 x[3,3] B 0.0 0.0 1.0
10 x[3,4] B 0.0 0.0 1.0
```

```
for c in m.solution.constraints:
  print c.idx, c.name, c.type, \
    c.activity,c.lowerBound, c.upperBound
```

Constraints:

```
0 sos_m[1] E 1.0 1.0 1.0
1 sos_m[2] E 1.0 1.0 1.0
2 sos_m[3] E 1.0 1.0 1.0
3 sos_l[1] L 1.0 -inf 1.0
4 sos_l[2] L 0.0 -inf 1.0
5 sos_l[3] L 1.0 -inf 1.0
6 sos_l[4] L 1.0 -inf 1.0
```

```
m = Cmpl("assignment.cmpl")
...
m.setOption("%display nonZeros")
m.setOption("%arg -solver cplex")
m.setOption("%display solutionPool")
m.solve()

for s in m.solutionPool:
  print "Solution number: ",s.idx+1
  print "Objective value: ",s.value
  print "Objective status: ",s.status


  print "Variables:"
  for v in s.variables:
    print  v.idx,v.name, v.type, \
      v.activity,v.lowerBound,\
      v.upperBound

  print "Constraints:"
  for c in s.constraints:
```

Solves the example from subchapter 4.1 and displays all information about variables and constraints of all solutions found

```
Solution number:  1
Objective value:  29.0
Objective status: integer optimal
solution

Variables:
3 x[1,4] B 1.0 0.0 1.0
5 x[2,3] B 1.0 0.0 1.0
7 x[3,1] B 1.0 0.0 1.0


Constraints:
0 sos_m[1] E 1.0 1.0 1.0
```

`print c.idx,c.name,c.type, \`
`  c.activity,c.lowerBound, \`
`  c.upperBound`
	`1 sos_m[2] E 1.0 1.0 1.0`
	`2 sos_m[3] E 1.0 1.0 1.0`
	`3 sos_l[1] L 1.0 -inf 1.0`
	
	`5 sos_l[3] L 1.0 -inf 1.0`
	`6 sos_l[4] L 1.0 -inf 1.0`
	
	`Solution number:  2`
	`Objective value:  29.0`
	`Objective status:integer feasible solution`
	`...`

`for s in m.solutionPool:`
 	
`  m.varByName(s.idx)`
`  m.conByName(s.idx)`
 	
`  print "Variables:"`
`  for c in combinations.values:`
`  print m.x[c].name,m.x[c].type, \`
`  m.x[c].activity,\`
`  m.x[c].lowerBound,\`
`  m.x[c].upperBound`
 	
`  print "Constraints:"`
`  for i in m.sos_m:`
`  print m.sos_m[i].name,\`
`  m.sos_m[i].type, \`
`  m.sos_m[i].activity,\`
`  m.sos_m[i].lowerBound,\`
`  m.sos_m[i].upperBound`
 	
`  for j in m.sos_l:`
`  print m.sos_l[j].name,\`
`  m.sos_l[j].type,\`
`  m.sos_l[j].activity,\`
`  m.sos_l[j].lowerBound,\`
`  m.sos_l[j].upperBound`
	As above but with direct access to the variable and constraint names
	Enables the direct access to the variable and constraint names of the current solution
	
	Iterates the variables `x[i,j]` over the value list of the `CmplSet` object `combinations`
	
	Iterates over the internal list of the indexing entries of the constraints with the name `sos_m`
	
	Iterates over the internal list of the indexing entries of the constraints with the name `sos_l`

`v = model.getVarByName("x");`
 	
`c = model.getConByName("line");`
 	
`for x in v:`
	`v` is assigned a list of `CmplSolution` objects for the variable array with the name `x`
	
	`c` is assigned a list of `CmplSolution` objects for the constraint array with the name `line`

`  print x.name, x.type ,x.activity, \`
`    x.lowerBound,x.upperBound ,x.marginal`
 	
`for x in c:`
`  print x.name , x.type ,x.activity, \`
`    x.lowerBound,x.upperBound ,x.marginal`
	Iterates over the list of `CmplSolution` objects for the variable array with the name `x`
	
	Iterates over the list of `CmplSolution` objects for the constrains array with the name `line`

4.3.3.5 Reading CMPL messages

R/o attributes:

Cmpl.**cmplMessages**

Description: Returns a list of `CmplMsg` objects that contain the CMPL messages

Return: `list` of *CmplMsg* objects — list of CMPL messages

CmplMsg.**type**

Description: Returns the type of the messages

Return: `str` — message type `warning|error`

CmplMsg.**file**

Description: Returns the name of the CMPL file in that the error or warning occurs

Return: `str` — CMPL file name or CmplData file name

CmplMsg.**line**

Description: Returns the line in the CMPL file in that the error or warning occurs

Return: `str` — line number

CmplMsg.**description**

Description: Returns a description of the error or warning message

Return: `str` — description of the error or warning

Examples:

`model = Cmpl("diet.cmpl")`
`...`
`model.solve()`
 	
`if model.cmplStatus==CMPL_WARNINGS:`
`  for m in model.cmplMessages:`
`    print m.type, \`
	If some warnings for the CMPL model `diet.cmpl` appear the messages will be shown.

m.file,\ m.line, \ m.description	

4.3.4 CmplExceptions

pyCMPL provides its own exception handling. If an error occurs either by using pyCmpl classes or in the CMPL model a `CmplException` is raised by pyCmpl automatically. This exception can be handled through using a try-except block.

```
try:
  # do something
except CmplException, e:
  print e.msg
```

4.4 jCMPL reference manual

To use the jCMPL functionalities a Java programme has to import jCMPL by `import jCMPL.*;` and to link your application against `jCmpl.jar` and the following jar files, that you can find in the CMPL applicatiopn folder in `jCmpl/Libs`: `commons-lang3`, `ws-commons-util`, `xmplrpc-client`, `xmlrpc-commons`.

4.4.1 CmplSets

The class `CmplSet` is intended to define sets that can be used with several `Cmpl` objects.

Setter methods:

`CmplSet(`*`setName[,rank]`***`)`**

Description: Constructor

Parameter:

`String` *`setName`*	name of the set Has to be equal to the corresponding name in the CMPL model.
`int` *`rank`*	optional - rank *n* for a *n*-tuple set (default 1)

Return: `CmplSet` object

`CmplSet``.`**`setValues(`***`setList`***`)`**

Description: Defines the values of an enumeration set

Parameter:

`Object` *`setList`*	for a set of *n*-tuples with *n*=1 - `List|Array` of single indexing entries `int|Integer|long|Long|String` for a set of *n*-tuples with *n*>1 – 2-dimensional `List|Array` that contain `int|Integer|long|Long|String` tuples

Return: -

CmplSet.**setValues(***startNumber,endNumber***)**

Description: Defines the values of an algorithmic set
(startNumber, startNumber+1, ..., endNumber)

Parameter:
- `int` *startNumber* — start value of the set
- `int` *endNumber* — end value of the set

Return: -

CmplSet.**setValues(***startNumber,step,endNumber***)**

Description: Defines the values of an algorithmic set
(startNumber, startNumber+step, ..., endNumber)

Parameter:
- `int` *startNumber* — start value of the set
- `int` *step* — positive value for increment; negative value for decrement
- `int` *endNumber* — end value of the set

Return: -

Getter methods:

CmplSet.**values()**

Description: List of the indexing entries of the set

Return: `List | Array of Object` — one-dimensional `List` or `Array` of single `int|Integer|long|Long|String` - for a set of *n*-tuples with *n*=1; two-dimensional `List` or `Array` of `int|Integer|long|Long|String` - for a set of *n*-tuples with *n*>1

CmplSet.**name()**

Description: Name of the set

Return: `String` — name of the CMPL set (not the name of the `CmplSet` object)

CmplSet.**rank()**

Description: Rank of the set

Return: `int` — number of *n* of a *n*-tuple set

CmplSet.**len()**

Description: Length of the set

Return: `int` — number of indexing entries

Examples:

`CmplSet s = new CmplSet("s");` `s.setValues(0,4);`	`s` is assigned $s \in (0,1,\ldots,4)$

`System.out.println( s.rank() );`
`System.out.println( s.len() );`
`System.out.println( s.name() );`
`System.out.println( s.values() );`
	`1`
	`5`
	`s`
	`[0, 1, 2, 3, 4]`

`CmplSet s = new CmplSet("a");`
`s.setValues(10,-2,0);`
 	
`System.out.println( s.rank() );`
`System.out.println( s.len() );`
`System.out.println( s.name() );`
`System.out.println( s.values() );`
	s is assigned $s \in (10, 8, \ldots, 0)$
	
	`1`
	`6`
	`a`
	`[10, 8, 6, 4, 2, 0]`

`CmplSet s = new CmplSet("FOOD");`
`Sring[] sVals`
`  = {"BEEF","CHK","FISH"};`
`s.setValues(sVals);`
 	
`System.out.println( s.rank() );`
`System.out.println( s.len() );`
`System.out.println( s.name() );`
 	
`for (String e: (String[]) s.values())`
`  System.out.println( e);`
	s is assigned $s \in ('BEEF', 'CHK', 'FISH')$
	
	`1`
	`3`
	`FOOD`
	
	`BEEF`
	`CHK`
	`FISH`

`CmplSet s = new CmplSet("FOOD");`
 	
`ArrayList nutrLst =`
`  new ArrayList<String>();`
`nutrLst.add("BEEF");`
`nutrLst.add("CHK");`
`nutrLst.add("FISH");`
`s.setValues(nutrLst);`
 	
`System.out.println( s.rank() );`
`System.out.println( s.len() );`
`System.out.println( s.name() );`
`System.out.println( s.values());`
	s is assigned $s \in ('BEEF', 'CHK', 'FISH')$
	
	`1`
	`3`
	`FOOD`
	`[BEEF, CHK, FISH]`

`CmplSet s = new CmplSet("c",3);`
`int[][] sVals = {  {1,1,1}, {1,1,2},`
`  {1,2,1} };`
 	
`s.setValues(sVals);`
	s is assigned a 3-tuple set of integers

`System.out.println( s.rank() );`
`System.out.println( s.len() );`
`System.out.println( s.name() );`
 	
`for (int i=0; i<s.len(); i++) {`
`  for (int j=0; j<s.rank(); j++)`
`  System.out.print( s.get(i,j) );`
`  System.out.print("\n");`
`}`
	`3`
	`3`
	`c`
	
	`111`
	`112`
	`121`

4.4.2 CmplParameters

The class `CmplParameters` is intended to define parameters that can be used with several `Cmpl` objects.

Setter methods:

`CmplParameter(`*`paramName`***`"[,`***`set1`***`,`***`set2`***`,...])`**

Description: Constructor

Parameter:

`String` *`paramName`*	name of the parameter Has to be equal to the corresponding name in the CMPL model.
`CmplSet` *`set1,set2,...`*	optional - set or sets through which the parameter array is defined (default `None`)

Return: `CmplParameter` object

*`CmplParameter`***`.setValues(`***`val`***`)`**

Description: Defines the values of a scalar parameter

Parameter:

`int|Integer|`
`long|Long|float|`
`Float|double|`
`Double|String`
`val`
	value of the scalar parameter

Return: -

*`CmplParameter`***`.setValues(`***`vals`***`)`**

Description: Defines the values of a parameter array

Parameter:

`Object` *`vals`*	one- our multidimensional `List|Array` of `int|Integer|long|Long|float|Float|double|Double|String`

Return: -

Getter methods:

CmplParameter.**values()**

Description: List of the values of a parameter

Return: `Object` - one- our multidimensional `List|Array` of `int|Integer|long|Long|float|Float|double|Double|String` - value list of the parameter array

CmplParameter.**value()**

Description: Value of a scalar parameter

Return: `int|Integer|long|Long|float|Float|double|Double|String` - value of the scalar parameter

CmplParameter.**setList()**

Description: List of sets through which the parameter array is defined

Return: `list` of `CmplSet` objects through which the parameter array is defined

CmplParameter.**name()**

Description: Name of the parameter

Return: `String` - name of the CMPL parameter (not the name of the `CmplParameter` object)

CmplParameter.**rank()**

Description: Rank of the parameter

Return: `int` - rank of the CMPL parameter

CmplParameter.**len()**

Description: Length of the parameter array

Return: `long` number of elements in the parameter array

Examples:

`CmplParameter p = new CmplParameter("p");`
`p.setValues(2);`
 	
`System.out.println( p.values());`
`System.out.println( p.value());`
`System.out.println( p.name());`
`System.out.println( p.rank());`
`System.out.println( p.len());`
	`p` is assigned `2`
	
	`2`
	`2`
	`p`
	`0`
	`1`

`CmplSet s = new CmplSet("s");` `s.setValues(0,4);`	

```
CmplParameter p = new CmplParameter("p",s);
int[] pVals = { 1,2,3,4,5 };
p.setValues(pVals);

for ( int val : (int[])p.values() )
  System.out.println(val);




System.out.println( p.name());
System.out.println( p.rank());
System.out.println( p.len());
```

p is assigned $(1,2,\ldots,5)$

```
1
2
3
4
5

p
1
5
```

```
CmplSet products =  new CmplSet("products");
products.setValues(1,3);
CmplSet machines =  new CmplSet("machines");
machines.setValues(1,2);

CmplParameter a = new
    CmplParameter("a",machines,products);
int[][] aVals = { {8,15,12}, {15,10,8} };
a.setValues(aVals);

for ( int i=0; i<machines.len(); i++ ) {
   for (int j=0; j<products.len(); j++)
     System.out.print( " " +
        ((int[][])a.values())[i][j]   );
   System.out.println();
}

System.out.println( a.name());
System.out.println( a.rank());
System.out.println( a.len());

for (CmplSet s : a.setList())
     System.out.println( s.values());
```

a is assigned a 2X3 matrix of integers

```
 8 15 12
 15 10 8

a
2
6

[1, 2]
[1, 2, 3]
```

```
CmplSet s = new CmplSet("s",2);
int[][] sVals = { {1,1}, {2,2} };
s.setValues(sVals);

CmplParameter p = new CmplParameter("p",s);
int[] pVals = { 1 , 1} ;
p.setValues(pVals);
```

s is assigned the indices of a matrix diagonal

s is assigned a 2x2 identity matrix

```
for ( int val : (int[])p.values() )
  System.out.println(val);
```

```
1
1
```

```
System.out.println( a.name());
System.out.println( a.rank());
System.out.println( a.len());
```

```
p
2
2
```

4.4.3 Cmpl

With the `Cmpl` class it is possible to define a CMPL model, to commit sets and parameters to this model, to start and control the solving process and to read the CMPL and solver messages and to have access to the solution(s) via `CmplMessages` and `CmplSolutions` objects.

4.4.3.1 Establishing models

Setter methods:

`Cmpl(`*`name`***`)`**

Description: Constructor
Parameter: `String` *name* filename of the CMPL model
Return: `Cmpl` object

`Cmpl``.`**`setSets(`***`set1[,set2,...]`***`)`**

Description: Committing `CmplSet` objects to the `Cmpl` model
Parameter: `CmplSet` *set1[,set2,...]* `CmplSet` object(s)
Return: -

`Cmpl``.`**`setParameters(`***`par1[,par2,...]`***`)`**

Description: Committing `CmplParameter` objects to the `Cmpl` model
Parameter: `CmplParameter` *par1[,par2,...]* `CmplParameter` object(s)
Return: -

Examples:

```
Cmpl m = new Cmpl("prodmix.cmpl");

CmplSet products =
  new CmplSet("products");
products.setValues(1,3);
```

`mplSet machines = new CmplSet("machines");`
`machines.setValues(1,2);`
 	
`CmplParameter c =`
`  new CmplParameter("c",products);`
`int[] cVals = {75,80,50};`
`c.setValues(cVals);`
 	
`CmplParameter b =`
`  new CmplParameter("b",machines);`
`int[] bVals = {1000,1000};`
`b.setValues(bVals);`
 	
`CmplParameter a =`
`  new CmplParameter("a",machines,products);`
`int[][] aVals = { {8,15,12}, {15,10,8} };`
 	
`a.setValues(aVals);`	

`m.setSets(products,machines);`	Commits the sets `products,machines` to the Cmpl object `m`
`m.setParameters(c,a,b);`	Commits the parameter `c,a,b` to the Cmpl object m

4.4.3.2 Manipulating models

Setter methods:

Cmpl.**setOption(***option***)**

Description: Sets a CMPL, display or solver option
Parameter: `String` *option* option in CmplHeader syntax
Return: `int` option id

Cmpl.**delOption(***optId***)**

Description: Deletes an option
Parameter: `Int` *optId* option id
Return: -

Cmpl.**delOptions()**

Description: Deletes all options

Parameter: -

Return: -

Cmpl.**setOutput(***ok[,leadStr]]***)**

Description: Turns the output of CMPL and the invoked solver on or off

Parameter: `boolean` *ok* `true|false`

`String` *leadStr* optional - Leading string for the output (default - model name)

Return: -

Cmpl.**setRefreshTime(***rTime***)**

Description: Refresh time for getting the output of CMPL and the invoked solver from a CMPLServer if the model is solved synchronously.

Parameter: `long` *rTime* refresh time in milliseconds (default 400)

Return: -

Getter methods:

Cmpl.**refreshTime()**

Description: Returns the refresh time for getting the output of CMPL and the invoked solver from a CMPLServer if the model is solved synchronously.

Return: `long` Refresh time in milliseconds

Examples:

`Cmpl m = new Cmpl("assignment.cmpl");`
 	
`long c1=m.setOption("%display nonZeros");`
`m.setOption("%arg -solver cplex");`
`m.setOption("%display solutionPool");`
 	
`m.delOption(c1);`
`m.delOptions();`
	
	Setting some options
	
	Deletes the first option
	Deletes all options

`Cmpl m = new Cmpl("assignment.cmpl");`
 	
`m.setOutput(True);`
 	
`m.setOutput(True,"my special model");`
	
	The stdOut and stdErr of CMPL and the invoked solver are shown for the `Cmpl` object `m`.
	
	As above but the output starts with the leading string `"my special model>"`.

`Cmpl m = new Cmpl("assignment.cmpl");`
`m.connect("http://194.95.45.70:8008");`
`m.setOutput(True);`
`m.setRefreshTime(500);`
	The `stdOut` and `stdErr` of CMPL and the invoked solver located at the specified CMPLServer will be refreshed every 500 millisecond.

4.4.3.3 Solving models

Setter Methods:

Cmpl.**solve()**

Description: Solves a `Cmpl` model either with a local installed CMPL or if the model is connected with a CMPLServer remotely.

Parameter: -

Return: - status of the model and the solver can be obtained by the methods `cmplStatus`, `cmplStatusText`,`solverStatus` and `solverStatusText`

Cmpl.**start()**

Description: Solves a `Cmpl` model in a separate thread either with a local installed CMPL or if the model is connected with a CMPLServer remotely.

Parameter: -

Return: - status of the model and the solver can be obtained by the methods `cmplStatus`, `cmplStatusText`,`solverStatus` and `solverStatusText`

Cmpl.**join()**

Description: Waits until the solving thread terminates.

Parameter: -

Return: - status of the model and the solver can be obtained by the methods `cmplStatus`, `cmplStatusText`,`solverStatus` and `solverStatusText`

Cmpl.**isAlive()**

Description: Return whether the thread is alive

Parameter: -

Return: `boolean` `true` or `false` - return whether the thread is alive or not

Cmpl.**connect(***cmplUrl***)**

Description: Connects a CMPLServer or CMPLGridScheduler under *cmplUrl* – first step of solving a model on a CMPLServer remotely

Parameter: `String` *`cmplUrl`* URL of the CMPLServer or CMPLGridScheduler

Return: –

`Cmpl``.`**`disconnect()`**

Description: Disconnects the connected CMPLServer or CMPLGridScheduler

Parameter: –

Return: –

`Cmpl``.`**`send()`**

Description: Sends the `Cmpl` model instance to the connected CMPLServer - first step of solving a model on a CMPLServer asynchronously (after connect())

Parameter: –

Return: – status of the model can be obtained by the methods `cmplStatus` and `cmplStatusText`

`Cmpl``.`**`knock()`**

Description: Knocks on the door of the connected CMPLServer or CMPLGridScheduler and asks whether the model is finished - second step of solving a model on a CMPLServer asynchronously

Parameter: –

Return: – status of the model can be obtained by the methods `cmplStatus` and `cmplStatusText`

`Cmpl``.`**`retrieve()`**

Description: Retrieves the `Cmpl` solution(s) if possible from the connected CMPLServer - last step of solving a model on a CMPLServer asynchronously

Parameter: –

Return: – status of the model and the solver can be obtained by the methods `cmplStatus`, `cmplStatusText`,`solverStatus` and `solverStatusText`

`Cmpl``.`**`cancel()`**

Description: Cancels the `Cmpl` solving process on the connected CMPLServer

Parameter: –

Return: – status of the model can be obtained by the methods `cmplStatus` and `cmplStatusText`

Getter methods:

Cmpl.**cmplStatus()**

Description: Returns the CMPL related status of the `Cmpl` object

Return: `int`

```
CMPL_UNKNOWN = 0
CMPL_OK = 1
CMPL_WARNINGS = 2
CMPL_FAILED = 3
CMPLSERVER_OK = 6
CMPLSERVER_ERROR = 7
CMPLSERVER_BUSY = 8
CMPLSERVER_CLEANED = 9
CMPLSERVER_WARNING = 10
PROBLEM_RUNNING = 11
PROBLEM_FINISHED = 12
PROBLEM_CANCELED = 13
PROBLEM_NOTRUNNING = 14
CMPLGRID_SCHEDULER_UNKNOWN = 15
CMPLGRID_SCHEDULER_OK = 16
CMPLGRID_SCHEDULER_ERROR = 17
CMPLGRID_SCHEDULER_BUSY = 18
CMPLGRID_SCHEDULER_SOLVER_NOT_AVAILABLE = 19
CMPLGRID_SCHEDULER_WARNING = 20
CMPLGRID_SCHEDULER_PROBLEM_DELETED = 21
```

Cmpl.**cmplStatusText()**

Description: Returns the CMPL related status text of the `Cmpl` object

Return: `String`

```
CMPL_UNKNOWN
CMPL_OK
CMPL_WARNINGS
CMPL_FAILED
CMPLSERVER_OK
CMPLSERVER_ERROR
CMPLSERVER_BUSY
CMPLSERVER_CLEANED
CMPLSERVER_WARNING
PROBLEM_RUNNING
PROBLEM_FINISHED
PROBLEM_CANCELED
PROBLEM_NOTRUNNING
CMPLGRID_SCHEDULER_UNKNOWN
CMPLGRID_SCHEDULER_OK
CMPLGRID_SCHEDULER_ERROR
CMPLGRID_SCHEDULER_BUSY
CMPLGRID_SCHEDULER_SOLVER_NOT_AVAILABLE
```

	CMPLGRID_SCHEDULER_WARNING
	CMPLGRID_SCHEDULER_PROBLEM_DELETED

Cmpl.**solverStatus()**

Description: Returns the solver related status of the `Cmpl` object

Return:	`int`	`SOLVER_OK = 4`
		`SOLVER_FAILED = 5`

Cmpl.**solverStatusText()**

Description: Returns the solver related status text of the `Cmpl` object

Return:	`String`	`SOLVER_OK`
		`SOLVER_FAILED`

Cmpl.**jobId()**

Description: Returns the jobId of the `Cmpl` problem at the connected CMPLServer

Return: `String` string of the jobId

Cmpl.**output()**

Description: Returns the output of CMPL and the invoked solver.
Intended to use if an application needs to parse the output.

Return: `String` string of output of CMPL and the invoked solver

Examples:

`Cmpl m = new Cmpl("assignment.cmpl");` `m.solve();`	Solves the `Cmpl` object `m` locally

`Cmpl m = new Cmpl("assignment.cmpl");`
`m.connect("http://194.95.45.70:8008");`
`m.solve();`
	Solves the `Cmpl` object `m` remotely and synchronously on the specified CMPLServer

`Cmpl m = new Cmpl("assignment.cmpl");`
`m.connect("http://194.95.45.70:8008");`
`m.send();`
`m.knock();`
`m.retrieve();`
	Solves the `Cmpl` object `m` remotely and asynchronously on the specified CMPLServer

`ArrayList<Cmpl> models =`
`  new ArrayList<Cmpl>();`
 	
`models.add(new Cmpl("m1.cmpl"));`
`models.add(new Cmpl("m2.cmpl"));`
`models.add(new Cmpl("m3.cmpl"));`
 	
`for (Cmpl c : models)`
`  c.start();`
	Starts all models in separate threads.

`for (Cmpl c : models)` `  c.join();`	Waits until the all solving threads are terminated.

`Cmpl m = new Cmpl("assignment.cmpl");`
`m.solve();`
`if (m.solverStatus() == Cmpl.SOLVER_OK)`
`  m.solutionReport();`
	Displays the optimal solution if the solver didn't fail.

4.4.3.4 Reading solutions

Setter methods:

Cmpl.**solutionReport()**

Description: Writes a standard solution report to stdOut

Parameter: -

Return: -

Cmpl.**saveSolution(***[solFileName]***)**

Description: Saves the solution(s) as `CmplSolutions` file

Parameter: `String` *solFileName* optional file name (default `<modelname>.csol`)

Return: -

Cmpl.**saveSolutionAscii(***[solFileName]***)**

Description: Saves the solution(s) as ASCII file

Parameter: `String` *solFileName* optional file name (default `<modelname>.sol`)

Return: -

Cmpl.**saveSolutionCsv(***[solFileName]***)**

Description: Saves the solution(s) as CSV file

Parameter: `String` *solFileName* optional file name (default `<modelname>.csv`)

Return: -

Getter methods:

Cmpl.**nrOfVariables()**

Description: Returns the number of variables of the generated and solved CMPL model

Return: `long` number of variables

Cmpl.**nrOfConstraints()**

Description: Returns the number of constraints of the generated and solved CMPL model

Return: `long` number of constraints

Cmpl.**objectiveName()**

Description: Returns the name of the objective function of the generated and solved CMPL model

Return: `String` objective name

Cmpl.**objectiveSense()**

Description: Returns the objective sense of the generated and solved CMPL model

Return: `String` objective sense

Cmpl.**nrOfSolutions()**

Description: Returns the number of solutions of the generated and solved CMPL model

Return: `int` number of solutions

Cmpl.**solver()**

Description: Returns the name of the invoked solver of the generated and solved CMPL model

Return: `String` invoked solver

Cmpl.**solverMessage()**

Description: Returns the message of the invoked solver of the generated and solved CMPL model

Return: `String` message of the invoked solver

Cmpl.**varDisplayOptions()**

Description: Returns a string with the display options for the variables of the generated and solved CMPL model

Return: `String` display options for the variables

Cmpl.**conDisplayOptions()**

Description: Returns a string with the display options for the constraints of the generated and solved CMPL model

Return: `String` display options for the constraints

Cmpl.**solution()**

Description: Returns the first (optimal) `CmplSolutions` object

Return: `CmplSolutions` first (optimal) solution

Cmpl.**solutionPool()**

Description: Returns a list of `CmplSolutions` objects

Return: `List of CmplSolutions` objects — list of `CmplSolution` object for solutions found

CmplSolutions.**status()**

Description: Returns a string with the status of the current solution provided by the invoked solver

Return: `String` — solution status

CmplSolutions.**value()**

Description: Returns the value of the objective function of the current solution

Return: `double` — objective function value

CmplSolutions.**idx()**

Description: Returns the index of the current solution

Return: `int` — index of the current solution

CmplSolutions.**variables()**

Description: Returns a list of `CmplSolElement` objects for the variables of the current solution

Return: `ArrayList<CmplSolElement>` — list of variables

CmplSolutions.**constraints()**

Description: Returns a list of `CmplSolElement` objects for the constraints of the current solution

Return: `ArrayList<CmplSolElement>` — list of constraints

Cmpl.**getVarByName(***name*, *[solIdx]***)**

Description: Returns a `CmplSolElement` object or `CmplSolArray` of `CmplSolElement` objects for the variable or variable array with the specified name

Parameter: `String name` — name of the variable or variable array
`int solIdx` — optional solution index (default `0`)

Return: `Object` — `CmplSolElement` for a single variable
`CmplSolArray` for a variable array

Cmpl.**getConByName(***[solIdx]***)**

Description: Returns a `CmplSolElement` object or `CmplSolArray` of `CmplSolElement` objects for the constraint or constraint array with the specified name

Parameter: `String name` — name of the constraint or constraint array
`int solIdx` — optional solution index (default `0`)

Return: `Object` `CmplSolElement` for a single constraint
`CmplSolArray` for a constraint array

`CmplSolElement.`**`idx()`**

Description: Index of the variable or constraint
Return: `int` index of the variable or constraint

`CmplSolElement.`**`name()`**

Description: Name of the variable or constraint
Return: `String` name of the variable or constraint

`CmplSolElement.`**`type()`**

Description: Type of the variable or constraint
Return: `String` type of the variable or constraint
`C|I|B` for variables
`L|E|G` for constraints

`CmplSolElement.`**`activity()`**

Description: Activity of the variable or constraint
Return: *`Object`* *`Double|Long`* Activity of the variable or constraint

`CmplSolElement.`**`lowerBound()`**

Description: Lower bound of the variable or constraint
Return: `double` lower bound of the variable or constraint

`CmplSolElement.`**`upperBound()`**

Description: Upper bound of the variable or constraint
Return: `double` upper bound of the variable or constraint

`CmplSolElement.`**`marginal()`**

Description: Marginal value (shadow prices or reduced costs) bound of the variable or constraint
Return: `double` marginal value of the variable or constraint

Examples:

```
Cmpl m = new Cmpl("assignment.cmpl");
...

m.solve();

System.out.printf("%s\n",m.solver());
System.out.printf("%s\n",m.solverMessage());
```

Solves the example from subchapter 4.1 and displays some information about the generated and solved model

```
CBC
```

```
System.out.printf("%d\n",m.nrOfVariables());
System.out.printf("%d\n",m.nrOfConstraints());
System.out.printf("%s\n",m.varDisplayOptions());
System.out.printf("%s\n",m.conDisplayOptions());
System.out.printf("%s\n",m.objectiveName());
System.out.printf("%s\n",m.objectiveSense());
System.out.printf("%f\n",m.solution().value());
System.out.printf("%s\n",m.solution().status());
System.out.printf("%d\n",m.nrOfSolutions());
System.out.printf("%d\n",m.solution().idx());
```

```
11
7
(all)
(all)
costs
min
29.000000
optimal
1
0
```

Displays all information about variables and constraints of the optimal solution

```
for (CmplSolElement v : m.solution().variables())
{
  System.out.printf("%8s %2s %2d %2.0f %2.0f%n",
     v.name(), v.type(),v.activity(),
     v.lowerBound(),v.upperBound());
}
```

Variables:

```
  x[1,1]  B  0  0  1
  x[1,2]  B  0  0  1
  x[1,3]  B  0  0  1
  x[1,4]  B  1  0  1
  x[2,1]  B  0  0  1
  x[2,3]  B  1  0  1
  x[2,4]  B  0  0  1
  x[3,1]  B  1  0  1
  x[3,2]  B  0  0  1
  x[3,3]  B  0  0  1
  x[3,4]  B  0  0  1
```

```
for (CmplSolElement c:m.solution().constraints())
{
  System.out.printf("%8s %2s %2.0f %2.0f %2.0f
     %n", c.name(), c.type(),c.activity(),
     c.lowerBound(),c.upperBound());
}
```

Constraints:

```
sos_m[1]  E  1  1  1
sos_m[2]  E  1  1  1
sos_m[3]  E  1  1  1
sos_l[1]  L  1 -Infinity  1
sos_l[2]  L  0 -Infinity  1
sos_l[3]  L  1 -Infinity  1
sos_l[4]  L  1 -Infinity  1
```

```
CmplSolArray x = (CmplSolArray)
m.getVarByName("x");
for(int[] tuple: (int[][]) combinations.values())
{  System.out.printf("%5s %2d %n",
     x.get(tuple).name(),
     x.get(tuple).activity() );
}
```

Direct access to the variable vector `x[]` by its name

```
Cmpl m = new Cmpl("assignment.cmpl");
...
m.setOption("%display nonZeros");
m.setOption("%arg -solver cplex");
```

Solves the example from subchapter 4.1 and displays all information about variables and constraints of all solu-

```
m.setOption("%display solutionPool");
m.solve();

for (CmplSolution s : m.solutionPool()) {
  System.out.printf("Solution number:  %d %n",
    (s.idx() + 1));
  System.out.printf("Objective value:  %f %n",
    s.value());
  System.out.printf("Objective status: %s %n",
    s.status());

  System.out.println("Variables:");
  for (CmplSolElement v : s.variables()) {
    System.out.printf("%8s %2s %2d %2.0f %2.0f
      %n", v.name(), v.type(), v.activity(),
      v.lowerBound(), v.upperBound());
  }
  System.out.println("Constraints:");
  for (CmplSolElement c : s.constraints()) {
    System.out.printf("%8s %2s %2.0f %2.0f %2.0f
      %n", c.name(), c.type(), c.activity(),
      c.lowerBound(), c.upperBound());
  }
}
```

```
tion found


Solution number:  1
Objective value:  29.000000
Objective status: integer
optimal solution


Variables:
  x[1,4]  B  1  0  1
  x[2,3]  B  1  0  1
  x[3,1]  B  1  0  1

Constraints:
sos_m[1]  E  1  1  1
sos_m[2]  E  1  1  1
sos_m[3]  E  1  1  1
sos_l[1]  L  1 -Infinity  1
sos_l[3]  L  1 -Infinity  1
sos_l[4]  L  1 -Infinity  1
Solution number:  2
Objective value:  29.000000
Objective status: integer
feasible solution
Variables:
  x[1,4]  B  1  0  1
 ...
```

4.4.3.5 Reading CMPL messages

Getter methods:

Cmpl.**cmplMessages()**

Description: Returns a list of `CmplMsg` objects that contain the CMPL messages

Return: `ArrayList<CmplMsg>` list of CMPL messages

CmplMsg.**type()**

Description: Returns the type of the messages

Return: `String` message type `warning|error`

CmplMsg.**file()**

Description: Returns the name of the CMPL file in that the error or warning occurs

Return: `String` CMPL file name or CmplData file name

CmplMsg.**line()**

Description: Returns the line in the CMPL file in that the error or warning occurs

Return: `String` line number

CmplMsg.**description()**

Description: Returns the a description of the error or warning message

Return: `String` description of the error or warning

Examples:

```
model = Cmpl("diet.cmpl")
...
model.solve();

if (model.cmplStatus()==Cmpl.CMPL_WARNINGS) {
     for (CmplMsg m: model.cmplMessages()) {
       System.out.printf("%s %s %s %s %s",
         m.type(), m.file(), m.line(),
         m.description());
     }
}
```

If some warnings for the CMPL model `diet.cmpl` appear the messages will be shown.

4.4.4 CmplExceptions

jCMPL provides its own exception handling. If an error occurs either by using jCmpl classes or in the CMPL model a `CmplException` is raised by jCmpl automatically. This exception can be handled through using a try-catch block.

```
try {
   // do something
} catch (CmplException e) {
   System.out.println(e);
}
```

4.5 Examples

4.5.1 The diet problem

4.5.1.1 Problem description and CMPL model

In this subchapter the jCMPL and jCMPL formulation of the diet problem already discussed in subchapter 2.8.1.1 is dealt with.

The first step is to formulate the CMPL model `diet.cmpl` where the sets and parameters that are created in the pyCmpl script have to be specified in the CMPL header entry `%data`:

```
%data : NUTR set, FOOD set, costs[FOOD], vitamin[NUTR,FOOD], vitMin[NUTR]

variables:
  x[FOOD]: integer[2..10];

objectives:
  cost: costs[]T * x[]->min;

constraints:
  $2$: vitamin[,] * x[] >= vitMin[];
```

4.5.1.2 pyCMPL

The corresponding pyCMPL script `diet.py` is formulated as follows:

```
#!/usr/bin/python

from pyCmpl import *

try:
  model = Cmpl("diet.cmpl")

  nutr = CmplSet("NUTR")
  nutr.setValues(["A", "B1", "B2", "C"])

  food = CmplSet("FOOD")
  food.setValues(["BEEF","CHK","FISH","HAM","MCH","MTL","SPG","TUR"])

  costs = CmplParameter("costs",food)
  costs.setValues([3.19, 2.59, 2.29, 2.89, 1.89, 1.99, 1.99, 2.49])

  vitmin = CmplParameter("vitMin",nutr)
  vitmin.setValues([ 700, 700, 700, 700])

  vitamin = CmplParameter("vitamin",nutr, food)
  vitamin.setValues([   [60,8,8,40,15,70,25,60], \
                        [20,0,10,40,35,30,50,20] , \
```

```
                          [10,20,15,35,15,15,25,15], \
                          [15,20,10,10,15,15,15,10]])

  model.setSets(nutr,food)
  model.setParameters(costs,vitmin,vitamin)

  model.solve()
  model.solutionReport()

except CmplException, e:
  print e.msg
```

Executing this pyCMPL model by using the command:

```
pyCmpl diet.py
```

leads to the following output created by pyCMPL's standard solution report:

```
-------------------------------------------------------------------------------------------------------------
Problem              diet.cmpl
Nr. of variables     8
Nr. of constraints   4
Objective name       cost
Solver name          CBC
Display variables    (all)
Display constraints  (all)
-------------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      101.14              (min!)

Variables
Name                 Type            Activity          LowerBound          UpperBound            Marginal
-------------------------------------------------------------------------------------------------------------
x[BEEF]                 I                   2                2.00               10.00                   -
x[CHK]                  I                   8                2.00               10.00                   -
x[FISH]                 I                   2                2.00               10.00                   -
x[HAM]                  I                   2                2.00               10.00                   -
x[MCH]                  I                  10                2.00               10.00                   -
x[MTL]                  I                  10                2.00               10.00                   -
x[SPG]                  I                  10                2.00               10.00                   -
x[TUR]                  I                   2                2.00               10.00                   -
-------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity          LowerBound          UpperBound            Marginal
-------------------------------------------------------------------------------------------------------------
line[A]                 G             1500.00              700.00                 inf                   -
line[B1]                G             1330.00              700.00                 inf                   -
line[B2]                G              860.00              700.00                 inf                   -
line[C]                 G              700.00              700.00                 inf                   -
-------------------------------------------------------------------------------------------------------------
```

4.5.1.3 jCmpl

The corresponding jCMPL programme `diet.java` is formulated as follows:

```
import jCMPL.*;
```

```
public class Diet {

  public static void main(String[] args) throws CmplException {
    try {
      Cmpl model = new Cmpl("diet.cmpl");

      CmplSet nutr = new CmplSet("NUTR");
      String[] nutrLst = {"A", "B1", "B2", "C"};
      nutr.setValues(nutrLst);

      CmplSet food = new CmplSet("FOOD");
      String[] foodLst = {"BEEF", "CHK", "FISH", "HAM", "MCH",
                          "MTL", "SPG", "TUR"};
      food.setValues(foodLst);

      CmplParameter costs = new CmplParameter("costs", food);
      Double[] costVec = {3.19, 2.59, 2.29, 2.89, 1.89, 1.99, 1.99, 2.49};
      costs.setValues(costVec);

      CmplParameter vitmin = new CmplParameter("vitMin", nutr);
      int [] vitminVec = { 700,700,700,700};
      vitmin.setValues(vitminVec);

      CmplParameter vitamin = new CmplParameter("vitamin", nutr, food);
      int[][] vitMat = {     {60, 8, 8, 40, 15, 70, 25, 60},
                             {20, 0, 10, 40, 35, 30, 50, 20},
                             {10, 20, 15, 35, 15, 15, 25, 15},
                             {15, 20, 10, 10, 15, 15, 15, 10}};
      vitamin.setValues(vitMat);

      model.setSets(nutr, food);
      model.setParameters(costs, vitmin, vitamin);

      model.solve();
      model.solutionReport();

    } catch (CmplException e) {
      System.out.println(e);
    }
  }
}
```

Executing this jCMPL programme leads to the following output created by jCMPL's standard solution report:

```
-------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------
Problem             diet.cmpl
Nr. of variables    8
Nr. of constraints  4
```

```
Objective name       cost
Solver name          CBC
Display variables    (all)
Display vonstraints  (all)
-----------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      101.14              (min!)

Variables
Name                 Type           Activity         LowerBound         UpperBound           Marginal
-----------------------------------------------------------------------------------------------------
x[BEEF]                 I                  2               2.00              10.00                  -
x[CHK]                  I                  8               2.00              10.00                  -
x[FISH]                 I                  2               2.00              10.00                  -
x[HAM]                  I                  2               2.00              10.00                  -
x[MCH]                  I                 10               2.00              10.00                  -
x[MTL]                  I                 10               2.00              10.00                  -
x[SPG]                  I                 10               2.00              10.00                  -
x[TUR]                  I                  2               2.00              10.00                  -
-----------------------------------------------------------------------------------------------------

Constraints
Name                 Type           Activity         LowerBound         UpperBound           Marginal
-----------------------------------------------------------------------------------------------------
line[A]                 G             1500.00             700.00           Infinity                  -
line[B1]                G             1330.00             700.00           Infinity                  -
line[B2]                G              860.00             700.00           Infinity                  -
line[C]                 G              700.00             700.00           Infinity                  -
-----------------------------------------------------------------------------------------------------
```

4.5.2 Transportation problem

4.5.2.1 Problem description and CMPL model

This subchapter discusses the pyCMPL formulation of the transportation problem from subchapter 2.8.1.6.

The CMPL model `transportation.cmpl` can be formulated as follows:

```
%data : plants set,centers set,routes set[2],c[routes], s[plants], d[centers]

variables:
  x[routes]: real[0..];
objectives:
  costs: sum{ [i,j] in routes : c[i,j]*x[i,j] } ->min;
constraints:
  supplies {i in plants : sum{j in routes *> [i,*] : x[i,j]} = s[i];}
  demands  {j in centers: sum{i in routes *> [*,j] : x[i,j]} <= d[j];}
```

4.5.2.2 pyCMPL

The corresponding pyCMPL script `transportation.py` is formulated as follows:

```
#!/usr/bin/python

from pyCmpl import *
```

```
try:
  model = Cmpl("transportation.cmpl")

  routes = CmplSet("routes",2)
  routes.setValues([[1,1],[1,2],[1,4],[2,2],[2,3],[2,4],[3,1],[3,3]])

  plants = CmplSet("plants")
  plants.setValues(1,3)

  centers = CmplSet("centers")
  centers.setValues(1,4)

  costs = CmplParameter("c",routes)
  costs.setValues([3,2,6,5,2,3,2,4])

  s = CmplParameter("s",plants)
  s.setValues([5000,6000,2500])

  d = CmplParameter("d",centers)
  d.setValues([6000,4000,2000,2500])

  model.setSets(routes, plants, centers)
  model.setParameters(costs,s,d)

  model.setOutput(True)
  model.setOption("%display nonZeros")
  model.solve()

  if model.solverStatus == SOLVER_OK:
    model.solutionReport()
  else:
    print "Solver failed " + model.solver + " " + model.solverMessage

except CmplException, e:
  print e.msg
```

Executing this pyCMPL model by using the command:

```
pyCmpl transportation.py
```

leads to the following output of CMPL and CBC (enabled with `model.setOutput(True)`) and the standard solution report:

```
transportation> CMPL model generation - running
transportation>
transportation> CMPL version: 1.9.0
transportation> Authors: Thomas Schleiff, Mike Steglich
transportation> Distributed under the GPLv3
transportation>
```

```
transportation> create model instance ...
transportation> write model instance ...
transportation> CMPL model generation - finished
transportation>
transportation> Solver - running
transportation>
transportation> Welcome to the CBC MILP Solver
transportation> Version: 2.8.8
transportation> Build Date: Jan  3 2014
transportation> Revision Number: 2001
transportation>
transportation> command line - /Applications/Cmpl/bin/../Thirdparty/CBC/cbc transportation_933604.mps min solve
gsolu transportation_933604.gsol (default strategy 1)
transportation> At line 2 NAME          transportation.cmpl
transportation> At line 3 ROWS
transportation> At line 12 COLUMNS
transportation> At line 29 RHS
transportation> At line 34 RANGES
transportation> At line 35 BOUNDS
transportation> At line 44 ENDATA
transportation> Problem transportation.cmpl has 7 rows, 8 columns and 16 elements
transportation> Coin0008I transportation.cmpl read with 0 errors
transportation> Presolve 6 (-1) rows, 7 (-1) columns and 14 (-2) elements
transportation> 0  Obj 16499.7 Primal inf 9500.2001 (3) Dual inf 1.9999999 (1)
transportation> 5  Obj 36500
transportation> Optimal - objective value 36500
transportation> After Postsolve, objective 36500, infeasibilities - dual 0 (0), primal 0 (0)
transportation> Optimal objective 36500 - 5 iterations time 0.002, Presolve 0.00
transportation> Total time (CPU seconds):       0.00   (Wallclock seconds):       0.00
transportation>
transportation>
transportation> CMPL: Time used for model generation: 0 seconds
transportation> CMPL: Time used for solving the model: 0 seconds
transportation>
transportation>
transportation> Solution written to cmplSolution file: transportation_933604.csol
transportation>
transportation> Solver - finished
transportation>
-------------------------------------------------------------------------------------------------------------
Problem              transportation.cmpl
Nr. of variables     8
Nr. of constraints   7
Objective name       costs
Solver name          CBC
Display variables    nonZeroVariables (all)
Display constraints  nonZeroConstraints (all)
-------------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      36500.00            (min!)

Variables
Name                 Type            Activity          LowerBound          UpperBound            Marginal
-------------------------------------------------------------------------------------------------------------
x[1,1]                  C             2500.00                0.00                 inf                0.00
x[1,2]                  C             2500.00                0.00                 inf                0.00
x[2,2]                  C             1500.00                0.00                 inf                0.00
x[2,3]                  C             2000.00                0.00                 inf                0.00
x[2,4]                  C             2500.00                0.00                 inf                0.00
x[3,1]                  C             2500.00                0.00                 inf                0.00
-------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity          LowerBound          UpperBound            Marginal
-------------------------------------------------------------------------------------------------------------
```

```
supplies[1]            E          5000.00          5000.00          5000.00             3.00
supplies[2]            E          6000.00          6000.00          6000.00             6.00
supplies[3]            E          2500.00          2500.00          2500.00             2.00
demands[1]             L          5000.00             -inf          6000.00             0.00
demands[2]             L          4000.00             -inf          4000.00            -1.00
demands[3]             L          2000.00             -inf          2000.00            -4.00
demands[4]             L          2500.00             -inf          2500.00            -3.00
---------------------------------------------------------------------------------------------------
```

4.5.2.3 jCMPL

The corresponding jCMPL script `transportation.java` is formulated as follows:

```
import jCMPL.*;

import java.util.ArrayList;

public class Transportation {

  public static void main(String[] args) throws CmplException {
    try {
      Cmpl model = new Cmpl("transportation.cmpl");

      CmplSet routes = new CmplSet("routes", 2);
      int[][] arcs = { {1, 1}, {1, 2}, {1, 4}, {2, 2}, {2, 3},
                       {2, 4}, {3, 1}, {3, 3}};
      routes.setValues(arcs);

      CmplSet plants = new CmplSet("plants");
      plants.setValues(1, 3);

      CmplSet centers = new CmplSet("centers");
      centers.setValues(1, 1, 4);

      CmplParameter costs = new CmplParameter("c", routes);
      Integer[] costArr = {3, 2, 6, 5, 2, 3, 2, 4};
      costs.setValues(costArr);

      CmplParameter s = new CmplParameter("s", plants);
      int[] sList = {5000,6000,2500};
      s.setValues(sList);

      CmplParameter d = new CmplParameter("d", centers);
      int[] dArr = {6000, 4000, 2000, 2500};
      d.setValues(dArr);

      model.setSets(routes, plants, centers);
      model.setParameters(costs, s, d);
```

```
        model.setOutput(true);
        model.setOption("%display nonZeros");
        model.solve();

        if (model.solverStatus() == Cmpl.SOLVER_OK) {
           model.solutionReport();
        } else {
           System.out.println("Solver failed " + model.solver() +
                              " " + model.solverMessage());
        }

     } catch (CmplException e) {
           System.out.println(e);

     }
   }
}
```

Executing this pyCMPL model by using the command:

```
pyCmpl transportation.py
```

leads to the following output of CMPL and CBC (enabled with `model.setOutput(True)`) and the standard solution report:

```
transportation> CMPL model generation - running
transportation> CMPL version: 1.9.0
transportation> Authors: Thomas Schleiff, Mike Steglich
transportation> Distributed under the GPLv3
transportation> create model instance ...
transportation> write model instance ...
transportation> CMPL model generation - finished
transportation> Solver - running
transportation> Welcome to the CBC MILP Solver
transportation> Version: 2.8.8
transportation> Build Date: Jan  3 2014
transportation> Revision Number: 2001
transportation> command line - /Applications/Cmpl/bin/../Thirdparty/CBC/cbc transportation_228086.mps min solve
gsolu transportation_228086.gsol (default strategy 1)
transportation> At line 2 NAME          transportation.cmpl
transportation> At line 3 ROWS
transportation> At line 12 COLUMNS
transportation> At line 29 RHS
transportation> At line 34 RANGES
transportation> At line 35 BOUNDS
transportation> At line 44 ENDATA
transportation> Problem transportation.cmpl has 7 rows, 8 columns and 16 elements
transportation> Coin0008I transportation.cmpl read with 0 errors
transportation> Presolve 6 (-1) rows, 7 (-1) columns and 14 (-2) elements
transportation> 0  Obj 16499.7 Primal inf 9500.2001 (3) Dual inf 1.9999999 (1)
transportation> 5  Obj 36500
transportation> Optimal - objective value 36500
transportation> After Postsolve, objective 36500, infeasibilities - dual 0 (0), primal 0 (0)
transportation> Optimal objective 36500 - 5 iterations time 0.002, Presolve 0.00
transportation> Total time (CPU seconds):       0.00   (Wallclock seconds):       0.00
transportation> CMPL: Time used for model generation: 0 seconds
transportation> CMPL: Time used for solving the model: 0 seconds
transportation> Solution written to cmplSolution file: transportation_228086.csol
transportation> Solver - finished
```

```
-----------------------------------------------------------------------------------------------------------
Problem              transportation.cmpl
Nr. of variables     8
Nr. of constraints   7
Objective name       costs
Solver name          CBC
Display variables    nonZeroVariables (all)
Display vonstraints  nonZeroConstraints (all)
-----------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      36500.00             (min!)

Variables
Name                 Type           Activity         LowerBound         UpperBound           Marginal
-----------------------------------------------------------------------------------------------------------
x[1,1]                  C            2500.00               0.00           Infinity               0.00
x[1,2]                  C            2500.00               0.00           Infinity               0.00
x[2,2]                  C            1500.00               0.00           Infinity               0.00
x[2,3]                  C            2000.00               0.00           Infinity               0.00
x[2,4]                  C            2500.00               0.00           Infinity               0.00
x[3,1]                  C            2500.00               0.00           Infinity               0.00
-----------------------------------------------------------------------------------------------------------

Constraints
Name                 Type           Activity         LowerBound         UpperBound           Marginal
-----------------------------------------------------------------------------------------------------------
supplies[1]             E            5000.00            5000.00            5000.00               3.00
supplies[2]             E            6000.00            6000.00            6000.00               6.00
supplies[3]             E            2500.00            2500.00            2500.00               2.00
demands[1]              L            5000.00          -Infinity            6000.00               0.00
demands[2]              L            4000.00          -Infinity            4000.00              -1.00
demands[3]              L            2000.00          -Infinity            2000.00              -4.00
demands[4]              L            2500.00          -Infinity            2500.00              -3.00
-----------------------------------------------------------------------------------------------------------
```

4.5.3 The shortest path problem

4.5.3.1 Problem description and CMPL model

Consider an undirected network $G=(V,A)$ where V is a set of nodes and A is a set of arcs joining pairs of nodes. The decision is to find the shortest path from a starting node s to a target node t. This problem can be formulated as an LP as follows (Hillier and Liebermann 2010, p. 383f.):

$$\sum_{(i,j)\in A} c_{ij} \cdot x_{ij} \rightarrow min!$$

$$s.t.$$

$$\sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} = \begin{cases} 1 & , if\ i=s \\ -1 & , if\ i=t \\ 0 & , otherwise \end{cases} ; \forall i \in V$$

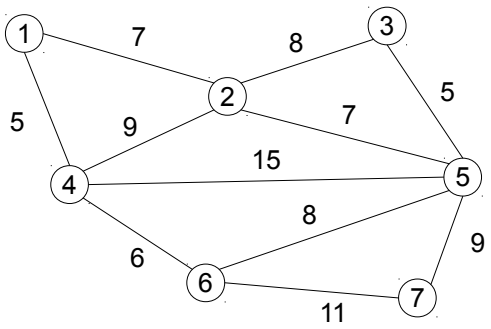
$$x_{ij} \geq 0; \forall (i,j) \in A$$

The decision variables are $x_{ij}; \forall \in A$ with $x_{ij}=1$ if the arc $i \rightarrow j$ is used. The parameters $c_{ij}; \forall \in A$ define the distance between the nodes i and j, but can also are interpreted as the time a vehicle takes to drive from node i to node j.

This CMPL model can be formulated as follows whilst the sets A an V and the parameters c_{ij}, t and s are defined in a pyCMPL script or jCMPL programme.

```
%data : A set[2], c[A], V set , s, t

parameters:
  { i in V: { i=s : rHs[i]:=1; |
       i=t : rHs[i]:=-1; |
       default: rHs[i]:=0;}  }

variables:
  x[A] :real[0..];

objectives:
  sum{ [i,j] in A: c[i,j]*x[i,j] } -> min;

constraints:
  node { i in V:  sum{ j in (A *> [i,*]) : x[i,j]  } -
           sum{ j in (A *> [*,i]) : x[j,i]  }  = rHs[i];}
```

To describe the formulation of the shortest path problem in pyCMPL and jCMPL the simple example shown in the following figure is used where the weights on the arcs are interpreted as the time in minutes a vehicle needs to travel from a node i to a node j.



It is assumed that the starting node is node 1 and the target node is node 7.

4.5.3.2 pyCMPL

The corresponding pyCMPL script `shortest-path.py` is formulated as follows:

```
#!/usr/bin/python

from pyCmpl import *

try:
  model = Cmpl("shortest-path.cmpl")

  routes = CmplSet("A",2)
  routes.setValues([ [1,2],[1,4],[2,1],[2,3],[2,4],[2,5],\
```

```
                      [3,2],[3,5],[4,1],[4,2],[4,5],[4,6],\
                      [5,2],[5,3],[5,4],[5,6],[5,7],\
                      [6,4],[6,5],[6,7],[7,5],[7,6] ])

  nodes = CmplSet("V")
  nodes.setValues(1,7)

  c = CmplParameter("c", routes)
  c.setValues([7,5,7,8,9,7,8,5,5,9,15,6,7,5,15,8,9,6,8,11,9,11])

  sNode = CmplParameter("s")
  sNode.setValues(1)

  tNode = CmplParameter("t")
  tNode.setValues(7)

  model.setSets(routes, nodes)
  model.setParameters(c,sNode,tNode)


  model.solve()
  print "Objective Value: ", model.solution.value

  for v in model.solution.variables:
    if v.activity>0:
      print v.name , " " , v.activity

except CmplException, e:
  print e.msg
```

Executing this pyCMPL script through using the command:

```
pyCmpl shortes-path.py
```

leads to the following output of the pyCMPL script:

```
Objective Value:  22.0
x[1,4]   1.0
x[4,6]   1.0
x[6,7]   1.0
```

The optimal route is 1→4→6→7 with a travelling time of 22 minutes.

4.5.3.3 jCMPL

The corresponding jCMPL programme `shortest-path.java` is formulated as follows:

```
package shortestpath;
```

```
import jCMPL.*;

public class ShortestPath {

  public static void main(String[] args) throws CmplException {

    try {
      Cmpl m = new Cmpl("shortest-path.cmpl");

      CmplSet routes = new CmplSet("A", 2);
      int[][] arcs = { {1, 2}, {1, 4}, {2, 1}, {2, 3}, {2, 4}, {2, 5},
                       {3, 2}, {3, 5}, {4, 1}, {4, 2}, {4, 5}, {4, 6},
                       {5, 2}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
                       {6, 4}, {6, 5}, {6, 7}, {7, 5}, {7, 6}};
      routes.setValues(arcs);

      CmplSet nodes = new CmplSet("V");
      nodes.setValues(1, 7);

      CmplParameter c = new CmplParameter("c", routes);
      Integer[] cArr = {7, 5, 7, 8, 9, 7, 8, 5, 5, 9, 15, 6, 7, 5, 15, 8, 9,
                        6, 8, 11, 9, 11};
      c.setValues(cArr);

      CmplParameter sNode = new CmplParameter("s");
      sNode.setValues(1);

      CmplParameter tNode = new CmplParameter("t");
      tNode.setValues(7);

      m.setSets(routes, nodes);
      m.setParameters(c, sNode, tNode);

      m.solve();

      if (m.solverStatus() == Cmpl.SOLVER_OK) {
        System.out.println("Objective value     :" + m.solution().value());
        for (CmplSolElement v : m.solution().variables()) {
          if ((Double) v.activity() > 0) {
            System.out.println(v.name() + " " + v.activity());
          }
        }
      } else {
        System.out.println("Solver failed " + m.solver() + " "
                           + m.solverMessage());
      }
```

```
    } catch (CmplException e) {
      System.out.println(e);
    }
  }
}
```

Executing this jCMPL programme leads to the following output of the pyCMPL script:

```
Objective value     :22.0
x[1,4] 1.0
x[4,6] 1.0
x[6,7] 1.0
```

As in pyCMPL the optimal route is 1→4→6→7 with a travelling time of 22 minutes.

4.5.4 Solving randomized shortest path problems in parallel

4.5.4.1 Problem description

For the last example it was shown that the optimal route travelling from node 1 to node 7 is 1→4→6→7. This solution is based on the assumption that the travelling times between nodes are certain. This example describes how a randomized shortest path problem can be solved where subproblems describing random situations are solved in own threads in parallel.

4.5.4.2 pyCMPL

Assuming that the staring node is node 1 and the target node is node 7 the corresponding pyCMPL script `shortest-path.py` is formulated as follows:

```
1  #!/usr/bin/python
2  from __future__ import division
3
4  from pyCmpl import *
5  import random
6
7  try:
8    routes = CmplSet("A",2)
9    routes.setValues([ [1,2],[1,4],[2,1],[2,3],[2,4],[2,5],\
10           [3,2],[3,5],[4,1],[4,2],[4,5],[4,6],\
11           [5,2],[5,3],[5,4],[5,6],[5,7],\
12           [6,4],[6,5],[6,7],[7,5],[7,6] ])
13
14   nodes = CmplSet("V")
15   nodes.setValues(1,7)
16
17   c = CmplParameter("c", routes)
18   c.setValues([7,5,7,8,9,7,8,5,5,9,15,6,7,5,15,8,9,6,8,11,9,11])
19
```

```
20    sNode = CmplParameter("s")
21    sNode.setValues(1)
22
23    tNode = CmplParameter("t")
24    tNode.setValues(7)
25
26    models= []
27    randC = []
28    for i in range(5):
29      models.append(Cmpl("shortest-path.cmpl"))
30      models[i].setOption("%display nonZeros")
31      models[i].setSets(routes, nodes)
32
33      tmpC =[]
34      for m in c.values:
35        tmpC.append( m + random.randint(-40,40)/10)
36
37      randC.append(CmplParameter("c", routes))
38      randC[i].setValues(tmpC)
39
40      models[i].setParameters(randC[i],sNode,tNode)
41
42    for m in models:
43      m.start()
44
45    for m in models:
46      m.join()
47
48    i = 0
49    for m in models:
50      print "problem : " , i , " needed time " , m.solution.value
51      for v in m.solution.variables:
52           print v.name , " " , v.activity
53      i = i + 1
54
56    except CmplException, e:
57      print e.msg
```

This script uses the same sets and parameters as before but for each of the 5 instantiated models in line 29 a new parameter array c is created whilst the original array c is changed by random numbers in line 35. In line 43 all of the models are starting and in line 46 the pyCmpl script is waiting for the termination of all of the models.

Executing this pyCMPL script through using the command:

```
pyCmpl shortes-path.py
```

can lead to the following output of the pyCMPL script, but every new run will show different results because of the random numbers.

```
problem :  0  needed time  23.7
x[1,4]   1.0
x[4,6]   1.0
x[6,7]   1.0
problem :  1  needed time  20.2
x[1,2]   1.0
x[2,5]   1.0
x[5,7]   1.0
problem :  2  needed time  13.3
x[1,4]   1.0
x[4,6]   1.0
x[6,7]   1.0
problem :  3  needed time  17.6
x[1,2]   1.0
x[2,5]   1.0
x[5,7]   1.0
problem :  4  needed time  20.7
x[1,4]   1.0
x[4,6]   1.0
x[6,7]   1.0
```

Depending on the uncertain traffic situations two different routes between the nodes $1 \rightarrow 7$ can be optimal: 1→4→6→7 and 1→2→5→7 .

4.5.4.3 jCMPL

Assuming that the staring node is node 1 and the target node is node 7 the corresponding jCMPL programme `shortest-path.java` is formulated as follows:

```
1  package shortestpath;
2  import jCMPL.*;
3  import java.util.ArrayList;
4  import java.util.logging.Level;
5  import java.util.logging.Logger;
6
7  public class shortestPathThreads {
8
9     public static void main(String[] args) throws CmplException {
10
11       try {
12         CmplSet routes = new CmplSet("A",2);
13         int[][] arcs = {   {1,2},{1,4},{2,1},{2,3},{2,4},{2,5},
14                            {3,2},{3,5},{4,1},{4,2},{4,5},{4,6},
15                            {5,2},{5,3},{5,4},{5,6},{5,7},
16                            {6,4},{6,5},{6,7},{7,5},{7,6}};
```

```
17        routes.setValues(arcs);
18
19        CmplSet nodes = new CmplSet("V");
20        nodes.setValues(1,7);
21
22        Integer[] cArr = {7,5,7,8,9,7,8,5,5,9,15,6,7,5,15,8,9,6,8,11,9,11};
23
24        CmplParameter sNode = new CmplParameter("s");
25        sNode.setValues(1);
26
27        CmplParameter tNode = new CmplParameter("t");
28        tNode.setValues(7);
29
30        ArrayList<Cmpl> models = new ArrayList<Cmpl>();
31        ArrayList<CmplParameter> randC = new ArrayList<CmplParameter>();
32
33        for (int i = 0; i  <= 5; i++) {
34
35          models.add(new Cmpl("shortest-path.cmpl") );
36          models.get(i).setSets(routes, nodes);
37
38          randC.add(new CmplParameter("c", routes) );
39
40          ArrayList<Double> tmpC = new ArrayList<Double>();
41          for (Integer cArr1 : cArr) {
42            tmpC.add(Double.valueOf(cArr1) +
43                 Double.valueOf( -40 +  (Math.random() *  40) )/10);
44          }
45          randC.get(i).setValues(tmpC);
46          models.get(i).setParameters(randC.get(i), sNode, tNode);
47          models.get(i).setOption("%display nonZeros");
48        }
49
50        for (Cmpl c : models) {
51          c.start();
52        }
53
54        for (Cmpl c : models) {
56          c.join();
57        }
58
59        int i = 0;
60        for (Cmpl c : models) {
61          System.out.println("model : " + String.valueOf(i) +
62               " needed time : " + c.solution().value());
63
```

```
64            for (CmplSolElement v : c.solution().variables()) {
65              System.out.println(v.name() + " " + v.activity());
66            }
67            i++;
68          }
69
70        } catch (CmplException e) {
71          System.out.println(e);
72        } catch (InterruptedException ex) {
73          Logger.getLogger(shortestPathThreads.class.getName()) ↲
              .log(Level.SEVERE, null, ex);
74        }
75      }
76  }
```

This script uses the same sets and parameters as before but for each of the 5 instantiated models in line 35 a new parameter array c is created whilst the original array c is changed by random numbers in line 42. In line 51 all of the models are starting and in line 56 the jCmpl programme is waiting for the termination of all of the models.

Executing this jCMPL programme can lead to the following output of the pyCMPL script, but every new run will show different results because of the random numbers.

```
model : 0 needed time : 12.4438
x[1,2] 1.0
x[2,5] 1.0
x[5,7] 1.0
model : 1 needed time : 14.9163
x[1,2] 1.0
x[2,5] 1.0
x[5,7] 1.0
model : 2 needed time : 15.2786
x[1,4] 1.0
x[4,6] 1.0
x[6,7] 1.0
model : 3 needed time : 15.253
x[1,4] 1.0
x[4,6] 1.0
x[6,7] 1.0
model : 4 needed time : 13.8339
x[1,4] 1.0
x[4,6] 1.0
x[6,7] 1.0
```

Depending on the uncertain traffic situations two different routes between the nodes $1 \rightarrow 7$ can be optimal: 1→4→6→7 and 1→2→5→7 .

4.5.5 Column generation for a cutting stock problem

4.5.5.1 Problem description and CMPL model

The following pyCMPL script and the corresponding jCMPL programme including the example are based on the AMPL formulation of a column generator for a cutting stock problem and is taken from (Fourer et.al. 2003, p. 304ff). In this cutting stock problem long raw rolls of paper have to be cut up into combinations of smaller widths that have to meet given orders and the objective is to minimize the waste.

In the example, the raw width is 110" and the demands for particular widths are given in the following table:

orders (demand)	withs
48	20"
35	45"
24	50"
10	55"
8	75"

Fourer, Gay & Kernigham use the Gilmore-Gomory procedure to define cutting patterns by involving two linear programmes.

The first model is a cutting optimisation model that finds the minimum number of raw rolls with a given set of possible cutting patterns subject to fulfilling the orders for the particular widths. This problem can be formulated as in the CMPL file `cut.cmpl` as follows:

```
%data :rollWidth, widths set, patterns set, orders[widths],nbr[widths,patterns]

variables:
  cut[patterns]: integer[0..];

objectives:
  number: sum{ j in patterns: cut[j] }->min;

constraints:
  fill {i in widths: sum{ j in patterns : nbr[i,j]*cut[j] } >= orders[i]; }
```

The parameter `rollWidth` defines the width of the raw rolls, the set `widths` defines the widths to be cut, the set `patterns` the set of the patterns, the parameter `orders` the number of orders per width and the parameters `nbr[i,j]` the number of rolls of width `i` in pattern `j`. The variables are the `cut[j]` and they define how many cuts of a pattern `j` are to be produced.

The second model is the pattern generation model that is indented to identify a new pattern that can be used in the cutting optimisation.

```
%data :  widths set, price[widths], rollWidth

variables:
  use[widths]: integer[0..];
  reducedCosts : real;
```

```
objectives:
  sum{ i in widths: price[i] * use[i]} -> max;

constraints:
  sum{ i in widths : i * use[i] } <= rollWidth;
```

This model in the CMPL file `cut-pattern.cmpl` requires as specified in the `%data` entry the set `widths`, the parameter `rollWidth` and a parameter vector `price`, that contains the marginals of the constraints `fill` of a solved `cut.cmpl` problem with a relaxation of the integer variables `cut[j]`.

It is a knapsack problem that "fills" a knapsack (here a raw roll with a given width `rollWidth`) with the most valuable things (here the desired widths via the variables `use[i]`) where the value of a width `i` is specified by the `price[i]`.

4.5.5.2 jCMPL

The relationship between these two CMPL models and the entire cutting optimisation procedure is controlled by the following pyCMPL script `cut.py`

```
1   #!/usr/bin/python
2
3   from pyCmpl import *
4   import math
5
6   try:
7     cuttingOpt = Cmpl("cut.cmpl")
8     patternGen = Cmpl("cut-pattern.cmpl")
9
10    cuttingOpt.setOption("%arg -solver cplex")
11    patternGen.setOption("%arg -solver cplex")
12
13    r = CmplParameter("rollWidth")
14    r.setValues(110)
15
16    w = CmplSet("widths")
17    w.setValues([ 20, 45, 50, 55, 75])
18
19    o = CmplParameter("orders",w)
20    o.setValues([ 48, 35, 24, 10, 8 ])
21
22    nPat=w.len
23    p = CmplSet("patterns")
24    p.setValues(1,nPat)
25
26    nbr = []
27    for i in range(nPat):
28      nbr.append( [ 0 for j in range(nPat) ] )
```

```
29
30    for i in w.values:
31      pos = w.values.index(i)
32      nbr[pos][pos] = int(math.floor( r.value / i ))
33
34    n = CmplParameter("nbr", w, p)
35    n.setValues(nbr)
36
37    price = []
38    for i in range(w.len):
39      price.append(0)
40
41    pr = CmplParameter("price", w)
42    pr.setValues(price)
43
44    cuttingOpt.setSets(w,p)
45    cuttingOpt.setParameters(r, o, n)
46
47    patternGen.setSets(w)
48    patternGen.setParameters(r,pr)
49
50    ri = cuttingOpt.setOption("%arg -integerRelaxation")
51
52    while True:
53      cuttingOpt.solve()
54      cuttingOpt.conByName()
55
56      for i in w.values:
57        pos = w.values.index(i)
58        price[pos] = cuttingOpt.fill[i].marginal
59
60      pr.setValues(price)
61
62      patternGen.solve()
63      patternGen.varByName()
64
65      if (1-patternGen.solution.value) < -0.00001:
66        nPat = nPat + 1
67        p.setValues(1,nPat)
68        for i in w.values:
69          pos = w.values.index(i)
70          nbr[pos].append(patternGen.use[i].activity)
71        n.setValues(nbr)
72      else:
73        break
74
```

```
75    cuttingOpt.delOption(ri)
76
77    cuttingOpt.solve()
78    cuttingOpt.varByName()
79
80    print "Objective value: " , cuttingOpt.solution.value , "\n"
81    print "Pattern:"
82
83    vStr="   | "
84    for j in p.values:
85      vStr+= " %d " % j
86    print vStr
87
88    print "----------------------------"
89    for i in range(len(w.values)):
90      vStr="%2d | " % w.values[i]
91      for j in p.values:
92        vStr += " %d " % nbr[i][j-1]
93      print vStr
94    print "\n"
95
96    for j in p.values:
97      if cuttingOpt.cut[j].activity>0:
98        print "%2d pieces of pattern: %d" % (cuttingOpt.cut[j].activity, j)
99        for i in range(len(w.values)):
100         print "   width ", w.values[i] , " - " , nbr[i][j-1]
101
102 except CmplException, e:
103   print e.msg
```

Cplex is chosen as solver for both in the lines 7 and 8 instantiated models (lines 10,11). In the next lines 13-20 the parameters `rollWidth` and `orders` and the set `widths` are created and the corresponding data are assigned. The lines 26-35 are intended to create an initial set of patterns whilst the matrix `nbr` contains only one pattern per width, where the diagonal elements are equal to the maximal possible number of rolls of the particular width. After creating the vector `price` with null values in the lines 37-42 all relevant sets and parameters are committed to both `Cmpl` objects (lines 44-48).

In the next lines the Gilmore-Gomory procedure is performed.

1. Solve the cutting optimisation problem `cut.cmpl` with an integer relaxation (line 50 and 53).
2. Assign the shadow prices `cuttingOpt.fill[i].marginal` to the corresponding elements `price[i]` for each pattern (lines 56-58).
3. Solve the pattern generation model `cut-pattern.cmpl` (line 62).
4. If (1 – optimal objective value) is approximately < 0 (line 65)

then add a new pattern using the activities `patternGen.use[i].activity` for all elements in `widths` (lines 68-70) and jump to step 1,

else

Solve the final cutting optimisation problem `cut.cmpl` as integer programme (lines 75 and 77)

After finding the final solution the next lines (lines 78-103) are intended to provide some information about the final integer solution.

Executing this pyCMPL model through using the command:

```
pyCmpl cut.py
```

leads to the following output of the pyCMPL script:

```
Objective value:  47.0

Pattern:
   |  1  2  3  4  5  6  7  8
---------------------------
20 |  5  0  0  0  0  1  1  3
45 |  0  2  0  0  0  0  2  0
50 |  0  0  2  0  0  0  0  1
55 |  0  0  0  2  0  0  0  0
75 |  0  0  0  0  1  1  0  0


 8 pieces of pattern: 3
   width  20  -  0
   width  45  -  0
   width  50  -  2
   width  55  -  0
   width  75  -  0
 5 pieces of pattern: 4
   width  20  -  0
   width  45  -  0
   width  50  -  0
   width  55  -  2
   width  75  -  0
 8 pieces of pattern: 6
   width  20  -  1
   width  45  -  0
   width  50  -  0
   width  55  -  0
   width  75  -  1
18 pieces of pattern: 7
   width  20  -  1
   width  45  -  2
   width  50  -  0
   width  55  -  0
```

```
   width  75  -  0
 8 pieces of pattern: 8
   width  20  -  3
   width  45  -  0
   width  50  -  1
   width  55  -  0
   width  75  -  0
```

4.5.5.3 jCMPL

The relationship between these `cut-pattern.cmpl` and `cut.cmpl` and the entire cutting optimisation procedure is controlled by the following jCMPL programme `CuttingStock.java`.

```
1   package cuttingstock;
2   import jCMPL.*;
3   import java.io.BufferedWriter;
4   import java.io.FileWriter;
5   import java.io.IOException;
6   import java.util.ArrayList;
7
8   public class CuttingStock {
9
10    public CuttingStock() throws CmplException {
11      try {
12        Cmpl cuttingOpt = new Cmpl("cut.cmpl");
13        Cmpl patternGen = new Cmpl("cut-pattern.cmpl");
14
15        cuttingOpt.setOption("%arg -solver cplex");
16        patternGen.setOption("%arg -solver cplex");
17
18        CmplParameter r = new CmplParameter("rollWidth");
19        r.setValues(110);
20
21        CmplSet w = new CmplSet("widths");
22        int[] wVals = {20, 45, 50, 55, 75};
23        w.setValues(wVals);
24
25        CmplParameter o = new CmplParameter("orders", w);
26        int[] oVals = {48, 35, 24, 10, 8};
27        o.setValues(oVals);
28
29        int nPat = w.len();
30
31        CmplSet p = new CmplSet("patterns");
32        p.setValues(1, nPat);
33
```

```
34          ArrayList<ArrayList<Long>> nbr = new ArrayList<ArrayList<Long>> ();
35
36          for (int i = 0; i < nPat; i++) {
37            ArrayList<Long> nbrRow = new ArrayList<Long>();
38            for (int j = 0; j < nPat; j++) {
39              if (i == j) {
40                Double nr =  Math.floor(((Integer) r.value()) /
41                                        ((int[]) w.values())[i]) ;
42                nbrRow.add( nr.longValue() );
43              } else {
44                nbrRow.add(Long.valueOf(0));
45              }
46            }
47            nbr.add(nbrRow);
48          }
49
50          CmplParameter n = new CmplParameter("nbr", w, p);
51          n.setValues(nbr);
52
53          Double[] price = new Double[w.len()];
54          for (int i = 0; i < price.length; i++) {
55            price[i] = 0.0;
56          }
57
58          CmplParameter pr = new CmplParameter("price", w);
59          pr.setValues(price);
60
61          cuttingOpt.setSets(w, p);
62          cuttingOpt.setParameters(r, o, n);
63
64          patternGen.setSets(w);
65          patternGen.setParameters(r, pr);
66
67          int ri =  cuttingOpt.setOption("%arg -integerRelaxation");
68
69          while (true) {
70            cuttingOpt.solve();
71
72            CmplSolArray fill=(CmplSolArray)cuttingOpt.getConByName("fill");
73
74            int pos = 0;
75            for (int with : (int[]) w.values()) {
76              price[pos] = fill.get(with).marginal();
77              pos++;
78            }
79
```

```
80            pr.setValues(price);
81
82            patternGen.solve();
83            CmplSolArray use = (CmplSolArray) patternGen.getVarByName("use");
84
85            if (1-patternGen.solution().value() < -0.00001) {
86              nPat++;
87              p.setValues(1, nPat);
88              for (int i = 0; i < w.len(); i++) {
89                ArrayList<Long> tmpList = nbr.get(i);
90                tmpList.add( (Long) use.get(w.get(i)).activity() );
91                nbr.set(i, tmpList);
92              }
93              n.setValues(nbr);
94            } else {
95                break;
96            }
97          }
98          cuttingOpt.delOption(ri);
99
100         cuttingOpt.solve();
101         CmplSolArray cut = (CmplSolArray) cuttingOpt.getVarByName("cut");
102
103         BufferedWriter out =
104           new BufferedWriter(new FileWriter("cuttingStock" + ".stdout"));
105
106         out.write(String.format("Objective value: %4.2f%n%n",
107                                 cuttingOpt.solution().value()));
108         out.write("Pattern:\n");
109         out.write("   | ");
110         for (int j : (ArrayList<Integer>) p.values()) {
111             ut.write(String.format(" %d ", j));
112         }
113         out.write("\n---------------------------\n");
114         for (int i = 0; i < w.len(); i++) {
115           out.write(String.format("%2d | ", w.get(i)));
116           for (int j : (ArrayList<Integer>) p.values()) {
117             out.write(String.format(" %d ", nbr.get(i).get(j - 1)));
118           }
119           out.write("\n");
120         }
121         out.write("\n");
122         for (int j : (ArrayList<Integer>) p.values()) {
123           if ((Long) cut.get(j).activity() > 0) {
124             out.write( String.format("%2d pieces of pattern: %d %n",
125                             (Long) cut.get(j).activity(), j));
```

```
126             for (int i=0; i<w.len(); i++) {
127               out.write(String.format("\twidth %d - %d%n", w.get(i) ,
128                                 nbr.get(i).get(j-1) ) );
129             }
130           }
131         }
132         out.close();
133       } catch (CmplException e) {
134         System.out.println(e);
135       } catch (IOException e) {
136         System.out.println("IO error" + e);
137       }
138    }
139 }
```

Cplex is chosen as solver for both instantiated models in the lines 15 and 16 (lines 12,13). In the next lines 18-27 the parameters `rollWidth` and `orders` and the set `widths` are created and the corresponding data are assigned. The lines 29-51 are intended to create an initial set of patterns whilst the matrix `nbr` contains only of one pattern per width, where the diagonal elements are equal to the maximal possible number of rolls of the particular width. After creating the vector `price` with null values in the lines 53-59 all relevant sets and parameters are committed to both `Cmpl` objects (lines 61-65).

In the next lines the Gilmore-Gomory procedure is performed.

5. Solve the cutting optimisation problem `cut.cmpl` with an integer relaxation (line 67 and 70).
6. Assign the shadow prices `cuttingOpt.fill[i].marginal` to the corresponding elements `price[i]` for each pattern (lines 75-78).
7. Solve the pattern generation model `cut-pattern.cmpl` (line 82).
8. If (1 – optimal objective value) is approximately < 0 (line 85)
   then add a new pattern using the activities `patternGen.use[i].activity` for all elements in `widths` (lines 88-92) and jump to step 1.

   else

   Solve the final cutting optimisation problem `cut.cmpl` as integer programme (line 98 and 100)

After finding the final solution the next lines (lines 101-139) are intended to provide some information about the final integer solution.

Executing this jCMPL model leads to the following output:

```
Objective value: 47.00

Pattern:
   |  1  2  3  4  5  6  7  8
---------------------------
20 |  5  0  0  0  0  1  1  3
45 |  0  2  0  0  0  0  2  0
```

```
50 |  0  0  2  0  0  0  0  1
55 |  0  0  0  2  0  0  0  0
75 |  0  0  0  0  1  1  0  0

 8 pieces of pattern: 3
   width 20 - 0
   width 45 - 0
   width 50 - 2
   width 55 - 0
   width 75 - 0
 5 pieces of pattern: 4
   width 20 - 0
   width 45 - 0
   width 50 - 0
   width 55 - 2
   width 75 - 0
 8 pieces of pattern: 6
   width 20 - 1
   width 45 - 0
   width 50 - 0
   width 55 - 0
   width 75 - 1
18 pieces of pattern: 7
   width 20 - 1
   width 45 - 2
   width 50 - 0
   width 55 - 0
   width 75 - 0
 8 pieces of pattern: 8
   width 20 - 3
   width 45 - 0
   width 50 - 1
   width 55 - 0
   width 75 - 0
```

5 Authors and Contact

- **CMPL**

  Thomas Schleiff - Halle(Saale), Germany

  Mike Steglich - Technical University of Applied Sciences Wildau, Germany - mike.steglich@th-wildau.de

- **Coliop, pyCMPL and CMPLServer**

  Mike Steglich

- **jCMPL**

  Mike Steglich

  Bernhard Knie - Technical University of Applied Sciences Wildau, Germany

- **Contact:**

  c/o Mike Steglich

  Professor of Business Administration, Quantitative Methods and Management Accounting

  Technical University of Applied Sciences Wildau
  Faculty of Business, Administration and Law
  Bahnhofstraße
  D-15745 Wildau

  Tel.: +493375 / 508-365
  Fax.: +493375 / 508-566

  mike.steglich@th-wildau.de

- **Support via mailing list**

  Please use our CMPL mailing list hosted at COIN-OR http://list.coin-or.org/mailman/listinfo/Cmpl to get a direct support, to post bugs or to communicate wishes.

6 Appendix

6.1 Selected CBC parameters

The CBC parameters are taken (mostly unchanged) from the CBC command line help. Only the CBC parameters that are useful in a CMPL context are described afterwards.

Usage CBC parameters:

```
%opt cbc solverOption [solverOptionValue]
```

Double parameters:

dualB(ound) *doubleValue*

Initially algorithm acts as if no gap between bounds exceeds this value

Range of values is 1e-20 to 1e+12, default 1e+10

dualT(olerance) *doubleValue*

For an optimal solution no dual infeasibility may exceed this value
Range of values is 1e-20 to 1e+12, default 1e-07

objective(Scale) *doubleValue*

Scale factor to apply to objective

Range of values is -1e+20 to 1e+20, default 1

primalT(olerance) *doubleValue*

For an optimal solution no primal infeasibility may exceed this value

Range of values is 1e-20 to 1e+12, default 1e-07

primalW(eight) *doubleValue*

Initially algorithm acts as if it costs this much to be infeasible

Range of values is 1e-20 to 1e+20, default 1e+10

rhs(Scale) *doubleValue*

Scale factor to apply to rhs and bounds

Range of values is -1e+20 to 1e+20, default 1

Branch and Cut double parameters:

allow(ableGap) *doubleValue*

Stop when gap between best possible and best less than this

Range of values is 0 to 1e+20, default 0

artif(icialCost) *doubleValue*

Costs >= these are treated as artificials in feasibility pump 0.0 off - otherwise variables with costs >= these are treated as artificials and fixed to lower bound in feasibility pump

Range of values is 0 to 1.79769e+308, default 0

cuto(ff) *doubleValue*

All solutions must be better than this value (in a minimization sense).

This is also set by code whenever it obtains a solution and is set to value of objective for solution minus cutoff increment.

Range of values is -1e+60 to 1e+60, default 1e+50

fix(OnDj) *doubleValue*

Try heuristic based on fixing variables with reduced costs greater than this

If this is set integer variables with reduced costs greater than this will be fixed before branch and bound - use with extreme caution!

Range of values is -1e+20 to 1e+20, default -1

fraction(forBAB) *doubleValue*

Fraction in feasibility pump

After a pass in feasibility pump, variables which have not moved about are fixed and if the pre-processed model is small enough a few nodes of branch and bound are done on reduced problem. Small problem has to be less than this fraction of original.

Range of values is 1e-05 to 1.1, default 0.5

inc(rement) *doubleValue*

A valid solution must be at least this much better than last integer solution

Whenever a solution is found the bound on solutions is set to solution (in a minimization sense) plus this. If it is not set then the code will try and work one out.

Range of values is -1e+20 to 1e+20, default 1e-05

inf(easibilityWeight) *doubleValue*

Each integer infeasibility is expected to cost this much

Range of values is 0 to 1e+20, default 0

integerT(olerance) *doubleValue*

For an optimal solution no integer variable may be this away from an integer value

Range of values is 1e-20 to 0.5, default 1e-06

preT(olerance) *doubleValue*

Tolerance to use in presolve

Range of values is 1e-20 to 1e+12, default 1e-08

pumpC(utoff) *doubleValue*

Fake cutoff for use in feasibility pump

0.0 off - otherwise add a constraint forcing objective below this value in feasibility pump

Range of values is -1.79769e+308 to 1.79769e+308, default 0

pumpI(ncrement) *doubleValue*

Fake increment for use in feasibility pump

0.0 off - otherwise use as absolute increment to cut off when solution found in feasibility pump

Range of values is -1.79769e+308 to 1.79769e+308, default 0

ratio(Gap) *doubleValue*

If the gap between best solution and best possible solution is less than this fraction of the objective value at the root node then the search will terminate.

Range of values is 0 to 1e+20, default 0

reallyO(bjectiveScale) *doubleValue*

Scale factor to apply to objective in place

Range of values is -1e+20 to 1e+20, default 1

sec(onds) *doubleValue*

maximum seconds

After this many seconds coin solver will act as if maximum nodes had been reached.

Range of values is -1 to 1e+12, default 1e+08

tighten(Factor) *doubleValue*

Tighten bounds using this times largest activity at continuous solution

Range of values is 0.001 to 1e+20, default -1

Integer parameters:

idiot(Crash) *integerValue*

This is a type of 'crash' which works well on some homogeneous problems. It works best on problems with unit elements and rhs but will do something to any model. It should only be used before primal. It can be set to -1 when the code decides for itself whether to use it, 0 to switch off or n > 0 to do n passes.

Range of values is -1 to 99999999, default -1

maxF(actor) *integerValue*

Maximum number of iterations between refactorizations

Range of values is 1 to 999999, default 200

maxIt(erations) *integerValue*

Maximum number of iterations before stopping

Range of values is 0 to 2147483647, default 2147483647

passP(resolve) *integerValue*

How many passes in presolve

Range of values is -200 to 100, default 5

pO(ptions) *integerValue*

If this is > 0 then presolve will give more information and branch and cut will give statistics

Range of values is 0 to 2147483647, default 0

slp(Value) *integerValue*

Number of slp passes before primal

If you are solving a quadratic problem using primal then it may be helpful to do some sequential Lps to get a good approximate solution.

Range of values is -1 to 50000, default -1

slog(Level) *integerValue*

Level of detail in (LP) Solver output

Range of values is -1 to 63, default 1

subs(titution) *integerValue*

How long a column to substitute for in presolve

Normally Presolve gets rid of 'free' variables when there are no more than 3 variables in column. If you increase this the number of rows may decrease but number of elements may increase.

Range of values is 0 to 10000, default 3

Branch and Cut integer parameters:

cutD(epth) *integerValue*

Depth in tree at which to do cuts

Cut generators may be - off, on only at root, on if they look possible and on. If they are done every node then that is that, but it may be worth doing them every so often. The original method was every so many nodes but it is more logical to do it whenever depth in tree is a multiple of K. This option does that and defaults to -1 (off -> code decides).

Range of values is -1 to 999999, default -1

cutL(ength) *integerValue*

Length of a cut

At present this only applies to Gomory cuts. -1 (default) leaves as is. Any value >0 says that all cuts <= this length can be generated both at root node and in tree. 0 says to use some dynamic lengths. If value >=10,000,000 then the length in tree is value%10000000 - so 10000100 means unlimited length at root and 100 in tree.

Range of values is -1 to 2147483647, default -1

dense(Threshold) *integerValue*

Whether to use dense factorization

Range of values is -1 to 10000, default -1

depth(MiniBab) *integerValue*

Depth at which to try mini BAB

Rather a complicated parameter but can be useful. -1 means off for large problems but on as if -12 for problems where rows+columns<500, -2 means use Cplex if it is linked in. Otherwise if negative then go into depth first complete search fast branch and bound when depth>= -value-2 (so -3 will use this at depth>=1). This mode is only switched on after 500 nodes. If you really want to switch it off for small problems then set this to -999. If >=0 the value doesn't matter very much. The code will do approximately 100 nodes of fast branch and bound every now and then at depth>=5. The actual logic is too twisted to describe here.

Range of values is -2147483647 to 2147483647, default -1

diveO(pt) *integerValue*

Diving options

If >2 && <8 then modify diving options

–3 only at root and if no solution,

–4 only at root and if this heuristic has not got solution,

–5 only at depth <4,

–6 decay, 7 run up to 2 times

if solution found 4 otherwise.

Range of values is -1 to 200000, default 3

hOp(tions) *integerValue*

Heuristic options

1 says stop heuristic immediately allowable gap reached. Others are for feasibility pump - 2 says do exact number of passes given, 4 only applies if initial cutoff given and says relax after 50 passes, while 8 will adapt cutoff rhs after first solution if it looks as if code is stalling.

Range of values is -9999999 to 9999999, default 0

hot(StartMaxIts) *integerValue*

Maximum iterations on hot start

Range of values is 0 to 2147483647, default 100

log(Level) *integerValue*

Level of detail in Coin branch and Cut output

If 0 then there should be no output in normal circumstances. 1 is probably the best value for most uses, while 2 and 3 give more information.

Range of values is -63 to 63, default 1

maxN(odes) *integerValue*

Maximum number of nodes to do

Range of values is -1 to 2147483647, default 2147483647

maxS(olutions) *integerValue*

Maximum number of solutions to get

You may want to stop after (say) two solutions or an hour. This is checked every node in tree, so it is possible to get more solutions from heuristics.

Range of values is 1 to 2147483647, default -1

passC(uts) *integerValue*

Number of cut passes at root node

The default is 100 passes if less than 500 columns, 100 passes (but stop if drop small if less than 5000 columns, 20 otherwise

Range of values is -9999999 to 9999999, default -1

passF(easibilityPump) *integerValue*

How many passes in feasibility pump

This fine tunes Feasibility Pump by doing more or fewer passes.

Range of values is 0 to 10000, default 30

passT(reeCuts) *integerValue*

Number of cut passes in tree

Range of values is -9999999 to 9999999, default 1

small(Factorization) *integerValue*

Whether to use small factorization

If processed problem <= this use small factorization

Range of values is -1 to 10000, default -1

strong(Branching) *integerValue*

Number of variables to look at in strong branching

Range of values is 0 to 999999, default 5

thread(s) *integerValue*

Number of threads to try and use

To use multiple threads, set threads to number wanted. It may be better to use one or two more than number of cpus available. If 100+n then n threads and search is repeatable (maybe be somewhat slower), if 200+n use threads for root cuts, 400+n threads used in sub-trees.

Range of values is -100 to 100000, default 0

trust(PseudoCosts) *integerValue*

Number of branches before we trust pseudocosts

Range of values is -3 to 2000000, default 5

Keyword parameters:

bscale *option*

Whether to scale in barrier (and ordering speed)

Possible options: off on off1 on1 off2 on2, default off

chol(esky) *option*

Which cholesky algorithm

Possible options: native dense fudge(Long_dummy) wssmp_dummy

crash *option*

Whether to create basis for problem

If crash is set on and there is an all slack basis then Clp will flip or put structural variables into basis with the aim of getting dual feasible. On the whole dual seems to be better without it and there are alternative types of 'crash' for primal e.g. 'idiot' or 'sprint'.

Possible options: off on so(low_halim) ha(lim_solow(JJF mods)), dfeault off

cross(over) *option*

Whether to get a basic solution after barrier

Interior point algorithms do not obtain a basic solution (and the feasibility criterion is a bit suspect (JJF)). This option will crossover to a basic solution suitable for ranging or branch and cut. With the current state of quadratic it may be a good idea to switch off crossover for quadratic (and maybe presolve as well) - the option maybe does this.

Possible options: on off maybe presolve, default on

dualP(ivot) *option*

Dual pivot choice algorithm

Possible options: auto(matic) dant(zig) partial steep(est), default auto(matic)

fact(orization) *option*

Which factorization to use

Possible options: normal dense simple osl, default normal

gamma((Delta)) *option*

Whether to regularize barrier

Possible options: off on gamma delta onstrong gammastrong deltastrong, default off

KKT *option*

Whether to use KKT factorization

Possible options: off on, default off

perturb(ation) *option*

Whether to perturb problem

Possible options: on off, default on

presolve *option*

Presolve analyzes the model to find such things as redundant equations, equations which fix some variables, equations which can be transformed into bounds etc etc. For the initial solve of any problem this is worth doing unless you know that it will have no effect. on will normally do 5 passes while using 'more' will do 10. If the problem is very large you may need to write the original to file using 'file'.

Possible options for presolve are: on off more file, default on

primalP(ivot) *option*

Primal pivot choice algorithm

Possible options: auto(matic) exa(ct) dant(zig) part(ial) steep(est) change sprint, default auto(matic)

scal(ing) *option*

Whether to scale problem

Possible options: off equi(librium) geo(metric) auto(matic) dynamic rows(only), default auto(matic)

spars(eFactor) *option*

Whether factorization treated as sparse

Possible options: on off, default on

timeM(ode) *option*

Whether to use CPU or elapsed time

cpu uses CPU time for stopping, while elapsed uses elapsed time. (On Windows, elapsed time is always used).

Possible options: cpu elapsed, default cpu

vector *option*

If this parameter is set to on ClpPackedMatrix uses extra column copy in odd format.

Possible options: off on, default off

Branch and Cut keyword parameters:

clique(Cuts) *option*

Whether to use Clique cuts

Possible options: off on root ifmove forceOn onglobal, default ifmove

combine(Solutions) *option*

Whether to use combine solution heuristic

This switches on a heuristic which does branch and cut on the problem given by just using variables which have appeared in one or more solutions. It obviously only tries after two or more solutions. See Rounding for meaning of on,both,before

Possible options: off on both before, default on

combine2(Solutions) *option*

Whether to use crossover solution heuristic

This switches on a heuristic which does branch and cut on the problem given by fixing variables which have same value in two or more solutions. It obviously only tries after two or more solutions. See Rounding for meaning of on,both,before

Possible options: off on both before, default off

cost(Strategy) *option*

How to use costs as priorities

This orders the variables in order of their absolute costs - with largest cost ones being branched on first. This primitive strategy can be surprsingly effective. The column order option is obviously not on costs but easy to code here.

Possible options: off pri(orities) column(Order?) 01f(irst?) 01l(ast?) length(?), default off

cuts(OnOff) *option*

Switches all cuts on or off

This can be used to switch on or off all cuts (apart from Reduce and Split). Then you can do individual ones off or on See branchAndCut for information on options.

Possible options: off on root ifmove forceOn, default on

Dins *option*

This switches on Distance induced neighborhood Search. See Rounding for meaning of on,both,before

Possible options: off on both before often, default off

DivingS(ome) *option*

This switches on a random diving heuristic at various times. C - Coefficient, F - Fractional, G - Guided, L - LineSearch, P - PseudoCost, V - VectorLength. You may prefer to use individual on/off See Rounding for meaning of on,both,before

Possible options: off on both before, default off

DivingC(oefficient) *option*

Whether to try DiveCoefficient

Possible options: off on both before, default on

DivingF(ractional) *option*

Whether to try DiveFractional

Possible options: off on both before, default off

DivingG(uided) *option*

Whether to try DiveGuided

Possible options: off on both before, default off

DivingL(ineSearch) *option*

Whether to try DiveLineSearch

Possible options: off on both before, default off

DivingP(seudoCost) *option*

Whether to try DivePseudoCost

Possible options: off on both before, default off

DivingV(ectorLength) *option*

Whether to try DiveVectorLength

Possible options: off on both before, default off

feas(ibilityPump) *option*

This switches on feasibility pump heuristic at root. This is due to Fischetti, Lodi and Glover and uses a sequence of Lps to try and get an integer feasible solution. Some fine tuning is available by passFeasibilityPump and also pumpTune. See Rounding for meaning of on,both,before

Possible options: off on both before, default on

flow(CoverCuts) *option*

This switches on flow cover cuts (either at root or in entire tree)

See branchAndCut for information on options.

Possible options: off on root ifmove forceOn onglobal, default ifmove

gomory(Cuts) *option*

Whether to use Gomory cuts

The original cuts - beware of imitations! Having gone out of favor, they are now more fashionable as LP solvers are more robust and they interact well with other cuts. They will almost always give cuts (although in this executable they are limited as to number of variables in cut). However the cuts may be dense so it is worth experimenting (Long allows any length). See branchAndCut for information on options.

Possible options: off on root ifmove forceOn onglobal forceandglobal forceLongOn long, default ifmove

greedy(Heuristic) *option*

Whether to use a greedy heuristic

Switches on a greedy heuristic which will try and obtain a solution. It may just fix a percentage of variables and then try a small branch and cut run. See Rounding for meaning of on,both,before

Possible options: off on both before, default on

heur(isticsOnOff) *option*

Switches most heuristics on or off

Possible options: off on, default on

knapsack(Cuts) *option*

This switches on knapsack cuts (either at root or in entire tree)

Possible options: off on root ifmove forceOn onglobal forceandglobal, default ifmove

lift(AndProjectCuts) *option*

Whether to use Lift and Project cuts

Possible options: off on root ifmove forceOn, default off

local(TreeSearch) *option*

This switches on a local search algorithm when a solution is found. This is from Fischetti and Lodi and is not really a heuristic although it can be used as one. When used from Coin solve it has limited functionality. It is not switched on when heuristics are switched on.

Possible options: off on, default off

mixed(IntegerRoundingCuts) *option*

This switches on mixed integer rounding cuts (either at root or in entire tree) See branchAndCut for information on options.

Possible options: off on root ifmove forceOn onglobal, default ifmove

naive(Heuristics) *option*

Really silly stuff e.g. fix all integers with costs to zero!. Do option does heuristic before preprocessing

Possible options: off on both before, default off

node(Strategy) *option*

What strategy to use to select nodes

Normally before a solution the code will choose node with fewest infeasibilities. You can choose depth as the criterion. You can also say if up or down branch must be done first (the up down choice will carry on after solution). Default has now been changed to hybrid which is breadth first on small depth nodes then fewest.

Possible options: hybrid fewest depth upfewest downfewest updepth downdepth, default fewest

pivotAndC(omplement) *option*

Whether to try Pivot and Complement heuristic

Possible options: off on both before, default off

pivotAndF(ix) *option*

Whether to try Pivot and Fix heuristic

Possible options: off on both before, default off

preprocess *option*

This tries to reduce size of model in a similar way to presolve and it also tries to strengthen the model - this can be very useful and is worth trying. Save option saves on file presolved.mps. equal will turn <= cliques into ==. sos will create sos sets if all 0-1 in sets (well one extra is allowed) and no overlaps. trysos is same but allows any number extra. equalall will turn all valid inequalities into equalities with integer slacks.

Possible options: off on save equal sos trysos equalall strategy aggregate forcesos, default sos

probing(Cuts) *option*

This switches on probing cuts (either at root or in entire tree) See branchAndCut for information on options. but strong options do more probing

Possible options: off on root ifmove forceOn onglobal forceonglobal forceOnBut forceOnStrong forceOnButStrong strongRoot, default forceOnStrong

rand(omizedRounding) *option*

Whether to try randomized rounding heuristic

Possible options: off on both before, default off

reduce(AndSplitCuts) *option*

This switches on reduce and split cuts (either at root or in entire tree) See branchAndCut for information on options.

Possible options: off on root ifmove forceOn, default off

residual(CapacityCuts) *option*

Residual capacity cuts. See branchAndCut for information on options.

Possible options: off on root ifmove forceOn, default off

Rens *option*

This switches on Relaxation enforced neighborhood Search. on just does 50 nodes 200 or 1000 does that many nodes. Doh option does heuristic before preprocessing

Possible options: off on both before 200 1000 10000 dj djbefore, default off

Rins *option*

This switches on Relaxed induced neighborhood Search. Doh option does heuristic before preprocessing

Possible options: off on both before often, default on

round(ingHeuristic) *option*

This switches on a simple (but effective) rounding heuristic at each node of tree. On means do in solve i.e. after preprocessing, Before means do if doHeuristics used, off otherwise, and both means do if doHeuristics and in solve.

Possible options: off on both before, default on

two(MirCuts) *option*

This switches on two phase mixed integer rounding cuts (either at root or in entire tree) See branchAndCut for information on options.

Possible options: off on root ifmove forceOn onglobal forceandglobal forceLongOn, default root

Vnd(VariableNeighborhoodSearch) *option*

Whether to try Variable Neighborhood Search

Possible options: off on both before intree, default off

Actions:

barr(ier)	Solve using primal dual predictor corrector algorithm
dualS(implex)	Do dual simplex algorithm
either(Simplex)	Do dual or primal simplex algorithm
initialS	Solve to continuous
	This just solves the problem to continuous - without adding any cuts
outDup	takes duplicate rows etc out of integer model
primalS	Do primal simplex algorithm
reallyS	Scales model in place
stat	Print some statistics
tightLP	Poor person's preSolve for now

Branch and Cut actions:

branch	Do Branch and Cut

6.2 Selected GLPK parameters

The following parameters are taken from the GLPK command line help.

Only the GLPK parameters that are useful in a CMPL context are described afterwards.

Usage GLPK parameters:

```
%opt glpk solverOption [solverOptionValue]
```

General options:

simplex	use simplex method (default)
interior	use interior point method (LP only)
scale	scale problem (default)
noscale	do not scale problem
ranges *filename*	write sensitivity analysis report to filename in
	printable format (simplex only)

tmlim *nnn*	limit solution time to nnn seconds
memlim *nnn*	limit available memory to nnn megabytes
wlp *filename*	write problem to filename in CPLEX LP format
wglp *filename*	write problem to filename in GLPK format
wcnf *filename*	write problem to filename in DIMACS CNF-SAT format
log *filename*	write copy of terminal output to filename

LP basis factorization options:

luf	LU + Forrest-Tomlin update (faster, less stable; default)
cbg	LU + Schur complement + Bartels-Golub update (slower, more stable)
cgr	LU + Schur complement + Givens rotation update (slower, more stable)

Options specific to simplex solver:

primal	use primal simplex (default)
dual	use dual simplex
std	use standard initial basis of all slacks
adv	use advanced initial basis (default)
bib	use Bixby's initial basis
steep	use steepest edge technique (default)
nosteep	use standard "textbook" pricing
relax	use Harris' two-pass ratio test (default)
norelax	use standard "textbook" ratio test
presol	use presolver (default; assumes scale and adv)
nopresol	do not use presolver
exact	use simplex method based on exact arithmetic
xcheck	check final basis using exact arithmetic

Options specific to interior-point solver:

nord	use natural (original) ordering
qmd	use quotient minimum degree ordering

amd	use approximate minimum degree ordering (default)
symamd	use approximate minimum degree ordering

Options specific to MIP solver:

nomip	consider all integer variables as continuous (allows solving MIP as pure LP)
first	branch on first integer variable
last	branch on last integer variable
mostf	branch on most fractional variable
drtom	branch using heuristic by Driebeck and Tomlin (default)
pcost	branch using hybrid pseudocost heuristic (may be useful for hard instances)
dfs	backtrack using depth first search
bfs	backtrack using breadth first search
bestp	backtrack using the best projection heuristic
bestb	backtrack using node with best local bound (default)
intopt	use MIP presolver (default)
nointopt	do not use MIP presolver
binarize	replace general integer variables by binary ones (assumes intopt)
fpump	apply feasibility pump heuristic
gomory	generate Gomory's mixed integer cuts
mir	generate MIR (mixed integer rounding) cuts
cover	generate mixed cover cuts
clique	generate clique cuts
cuts	generate all cuts above
mipgap *tol*	set relative mip gap tolerance to tol
minisat	translate integer feasibility problem to CNF-SAT and solve it with MiniSat solver
objbnd *bound*	add inequality obj <= bound (minimization) or obj >= bound (maximization) to integer feasibility problem (assumes minisat)

References

- Achterberg, T. 2009. *SCIP - solving constraint integer programs.* Mathematical Programming Computation Volume 1 Number 1. 1–41.
- Coulouris, G.F.; J. Dollimore, T. Kindberg, G. Blai. 2012. *Distributed Systems : Concepts and Design*, 5th ed., Addison-Wesley.
- Fourer, R., D. M. Gay, B. W. Kernighan. 2003. *AMPL: A Modeling Language for Mathematical Programming*, 2nd ed. Duxbury Press, Pacific Grove, CA.
- Anderson, D. R., D. J. Sweeney, Th. A. Williams, K. Martin. 2011. *An Introduction to Management Science : Quantitative Approaches to Decision Making.* 13th ed.. South-Western.
- Fourer, R, J. Ma, R. K. Martin. 2010. *Optimization Services: A Framework for Distributed Optimization.* Operations Research 58(6). 1624-1636.
- GLPK. 2014. *GNU Linear Programming Kit Reference Manual for GLPK Version 4.54.*
- Hillier, F. S., G. J. Lieberman. 2010. *Introduction to Operations Research.* 9th ed.. McGraw-Hill Higher Education.
- Foster, I., C. Kesselman (editors). 2004. *The Grid2: 2nd Edition: Blueprint for a New Computing Infrastructure*, Kindle ed., Morgan Kaufmann Publishers Inc.
- Kshemkalyani, A.D., M. Singhal, M. 2008. *Distributed Computing – Principles, Algorithms, and Systems*, Kindle ed., Cambridge University Press.
- St. Laurent, S., J. Johnston, E. Dumbill. 2001. *Programming Web Services with XML-RPC*, 1st ed., O'Reilly.
- Rogge, R., M. Steglich. 2007. *Betriebswirtschaftliche Entscheidungsmodelle zur Verfahrenswahl sowie Auflagen- und Lagerpolitiken*, Diskussionsbeiträge zu Wirtschaftsinformatik und Operations Research 10/2007, Martin-Luther-Universität Halle-Wittenberg.