# MIP formulations and linearizations

Quick reference

9.7

QUICK REFERENCE

FICO® Xpress Optimization

FICO

FICO® Xpress Optimization 9.7

Last Revised: 29 July, 2025

## How to Contact the Xpress Team

### Sales and Maintenance

If you need information on other Xpress Optimization products, or you need to discuss maintenance contracts or other sales-related items, contact FICO by:

- Phone: +1 (408) 535-1500 or +44 207 940 8718
- Web: www.fico.com/optimization and use the available contact forms

### Product Support

*Customer Self Service Portal (online support):* www.fico.com/en/product-support

*Email:* Support@fico.com (Please include *'Xpress'* in the subject line)

For the latest news and Xpress software and documentation updates, please visit the Xpress website at http://www.fico.com/xpress or subscribe to our mailing list.

## FICO® Xpress Optimization

# MIP formulations and linearizations

### Quick reference

**Release 9.7**

29 July, 2025

# Contents

# 1 Introduction

This quick reference guide presents a collection of MIP model formulations for Xpress Optimizer, including standard linearization techniques involving binary variables, the use of more specific modeling objects such as SOS and partial integer variables, and reformulations of logic constraints through indicator constraints.

## 1.1 Integer Programming entities supported in Xpress

- *Binary variables (BV)* – decision variables that must take either the value 0 or the value 1, sometimes called 0/1 variables;
- *Integer variables (UI)* – decision variables that must take on integer values. Some upper limit must be specified;
- *Partial integer variables (PI)* – decision variables that must take integer values below a specified limit but can take any value above that limit;
- *Semi-continuous variables (SC)* – decision variables that must take on either the value 0, or any value in a range whose lower an upper limits are specified. SCs help model situations where, if a variable is to be used at all, it has to be at some minimum level;
- *Semi-continuous integer variables (SI)* – decision variables that must take either the value 0, or any integer value in a range whose lower and upper limits are specified;
- *Special ordered sets of type one (SOS1)* – an ordered set of variables of which at most one can take a nonzero value;
- *Special ordered sets of type two (SOS2)* – an ordered set of variables of which at most two can be nonzero, and if two are nonzero, they must be consecutive in their ordering.

**Remarks**

- The solution values of binary and integer variables are real valued, not integer valued.
- At an optimal MIP solution, the actual values of the binary and integer variables will be integer – to within a certain tolerance.

## 1.2 Integer Programming entities in Mosel

**Definition**: integer programming types are defined as unary constraints on previously declared decision variables of type `mpvar`; name the constraints if you want to be able to access/modify them.

```
model "intromip"
 uses "mmxprs"
 declarations
  d: mpvar
  ifmake: array(PRODS,LINES) of mpvar
  x: mpvar
 end-declarations

 d is_binary                        ! Single binary variable
 forall(p in PRODS, l in LINES)
  ifmake(p,l) is_binary             ! An array of binaries

 ACtr:= x is_integer                ! An integer variable
```

```
 x >= MINVAL                        ! Lower bound on the variable
 x <= MAXVAL                        ! Upper bound on the variable
 ! MINVAL,MAXVAL: values between -MAX_REAL and MAX_REAL
 ...
 ACtr:= x is_partint 10             ! Change type to partial integer
 ...
 ACtr:= 0                           ! Delete constraint
! Equivalently:
 ACtr:= x is_continuous             ! Change type to continuous
 ...
```

**Solving**: with Xpress Optimizer (Mosel module *mmxprs*) any problem containing integer programming entities is automatically solved as a MIP problem, to solve just the LP relaxation use option `XPRS_LPSTOP` (if following up with MIP search) or `XPRS_LIN` (ignore all MIP information) for `maximize` / `minimize`.

```
minimize(d)                         ! Solve the MIP problem
minimize(XPRS_LPSTOP, d)            ! Solve the LP releaxation
```

**Accessing the solution**: for obtaining solution values of decision variables and linear expressions use `getsol` (alternative syntax: `.sol`); the MIP problem status is returned by the function `getparam("XPRS_MIPSTATUS")`

```
case getparam("XPRS_MIPSTATUS") of
 XPRS_MIP_NOT_LOADED,
   XPRS_MIP_LP_NOT_OPTIMAL: writeln("Solving not started")
 XPRS_MIP_LP_OPTIMAL:       writeln("Root LP solved")
 XPRS_MIP_UNBOUNDED:        writeln("LP unbounded")
 XPRS_MIP_NO_SOL_FOUND,
   XPRS_MIP_INFEAS:         writeln("MIP search started, no solution")
 XPRS_MIP_SOLUTION,
   XPRS_MIP_OPTIMAL:        writeln("MIP solution: ", , getobjval)
end-case

writeln("x: ", getsol(x))
writeln("d: ", d.sol)
```

## 1.3 Integer Programming entities in the Xpress Python API

**Definition**: Integer Programming types are specified when creating decision variables; types may be changed with `vartype`.

```
import xpress as xp

MAXVAL = 50
NP = 5
NL = 6

p = xp.problem()

# A single binary variable
d = p.addVariable(vartype=xp.binary, name="d")

# A NumPy array of variables
ifmake = p.addVariables(NP, NL, vartype=xp.binary, name="ifmake")

 # An integer variable with an upper bound
x = p.addVariable(lb=0, ub=MAXVAL, vartype=xp.integer, name="x")

x.vartype = xp.partiallyinteger    # Change type to partial integer
x.threshold = 10
```

```
x.vartype = xp.continuous          # Change type to continuous
```

**Solving**: to solve a MIP problem use method `optimize` of `xp.problem`. This call is usually preceded by the definition of the objective function via `setObjective` that also allows you to change the sense of the optimization (the default optimization direction is minimization).

```
p.setObjective(d)
p.optimize()
```

**Accessing the solution**: for obtaining solution values of decision variables use `getSolution`; the MIP problem status is returned by `p.attributes.solstatus`. Import `SolStatus` from `xpress.enums` at the top of the script to access the possible solution values.

```
from xpress.enums import SolStatus

match p.attributes.solstatus:
    case SolStatus.FEASIBLE | SolStatus.OPTIMAL:
        print("MIP solution: ", p.attributes.objval())
    case SolStatus.INFEASIBLE:
        print("Problem is infeasible")
    case SolStatus.UNBOUNDED:
        print("LP unbounded")
    case SolStatus.NOTFOUND:
        print("Solution not found")

print(x.name,": ",p.getSolution(x))
```

## 1.4 Integer Programming entities in the Xpress C++ API

The code extracts for the object-oriented APIs of Xpress Solver shown in this document are formulated for the C++ interface. The other object-oriented APIs (Java, C#) work similarly, please refer to the Xpress Java API user guide and Xpress C# API user guide for further detail.

**Definition**: Integer Programming types are specified when creating decision variables; types may be changed with `setType`.

```
#include <iostream>
#include <xpress.hpp>
using namespace xpress;
using namespace xpress::objects;
using namespace std;

int main()
{
  XpressProblem prob;

  Variable d = prob.addVariable(ColumnType::Binary, "d"); // Single binary variable

  auto ifmake = prob.addVariables(NP, NL)  // 2-dim array of binary variables
    .withType(ColumnType::Binary)
    .withName("ifmake_%d_%d")    .toArray();

  // An integer variable
  Variable x = prob.addVariable(0, MAXVAL, ColumnType::Integer, "x");
  ...
  x.setType(ColumnType::PartialInteger);   // Change type to partial integer
  x.setLimit(10);                          // Set the partial integer limit value
  ...
  x.setType(ColumnType::Continuous);       // Change type to continuous
  ...
```

**Solving**: to solve a MIP problem use method `optimize` of `XpressProblem`. This call is usually preceded by the definition of the objective function via `setObjective` that also allows you to change the sense of the optimization (the default optimization direction is minimization). To solve just the LP relaxation use `lpOptimize`.

```
prob.setObjective(d, ObjSense::Maximize);
prob.optimize();
```

**Accessing the solution**: for obtaining solution values of decision variables use `getSolution`; the MIP problem status is returned by the attribute `getMipStatus`.

```
auto mipStatus = prob.attributes.getMipStatus();
switch (mipStatus) {
  case MIPStatus::NotLoaded:
  case MIPStatus::LPNotOptimal:
    cout << "Solving not started" << endl;
    break;
  case MIPStatus::LPOptimal:
    cout << "Root LP solved" << endl;
    break;
  case MIPStatus::Unbounded:
    cout << "LP unbounded" << endl;
    break;
  case MIPStatus::NoSolutionFound:
  case MIPStatus::Infeasible:
    cout << "MIP search started, no solution" << endl;
    break;
  case MIPStatus::Solution:
  case MIPStatus::Optimal:
    cout << "MIP solution: " << prob.attributes.getObjVal() << endl;
    break;
}

cout << x.getName() << ": " << x.getSolution() << endl;
```

# 2 Binary variables

*Binary decision variables*

- take value 0 or 1
- model a discrete decision
  - yes/no
  - on/off
  - open/close
  - build or don't build
  - strategy A or strategy B

## 2.1 Logical conditions

Projects A, B, C, D, ... with associated binary variables $a, b, c, d, \ldots$ which are 1 if we decide to do the project and 0 if we decide not to do the project.

| | |
|---|---|
| At most N of A, B, C,... | $a + b + c + ... \leqslant N$ |
| At least N of A, B, C,... | $a + b + c + ... \geqslant N$ |
| Exactly N of A, B, C,... | $a + b + c + ... = N$ |
| If A then B | $b \geqslant a$ |
| Not B | $not(b) = 1 - b$ |
| If A then not B | $a + b \leqslant 1$ |
| If not A then B | $a + b \geqslant 1$ |
| If A then B, and if B then A | $a = b$ |
| If A then B and C; A only if B and C | $b \geqslant a$ and $c \geqslant a$<br>or alternatively: $a \leqslant (b + c)/2$ |
| If A then B or C | $b + c \geqslant a$ |
| If B or C then A | $a \geqslant b$ and $a \geqslant c$<br>or alternatively: $a \geqslant \frac{1}{2} \cdot (b + c)$ |
| If B and C then A | $a \geqslant b + c - 1$ |
| If two or more of B, C, D or E then A | $a \geqslant \frac{1}{3} \cdot (b + c + d + e - 1)$ |
| If M or more of N projects (B, C, D, ...) then A | $a \geqslant \frac{b+c+d+...-M+1}{N-M+1}$ |

## 2.2 Minimum values

**y** $=$ **min**$\{\mathbf{x_1}, \mathbf{x_2}\}$ for two continuous variables $x_1, x_2$

- Must know lower and upper bounds

$L_1 \leqslant x_1 \leqslant U_1$ [1.1]
$L_2 \leqslant x_2 \leqslant U_2$ [1.2]

- Introduce binary variables $d_1, d_2$ to mean

$d_i$ 1 if $x_i$ is the minimum value;
0 otherwise

- MIP formulation:

$y \leqslant x_1$ [2.1]
$y \leqslant x_2$ [2.2]
$y \geqslant x_1 - (U_1 - L_{min}) \cdot (1 - d_1)$ [3.1]
$y \geqslant x_2 - (U_2 - L_{min}) \cdot (1 - d_2)$ [3.2]
$d_1 + d_2 = 1$ [4]

- Generalization to **y** $=$ **min**$\{\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_n}\}$

$L_i \leqslant x_i \leqslant U_i$ [1.i]
$y \leqslant x_i$ [2.i]
$y \geqslant x_i - (U_i - L_{min}) \cdot (1 - d_i)$ [3.i]
$\sum_i d_i = 1$ [4]

- See Section 3.7 for an alternative formulation via general constraints

## 2.3 Maximum values

$\mathbf{y = max\{x_1, x_2, ..., x_n\}}$ for continuous variables $x_1, ..., x_n$

- Must know lower and upper bounds

  $L_i \leqslant x_i \leqslant U_i$ [1.i]

- Introduce binary variables $d_1, ..., d_n$
  $d_i = 1$ if $x_i$ is the maximum value, 0 otherwise
- MIP formulation

  $L_i \leqslant x_i \leqslant U_i$ [1.i]
  $y \geqslant x_i$ [2.i]
  $y \leqslant x_i + (U_{max} - L_i) \cdot (1 - d_i)$ [3.i]
  $\sum_i d_i = 1$ [4]

- See Section 3.7 for an alternative formulation via general constraints

## 2.4 Absolute values

$\mathbf{y = |x_1 - x_2|}$ for two variables $x_1, x_2$ with $0 \leqslant x_i \leqslant U$

- Introduce binary variables $d_1, d_2$ to mean

  $d_1$ : 1 if $x_1 - x_2$ is the positive value
  $d_2$ : 1 if $x_2 - x_1$ is the positive value

- MIP formulation

  $0 \leqslant x_i \leqslant U$ [1.i]
  $0 \leqslant y - (x_1 - x_2) \leqslant 2 \cdot U \cdot d_2$ [2]
  $0 \leqslant y - (x_2 - x_1) \leqslant 2 \cdot U \cdot d_1$ [3]
  $d_1 + d_2 = 1$ [4]

- See Section 3.7 for an alternative formulation via general constraints

## 2.5 Logical AND

$\mathbf{d = min\{d_1, d_2\}}$ for two binary variables $d_1, d_2$, or equivalently
$\mathbf{d = d_1 \cdot d_2}$ (see Section 2.8), or
$\mathbf{d = d_1}$ AND $\mathbf{d_2}$ as a logical expression

- IP formulation

  $d \leqslant d_1$ [1.1]
  $d \leqslant d_2$ [1.2]
  $d \geqslant d_1 + d_2 - 1$ [2]
  $d \geqslant 0$ [3]

- Generalization to $\mathbf{d} = \mathbf{min}\{\mathbf{d_1}, \mathbf{d_2}, ..., \mathbf{d_n}\}$

$$
\begin{aligned}
& d \leqslant d_i && [1.i] \\
& d \geqslant \textstyle\sum_i d_i - (n-1) && [2] \\
& d \geqslant 0 && [3]
\end{aligned}
$$

  Note: equivalent to $\mathbf{d} = \mathbf{d_1} \cdot \mathbf{d_2} \cdot ... \cdot \mathbf{d_n}$
  and (as a logical expression): $\mathbf{d} = \mathbf{d_1}$ AND $\mathbf{d_2}$ AND ... AND $\mathbf{d_n}$

- See Section 5 for an alternative formulation via Boolean variables

## 2.6 Logical OR

$\mathbf{d} = \mathbf{max}\{\mathbf{d_1}, \mathbf{d_2}\}$ for two binary variables $d_1$, $d_2$, or
$\mathbf{d} = \mathbf{d_1}$ OR $\mathbf{d_2}$ as a logical expression

- IP formulation

$$
\begin{aligned}
& d \geqslant d_1 && [1.1] \\
& d \geqslant d_2 && [1.2] \\
& d \leqslant d_1 + d_2 && [2] \\
& d \leqslant 1 && [3]
\end{aligned}
$$

- Generalization to $\mathbf{d} = \mathbf{max}\{\mathbf{d_1}, \mathbf{d_2}, ..., \mathbf{d_n}\}$

$$
\begin{aligned}
& d \geqslant d_i && [1.i] \\
& d \leqslant \textstyle\sum_i d_i && [2.i] \\
& d \leqslant 1 && [3]
\end{aligned}
$$

  Note: equivalent to $\mathbf{d} = \mathbf{d_1}$ OR $\mathbf{d_2}$... OR $\mathbf{d_n}$

- See Section 5 for an alternative formulation via Boolean variables

## 2.7 Logical NOT

$\mathbf{d} =$ NOT $\mathbf{d_1}$ for one binary variable $d_1$

- IP formulation

$$d = 1 - d_1$$

## 2.8 Product values

$\mathbf{y} = \mathbf{x} \cdot \mathbf{d}$ for one continuous variable $x$, one binary variable $d$

- Must know lower and upper bounds

$$L \leqslant x \leqslant U$$

- MIP formulation:

$$L \cdot d \leqslant y \leqslant U \cdot d \quad [1]$$
$$L \cdot (1-d) \leqslant x - y \leqslant U \cdot (1-d) \quad [2]$$

Product of two binaries: $d_3 = d_1 \cdot d_2$

- MIP formulation:

$$d_3 \leqslant d_1$$
$$d_3 \leqslant d_2$$
$$d_3 \geqslant d_1 + d_2 - 1$$

## 2.9 Disjunctions

**Either $5 \leqslant x \leqslant 10$ or $80 \leqslant x \leqslant 100$**

- Introduce a new binary variable:
  $ifupper$: 0 if $5 \leqslant x \leqslant 10$; 1 if $80 \leqslant x \leqslant 100$
- MIP formulation:

$$x \leqslant 10 + (100 - 10) \cdot ifupper \quad [1]$$
$$x \geqslant 5 + (80 - 5) \cdot ifupper \quad [2]$$

- Generalization to **Either $L_1 \leqslant \sum_i A_i \cdot x_i \leqslant U_1$ or $L_2 \leqslant \sum_i A_i \cdot x_i \leqslant U_2$ (with $U_1 \leqslant L_2$)**

$$\sum_i A_i \cdot x_i \leqslant U_1 + (U_2 - U_1) \cdot ifupper \quad [1]$$
$$\sum_i A_i \cdot x_i \geqslant L_1 + (L_2 - L_1) \cdot ifupper \quad [2]$$

## 2.10 Minimum activity level

Continuous production rate $make$ that may be 0 (the plant is not operating) or between allowed production limits $MAKEMIN$ and $MAKEMAX$

- Introduce a binary variable $ifmake$ to mean

  $ifmake$: 0 if plant is shut
  1 plant is open

  MIP formulation:

$$make \geqslant MAKEMIN \cdot ifmake \quad [1]$$
$$make \leqslant MAKEMAX \cdot ifmake \quad [2]$$

  Note: see Section 3.5 for an alternative formulation using semi-continuous variables

- The $ifmake$ binary variable also allows us to model fixed costs
  - $FCOST$: fixed production cost
  - $VCOST$: variable production cost

  MIP formulation:

$$cost = FCOST \cdot ifmake + VCOST \cdot make \quad [3]$$
$$make \geqslant MAKEMIN \cdot ifmake \quad [1]$$
$$make \leqslant MAKEMAX \cdot ifmake \quad [2]$$

# 3 MIP formulations using other entities

In principle, all you need in building MIP models are continuous variables and binary variables. But it is convenient to extend the set of modeling entities to embrace objects that frequently occur in practice.

*Integer decision variables*

- values 0, 1, 2, ... up to small upper bound
- model discrete quantities
- try to use *partial integer variables* instead of integer variables with a very large upper bound

*Semi-continuous variable*

- may be zero, or any value between the intermediate bound and the upper bound
- *Semi-continuous integer variables* also available: may be zero, or any integer value between the intermediate bound and the upper bound

*Special ordered sets*

- set of decision variables
- each variable has a different ordering value, which orders the set
- Special ordered sets of type 1 (SOS1): at most one variable may be non-zero
- Special ordered sets of type 2 (SOS2): at most two variables may be non-zero; the non-zero variables must be adjacent in ordering

*Indicator constraints*

- associate a binary variable with a linear or nonlinear constraint
- model an implication: the constraint is active only if the condition is true

*General constraints*

- specific constraint relations that are recognized by MIP solvers
- piecewise linear: can be used in place of SOS-2 formulations
- absolute value, minimum value, maximum value of discrete or continuous decision variables
- logical constraints: 'and' and 'or' over binary variables

## *3.1 Batch sizes*

Must deliver in batches of 10, 20, 30, ...

- Decision variables

  $\mathrm{nship}$ number of batches delivered: integer
  $\mathrm{ship}$ quantity delivered: continuous

- Constraint formulation

  $$\mathrm{ship} = 10 \cdot \mathrm{nship}$$

## 3.2 Ordered alternatives

Suppose you have $N$ possible investments of which at most one can be selected. The capital cost is $CAP_i$ and the expected return is $RET_i$.

- Often use binary variables to choose between alternatives. However, SOS1 are more efficient to choose between a set of graded (ordered) alternatives.
- Define a variable $d_i$ to represent the decision, $d_i = 1$ if investment $i$ is picked
- Binary variable (standard) formulation

  $d_i$ : binary variables

  Maximize: $ret = \sum_i RET_i \cdot d_i$
  $\sum_i d_i \leqslant 1$
  $\sum_i CAP_i \cdot d_i \leqslant MAXCAP$

- SOS1 formulation

  {$d_i$; ordering value $CAP_i$} : SOS1

  Maximize: $ret = \sum_i RET_i \cdot d_i$
  $\sum_i d_i \leqslant 1$
  $\sum_i CAP_i \cdot d_i \leqslant MAXCAP$

**Special ordered sets in Mosel**

- special ordered sets are a special type of linear constraint
- the set includes all variables in the constraint
- the coefficient of a variable is used as the ordering value (*i.e.*, each value must be unique)

```
declarations
  I=1..4
  d: array(I) of mpvar
  CAP: array(I) of real
  My_Set, Ref_row: linctr
end-declarations

My_Set:= sum(i in I) CAP(i)*d(i) is_sos1
```

or alternatively (must be used if a coefficient is 0):

```
Ref_row:= sum(i in I) CAP(i)*d(i)
makesos1(My_Set, union(i in I) {d(i)}, Ref_row)
```

**Special ordered sets in the Python API**

- special ordered sets are defined as constraints, specifying the set type ('1' or '2'), a list of variables and the corresponding weight coefficients

```
import xpress as xp

I = range(4)
CAP = [10, 20, 100, 250]

p = xp.problem()

# Create the decision variables
d = [prob.addVariable(name="d_{}".format(i)) for i in I]

# Define a SOS-1 with weights CAP
p.addSOS(d, CAP, name="My_Set", type=1)
```

**Special ordered sets in the C++ API**

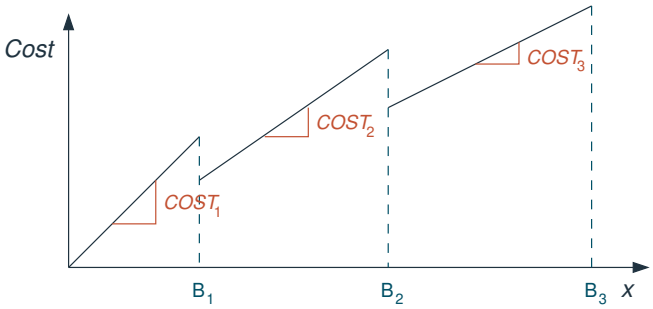
- special ordered sets are defined as constraints, specifying the set type ('SOS1' or 'SOS2'), a list of variables and the corresponding weight coefficients

```
static std::vector<int> I = {0, 1, 2, 3};
int main()
{
  XpressProblem prob;
  std::vector<double> CAP = {10, 20, 100, 250};

  // Create the decision variables
  auto d = prob.addVariables(I.size()).withName("d_%d").toArray();

  // Define a SOS-1 with weights CAP
  prob.addConstraint(SOS::sos(SetType::SOS1, d, CAP, "My_Set"));
```

# 3.3 Price breaks

## 3.3.1 All items discount

**All items discount**: when buying a certain number of items we get discounts on *all* items that we buy if the quantity we buy lies in certain price bands.




| | |
|---|---|
| less than $B_1$ | $COST_1$ each |
| $\geqslant B_1$ and $< B_2$ | $COST_2$ each |
| $\geqslant B_2$ and $< B_3$ | $COST_3$ each |

Formulation with binary variables or Special Ordered Sets of type 1 (SOS1):

- Define binary variables $b_i$ (i=1,2,3), where $b_i$ is 1 if we pay a unit cost of $COST_i$.
- Real decision variables $xp_i$ represent the number of items bought at price $COST_i$.

- The quantity bought is given by $x = \sum_i xp_i$, with a total price of $\sum_i COST_i \cdot xp_i$
- MIP formulation :

  $\sum_i b_i = 1$
  $xp_1 \leqslant B_1 \cdot b_1$
  $B_{i-1} \cdot b_i \leqslant xp_i \leqslant B_i \cdot b_i$ for $i = 2, 3$

  where the variables $b_i$ are either defined as binaries, or they form a Special Ordered Set of type 1 (SOS1), where the order is given by the values of the breakpoints $B_i$.

Formulation as piecewise linear expression:

- Specification as list of piecewise linear segments with associated intervals:

  $\bigcup_{i=1,2,3}[B_{i-1}, B_i[: COST_i \cdot x$

  **Implementation with Mosel:**

```
TotalCost:= pwlin(union(i in 1..3) [pws(B(i-1), COST(i)*x)])
```

  **Implementation with the Python API:**

```
totalcost = xp.pwl({(B[i-1],B[i]): COST[i]*x[i] for i in range(1,3)})
```

- Specification as list of points:

  $\bigcup_{i=1,2,3} [B_{i-1}, COST_i \cdot B_{i-1}, B_i, COST_i \cdot B_i]$

  **Implementation with Mosel:**

```
TotalCost:= pwlin(x, union(i in 1..3) [B(i-1), COST(i)*B(i-1), B(i), COST(i)*B(i)])
```

  **Implementation with the Python API:** In Python, it is possible to specify piecewise linear functions as a list of points using the `problem.addpwlcons` method. The example assumes that `B` and `COST` are array (lists) containing the x-values and y-values of the breakpoints, respectively.

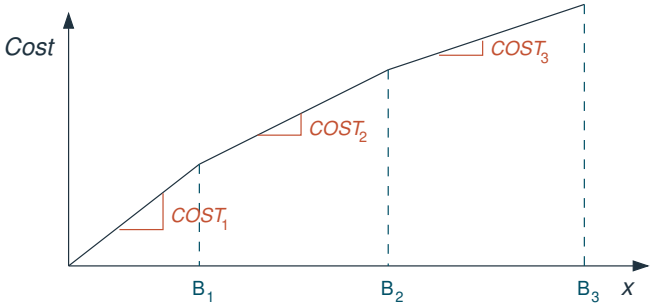
```
prob.addpwlcons([x], [totalcost], [0], B, COST)
```

  **Implementation with the C++ API:**

```
Variable x = prob.addVariable();
Variable fx = prob.addVariable();
std::vector<double> B = ...;
std::vector<double> COST = ...;
std::vector<double> breakX = {0, B[0], B[0], B[1], B[1], B[2]};
std::vector<double> breakY = {0, B[0]*COST[0], B[0]*COST[1], B[1]*COST[1],
                              B[1]*COST[2], B[2]*COST[2]};
prob.addConstraint(fx.pwlOf(x, breakX, breakY));
```

### 3.3.2 Incremental pricebreaks

**Incremental pricebreaks**: when buying a certain number of items we get discounts incrementally. The unit cost for items between 0 and $B_1$ is $COST_1$, items between $B_1$ and $B_2$ cost $COST_2$ each, *etc.*




Formulation with Special Ordered Sets of type 2 (SOS2):

- Associate real valued decision variables $w_i$ ($i = 0, 1, 2, 3$) with the quantity break points $B_0 = 0$, $B_1$, $B_2$ and $B_3$.
- Cost break points $CBP_i$ (=total cost of buying quantity $B_i$):

  $$CBP_0 = 0$$
  $$CBP_i = CBP_{i-1} + COST_i \cdot (B_i - B_{i-1}) \text{ for } i = 1, 2, 3$$

- Constraint formulation:

  $$\sum_i w_i = 1$$
  $$TotalCost = \sum_i CBP_i \cdot w_i$$
  $$x = \sum_i B_i \cdot w_i$$

  where the $w_i$ form a SOS2 with reference row coefficients given by the coefficients in the definition of the total amount $x$.
  For a solution to be valid, at most two of the $w_i$ can be non-zero, and if there are two non-zero they must be contiguous, thus defining one of the line segments.

**Implementation with Mosel:** `is_sos2` cannot be used here due to the 0-valued coefficient of $w_0$

```
Defx := x = sum(i in 1..3) B(i)*w(i)
makesos2(My_Set, union(i in 0..3) {w(i)}, Defx)
sum(i in 1..3) w(i) = 1
```

**Implementation with the Python API:**

```
p.addSOS(w, B, type=2)
p.addConstraint(xp.Sum(w) == 1)
```

**Implementation with the C++ API:**

```
std::vector<int> I = {0, 1, 2};
std::vector<double> B = ...;
auto x = prob.addVariable();
auto w = prob.addVariables(I.size()).withName("w_%d").toArray();
prob.addConstraint(SOS::sos(SetType::SOS2, w, B, "Defx"));
prob.addConstraint(sum(I, [&](auto i) { return w[i];}) == 1.0);
```

Formulation using binaries:

- Define binary variables $b_i$ (i=1,2,3), where $b_i$ is 1 if we have bought any items at a unit cost of $COST_i$.
- Real decision variables $xp_i$ (i=1,..3) for the number of items bought at price $COST_i$.
- Total amount bought: x = $\sum_i xp_i$
- Constraint formulation:

  $(B_i - B_{i-1}) \cdot b_{i+1} \leqslant xp_i \leqslant (B_i - B_{i-1}) \cdot b_i$ for $i = 1, 2$
  $xp_3 \leqslant (B_3 - B_2) \cdot b_3$
  $b_1 \geqslant b_2 \geqslant b_3$

Formulation as piecewise linear expression:

- Specification as list of slopes with associated intervals:

  $\bigcup_{i=1,2,3}[B_{i-1}, B_i[: COST_i$

  **Implementation with Mosel:** only points of slope changes are specified, start value is 0

```
TotalCost:= pwlin(x, union(i in 1..2) [B(i)], union(i in 1..3) [COST(i)])
```

  **Python** does not support specifying piecewise linear functions as a list of slopes

- Specification as list of piecewise linear segments with associated intervals:

  $\bigcup_{i=1,2,3}[B_{i-1}, B_i[: CBP_{i-1} + COST_i \cdot (x - B_{i-1})$

  **Implementation with Mosel:**

```
TotalCost:= pwlin(union(i in 1..3) [pws(B(i-1), CBP(i-1)+COST(i)*(x-B(i-1)))])
```

  **Implementation with the Python API:**

```
totalcost = xp.pwl({(B[i-1],B[i]): COST[i-1]+COST[i]*(x[i]-B[i-1])) for i in range(1,3)})
```

- Specification as list of points:

  $\bigcup_{i=0,1,2,3} [B_i, CBP_i]$

  **Implementation with Mosel:**

```
TotalCost:= pwlin(x, union(i in 0..3) [B(i), CBP(i)])
```

  **Implementation with the Python API:** In Python, it is possible to specify piecewise linear functions as a list of points using the `problem.addpwlcons` method. The example assumes that `B` and `CBP` are array (lists) containing the x-values and y-values of the breakpoints, respectively.

```
prob.addpwlcons([x], [totalcost], [0], B, CBP)
```

  **Implementation with the C++ API:**

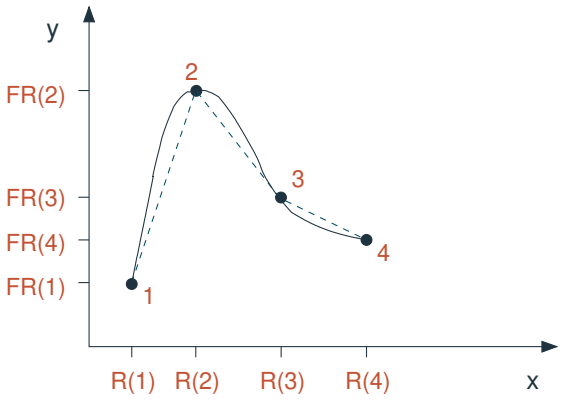
```
Variable x = prob.addVariable();
Variable fx = prob.addVariable();
int npieces = 3;
std::vector<double> B = ...;
std::vector<double> COST = ...;
std::vector<double> breakX = {0, B[0], B[1], B[2]};
std::vector<double> breakY = {0};
for (int i = 1; i <= npieces; i++)  breakY[i] = breakY[i-1] + COST[i-1]*(B[i] - B[i-1]);
prob.addConstraint(fx.pwlOf(x, breakX, breakY));
```

## 3.4 Non-linear functions

Can model non-linear functions in the same way as incremental pricebreaks

- approximate the non-linear function with a piecewise linear function
- use an SOS2 to model the piecewise linear function
- alternatively, formulate as piecewise linear expression by specifying a list of points
- note that certain nonlinear functions (*e.g.* absolute value, minimum value, maximum value) are recognized by MIP solvers and can be used directly in the formulation of MIP models, see Section 3.7

### 3.4.1 Non-linear function in a single variable




- x-coordinates of the points: $R_1$, ..., $R_4$
  y-coordinates $FR_1$, ..., $FR_4$. So point 1 is $(R_1, FR_1)$ *etc.*
- Let weights (decision variables) associated with point $i$ be $w_i$ (i=1,...,4)
- Form convex combinations of the points using weights $w_i$ to get a combination point (x,y):

$$x = \sum_i w_i \cdot R_i$$
$$y = \sum_i w_i \cdot FR_i$$
$$\sum_i w_i = 1$$

where the variables $w_i$ form an SOS2 set with ordering coefficients defined by values $R_i$.

**Mosel implementation (SOS-2):**

```
 declarations
  I=1..4
  x,y: mpvar
  w: array(I) of mpvar
  R,FR: array(I) of real
 end-declarations

! ...assign values to arrays R and FR...

! Define the SOS-2 with "reference row" coefficients from R
 Defx:= sum(i in I) R(i)*w(i) is_sos2
 sum(i in I) w(i) = 1

! The variable and the corresponding function value we want to approximate
 x = Defx
 y = sum(i in I) FR(i)*w(i)
```

### Mosel implementation (piecewise linear):

```
 uses "mmxnlp"
 declarations
  I=1..4
  x,y: mpvar
  R,FR: array(I) of real
 end-declarations

! ...assign values to arrays R and FR...

! Define the piecewise linear expression
 y = pwlin(x, sum(i in I) [R(i), FR(i)])

! Only consider the x-values interval defined by the breakpoints
 setrange(x, R(1), R(4))
```

### Python implementation (SOS-2):

```
import xpress as xp

I = range(4)
R = [1, 2.5, 4.5, 6.5]
FR = [1.5, 6, 3.5, 2.5]

p = xp.problem()

# Create the decision variables
x = p.addVariable(lb=R[0],ub=R[3],name="x") # x-values interval defined by the breakpoints
y = p.addVariable(name="y")
w = [p.addVariable(name="w{0}".format(i)) for i in I]

# Define the SOS-2 with weights R
p.addSOS(w, R, name="Defx", type=2)

# Weights must sum up to 1
p.addConstraint(xp.Sum(w[i] for i in I) == 1.0)

# The variable and the corresponding function value we want to approximate
p.addConstraint(x == xp.Sum(R[i]*w[i] for i in I))
p.addConstraint(y == xp.Sum(FR[i]*w[i] for i in I))
...
```

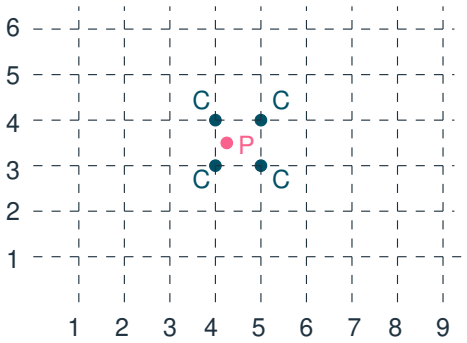
### C++ implementation (SOS-2):

```
#include <iostream>
#include <xpress.hpp>
```

```
using namespace xpress;
using namespace xpress::objects;
using xpress::objects::utils::sum;
using namespace std;

static std::vector<int> I = {0, 1, 2, 3};

int main()
{
  XpressProblem prob;
  std::vector<double> R = {1, 2.5, 4.5, 6.5};
  std::vector<double> FR = {1.5, 6, 3.5, 2.5};

  // Create the decision variables
  auto x = prob.addVariable(R[0], R[3], ColumnType::Continuous, "x");
                          // x-values interval defined by the breakpoints
  auto y = prob.addVariable("y");
  auto w = prob.addVariables(I.size()).withName("w_%d").toArray();

  // Define the SOS-2 with weights R
  prob.addConstraint(SOS::sos(SetType::SOS2, w, R, "Defx"));

  // Weights must sum up to 1
  prob.addConstraint(sum(I, [&](auto i) { return w[i];}) == 1.0);

  // The variable and the corresponding function value we want to approximate
  prob.addConstraint(x == sum(I, [&](auto i) { return R[i]*w[i];}));
  prob.addConstraint(y == sum(I, [&](auto i) { return FR[i]*w[i];}));
...
```

## 3.4.2 Non-linear function in two variables

Interpolation of a function f in two variables: approximate f at a point P by the corners C of the enclosing square of a rectangular grid (NB: the representation of P=(x,y) by the four points C obviously means a fair amount of degeneracy).




- x-coordinates of grid points: $X_1, ..., X_n$
  y-coordinates of grid points: $Y_1, ..., Y_m$. So grid points are $(X_i, Y_j)$.
- Function evaluation at grid points: $FXY_{11}, ..., FXY_{nm}$
- Define weights (decision variables) associated with x and y coordinates, $wx_i$ respectively $wy_j$, and for each grid point $(X(i), Y(j))$ define a variable $wxy_{ij}$
- Form convex combinations of the points using the weights to get a combination point (x,y) and the corresponding function approximation:

$$x = \sum_i wx_i \cdot X_i$$
$$y = \sum_j wy_j \cdot Y_j$$
$$f = \sum_{ij} wxy_{ij} \cdot FXY_{ij}$$
$$\forall i = 1, ..., n : \sum_j wxy_{ij} = wx_i$$
$$\forall j = 1, ..., m : \sum_i wxy_{ij} = wy_j$$
$$\sum_i wx_i = 1$$
$$\sum_j wy_j = 1$$

where the variables $wx_i$ form an SOS2 set with ordering coefficients defined by values $X_i$, and the variables $wy_j$ are a second SOS2 set with coordinate values $Y_j$ as ordering coefficients.

**Mosel implementation:**

```
declarations
 RX,RY:range
 X: array(RX) of real          ! x coordinate values of grid points
 Y: array(RY) of real          ! y coordinate values of grid points
 FXY: array(RX,RY) of real     ! Function evaluation at grid points
end-declarations

! ... initialize data

declarations
 wx: array(RX) of mpvar        ! Weight on x coordinate
 wy: array(RY) of mpvar        ! Weight on y coordinate
 wxy: array(RX,RY) of mpvar    ! Weight on (x,y) coordinates
 x,y,f: mpvar
end-declarations

! Definition of SOS (assuming coordinate values <>0)
sum(i in RX) X(i)*wx(i) is_sos2
sum(j in RY) Y(j)*wy(j) is_sos2

! Constraints
forall(i in RX) sum(j in RY) wxy(i,j) = wx(i)
forall(j in RY) sum(i in RX) wxy(i,j) = wy(j)
sum(i in RX) wx(i) = 1
sum(j in RY) wy(j) = 1

! Then x, y and f can be calculated using
x = sum(i in RX)  X(i)*wx(i)
y = sum(j in RY)  Y(j)*wy(j)
f = sum(i in RX,j in RY) FXY(i,j)*wxy(i,j)

! f can take negative or positive values (unbounded variable)
f is_free
```

**Python implementation**

```
import xpress as xp

NX = 10
NY = 10
RX = range(NX)
RY = range(NY)

X = [i+1 for i in RX]
Y = [j+1 for j in RY]
FXY = [[(i-4)*(j-4) for i in RX] for j in RY]

p = xp.problem()
```

```
# Create the decision variables
x = p.addVariable(name="x")
y = p.addVariable(name="y")
f = p.addVariable(lb=-xp.infinity, ub=xp.infinity, name="f")
wx = [p.addVariable(name="wx_{}".format(i)) for i in RX]
wy = [p.addVariable(name="wy_{}".format(i)) for i in RY]
wxy = [[p.addVariable(name="wxy_{}_{}".format(i,j)) for i in RX] for j in RY]

# Define the SOS-2 for coordinate x
p.addSOS(wx, X, name="sos_x", type=2)
# Define the SOS-2 for coordinate y
p.addSOS(wy, Y, name="sos_y", type=2)

# Weights must sum up to 1 along the two coordinates
p.addConstraint(xp.Sum(wx[i] for i in RX) == 1.0)
p.addConstraint(xp.Sum(wy[j] for j in RY) == 1.0)

# wx, wy and wxy must be consistent
p.addConstraint(wx[i] == xp.Sum(wxy[i][j] for j in  RY) for i in RX)
p.addConstraint(wy[j] == xp.Sum(wxy[i][j] for i in  RX) for j in RY)

# The coordinates and the corresponding function value we want to approximate
p.addConstraint(x == xp.Sum(X[i]*wx[i] for i in RX))
p.addConstraint(y == xp.Sum(Y[j]*wy[j] for j in RY))
p.addConstraint(f == xp.Sum(FXY[i][j]*wxy[i][j] for i in RX for j in RY))
```

### C++ implementation

```
#include <iostream>
#include <xpress.hpp>
using namespace xpress;
using namespace xpress::objects;
using xpress::objects::utils::sum;
using namespace std;

int main()
{
  XpressProblem prob;

  // Problem data
  static const int NX = 10;
  static const int NY = 10;

  std::array<double,NX> X;
  for (int i = 0; i < NX; i++)  X[i] = (double)(i+1);
  std::array<double,NY> Y;
  for (int j = 0; j < NY; j++)  Y[j] = (double)(j+1);
  std::array<std::array<double,NY>, NX> FXY;
  for (int i = 0; i < NX; i++) {
    for (int j = 0; j < NY; j++) {
      FXY[i][j] = (double)(i-4)*(j-4);
    }
  }

  // Create the decision variables
  auto x = prob.addVariable("x");
  auto y = prob.addVariable("y");
  auto f =
    prob.addVariable(XPRS_MINUSINFINITY, XPRS_PLUSINFINITY, ColumnType::Continuous, "f");
  auto wx = prob.addVariables(NX).withName("wx_%d").toArray();
  auto wy = prob.addVariables(NY).withName("wy_%d").toArray();
  auto wxy = prob.addVariables(NX, NY).withName("wxy_%d_%d").toArray();

  // Define the SOS-2 for coordinate x
  prob.addConstraint(SOS::sos(SetType::SOS2, wx, X, "sos_x"));
  // Define the SOS-2 for coordinate y
```

```
  prob.addConstraint(SOS::sos(SetType::SOS2, wy, Y, "sos_y"));

  // Weights must sum up to 1 along the two coordinates
  prob.addConstraint(sum(NX, [&](auto i) { return wx[i]; }) == 1.0);
  prob.addConstraint(sum(NY, [&](auto j) { return wy[j]; }) == 1.0);

  // wx, wy and wxy must be consistent
  prob.addConstraints(NX, [&](auto i) {
    return wx[i] == sum(NY, [&](auto j) { return wxy[i][j]; });
  });
  prob.addConstraints(NY, [&](auto j) {
    return wy[j] == sum(NX, [&](auto i) { return wxy[i][j]; });
  });

  // The coordinates and the corresponding function value we want to approximate
  prob.addConstraint(x == sum(NX, [&](auto i) { return X[i]*wx[i]; }));
  prob.addConstraint(y == sum(NY, [&](auto j) { return Y[j]*wy[j]; }));
  prob.addConstraint(f == sum(NX, [&](auto i) {
    return sum(NY, [&](auto j) { return FXY[i][j]*wxy[i][j]; });
  }));
...
```

## 3.5 Minimum activity level

Continuous production rate $\mathrm{make}$. May be 0 (the plant is not operating) or between allowed production limits $\mathrm{MAKEMIN}$ and $\mathrm{MAKEMAX}$

- Can impose using a *semi-continuous variable*: may be zero, or any value between the intermediate bound and the upper bound
- Semi-continuous variables are slightly more efficient than the alternative binary variable formulation that we saw before. But if you incur fixed costs on any non-zero activity, you must use the binary variable formulation (see Section 2.10).

**Mosel:**

```
make is_semcont MAKEMIN
make <= MAKEMAX
```

**Python:**

```
make = p.addVariable(vartype=xp.semicontinuous, threshold=MAKEMIN, ub=MAKEMAX, name="make")
```

**C++ API:**

```
auto make = prob.addVariable(0, MAKEMAX, ColumnType::SemiContinuous, "make");
make.setLimit(MAKEMIN);
```

## 3.6 Partial integer variables

- In general, try to keep the upper bound on integer variables as small as possible. This reduces the number of possible integer values, and so reduces the time to solve the problem.
- Sometimes this is not possible – a variable has a large upper bound and must take integer values. ⇒ Try to use *partial integer variables* instead of integer variables with a very large upper bound: takes integer values for small values, where it is important to be precise, but takes real values for larger values, where it is OK to round the value afterwards.

- For example, it may be important to clarify whether the value is 0, 1, 2, ..., 10, but above 10 it is OK to get a real value and round it.

**Mosel:**

```
x is_partint 10          ! x is integer valued from 0 to 10
x <= 20                  ! x takes real values from 10 to 20
```

**Python:**

```
make = p.addVariable(vartype=xp.partiallyinteger, threshold=10, ub=20, name="make")
```

**C++ API:**

```
auto x = prob.addVariable(0, 20, ColumnType::PartialInteger, "x");   // x has upper bound 20
x.setLimit(10);                                                      // x is integer valued from 0 to 10
```

## 3.7 General constraints

Certain nonlinear constraint relations (refered to as *general constraints*) are recognized by MIP solvers and they are treated as MIP modeling constructs. These include

- piecewise linear expressions (see examples in Sections 3.3 and 3.4 above)
- absolute value, minimum value, maximum value of discrete or continuous decision variables
- logical constraints: 'and' and 'or' over binary variables (see Section 5 )

**Mosel:** The Mosel implementation of the numerical constraints `abs`, `fmin` and `fmax` makes it possible to use linear expressions as argument to these functions. Note that models using these general constraints need to load the *mmxnlp* module in addition to *mmxprs* although the problems will be solved by the Xpress MIP solver.

```
uses "mmxnlp"

public declarations
  R=1..3
  x: array(R) of mpvar
  y,z: mpvar
end-declarations

forall(i in R) x(i) <= 20
abs(x(1)-2*x(2)) <= 10                    ! Absolute value constraint
MinCtr:= fmin(union(i in R) [x(i)]) >= 5   ! Minimum value constraint
y = fmax(x(3), 20, x(1)-z)                ! Maximum value constraint

maximize(sum(i in R) x(i))                ! Solve as MIP problem
if getprobstat=XPRS_OPT then              ! Solution reporting
  writeln("Solution: ", getobjval)
  forall(i in R) write("x", i, "=", x(i).sol, ", ")
  writeln("y=", y.sol, ", z=", z.sol)
  writeln("abs=", getsol(abs(x(1)-2*x(2))), ", Min of x(i)=", MinCtr.sol+5)
else
  writeln("No solution")
end-if
```

**Python:** The Python implementation of the numerical constraints `xp.abs`, `xp.min` and `xp.max` makes it possible to use linear expressions as argument to these functions.

```
import xpress as xp

R = range(3)

p = xp.problem()

x = [p.addVariable(name="x_{}".format(i)) for i in R]
y = p.addVariable()
z = p.addVariable()

p.addConstraint(x[i] <= 20 for i in R)
p.addConstraint(xp.abs(x[0] - 2*x[1]) <= 10)      # Absolute value constraint
p.addConstraint(xp.min(x) >= 5)                   # Minimum value constraint
p.addConstraint(y == xp.max(x[2], 20, x[0]-z))    # Maximum value constraint

p.setObjective(xp.Sum(x), sense=xp.maximize)

p.optimize()                                      # Solve as MIP problem

if p.attributes.solstatus in [SolStatus.FEASIBLE",SolStatus."OPTIMAL"]:
    print("Solution:", p.attributes.objval())
    for i in R:
        print(x[i].name,"=",p.getSolution(x[i]))
    print("y=", p.getSolution(y), ", z=", p.getSolution(z))
    print("abs=", p.getSolution(xp.abs(x[0]-2*x[1])), ", Min of x(i)=", min(p.getSolution(x)))
else:
    print("No solution")
```

**C++ API:** The implementation of general constraints with the C++ API introduces auxiliary variables for the formulation of the constraints since `absOf`/`minOf`/`maxOf` are not defined for expressions, they expect decision variables as their arguments.

```
using namespace xpress;
using namespace xpress::objects;
using xpress::objects::utils::sum;
using namespace std;

int main()
{
  XpressProblem prob;

  // Create the decision variables
  auto x = prob.addVariables(3).withName("x_%d").withUB(20).toArray();
  auto y = prob.addVariable("y");
  auto z = prob.addVariable("z");

  // abs(x_0 - 2*x_1) <= 10
  auto diff1 = prob.addVariable("diff1");
  auto absOfDiff1 = prob.addVariable("absOfDiff1");
  prob.addConstraint(diff1 == x[0] - 2*x[1]);
  prob.addConstraint(absOfDiff1.absOf(diff1));
  absOfDiff1.setUB(10);

  // min(x_i) >= 5
  auto minOfX = prob.addVariable("minOfX");
  prob.addConstraint(minOfX.minOf(x));
  minOfX.setLB(5);

  // y = max(x_2, 20, x_0 - z)
  auto diff2 = prob.addVariable("diff2");
  prob.addConstraint(diff2 == x[0] - z);
  std::array<Variable,2> maxVarArgs = {x[2], diff2};
  prob.addConstraint(y.maxOf(maxVarArgs, 20));

  // Objective
  prob.setObjective(sum(x), ObjSense::Maximize);
```

```
  // Solve the problem
  prob.optimize();
...
```

# 4 Indicator constraints

- Indicator constraints associate a binary variable $b$ with a linear or nonlinear constraint $C$.
- An indicator constraint models an implication:
  'if $b = 1$ then $C$', in symbols: $b \rightarrow C$, or
  'if $b = 0$ then $C$', in symbols: not$(b) \rightarrow C$
  (the constraint $C$ is active only if the condition is true)
- Indicator constraints can be used for the composition of logic expressions.

**Indicator constraints in Mosel:** for the definition of indicator constraints (function `indicator` of module *mmxprs*) you need a binary variable (type `mpvar`) and a linear or nonlinear constraint (type `linctr` or `nlctr`). You also have to specify the type of the implication (1 for $b \rightarrow C$ and -1 for not$(b) \rightarrow C$). The subroutine `indicator` returns a new constraint of type `logctr` (the type `logctr` and the corresponding subroutines including `indicator` are documented in the chapter *mmxprs* of the Mosel Language Reference Manual).

```
uses "mmxprs"

declarations
 R=1..2
 C: array(range) of linctr
 L: array(range) of logctr
 x, b: array(R) of mpvar
end-declarations

forall(i in R) b(i) is_binary   ! Variables for indicator constraints

C(2):= x(2)<=5

! Define 2 indicator constraints
L(1):= indicator(1, b(1), x(1)+x(2)>=12)    ! b(1)=1 -> x(1)+x(2)>=12
indicator(-1, b(2), C(2))                   ! b(2)=0 -> x(2)<=5

C(2):=0                          ! Delete auxiliary constraint definition
```

The module *mmxnlp* must be used in place of *mmxprs* if the indicator constraints are defined over nonlinear constraints:

```
uses "mmxnlp"

declarations
 R=1..2
 C: array(range) of nlctr
 L: array(range) of logctr
 x, b: array(R) of mpvar
end-declarations

forall(i in R) b(i) is_binary   ! Variables for indicator constraints

C(2):= sin(x(2))>=0.5

! Define 2 indicator constraints
L(1):= indicator(1, b(1), x(1)*x(2)>=12)    ! b(1)=1 -> x(1)*x(2)=12
indicator(-1, b(2), C(2))                   ! b(2)=0 -> sin(x(2))>=0.5
```

```
C(2):=0                              ! Delete auxiliary constraint definition
```

**Indicator constraints in the Python API:** indicator constraints are defined via the method `addIndicator` of `xp.problem`. All arguments can be single indicator constraints or lists, tuples, or NumPy arrays created as indicator constraints. An indicator constraint is a tuple of two elements, the first being a condition (i.e. a binary variable being 0 or 1) and the second being the constraint.

```
import xpress as xp

N = 2

p = xp.problem()

# Create the decision variables
x = [p.addVariable(name="x_{}".format(i),vartype=xp.continuous) for i in range(N)]
b = [p.addVariable(name="b_{}".format(i),vartype=xp.binary) for i in range(N)]

# b[0] = 1 -> x[0]+x[1] >= 12
p.addIndicator(b[0] == 1, x[0] + x[1] >= 12)
# b[1] = 0 -> x[1] <= 5
p.addIndicator(b[1] == 0, x[1] <= 5)
```

**Indicator constraints in the C++ API:** indicator constraints are defined via the method `addConstraint` of `XpressProblem`.The type of the implication is specified by using either `ifThen` (for $b \rightarrow C$) or `ifNotThen` (for not$(b) \rightarrow C$).

```
XpressProblem prob;

// Create the decision variables
auto x = prob.addVariables(N).withType(ColumnType::Continuous).withName("x_%d").toArray();
auto b = prob.addVariables(N).withType(ColumnType::Binary).withName("b_%d").toArray();

// Define 2 indicator constraints:
// b[0] = 1 -> x[0]+x[1] >= 12
prob.addConstraint(b[0].ifThen(x[0] + x[1] >= 12));
// b[1] = 0 -> x[1] <= 5
prob.addConstraint(b[1].ifNotThen(x[1] <= 5));
```

## 4.1 Inverse implication

$b \leftarrow a \cdot x \geqslant c$

- Model as

  not$(b) \rightarrow a \cdot x \leqslant c - m$

  where $m$ is a sufficiently small value (slightly larger than the feasibility tolerance)

$b \leftarrow a \cdot x \leqslant c$

- Model as

  not$(b) \rightarrow a \cdot x \geqslant c + m$

$b \leftarrow a \cdot x = c$

- Model as

$\text{not}(b) \rightarrow b_1 + b_2 = 1$
$b_1 \rightarrow a \cdot x \geqslant c + m$
$b_2 \rightarrow a \cdot x \leqslant c - m$

## 4.2 Logical constructs

Indicator constraints can be used for the formulation of logical constructs composed of linear or nonlinear equality or inequality constraints and the logic functions 'implies', 'not', 'and', 'or', or 'xor' by applying a set of *recipes*.

In the following we shall be using the following definitions:

| | |
|---|---|
| *C (constraint)* | a linear or nonlinear equality or inequality constraint |
| *IC (indicator contraint)* | can be either like $y \rightarrow C$ or $not(y) \rightarrow C$ |
| *LE (logical expression)* | either a C or one of the following functions (of C's or LE's): AND, OR, XOR, NOT, IMPLIES, such as `IMPLIES(x(1)>=10, AND(x(1)+x(2)>=12, NOT(x(2)<=5)))` |

To model logical constructs we associate a binary indicator variable $y(e)$ to each LE $e$, representing its truth value. For each pair $(y(e), e)$, we may need to enforce either that $y(e) \rightarrow e$ or $y(e) \leftarrow e$ or both, that is, $y(e) \leftrightarrow e$.

We will write

- $is(e)$ (for **Implie*S***) to refer to the implication $y(e) \rightarrow e$, and
- $id(e)$ (for **Implie*D***) to refer to the implication $y(e) \leftarrow e$ which is equivalent to $not(y(e)) \rightarrow not(e)$.

To model a logical construct, we proceed recursively from the *outer LE*. Let $e$ be the outer LE, then we can model it as follows:

add constraint $y(e) = 1$
model $is(e)$ where the recipe to "model $is(e)$" is given below.

### 4.2.1 Rules to model logical constructs

The following set of rules can be applied to model each type of LE (with $y$ being its associated indicator variable).

- C (***e.g.*** $a \cdot x \geqslant b$)
  - *is* direction

    model it in the obvious way as an indicator constraint (***e.g.*** $y \rightarrow a \cdot x \geqslant b$)
  - *id* direction

    model it as an inverse implication (***e.g.*** $not(y) \rightarrow a \cdot x \leqslant b - m$)

    Here $m$ is a parameter whose default value should be somewhat larger than the feasibility tolerance. When the involved constraints are all integer, $m$ could be set equal to 1.
- AND$(e_1, e_2, ..., e_n)$

- *is* direction

  add $n$ contraints: $y \rightarrow y(e_i) = 1 \forall i$
  model $is(e_i) \forall i$

- *id* direction

  add contraint: not$(y) \rightarrow \sum_{i=1}^{n} y(e_i) \leqslant n - 1$
  model $id(e_i) \forall i$

- OR$(e_1, e_2, ..., e_n)$

  - *is* direction

    add contraint: $y \rightarrow \sum_{i=1}^{n} y(e_i) \geqslant 1$
    model $is(e_i) \forall i$

  - *id* direction

    add contraint: not$(y) \rightarrow \sum_{i=1}^{n} y(e_i) = 0$
    model $id(e_i) \forall i$

- XOR$(e_1, e_2, ..., e_n)$

  - *is* direction

    add constraint: $y \rightarrow \sum_{i=1}^{n} y(e_i) = 1$
    model $is(e_i)$ and $id(e_i) \forall i$

  - *id* direction

    add two auxiliary variables $y'$ and $y"$ and the constraints
    $y' \rightarrow \sum_{i=1}^{n} y(e_i) = 0$
    $y" \rightarrow \sum_{i=1}^{n} y(e_i) \geqslant 2$
    $not(y) \rightarrow y' + y" = 1$
    model $is(e_i)$ and $id(e_i) \forall i$

- NOT$(e)$

  - *is* direction

    add constraint: $y + y(e) = 1$
    model $id(e)$

  - *id* direction

    add constraint: $y + y(e) = 1$
    model $is(e)$

- IMPLIES$(e_1, e_2)$ (same as: $e_1 \rightarrow e_2$)

  - *is* direction

    add constraint: $y \rightarrow y(e_1) \leqslant y(e_2)$
    model $id(e_1)$ and $is(e_2)$

  - *id* direction

    add constraints:
    $not(y) \rightarrow y(e_1) = 1$
    $not(y) \rightarrow y(e_2) = 0$
    model $is(e_1)$ and $id(e_2)$

### 4.2.2 Example

By applying the reformulation rules from the previous section to an expression `LE` that is defined as `LE = IMPLIES(x(1)>=10, AND(x(1)+x(2)>=12, NOT(x(2)<=5)))` we obtain the following:

```
start:
  associate y to the outermost expression "LE=IMPLIES(...)"
  add constraint: y = 1
  model is(LE)

model is(LE) yields:
  associate y1 to "e1=x(1)>=10" and y2 to "e2=AND(...)"
  add constraint: y -> y1 = y2
  model id(e1) and is(e2)

model id(e1) yields:
  add constraint: not(y1) -> x(1)=10-m

model is(e2) yields:
  associate y3 to "e3=x(1)+x(2)>=12" and y4 to "e4=NOT(x(2)=5)"
  add constraints:
  y2 -> y3=1
  y2 -> y4=1
  model is(e3) and is(e4)

model is(e3) yields:
  add constraint: y3 -> x(1)+x(2)>=12

model is(e4) yields:
  associate y5 with "e5=x(2)=5"
  add constraint: y4 + y5 = 1
  model id(e5)

model id(e5) yields:
  add constraint: not(y5) -> x(2)>=5+m
```

In summary, the resulting model formulation is as follows (where the outer fixed indicator constraint can be removed):

```
binaries y, y1, y2, y3, y4, y5
y = 1
y -> y1 = y2
not(y1) -> x(1)=10-m
y2 -> y3=1
y2 -> y4=1
y3 -> x(1)+x(2)>=12
y4 + y5 = 1
not(y5) -> x(2)>=5+m
```

# 5 Boolean variables and logical constraints

The Mosel module *mmxprs* defines the entity type `boolvar` for representing a *pseudo boolean decision variable*. This type supports the operators `and`, `or` and `not` for building logical expressions and can be combined with ordinary Boolean variables (type `boolean`). A logical constraint is specified either by associating a pseudo boolean variable to a logical expression or by forcing the truth value (i.e. true or false) of an expression as shown in the code example below. When a logical expression is used on its own as a statement it is implicitly turned into a constraint (forced to `true`) and added to the constraint store.

```
uses "mmxprs"

public declarations
```

```
  R=1..5
  bv: array(R) of boolvar
  LC1, LC2: logctr
end-declarations

! Simple clause, same as:  bv(1) and not bv(5) = true
bv(1) and not bv(5)

! Association of clauses
bv(3)=(not bv(4))

! The opposite of 'bv(1) or bv(3)' must be false
(not (bv(1) or bv(3)))=false

! Defining a logic expression (not recorded in the constraint store)
LC1:= and(i in 1..3) bv(i) or and(i in 4..5) not bv(i)

! Turn expression into a constraint
LC1:= LC1=true

! A named logic expression (this defines a constraint)
LC2:= (or(i in 1..3) not bv(i)) = false

! Solve as feasibility (SAT) problem
maximise(0)
if getprobstat=XPRS_OPT then
  writeln("Problem is feasible")
else
  writeln("Problem is unsatisfiable")
end-if
```

## 5.1 Correspondence with MIP

Each `boolvar` is represented in the MIP problem by two binary variables (`mpvar`): one for the value itself and second one for its negation. These decision variables can be accessed from the model using the function `getvar` such that they can be used in linear constraints. The solution value of a `boolvar` is of type `boolean` and can be obtained using `getsol`.

```
! Retrieve associated binary variables for formulation of an objective function
Obj:=sum(i in R) bv(i).var

! Solve as optimization problem
maximise(Obj)
if getprobstat=XPRS_OPT then
  writeln("Solution: ", getobjval)
  forall(i in R) writeln(i, ": ", bv(i).sol)
end-if
```

The problem matrix can be output from the solver to inspect the resulting formulation: note that the logic relations are represented via *general constraints* by the MIP solver.

```
loadprob(Obj)
writeprob("testout.lp","l")
```