

# ZIMPL User Guide

(Zuse Institute Mathematical Programming Language)

Thorsten Koch

for Version 2.02  
15. May 2004

## Contents

<b>1</b>	<b>Preface</b>	<b>3</b>
<b>2</b>	<b>Introduction</b>	<b>4</b>
<b>3</b>	<b>Invocation</b>	<b>5</b>
<b>4</b>	<b>Format</b>	<b>6</b>
4.1	Expressions . . . . .	7
4.2	Sets . . . . .	8
4.3	Parameters . . . . .	11
4.4	Variables . . . . .	12
4.5	Objective . . . . .	12
4.6	Constraints . . . . .	13
4.7	Details on <code>sum</code> and <code>forall</code> . . . . .	13
4.8	Details on <code>if</code> in constraints . . . . .	13
4.9	Initializing sets and parameters from a file . . . . .	14
4.10	Function Definitions . . . . .	15
4.11	Extended Constraints (experimental) . . . . .	16
4.12	Extended Functions (experimental) . . . . .	16
4.13	The <code>do print</code> and <code>do check</code> commands . . . . .	17
<b>5</b>	<b>Examples</b>	<b>18</b>
5.1	Diet problem . . . . .	18
5.2	TSP . . . . .	20
5.3	Capacitated Facility Location Problem . . . . .	21
5.4	$n$ -Queens Problem . . . . .	23



### Abstract

ZIMPL is a little language to translate the mathematical model of a problem into a linear or (mixed-)integer mathematical program expressed in LP or MPS file format which can be read by a LP or MIP solver.

## 1 Preface

*May the source be with you, Luke!*

Most of the things in ZIMPL (and a lot more) can be found in the excellent book about the modeling language AMPL from Robert Fourer, David N. Gay and Brian W. Kernighan [FGK03]. But having the source code of a program has its advantages. The possibility to run it regardless of architecture and operating system, the ability to modify it to suite the needs, and not having to hassle with license managers may make a much less powerful program the better choice. And so ZIMPL came into being.

By now ZIMPL has grown up and matured. It has been used in several industry projects and university lectures, showing that it is able to cope with large scale models and also with students. This would have not been possible without my early adopters Armin Fügenschuh, Marc Pfetsch, Sascha Lukac, Daniel Junglas and Tobias Achterberg. Thanks for there comments and bug reports.

ZIMPL is licensed under the GNU general public license version 2. For more information on free software see <http://www.gnu.org>. The latest version of ZIMPL can be found at <http://www.zib.de/koch/zimpl>. If you find any bugs, please send an email to <mailto:koeh@zib.de>. But do not forget to include an example that shows the problem. If somebody extends ZIMPL, I am interested in getting patches to include them in the main distribution.

## 2 Introduction

A linear program (LP) might look like this:

$$\begin{array}{ll} \min & 2x + 3y \\ \text{subject to} & x + y \leq 6 \\ & x, y \geq 0 \end{array}$$

The usual format to store the description of such a problem is MPS invented by IBM [IBM97] long ago. Nearly all available LP and MIP solvers can read this format. While MPS is a nice format to punch into a punch card and at least a reasonable format to read for a computer it is quite unreadable for humans.

```
NAME          ex1.mps
ROWS
  N  OBJECTIV
  L  c1
COLUMNS
  x      OBJECTIV      2
  x      c1             1
  y      OBJECTIV      3
  y      c1             1
RHS
  RHS    c1             6
ENDATA
```

Another possibility is the LP format [ILO02], which is more readable<sup>1</sup> but is only supported by a few solvers.

```
Minimize
  cost:  +2 x +3 y
Subject to
  c1:  +1 x +1 y <= 6
End
```

But since each coefficient of the matrix  $A$  must be stated explicitly it is also not a desirable choice to develop a mathematical model.

Now, with ZIMPL it is possible to write this:

```
var x;
var y;
minimize cost: 2 * x + 3 * y;
subto c1: x + y <= 6;
```

---

<sup>1</sup> The LP format has also some idiosyncratic restrictions. For example variables should not be named e12 or the like. And it is not possible to specify ranged constraints.

and have it automatically translated into MPS or LP format. While this looks not much different from what is in the LP format, the difference can be seen, if we use indexed variables. Here is an example. This is the LP

$$\begin{array}{ll}\min & 2x_1 + 3x_2 + 1.5x_3 \\ \text{subject to} & \sum_{i=1}^3 x_i \leq 6 \\ & x_i \geq 0\end{array}$$

And this is how to tell it to ZIMPL

```
set I      := { 1 to 3 };
param c[I] := <1> 2, <2> 3, <3> 1.5;
var  x[I] >= 0;
minimize value: sum <i> in I : c[i] * x[i];
subto      cons: sum <i> in I : x[i] <= 6;
```

### 3 Invocation

To run ZIMPL on a model given in the file `ex1.zpl` type the command

```
zimpl ex1.zpl
```

The general case is `zimpl [options] <input-files>`.

It is possible to give more than one input file. They are read one after the other as if they were all one big file. If any error occurs while processing, ZIMPL will print out an error message and abort. In case everything goes well, the results will be written into two or three files, depending on the options specified.

The first file is the problem generated from the model in either LP or MPS format with extension `.lp` or `.mps`. The next one is the “table” file, which has the extension `.tbl`. This file lists all variable and constraint names used in the model and their corresponding names in the problem file.

The reason for this name translation is that the MPS format can only handle names up to eight characters long. Also the the LP format restricts the length of names to 16 characters.

The third file is an optional CPLEX branching order file.

The following options are possible (only the first two are normally of interest):

-t <i>format</i>	Selects the output format. Can be either <code>lp</code> which is default, or <code>mps</code> or <code>hum</code> which is only human readable.
-o <i>name</i>	Sets the base-name for the output files. Defaults to the name of the first input file stripped off its path and extension.
-F <i>filter</i>	The output is piped through a filter. A <code>%s</code> in the string is replaced by the output filename. For example <code>-F "gzip -c &gt;%s.gz"</code> would compress all the output files.
-n <i>cform</i>	Select the format for the generation of constraint names. Can be either <code>cm</code> which will number them $1 \dots n$ with a 'c' in front. <code>cn</code> will use the name supplied in the <code>subto</code> statement and number them $1 \dots n$ within the statement. <code>cf</code> will use the name given with the <code>subto</code> , then a $1 \dots n$ number like in <code>cm</code> and then append all the local variables from the <code>forall</code> statements.
-v <i>l..5</i>	Set the verbosity level. 0 is quiet, 1 is default, 2 is verbose, 3 is chatter, and 5 is debug.
-D <i>name=val</i>	Sets the parameter <i>name</i> to the specified value. This is equivalent with having this line in the ZIMPL program: <code>param name:=val.</code>
-b	Enables bison debugging output.
-f	Enables flex debugging output.
-h	Prints a help message.
-m	Writes a CPLEX <code>mst Mip S</code> Start file.
-O	Optimize the generated LP by doing some presolve analysis.
-r	Writes a CPLEX <code>ord</code> branching order file.
-V	Prints the version number.

A typical invocation is for example:

```
zimpl -o hardone -t mps data.zpl model.zpl
```

This reads the files `data.zpl` and `model.zpl` and produces `hardone.mps` and `hardone.tbl`.

If MPS-output is specified for a maximization problem, the objective function will be inverted.

## 4 Format

Each ZPL-file consists of six types of statements:

- ▶ Sets
- ▶ Parameters
- ▶ Variables

- Objective
- Constraints
- Function definitions

Each statement ends with a semicolon `;`. Everything from a number-sign `#` to the end of the line is treated as a comment and is ignored.

If a line starts with the word `include` followed by a filename in double quotation marks then this file is read instead of the line.

## 4.1 Expressions

ZIMPL works on its lowest level with two types of data: Strings and numbers.

Wherever a number or string is required, it is also possible to use a parameter of the corresponding value type. Usually, expressions are allowed instead of just a number or a string. The precedence of operators should be the usual one, but parenthesis can always be used to specify the evaluation order explicitly. If in doubt use parenthesis.

### Numeric expressions

A number in ZIMPL can be given in the usual format, e. g. as 2, -6.5 or 5.234e-12. Numeric expressions consist of numbers, numeric valued parameters, and any of the following operators and functions:

$a^b$ , <code>a**b</code>	$a$ to the power of $b$	$a^b$
<code>a+b</code>	addition	$a + b$
<code>a-b</code>	subtraction	$a - b$
<code>a*b</code>	multiplication	$a \cdot b$
<code>a/b</code>	division	$a/b$
<code>a mod b</code>	modulo	$a \bmod b$
<code>a div b</code>	integer division	
<code>abs(a)</code>	absolute value	$ a $
<code>floor(a)</code>	round down	$\lfloor a \rfloor$
<code>ceil(a)</code>	round up	$\lceil a \rceil$
<code>a!</code>	factorial	$a!$
<code>min(S)</code>	minimum of a set	$\min_{s \in S}$
<code>max(S)</code>	maximum of a set	$\max_{s \in S}$
<code>min(a,b,c,...,n)</code>	minimum of a list	$\min(a, b, c, \dots, n)$
<code>max(a,b,c,...,n)</code>	maximum of a list	$\max(a, b, c, \dots, n)$
<code>card(S)</code>	cardinality of a set	$ S $
<code>ord(A,n,c)</code>	ordinal	$c$ -th component of the $n$ -th element of set $A$ .
<code>if a then b else c</code>	conditional	

The following functions are only computed with normal double precision floating point arithmetic. So be careful:

<code>sqrt(a)</code>	square root	$\sqrt{a}$
<code>log(a)</code>	logarithm to base 10	$\log_{10} a$
<code>ln(a)</code>	natural logarithm	$\ln a$
<code>exp(a)</code>	exponential function	$e^a$

### String expressions

A string is delimited by double quotation marks ", e.g., "Hallo".

### Variant expressions

The following is either a numeric or a string expression, depending if *expression* is a string or a numeric expression.

*if boolean-expression then expression else expression end.*

The same is true for the `ord(set,tuple-number,component-number)` function, which evaluates to a specific element of a set. Please note, that sets have no specific order.

### Boolean expressions

These evaluate either to *true* or *false*. For numbers and strings the relational operators `<`, `<=`, `=`, `!=`, `>=`, and `>` are defined. Combinations of boolean expressions with `and`, `or`, and `xor`<sup>2</sup> and negation with `not` are possible. The expression *tuple in set-expression* can be used to test set membership of a tuple.

## 4.2 Sets

Sets consist of tuples. The tuples in a sets are unordered, i.e., each tuple can only be once in a set. Sets are delimited by braces, { and }, resp. Tuples consist of components. The components are ordered. Each tuple of a specific set has the same number of components. The components are either numbers or strings. The type of the *n*-th component for all tuples of a set must be the same. A tuple starts and ends with `<` and `>`, resp., e.g. `<1, 2, "x">`. The components are separated by commas. If tuples are one-dimensional, it is possible to omit the tuple delimiters in a list of elements, but then they must be omitted from all tuples of the set, e.g. `{1, 2, 3}`.

Sets can be defined with the `set` statement. It consists of the keyword `set`, the name of the set, an assignment operator (`:=`) and a valid set expression.

Sets are referenced by use of an *template* tuple, consisting of placeholders that are replaced by the value of the components of the respective tuple. For example a set *S* consisting of two-dimensional tuples could be referenced by `<a,b> in S`. If any of

---

<sup>2</sup> $a \wedge \bar{b} \vee \bar{a} \wedge b$



the placeholders are actual values, only those tuples will be extracted that match these values. For example `<1,b> in S` will only get those tuples whose first component is 1. Please note that if one of the placeholders is the name of an already defined parameter, set or variable, it will be substituted. This will result either in an error or an actual value.

### Examples

```
set A := { 1, 2, 3 };
set B := { "hi", "ha", "ho" };
set C := { <1,2,"x">, <6,5,"y">, <787,12.6,"oh"> };
```

For set expressions these functions and operators are defined:

<code>A*B,</code>		
<code>A cross B</code>	Cross product	$\{(x,y)   x \in A \wedge y \in B\}$
<code>A+B,</code>		
<code>A union B</code>	Union	$\{x   x \in A \vee x \in B\}$
<code>A inter B</code>	Intersection	$\{x   x \in A \wedge x \in B\}$
<code>A\B, A-B,</code>		
<code>A without B</code>	Difference	$\{x   x \in A \wedge x \notin B\}$
<code>A symdiff B</code>	Symmetric difference	$\{x   (x \in A \wedge x \notin B) \vee (x \in B \wedge x \notin A)\}$
<code>{n..m},</code>		
<code>{n to m by s}</code>	Generate, (default $s = 1$ )	$\{x   x = n + is \leq m, i \in \mathbb{N}_0\}$
<code>proj(A, t)</code>	Projection	The new set will consist of $n$ -tuples, with the $i$ -th component being the $e_i$ -th component of $A$ .
	$t = (e_1, \dots, e_n)$	
<code>if a then</code>		
<code>  b else c</code>	Conditional	

An example for the use of the `if boolean-expression then set expression else set expression end` can be found below with the examples for indexed sets.

### Examples

```
set D := A cross B;
set E := { 6 to 9 } union A without { <2>, <3> };
set F := { 1 to 9 } * { 10 to 19 } * { "A", "B" };
set G := proj(F, <3,1>)
# will give: <"A",1>, <"A",2> ... <"B",9>
```

## Conditional sets

It is possible to restrict a set to tuples that satisfy a boolean expression. The expression given by the `with` clause is evaluated for each tuple in the set and only tuples for which the expression evaluated to `true` are included in the new set.

## Examples

```
set F := { <i,j> in Q with i > j and i < 5 };
set A := { "a", "b", "c" };
set B := { 1, 2, 3 };
set V := { <a,2> in A*B with a == "a" or a == "b" };
# will give: <"a",2>, <"b",2>
```

## Indexed sets

It is possible to index one set with another set. Here is a list of functions that use this:

<code>powerset(A)</code>	Generates all subsets of $A$	$\{X   X \subseteq A\}$
<code>subset(A, n)</code>	Generates all subsets of $A$ with $n$ elements	$\{X   X \subseteq A \wedge  X  = n\}$
<code>indexset(A)</code>	The index set of $A$	$\{1 \dots  A \}$

Indexed sets are accessed by adding the index of the set in brackets `[` and `]`, like `S[7]`. There are three possibilities how to assign to an indexed set:

- ▶ The assignment expression is a list of comma separated pairs, consisting of a tuple from the index set and a set expression to assign.
- ▶ A set reference expression is given as index, then the assignment expression is evaluated for each index tuple.
- ▶ By use of a function that returns an indexed set.

## Examples

```
set I          := { 1..3 };
set A[I]       := <1> { "a", "b" },
                 <2> { "c", "e" }, <3> { "f" };
set B[<i> in I] := { 3 * i };
set P[]        := powerset(I);
set J          := indexset(P);
set S[]        := subset(I, 2);
set K[<i> in I] := if i mod 2 == 0 then { i } else { -i } end;
```

### 4.3 Parameters

Parameters can be declared with or without an indexing set. Without indexing the parameter is just one value, which is either a number or a string. For indexed parameters there is one value for each member of the set. It is possible to declare a *default* value.

Parameters are declared in the following way: The keyword `param` is followed by the name of the parameter optionally followed by the indexing set. Then after the assignment sign comes a list of pairs. The first element of each pair is a tuple from the index set, the second element is the value of the parameter for this index.

#### Examples

```
set A := { 12 .. 30 };
set C := { <1,2,"x">, <6,5,"y"> };
param q := 5;
param u[A] := <1> 17, <2> 29, <3> 12 default 99;
param w[C] := <1,2,"x"> 1/2, <6,5,"y"> 2/3;
param x[<i> in { 1 .. 8 } with i mod 2 == 0] := 3 * i;
```

In the example, no value is given for index `<787,12.6,"oh">` of parameter `w`, that is assignments need not to be complete. This is correct as long as it is never referenced.

#### Parameter tables

It is possible to initialize multi-dimensional indexed parameters from tables. This is especially useful for two-dimensional parameters. The data is put in a table structure with `|` signs on each margin. Then a head line with column indices has to be added, and one index for each row of the table is needed. The column index has to be one-dimensional, but the row index can be multi-dimensional. The complete index for the entry is built by appending the column index to the row index. The value entries are separated by commas. Any valid expression is allowed here. As can be seen in the third example below, it is possible to add a list of entries after the table.

#### Examples

```
set I := { 1 .. 10 };
set J := { "a", "b", "c", "x", "y", "z" };

param h[I*J] :=
    | "a", "c", "x", "z" |
    |1| 12, 17, 99, 23 |
    |3| 4, 3, -17, 66*5.5 |
    |5| 2/3, -.4, 3, abs(-4) |
    |9| 1, 2, 0, 3 | default -99;

param g[I*I*I] :=
    | 1, 2, 3 |
```

```
| 1,1 | 0, 1, 0 |  
| 1,2 | 1, 1, 1 |  
| 1,3 | 0, 0, 1 |  
| 2,1 | 1, 0, 1 |;
```

```
param k[I*I] := | 7, 8, 9 |  
                | 4 | 89, 67, 55 |  
                | 5 | 12, 13, 14 |, <1,2> 17, <3,4> 99;
```

## 4.4 Variables

Like parameters, variables can be indexed. A variable has to be one out of three possible types: Continuous (called *real*), binary or integer. The default is real. Variables may have lower and upper bounds. Defaults are zero as lower and infinity as upper bound. Binary variables are always bounded between zero and one. It is possible to compute the value of the lower or upper bounds depending on the index for the variable (see last declaration in the example). Bounds can also be set to *infinity* and *-infinity*.

### Examples

```
var x1;  
var x2 binary;  
var y[A] real >= 2 <= 18;  
var z[<a,b,c> in C] integer  
    >= a * 10  
    <= if b <= 3 then p[b] else 10 end;
```

Remember: if nothing is specified a lower bound of zero is assumed.

## 4.5 Objective

There must be at most one objective statement in a model. The objective can be either minimize or maximize. Following the keyword is a name, a colon (:) and then a term consisting of variables.

### Example

```
minimize cost: 12 * x1 -4.4 * x2  
    + sum <a> in A : u[a] * y[a]  
    + sum <a,b,c> in C with a in E and b > 3 : -a/2 * z[a,b,c];
```

## 4.6 Constraints

The general format for a constraint is `subto name: term sense term`. Name can be any name starting with a letter. The term is defined as in the objective. Sense is one of `<=`, `>=` and `==`. Many constraints can be generated with one statement by the use of the `forall` instruction, see below.

### Examples

```
subto time : 3 * x1 + 4 * x2 <= 7;
subto space: sum <a> in A : 2 * u[a] * y[a] >= 50;
subto weird: forall <a> in A :
    sum <a,b,c> in C : z[a,b,c] == 55;
subto c21: 6 * (sum <i> in A : x[i]
    + sum <j> in B : y[j]) >= 2;
subto c40: x[1] == a[1] +
    2 * sum <i> in A do 2*a[i]*x[i]*3 + 4;
```

## 4.7 Details on `sum` and `forall`

The general forms are

`forall index do term` and `sum index do term`.

It is possible to nest several `forall` instructions. The general form of *index* is *tuple* `in set` with *boolean-expression*.

It is allowed to write a colon (`:`) instead of `do` and a vertical bar (`|`) instead of `with`. The number of components in the *tuple* and in the components of the members of the *set* must match. The *with* part of an *index* is optional. The *set* can be any expression giving a set.

### Examples

```
forall <i,j> in X cross { 1 to 5 } without { <2,3> }
    with i > 5 and j < 2 do
        sum <i,j,k> in X cross { 1 to 3 } cross Z do
            p[i] * q[j] * w[j,k] >= if i == 2 then 17 else 53;
```

Note that in the example *i* and *j* are set by the `forall` instruction. So they are fixed for all invocations of `sum`.

## 4.8 Details on `if` in constraints

It is possible to put two variants of a constraint into an `if`-statement. The same applies for *terms*. It is also possible to have a `forall` statement inside the result part of an `if`.

### Examples

```
subto c1: forall <i> in I do
  if (i mod 2 == 0) then 3 * x[i] >= 4
                        else -2 * y[i] <= 3 end;
subto c2: sum <i> in I :
  if (i mod 2 == 0) then 3 * x[i] else -2 * y[i] end <= 3;
```

## 4.9 Initializing sets and parameters from a file

It is possible to load the values for a set or a parameter from a file. The syntax is

```
read filename as template [skip n] [use n] [fs s] [comment s]
```

*filename* is the name of the file to read.

*template* is a string with a template for the tuples to generate. Each input line from the file is split into fields. The splitting is done according to the following rules: Whenever a space, tab, comma, semicolon or double colon is encountered a new field is started. Text that is enclosed in double quotes is not split, the quotes are always removed. When a field is split all space and tab characters around the splitting are removed. If the split is due to a comma, semicolon or double colon, each occurrence of these characters starts a new field.

### Examples

All these lines have three fields:

```
Hallo;12;3
Moin 7 2
"Hallo, Peter"; "Nice to meet you" 77
,,2
```

For each component of the tuple, the number of the field to use for the value is given, followed by either an *n* if the field should be interpreted as a number or *s* for a string. Have a look at the example, it is quite obvious how it works. After the template some optional modifiers can be given. The order does not matter.

*skip n* instructs to skip the first *n* lines of the file.

*use n* limits the number of lines to use to *n*.

*comment s* sets a list of characters that start comments in the file. Each line is ended when any of the comment characters is found.

When a file is read, empty lines are skipped and not counted for the *use* clause. They are counted for the *skip* clause.

### Examples

```
set P := { read "nodes.txt" as "<ls>" };
```

```
nodes.txt:
    Hamburg    ->    <"Hamburg">
    München    ->    <"München">
    Berlin     ->    <"Berlin">

set Q := { read "blabla.txt" as "<1s,5n,2n>" skip 1 use 2 };

blabla.txt:
    Name;Nr;X;Y;No      ->    skip
    Hamburg;12;x;y;7    ->    <"Hamburg",7,12>
    Bremen;4;x;y;5      ->    <"Bremen",5,4>
    Berlin;2;x;y;8      ->    skip

param cost[P] := read "cost.txt" as "<1s> 2n" comment "#";

cost.txt:
    # Name Price      ->    skip
    Hamburg 1000      ->    <"Hamburg"> 1000
    München 1200      ->    <"München"> 1200
    Berlin  1400      ->    <"Berlin">  1400

param cost[Q] := read "haha.txt" as "<3s,1n,2n> 4s";

haha.txt:
    1:2:ab:con1      ->    <"ab",1,2> "con1"
    2:3:bc:con2      ->    <"bc",2,3> "con1"
    4:5:de:con3      ->    <"de",4,5> "con1"
```

As with table format input, it is possible to add a list of tuples or parameter entries after a read statement.

## Examples

```
set A := { read "test.txt" as "<2n>", <5>, <6> };
param winniepoh[X] :=
    read "values.txt" as "<1n,2n> 3n", <1,2> 17, <3,4> 29;
```

## 4.10 Function Definitions

It is possible to define functions within ZIMPL. The value a function returns has to be either a number, a string or a set. The arguments of a function can only be numbers or strings, but within the function definition it is possible to access all otherwise declared sets, parameters and variables.

The definition of a function has to start with `defnumb`, `defstrg` or `defset`, depending on the return value. Then follows the name of the function and a list of

argument names put in parenthesis.

After this comes an assignment operator ( $:=$ ) and a valid expression or set expression.

### Examples

```
defnumb dist(a,b)  := a*a + b*b;
defstrg huehott(a) := if a < 0 then "hue" else "hott" end;
defset  bigger(i)  := { <j> in K with j > i };
```

## 4.11 Extended Constraints (experimental)

ZIMPL has the possibility to generate systems of constraints that mimic conditional constraints. The general syntax is as follows, note that the `else` part is optional:

*vif boolean-constraint then constraint [ else constraint ] end*

where *boolean-constraint* consists of linear expression involving variables. All these variables have to be bounded integer or binary variables. It is not possible to use any continuous variables or integer variables with infinite bounds in a boolean-constraint. All comparison operators ( $<$ ,  $\leq$ ,  $==$ ,  $!=$ ,  $\geq$ ,  $>$ ) are allowed. Also combination of several terms with `and`, `or`, and `xor` and negation with `not` is possible. The conditional constraints (those which follow after `then` or `else`, may include bounded continuous variables.

Be aware that using this construct will lead to the generation of several additional constraints and variables.

### Examples

```
var x[i] integer >= 0 <= 20;

subto c1: vif 3 * x[1] + x[2] != 7
  then sum <i> in I : y[i] <= 17
  else sum <k> in K : z[k] >= 5 end;

subto c2: vif x[1] == 1 and x[2] > 5 then x[3] == 7 end;

subto c3: forall <i> in I :
  vif x[i] >= 2 then x[i + 1] <= 4 end;
```

## 4.12 Extended Functions (experimental)

It is possible to use special functions on terms with variables that will automatically converted to a system of inequalities. The arguments of these functions have to be linear terms consisting of bounded integer or binary variables.



The following functions are defined:

`vabs(t)` Absolute value  $|t|$

Be aware that using this construct will lead to the generation of several additional constraints and variables.

### Examples

```
var x[i] integer >= -5 <= 5;

subto c1: vabs(x[1]) >= 5;
subto c2: vabs(sum <i> in I : x[i]) <= 15;
subto c2: vif vabs(x[1] + x[2]) > 2 then x[3] == 7 end;
```

## 4.13 The `do print` and `do check` commands

The `do` command is special. It has two possible incarnations:

`print` and `check`. `print` will print to the standard output stream whatever numerical, string, Boolean or set expression, or tuple follows it. This can be used for example to check if a set has the expected members, or if some computation has the anticipated result.

`check` always precedes a Boolean expression. If this expression does not evaluate to *true*, the program is aborted with an appropriate error message. This can be used to assert that specific conditions are met.

It is possible to use a `forall` clause before a `print` oder `check` statement.

### Examples

```
set I := { 1..10 };

do print I;
do forall <i> in I with i > 5 do print sqrt(i);

do forall <p> in P do check sum <p,i> in PI : 1 >= 1;
```

## 5 Examples

In this section we will show examples how to translate a problem into ZIMPL format.

### 5.1 Diet problem

This is the first example in [Chv83, Chapter 1, page 3]. It is a classic so-called *diet*-problem, see for example [Dan90] about the practice implications.

Given a set of foods  $F$  and a set of nutrients  $N$ , we have a table  $\pi_{fn}$  of the amount of nutrient  $n$  in food  $f$ . Now  $\Pi_n$  defines how much intake of each nutriment is needed. And  $\Delta_f$  describes the maximum number of servings of each food.

Now given prices  $c_f$  for each food, we have to find a selection of foods that obeys the restrictions and has minimal cost.

$$\min_{f \in F, n \in N} c_{fn} x_{fn} \quad \text{subject to} \quad (1)$$

$$\sum_{f \in F} \pi_f x_{fn} \geq \Pi_n \quad \forall n \in N \quad (2)$$

$$0 \leq x_{fn} \leq \Delta_f \quad \forall f \in F, n \in N \quad (3)$$

$$x_{fn} \in \mathbb{N} \quad (4)$$

As (4) implies, only complete servings can be obtained. Half an egg is not an option. Now translating this to ZIMPL looks as follows:

```
set Food := { "Oatmeal", "Chicken", "Eggs", "Milk", "Pie", "Pork" };
set Nutrients := { "Energy", "Protein", "Calcium" };
set Attr      := Nutrients + { "Servings", "Price" };
```

```
param needed[Nutrients] :=
  <"Energy"> 2000, <"Protein"> 55, <"Calcium"> 800;
```

```
param data[Food * Attr] :=
  | "Servings", "Energy", "Protein", "Calcium", "Price" |
  | "Oatmeal"   | 4 , 110 , 4 , 2 , 3 |
  | "Chicken"   | 3 , 205 , 32 , 12 , 24 |
  | "Eggs"      | 2 , 160 , 13 , 54 , 13 |
  | "Milk"      | 8 , 160 , 8 , 284 , 9 |
  | "Pie"       | 2 , 420 , 4 , 22 , 20 |
  | "Pork"      | 2 , 260 , 14 , 80 , 19 |;
# (kcal) (g) (mg) (cents)
var x[<f> in Food] integer >= 0 <= data[f, "Servings"];
```

```
minimize cost: sum <f> in Food : data[f, "Price"] * x[f];
```

```
subto need :
```

## ZIMPL

---

```
forall <n> in Nutrients do  
  sum <f> in Food : data[f, n] * x[f] >= needed[n];
```

## 5.2 TSP

In this example we show how to generate an exponential description of the *Traveling Salesmen Problem* as given for example in [Sch03, Section 58.5]. The data is read in from a file that gives the number of the city and the x and y coordinate. Distances between cities are geometric. A suitable data file would look like this:

```
#City      x y
"Sylt"      1 1
"Flensburg" 3 1
"Neumünster" 2 2
"Husum"     1 3
"Schleswig" 3 3
"Ausacker"  2 4
```

The formulation in ZIMPL follows below. Please note that  $P[ ]$  holds all subsets of the cities. So don't try to solve a 52 city TSP this way. It won't work.

```
set V      := { read "tsp.dat" as "<ls>" comment "#" };
set E      := { <i,j> in V * V with i < j };
set P[]    := powerset(V);
set K      := indexset(P);

param px[V] := read "tsp.dat" as "<ls> 2n" comment "#";
param py[V] := read "tsp.dat" as "<ls> 3n" comment "#";

defnomb dist(a,b) := sqrt((px[a]-px[b])^2 + (py[a]-py[b])^2);

var x[E] binary;

minimize cost: sum <i,j> in E : dist(i,j) * x[i, j];

subto two_connected:
  forall <v> in V do
    (sum <v,j> in E : x[v,j])
    + (sum <i,v> in E : x[i,v]) == 2;

subto no_subtour:
  forall <k> in K with
    card(P[k]) > 2 and card(P[k]) < card(V) - 2 do
    sum <i,j> in E with <i> in P[k] and <j> in P[k] : x[i,j]
    <= card(P[k]) - 1;
```

### 5.3 Capacitated Facility Location Problem

Here we have a formulation for the *Capacitated Facility Location Problem*. Of course this is also kind of a *bin packing* problem with packing costs and variable sized bins, or a *cutting stock* problem with cutting costs.

Given a set of possible plants  $P$  to build, and a set of stores  $S$  with a certain demand  $\delta_s$  that has to be satisfied, we have to decide which plant should serve which store. We have costs  $c_p$  for building plant  $p$  and  $c_{ps}$  for transporting the goods from plant  $p$  to store  $s$ . Each plant has only a limited capacity  $\kappa_p$ . And we insist that each store is served by exactly one plant. Of course we are looking for the cheapest solution:

$$\min_{p \in P, s \in S} c_p + c_{ps} \quad \text{subject to} \quad (5)$$

$$\sum_{p \in P} x_{ps} = 1 \quad \forall s \in S \quad (6)$$

$$x_{ps} \leq z_s \quad \forall s \in S, p \in P \quad (7)$$

$$\sum_{s \in S} \delta_s x_{ps} \leq \kappa_p \quad \forall p \in P \quad (8)$$

$$x_{ps}, z_p \in \{0, 1\} \quad (9)$$

We have binary variables  $z_p$ , which are set to one, iff plant  $p$  is to be build. And we have binary variables  $x_{ps}$ , which are set to one iff plant  $p$  serves shop  $s$ . Equation (6) demands that each store is assigned to exactly one plant. Equation (7) makes sure that a plant that serves a shop is built. And Equation (8) restricts the shops that are served by a plant to the plants capacity. Putting this into ZIMPL yields:

```
set PLANTS := { "A", "B", "C", "D" };
set STORES := { 1 .. 9 };
set PS      := PLANTS * STORES;

# How much does it cost to build a plant and what capacity
# will it then have?
param building[PLANTS] := <"A">500,<"B">600,<"C">700,<"D">800;
param capacity[PLANTS] := <"A"> 40,<"B"> 55,<"C"> 73,<"D"> 90;

# Here is the demand of each store
param demand [STORES] := <1> 10,<2> 14,<3> 17,<4> 8,<5> 9,
                        <6> 12,<7> 11,<8> 15,<9> 16;

# Transportation cost from each plant to each store
param transport[PS] :=
    | 1, 2, 3, 4, 5, 6, 7, 8, 9 |
    | "A" | 55, 4, 17, 33, 47, 98, 19, 10, 6 |
    | "B" | 42, 12, 4, 23, 16, 78, 47, 9, 82 |
```

## ZIMPL

---

```
|"C"| 17, 34, 65, 25, 7, 67, 45, 13, 54 |
|"D"| 60, 8, 79, 24, 28, 19, 62, 18, 45 |;

var x[PS]      binary;  # Is plant p suppling store s ?
var z[PLANTS] binary;  # Is plant p build ?

# We want it cheap
minimize cost: sum <p> in PLANTS : building[p] * z[p]
              + sum <p,s> in PS : transport[p,s] * x[p,s];

# Each store is supplied by exactly one plant
subto assign:
  forall <s> in STORES : sum <p> in PLANTS : x[p,s] == 1;

# To be able to supply a store, a plant must be build
subto build: forall <p,s> in PS : x[p,s] <= z[p];

# The plant must be able to meet the demands from all stores
# that are connected to it
subto limit: forall <p> in PLANTS :
  sum <s> in S : demand[s] * x[p,s] <= capacity[p];
```

The optimal solution in this case is to build plants A and C. Stores 2, 3, and 4 are served by plant A, the rest by plant C. Total cost would be 1457.

## 5.4 $n$ -Queens Problem

We now show two formulations of the  $n$ -Queens problems using extended constraints and functions.

The first formulation uses one general integer variable for each row of the board. Each variable can take the value of a column. So we have  $n$  variables with bounds  $1 \dots n$ . Next we use the `vabs` extended function to model an *all different* constraint on the variables (see constraint `c1`). This makes sure that no queen is located on the same column than any other queen. The second constraint (`c2`) is used to block all the diagonals of a queen. This is done by demanding that the absolute row and the column distance of each pair of queens is different. We model  $a \neq b$  by  $\text{abs}(a - b) \geq 1$ .

Note that this formulation only works if a queen can be placed in each row, i.e. the board size has to be at least  $4 \times 4$ .

```
param queens := 8;

set I := { 1 .. queens };
set P := { <i,j> in I * I with i < j };

var x[I] integer >= 1 <= queens;

subto c1: forall <i,j> in P do vabs(x[i] - x[j]) >= 1;
subto c2: forall <i,j> in P do
    vabs(vabs(x[i] - x[j]) - abs(i - j)) >= 1;
```

Here we do the same with one binary variable for each field of the board. The variable is one iff a queen is on this field. We compute in advance which other fields are blocked if a queen is placed on a particular field. Then we use the extended `vif` constraint to set the variables of the blocked fields to zero if a queens is placed.

```
param columns := 8;

set I := { 1 .. columns };
set IxI := I * I;

set TABU[<i,j> in IxI] := { <m,n> in IxI with
    (m != i or n != j) and
    (m == i or n == j or abs(m - i) == abs(n - j)) };

var x[IxI] binary;

maximize queens: sum <i,j> in IxI : x[i,j];

subto c1: forall <i,j> in IxI do
    vif x[i,j] == 1 then
        sum <m,n> in TABU[i,j] : x[m,n] <= 0 end;
```

## 6 Error messages

Here is a complete list of the incomprehensible error messages ZIMPL can produce:

### 101 Bad filename

The name given with the `-o` option is either missing, a directory name, or starts with a dot.

### 102 File write error

Some error occurred when writing to an output file. A description of the error follows on the next line. For the meaning consult your OS documentation.

### 103 Output format not supported, using LP format

You tried to select another format than `lp`, `mps`, or `hum`.

### 104 File open failed

Some error occurred when trying to open a file for writing. A description of the error follows on the next line. For the meaning consult your OS documentation.

### 105 Duplicate constraint name “xxx”

Two `subto` statements have the same name.

### 106 Empty LHS, constraint trivially violated

One side of your constraint is empty and the other not equal to zero. Most frequently this happens, when a set to be summed up is empty.

### 107 Range must be $l \leq x \leq u$ , or $u \geq x \geq l$

If you specify a range you must have the same comparison operators on both sides.

### 108 Empty Term with nonempty LHS/RHS, constraint trivially violated

The middle of your constraint is empty and either the left- or right-hand side of the range is not zero. This most frequently happens, when a set to be summed up is empty.

### 109 LHS/RHS contradiction, constraint trivially violated

The lower side of your range is bigger than the upper side, e.g.  $15 \leq x \leq 2$ .

### 110 Division by zero

You tried to divide by zero. This is not a good idea.

### 111 Modulo by zero

You tried to compute a number modulo zero. This does not work well.



**112 Exponent value xxx is too big or not an integer**

It is only allowed to raise a number to the power of integers. Also trying to raise a number to the power of more than two billion is prohibited.<sup>3</sup>

**113 Factorial value xxx is too big or not an integer**

You can only compute the factorial of integers. Also computing the factorial of a number bigger than two billion is generally a bad idea. See also Error 115.

**114 Negative factorial value**

To compute the factorial of a number it has to be positive. In case you need it for a negative number, remember that for all even numbers the outcome will be positive and for all odd number negative.

**115 Timeout!**

You tried to compute a number bigger than 1000!. See also the footnote to Error 112.

**116 Illegal value type in min: xxx only numbers are possible**

You tried to build the minimum of some strings.

**117 Illegal value type in max: xxx only numbers are possible**

You tried to build the maximum of some strings.

**118 Comparison of different types**

You tried to compare apples with oranges, i.e. numbers with strings. Note that the use of an undefined parameter could also lead to this message.

**119 Union of incompatible sets**

To unite two sets, both must have the same dimension tuples, i. e. the tuples must have the same number of components.

**120 Minus of incompatible sets**

To subtract two sets, both must have the same dimension tuples.

**121 Intersection of incompatible sets**

To intersect two sets, both must have the same dimension tuples.

**122 Symmetric Difference of incompatible sets**

To build the symmetric difference of two sets, both must have the same dimension tuples.

**123 “from” value xxx in range too big or not an integer**

To generate a set, the “from” number must be an integer with an absolute value of less than two billion.

---

<sup>3</sup>The behavior of this operation could easily be implemented as `for( ; )` or more elaborate as `void f(){f();}`.

**124 “upto” value xxx in range too big or not an integer**

To generate a set, the “upto” number must be an integer with an absolute value of less than two billion.

**125 “step” value xxx in range too big or not an integer**

To generate a set, the “step” number must be an integer with an absolute value of less than two billion.

**126 Zero “step” value in range**

The given “step” value for the generation of a set is zero. So the “upto” value can never be reached.

**127 Illegal value type in tuple: xxx only numbers are possible**

The selection tuple in a call to the `proj` function can only contain numbers.

**128 Index value xxx in proj too big or not an integer**

The value given in a selection tuple of a `proj` function is not an integer or bigger than two billion.

**129 Illegal index xxx, set has only dimension yyy**

The index value given in a selection tuple is bigger than the dimension of the tuples in the set.

**131 Illegal element xxx for symbol**

The index tuple used in the initialization list of a index set, is not member of the index set of the set. E.g, set `A[{ 1 to 5 }]`  $:= \langle 1 \rangle \{ 1 \}, \langle 6 \rangle \{ 2 \};$

**132 Values in parameter list missing, probably wrong read template**

Probably the template of a read statement looks like "`<1n>`" only having a tuple, instead of "`<1n> 2n`".

**133 Unknown local symbol xxx**

A (local) symbol was used, that is not defined anywhere in scope.

**134 Illegal element xxx for symbol**

The index tuple given in the initialization is not member of the index set of the parameter.

**135 Index set for parameter xxx is empty**

The attempt was made to declare an indexed parameter with the empty set as index set. Most likely the index set has a `with` clause which has rejected all elements.

**136 Lower bound for var xxx set to infinity – ignored (warning)**

In the ZIMPL code something like  `$\geq$  infinity` must have appeared. This makes no sense and is therefore ignored.

**137 Upper bound for var xxx set to -infinity – ignored** (warning)

In the ZIMPL code something like  $\leq -\text{infinity}$  must have appeared. This makes no sense and is therefore ignored.

**138 Priority/Startval for continuous var xxx ignored** (warning)

There has been set a priority or a starting value for a continuous (real) variable. This is not possible and therefore ignored.

**139 Lower bound for integral var xxx truncated to yyy** (warning)

An integral variable can only have an integral bound. So the given non integral bound was adjusted.

**140 Upper bound for integral var xxx truncated to yyy** (warning)

An integral variable can only have an integral bound. So the given non integral bound was adjusted.

**141 Infeasible due to conflicting bounds for var xxx**

The upper bound given for a variable was smaller than the lower bound.

**142 Unknown index xxx for symbol yyy**

The index tuple given is not member of the index set of the symbol.

**143 Size for subsets xxx is too big or not an integer**

The cardinality for the subsets to generate must be given as an integer smaller than two billion.

**144 Tried to build subsets of empty set**

The set given to build the subsets of, was the empty set.

**145 Illegal size for subsets xxx, should be between 1 and yyy**

The cardinality for the subsets to generate must be between 1 and the cardinality of the base set.

**146 Tried to build powerset of empty set**

The set given to build the powerset of, was the empty set.

**147 use value xxx is too big or not an integer**

The use value must be given as an integer smaller than two billion.

**148 use value xxx is not positive**

Negative or zero values for the use parameter are not allowed.

**149 skip value xxx is too big or not an integer**

The skip value must be given as an integer smaller than two billion.

**150 skip value xxx is not positive**

Negative or zero values for the skip parameter are not allowed.

**151 Not a valid read template**

A read template must look something like "`<1n, 2n>`". There have to be a `<` and a `>` in this order.

**152 Invalid read template syntax**

Apart from any delimiters like `<`, `>`, and commas a template must consists of number character pairs like `1n`, `3s`.

**153 Invalid field number xxx**

The field numbers in a template have to be between 1 and 255.

**154 Invalid field type xxx**

The only possible field types are `n` and `s`.

**155 Invalid read template, not enough fields**

There has to be at least one field inside the delimiters.

**156 Not enough fields in data**

The template specified a field number that is higher than the actual number of field found in the data.

**157 Not enough fields in data (value)**

The template specified a field number that is higher than the actual number of field found in the data. The error occurred after the index tuple in the value field.

**158 Read from file found no data**

Not a single record could be read out of the data file. Either the file is empty, or all lines are comments.

**159 Type error, expected xxx got yyy**

The type found was not the expected one, e.g. subtracting a string from a number would result in this message.

**160 Comparison of elements with different types xxx / yyy (warning)**

Two elements from different tuples were compared and found to be of different types. Probably you have an index expression with a constant element, that does not match the elements of the set. Example: `set B:={<i, "a"> in {<1, 2>}};`

**161 Line xxx: Unterminated string**

This line has an odd number of `"` characters. A String was started, but not ended.

**162 Line xxx: Trailing "yyy" ignored (warning)**

Something was found after the last semicolon in the file.

**163 Line xxx: Syntax Error**

A new statement was not started with one of the keywords: set, param, var, minimize, maximize, subto, or do.

**164 Duplicate element xxx for set rejected** (warning)

An element was added to a set that was already in it.

**165 Comparison of different dimension sets** (warning)

Two sets were compared, but have different dimension tuples. (This means they never had a chance to be equal, other than being empty sets.)

**166 Duplicate element xxx for symbol yyy rejected** (warning)

An element that was already there was added to a symbol.

**167 Comparison of different dimension tuples** (warning)

Two tuples with different dimensions were compared. Probably you have an index expression which does not match the dimension of the tuples in the set.  
Example: `set B:={<1> in {<1,2>}};`

**168 No program statements to execute**

No ZIMPL statements were found in the files loaded.

**169 Execute must return void element**

This should not happen. If you encounter this error please email the .zpl file to <mailto:koch@zib.de>.

**170 Use of uninitialized local variable xxx in call of define yyy**

A define was called and one of the arguments was a “name” (of a variable) for which no value was defined.

**171 Wrong number of arguments (xxx instead of yyy) for call of define zzz**

A define was called with a different number of arguments than in its definition.

**172 Wrong number of entries (xxx) in table line, expected yyy entries**

Each line of a parameter initialization table must have exactly the same number of entries as the index (first) line of the table.

**173 Illegal type in element xxx for symbol**

A parameter can only have a single value type. Either numbers or strings. In the initialization both types were present.

**174 Numeric field xxx read as "yyy". This is not a number**

It was tried to read a field with an 'n' designation in the template, but what was read is not a valid number.

**175 Illegal syntax for command line define "xxx" – ignored** (warning)

A parameter definition using the command line `-D` flag, must have the form `name=value`. The name must be a legal identifier, i.e. it has to start with a letter and may consist only out of letters and numbers including the underscore.

**176 Empty LHS, in Boolean constraint** (warning)

The left hand side, i.e. the term with the variables, is empty.

**177 Boolean constraint not all integer**

No continuous (real) variables are allowed in a Boolean constraint.

**178 Conditional always true or false due to bounds** (warning)

All or part of a Boolean constraint are always either true or false, due to the bounds of variables.

**179 Conditional only possible on bounded constraints**

A Boolean constraint has at least one variable without finite bounds.

**180 Conditional constraint always true due to bounds** (warning)

The result part of a conditional constraint is always true anyway. This is due to the bounds of the variables involved.

**181 Empty LHS, not allowed in conditional constraint**

The result part of a conditional constraint may not be empty.

**182 Empty LHS, in variable vabs**

There are no variables in the argument to a `vabs` function. Either everything is zero, or just use `abs`.

**183 vabs term not all integer**

There are non integer variables in the argument to a `vabs` function. Due to numerical reasons continuous variables are not allowed as arguments to `vabs`.

**184 vabs term not bounded**

The term inside a `vabs` has at least one unbounded variable.

**185 Term in Boolean constraint not bounded**

The term inside a `vif` has at least one unbounded variable.

**186 Minimizing over empty set – zero assumed** (warning)

The index expression for the minimization was empty. The result used for this expression was zero.

**187 Maximizing over empty set – zero assumed** (warning)

The index expression for the maximization was empty. The result used for this expression was zero.

**188 Index tuple has wrong dimension**

The number of elements in an index tuple is different from the dimension of the tuples in the set that is indexed.

**189 Tuple number xxx is too big or not an integer**

The tuple number must be given as an integer smaller than two billion.

**190 Component number xxx is too big or not an integer**

The component number must be given as an integer smaller than two billion.

**191 Tuple number xxx is not a valid value between 1..yyy**

The tuple number must be between one and the cardinality of the set.

**192 Component number xxx is not a valid value between 1..yyy**

The component number must be between one and the dimension of the set.

**193 Different dimension tuples in set initialization**

The tuples that should be part of the list have different dimension.

**194: Indexing tuple xxx has wrong dimension yyy, expected zzz**

The index tuple of an entry in a parameter initialization list must have the same dimension as the indexing set of the parameter. This is just another kind of error 134.

**195: Empty index set for parameter**

The index set for a parameter is empty.

**196: Indexing tuple xxx has wrong dimension yyy, expected zzz**

The index tuple of an entry in a set initialization list must have the same dimension as the indexing set of the set. If you use a powerset or subset instruction, the index set has to be one dimension.

**197: Empty index set for set**

The index set for a set is empty.

**700 log(): OS specific domain or range error message**

Function log was called with a zero or negative argument, or the argument was too small to be represented as a double.

**701 sqrt(): OS specific domain error message**

Function sqrt was called with a negative argument.

**702 ln(): OS specific domain or range error message**

Function ln was called with a zero or negative argument, or the argument was too small to be represented as a double.

**800 parse error: expecting xxx (or yyy)**

Parsing error. What was found was not what was expected. The statement you entered is not valid.

**801 Parser failed**

The parsing routine failed. This should not happen. If you encounter this error please email the .zpl file to <mailto:koch@zib.de>.

**900 Check failed!**

A check instruction did not evaluate to true.



## References

- [Chv83] Vašek Chvátal. *Linear Programming*. H.W. Freeman and Company, New York, 1983.
- [Dan90] Georg B. Danzig. The diet problem, 1990.
- [FGK03] R. Fourier, D. M. Gray, and B. W. Kernighan. *AMPL: A Modelling Language for Mathematical Programming*. Brooks/Cole—Thomson Learning, second edition, 2003.
- [GNU03] GNU multiple precision arithmetic library (GMP), version 4.1.2., 2003. Code and documentation available at <http://www.swox.com/gmp>.
- [IBM97] IBM optimization library guide and reference, 1997. For an online reference see <http://www6.software.ibm.com/sos/features/featur11.htm>.
- [ILO02] ILOG CPLEX Division, 889 Alder Avenue, Suite 200, Incline Village, NV 89451, USA. *ILOG CPLEX 8.0 Reference Manual*, 2002. Information available at <http://www.cplex.com>.
- [Sch03] Alexander Schrijver. *Combinatorial Optimization*. Springer, 2003.
- [vH99] Pascal van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, Cambridge, Massachusetts, 1999.
- [XPR99] *XPRESS-MP Release 11 Reference Manual*. Dash Associates, 1999. Information available at <http://www.dashoptimization.com/>.