FICO^{TM} Xpress Optimization Suite

MIP formulations and linearizations

Quick reference

Last update 29 June, 2009

Published by Fair Isaac Corporation

How to Contact the Xpress Team

Information, Sales and Licensing

USA, CANADA AND ALL AMERICAS

Email: XpressSalesUS@fico.com

WORLDWIDE

Email: XpressSalesUK@fico.com

Tel: +44 1926 315862
Fax: +44 1926 315854

FICO, Xpress team
Leam House, 64 Trinity Street
Leamington Spa
Warwickshire CV32 5YN
UK

Product Support

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

Telephone:

NORTH AMERICA
Tel (toll free): +1 (877) 4FI-SUPP
Fax: +1 (402) 496-2224

EUROPE, MIDDLE EAST, AFRICA
Tel: +44 (0) 870-420-3777
UK (toll free): 0800-0152-153
South Africa (toll free): 0800-996-153
Fax: +44 (0) 870-420-3778

ASIA-PACIFIC, LATIN AMERICA, CARIBBEAN
Tel: +1 (415) 446-6185
Brazil (toll free): 0800-891-6146

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 Suite

MIP formulations and linearizations

Quick reference

29 June, 2009

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.

```
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_TOP` (if following up with MIP search) or `XPRS_LIN` (ignore all MIP information) for `maximize` / `minimize`.

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("LP unbounded or infeasible")
 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 BCL

The BCL code extracts in this document are formulated for the BCL C++ interface. The other BCL interfaces (C, Java, C#, VB) work similarly, please refer to the Xpress documentation for further detail.

Definition: Integer Programming types are specified when creating decision variables (type `XPRBvar`); types may be changed with `setType`.

```
XPRBprob prob("test");
XPRBvar d, ifmake[NP][NL], x;
int p,l;

d = prob.newVar("d", XPRB_BV);          // Single binary variable

for (p = 0; p < NP; p++)                 // An array of binaries
 for (l = 0; l < NL; l++)
  ifmake[p][l] = prob.newVar("ifmake", XPRB_BV);

x = prob.newVar("x", XPRB_UI, MINVAL, MAXVAL);  // An integer variable
// MINVAL,MAXVAL: reals between -XPRB_INFINITY and XPRB_INFINITY
...
x.setType(XPRB_PI);                     // Change type to partial integer
x.setLim(10);
...
x.setType(XPRB_PL);                     // Change type to continuous
```

Solving: use option "g" in `solve` / `minim` / `maxim` to solve a problem as a MIP problem (per default only the LP relaxation is solved).

```
prob.minim("g");
```

Accessing the solution: for obtaining solution values of decision variables and linear expressions use `getSol`; the MIP problem status is returned by `getMIPstatus`.

```
int mipstatus = prob.getMIPStat();
switch (mipstatus) {
 case XPRB_MIP_NOT_LOADED:
 case XPRB_MIP_LP_NOT_OPTIMAL:
   cout << "Solving not started" << endl;
   break;
 case XPRB_MIP_LP_OPTIMAL:
   cout << "LP unbounded or infeasible" << endl;
   break;
 case XPRB_MIP_NO_SOL_FOUND:
 case XPRB_MIP_INFEAS:
   cout << "MIP search started, no solution" << endl;
   break;
 case XPRB_MIP_SOLUTION:
```



```
    case XPRB_MIP_OPTIMAL:
      cout << "MIP solution: " << prob.getObjVal() << endl;
      break;
   }
   cout << x.getName() << ": " << x.getSol() << 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*, ... 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,...	$a + b + c + \ldots \leq N$
At least N of A, B, C,...	$a + b + c + \ldots \geq N$
Exactly N of A, B, C,...	$a + b + c + \ldots = N$
If A then B	$b \geq a$
Not B	$\bar{b} = 1 - b$
If A then not B	$a + b \leq 1$
If not A then B	$a + b \geq 1$
If A then B, and if B then A	$a = b$
If A then B and C; A only if B and C	$b \geq a$ and $c \geq a$
	or alternatively: $a \leq (b + c) / 2$
If A then B or C	$b + c \geq a$
If B or C then A	$a \geq b$ and $a \geq c$
	or alternatively: $a \geq \frac{1}{2} \cdot (b + c)$
If B and C then A	$a \geq b + c - 1$
If two or more of B, C, D or E then A	$a \geq \frac{1}{3} \cdot (b + c + d + e - 1)$
If M or more of N projects (B, C, D, ...) then A	$a \geq \frac{b+c+d+\ldots-M+1}{N-M+1}$

2.2 Minimum values

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

- Must know lower and upper bounds

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

- Introduce binary variables d_1, d_2 to mean

d_i 1 if x_i is the minimum value;
0 otherwise

- MIP formulation:

$$\begin{array}{ll} y \le x_1 & [2.1] \\ y \le x_2 & [2.2] \\ y \ge x_1 - (U_1 - L_{min})(1 - d_1) & [3.1] \\ y \ge x_2 - (U_2 - L_{min})(1 - d_2) & [3.2] \\ d_1 + d_2 = 1 & [4] \end{array}$$

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

$$\begin{array}{ll} L_i \le x_i \le U_i & [1.i] \\ y \le x_i & [2.i] \\ y \ge x_i - (U_i - L_{min})(1 - d_i) & [3.i] \\ \sum_i d_i = 1 & [4] \end{array}$$

2.3 Maximum values

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

- Must know lower and upper bounds

$$L_i \le x_i \le U_i \qquad [1.i]$$

- Introduce binary variables $d_1, \ldots, d_n$
  $d_i = 1$ if x_i is the minimum value, 0 otherwise

- MIP formulation

$$\begin{array}{ll} L_i \le x_i \le U_i & [1.i] \\ y \ge x_i & [2.i] \\ y \le x_i + (U_{max} - L_i)(1 - d_i) & [3.i] \\ \sum_i d_i = 1 & [4] \end{array}$$

2.4 Absolute values

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

- 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

$$
\begin{array}{ll}
0 \leq x_i \leq U & [1.i] \\
0 \leq y - (x_1 - x_2) \leq 2 \cdot U \cdot d_2 & [2] \\
0 \leq y - (x_2 - x_1) \leq 2 \cdot U \cdot d_1 & [3] \\
d_1 + d_2 = 1 & [4]
\end{array}
$$

2.5 Logical AND

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

- IP formulation

$$
\begin{array}{ll}
d \leq d_1 & [1.1] \\
d \leq d_2 & [1.2] \\
d \geq d_1 + d_2 - 1 & [2] \\
d \geq 0 & [3]
\end{array}
$$

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

$$
\begin{array}{ll}
d \leq d_i & [1.i] \\
d \geq \sum_i d_i - (n - 1) & [2] \\
d \geq 0 & [3]
\end{array}
$$

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

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

$$
\begin{array}{ll}
d \geq d_1 & [1.1] \\
d \geq d_2 & [1.2] \\
d \leq d_1 + d_2 & [2] \\
d \leq 1 & [3]
\end{array}
$$

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

$$
\begin{array}{ll}
d \geq d_i & [1.i] \\
d \leq \sum_i d_i & [2.i] \\
d \leq 1 & [3]
\end{array}
$$

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

2.7 Logical NOT

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

- IP formulation

  $d = 1 - d_1$

2.8 Product values

y = x · d for one continuous variable x, one binary variable d

- Must know lower and upper bounds

  $L \leq x \leq U$

- MIP formulation:

  $Ld \leq y \leq Ud$ [1]
  $L(1-d) \leq x - y \leq U(1-d)$ [2]

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

- MIP formulation:

  $d_3 \leq d_1$
  $d_3 \leq d_2$
  $d_3 \geq d_1 + d_2 - 1$

2.9 Disjunctions

Either $\mathbf{5 \leq x \leq 10}$ or $\mathbf{80 \leq x \leq 100}$

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

  $x \leq 10 + (100 - 10) \cdot ifupper$ [1]
  $x \geq 5 + (80 - 5) \cdot ifupper$ [2]

- Generalization to **Either $\mathbf{L_1 \leq \sum_i A_i x_i \leq U_1}$ or $\mathbf{L_2 \leq \sum_i A_i x_i \leq U_2}$ (with $\mathbf{U_1 \leq L_2}$)**

  $\sum_i A_i x_i \leq U_1 + (U_2 - U_1) \cdot ifupper$ [1]
  $\sum_i A_i x_i \geq L_1 + (L_2 - L_1) \cdot ifupper$ [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:

  $$\begin{array}{ll} make \geq MAKEMIN \cdot ifmake & [1] \\ make \leq MAKEMAX \cdot ifmake & [2] \end{array}$$

  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:

  $$\begin{array}{ll} cost = FCOST \cdot ifmake + VCOST \cdot make & [3] \\ make \geq MAKEMIN \cdot ifmake & [1] \\ make \leq MAKEMAX \cdot ifmake & [2] \end{array}$$

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

3.1 Batch sizes

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

- Decision variables

  nship number of batches delivered: integer
  ship quantity delivered: continuous

- Constraint formulation

  $ship = 10 \cdot nship$

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: $ret = \sum_i RET_i d_i$
  $\sum_i d_i \leq 1$
  $\sum_i CAP_i d_i \leq MAXCAP$

- SOS1 formulation

  $\{\mathrm{d}_i$; ordering value $\mathrm{CAP}_i\}$: SOS1

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

- 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):

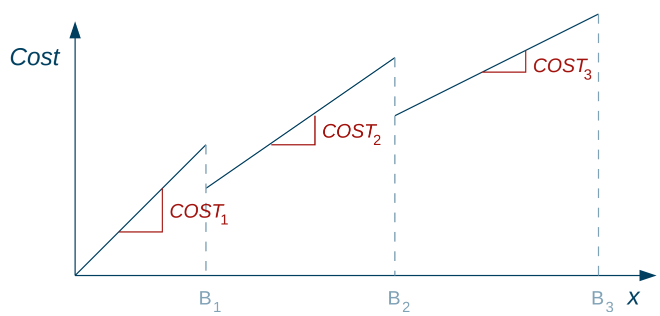
```
Ref_row:= sum(i in I) CAP(i)*d(i)
makesos1(My_Set, union(i in I) d(i), Ref_row)
```

- Special ordered sets in BCL
  - a special ordered set is an object of type `XPRBsos`
  - the set includes all variables from the specified linear expression or constraint that have a coefficient different from 0
  - the coefficient of a variable is used as the ordering value (*i.e.*, each value must be unique)

```
XPRBprob prob("testsos");
XPRBvar d[I];
XPRBexpr le;
XPRBsos My_Set;
double CAP[I];
int i;

for(i=0; i<I; i++) d[i] = prob.newVar("d");

for(i=0; i<I; i++) le += CAP[i]*d[i];
My_Set = prob.newSos("My_Set", XPRB_S1, le);
```

3.3 Price breaks

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
$\geq B_1$ and $< B_2$	$COST_2$ each
$\geq B_2$ and $< B_3$	$COST_3$ each

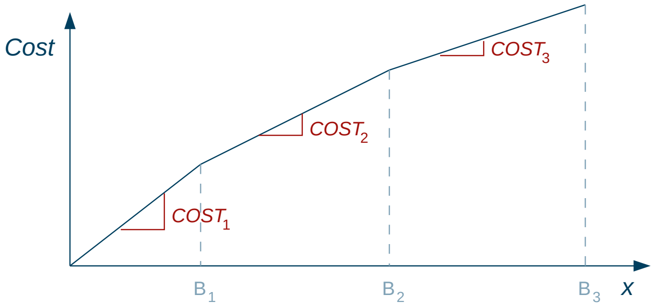
- Define binary variables b_i (i=1,2,3), where b_i is 1 if we pay a unit cost of $COST_i$.
- Real decision variables x_i represent the number of items bought at price $COST_i$.

- The total cost is given by $x = \sum_i x_i$
- MIP formulation :

  $\sum_i b_i = 1$
  $x_1 \leq B_1 \cdot b_1$
  $B_{i-1} \cdot b_i \leq x_i \leq B_i \cdot b_i$ 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.

Incremental pricebreaks: when buying a certain number of items we get discounts incrementally. The unit cost for items between 0 and B_1 is C_1, items between B_1 and B_2 cost C_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):

  $CBP_0 = 0$
  $CBP_i = CBP_{i-1} + C_i \cdot (B_i - B_{i-1})$ 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)
```

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 x_i (i=1,..3) for the number of items bought at price $COST_i$.
- Total amount bought: $x = \sum_i x_i$
- Constraint formulation:

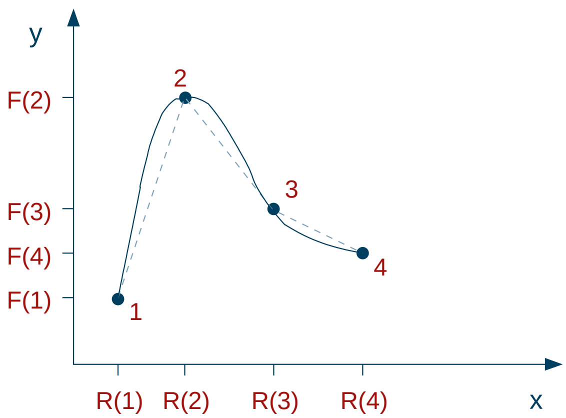
$$(B_i - B_{i-1}) \cdot b_{i+1} \leq x_i \leq (B_i - B_{i-1}) \cdot b_i \text{ for } i = 1, 2$$
$$x_3 \leq (B_3 - B_2) \cdot b_3$$
$$b_1 \geq b_2 \geq b_3$$

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

Non-linear function in a single variable



- x-coordinates of the points: R_1, ..., R_4
  y-coordinates F_1, ..., F_4. So point 1 is (R_1, F_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):

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

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

- Mosel implementation:

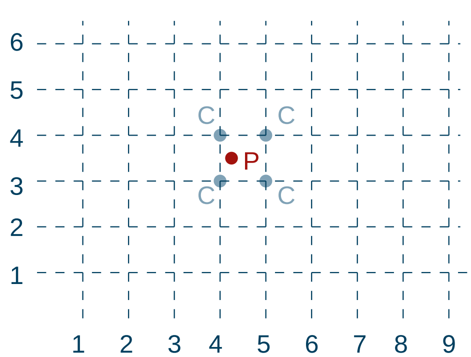
```
declarations
 I=1..4
 x,y: mpvar
 w: array(I) of mpvar
 R,F: array(I) of real
end-declarations

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

! 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) F(i)*w(i)
```

- BCL implementation:

```
 XPRBprob prob("testsos");
 XPRBvar x, y, w[I];
 XPRBexpr Defx, le, ly;
 double R[I], F[I];
 int i;

// ...assign values to arrays R and F...

// Create the decision variables
 x = prob.newVar("x"); y = prob.newVar("y");
 for(i=0; i<I; i++) w[i] = prob.newVar("w");

// Define the SOS-2 with "reference row" coefficients from R
 for(i=0; i<I; i++) Defx += R[i]*w[i];
 prob.newSos("Defx", XPRB_S2, Defx);
 for(i=0; i<I; i++) le += w[i];
 prob.newCtr("One", le == 1);

// The variable and the corresponding function value we want to approximate
 prob.newCtr("Eqx", x == Defx);
 for(i=0; i<I; i++) ly += F[i]*w[i];
 prob.newCtr("Eqy", y == ly);
```

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:

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

  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
```

- BCL implementation:

```
 XPRBprob prob("testsos");
 XPRBvar x, y, f;
 XPRBvar wx[NX], wy[NY], wxy[NX][NY];        // Weights on coordinates
 XPRBexpr Defx, Defy, le, lexy, lx, ly;
 double DX[NX], DY[NY];
 double FXY[NX][NY];
 int i,j;

// ... initialize data arrays DX, DY, FXY

// Create the decision variables
 x = prob.newVar("x"); y = prob.newVar("y");
 f = prob.newVar("f", XPRB_PL, -XPRB_INFINITY, XPRB_INFINITY);   // Unbounded variable
 for(i=0; i<NX; i++) wx[i] = prob.newVar("wx");
 for(j=0; j<NY; j++) wy[j] = prob.newVar("wy");
 for(i=0; i<NX; i++)
  for(j=0; j<NY; j++) wxy[i][j] = prob.newVar("wxy");

// Definition of SOS
 for(i=0; i<NX; i++) Defx += X[i]*wx[i];
 prob.newSos("Defx", XPRB_S2, Defx);
 for(j=0; j<NY; j++) Defy += Y[j]*wy[j];
 prob.newSos("Defy", XPRB_S2, Defy);

// Constraints
 for(i=0; i<NX; i++) {
  le = 0;
  for(j=0; j<NY; j++) le += wxy[i][j];
  prob.newCtr("Sumx", le == wx[i]);
 }
 for(j=0; j<NY; j++) {
  le = 0;
  for(i=0; i<NX; i++) le += wxy[i][j];
  prob.newCtr("Sumy", le == wy[j]);
 }
 for(i=0; i<NX; i++) lx += wx[i];
 prob.newCtr("Convx", lx == 1);
 for(j=0; j<NY; j++) ly += wy[j];
 prob.newCtr("Convy", ly == 1);

// Calculate x, y and the corresponding function value f we want to approximate
 prob.newCtr("Eqx", x == Defx);
 prob.newCtr("Eqy", y == Defy);
 for(i=0; i<NX; i++)
  for(j=0; j<NY; j++) lexy += FXY[i][j]*wxy[i][j];
 prob.newCtr("Eqy", f == lexy);
```

3.5 Minimum activity level

Continuous production rate *make*. May be 0 (the plant is not operating) or between allowed production limits *MAKEMIN* and *MAKEMAX*

- Can impose using a *semi-continuous variable*: may be zero, or any value between the intermediate bound and the upper bound
- Mosel:

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

- BCL:

```
make = prob.newVar("make", XPRB_SC, 0, MAKEMAX);
make.setLim(MAKEMIN);
```

- 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).

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.
  $\Rightarrow$ 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
```

- BCL:

```
x = prob.newVar("x", XPRB_PI, 0, 20);
x.setLim(10);
```

4 Indicator constraints

Indicator constraints

- associate a binary variable b with a linear constraint C
- model an implication:
  'if $b = 1$ then C', in symbols: $b \rightarrow C$, or
  'if $b = 0$ then C', in symbols: $\bar{b} \rightarrow C$
  (the constraint C is active only if the condition is true)
- use indicator constraints 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 inequality constraint (type `linctr`). You also have to specify the type of the implication (1 for $b \rightarrow C$ and -1 for $\bar{b} \rightarrow C$). The subroutine `indicator` returns a new constraint of type `logctr` that can be used in the composition of other logic expressions (see Section 4.2 below).

```
uses "mmxprs"

declarations
 R=1..10
 C: array(range) of linctr
 L: array(range) of logctr
 x: array(R) of mpvar
 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
```

Indicator constraints in BCL: an indicator constraint is defined by associating a binary decision variable (`XPRBvar`) and an integer flag (1 for $b \rightarrow C$ and -1 for $\bar{b} \rightarrow C$) with a linear inequality or range constraint (`XPRBctr`). By defining an indicator constraint (function `XPRBsetindicator` or method `XPRBctr.setIndicator()` depending on the host language) the type of the constraint itself gets changed; it can be reset to 'standard constraint' by calling the `setIndicator` function with flag value 0.

```
 XPRBprob prob("testind");
 XPRBvar x[N], b[N];
 XPRBctr IndCtr[N];
 int i;

// Create the decision variables
 for(i=0;i<N;i++) x[i] = prob.newVar("x", XPRB_PL);  // Continuous variables
 for(i=0;i<N;i++) b[i] = prob.newVar("b", XPRB_BV);  // Indicator variables

// Define 2 linear inequality constraints
 IndCtr[0] = prob.newCtr("L1", x[0]+x[1]>=12);
 IndCtr[1] = prob.newCtr("L2", x[1]<=5);

// Turn the 2 constraints into indicator constraints
 IndCtr[0].setIndicator(1, b[0]);              // b(0)=1 -> x(0)+x(1)>=12
 IndCtr[1].setIndicator(-1, b[1]);             // b(1)=0 -> x(1)<=5
```

4.1 Inverse implication

$b \leftarrow ax \geq b$

- Model as

  $\bar{b} \rightarrow ax \leq b - m$

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

$b \leftarrow ax \leq b$

- Model as

  $\bar{b} \rightarrow ax \geq b + m$

$b \leftarrow ax = b$

- Model as

$\bar{b} \rightarrow b_1 + b_2 = 1$
$b_1 \rightarrow ax \geq b + m$
$b_2 \rightarrow ax \leq b - m$

4.2 Logic constructs

Mosel provides the type `logctr` for defining and working with logic constraints in MIP models. The implementation of these constraints is based on indicator constraints. Logic constraints are composed with linear constraints using the operations `and`, `or`, `xor`, `implies`, and `not` as shown in the following example. Mosel models using logic constraints must include the package `advmod` instead of the Optimizer library `mmxprs`.

```
 uses "advmod"

! **** 'implies', 'not', and 'and' ****
 declarations
  R = 1..3
  C: array(range) of linctr
  x: array(R) of mpvar
 end-declarations

 C(1):= x(1)>=10
 C(2):= x(2)<=5
 C(3):= x(1)+x(2)>=12

 implies(C(1), C(3) and not C(2))
 forall(j in 1..3) C(j):=0              ! Delete the auxiliary constraints

! Same as:
 implies(x(1)>=10, x(1)+x(2)>=12 and not x(2)<=5)

! **** 'or' and 'xor' ****
 declarations
  p: array(1..6) of mpvar
 end-declarations

 forall(i in 1..6) p(i) is_binary

! Choose at least one of projects 1,2,3 (option A)
! or at least two of projects 2,4,5,6 (option B)
  p(1) + p(2) + x(3) >= 1 or p(2) + p(4) + p(5) + p(6) >= 2

! Choose either option A or option B, but not both
  xor(p(1) + p(2) + p(3) >= 1, x(2) + p(4) + p(5) + p(6) >= 2)
```

These logic constructs, particularly the logic `or`, can be used for the formulation of minimum or maximum values of a set of variables and also for absolute values:

- **Minimum values: y = min$\{$x$_1$, x$_2$, . . . , x$_n\}$** for continuous variables $x_1, \ldots, x_n$
  - Logic formulation:

    $y \leq x_i \forall i = 1, \ldots, n$
    $y \geq x_1$ or ... or $y \geq x_n$

- **Maximum values: $\mathbf{y = max\{x_1, x_2, \ldots, x_n\}}$** for continuous variables $x_1, \ldots, x_n$
  - Logic formulation:

    $y \geq x_i \quad \forall i = 1, \ldots, n$
    $y \leq x_1$ or ... or $y \leq x_n$

- **Absolute values: $\mathbf{y = |x_1 - x_2|}$** for two variables x_1, x_2
  - Modeling $y = |x_1 - x_2|$ is equivalent to $y = max\{x_1 - x_2, x_2 - x_1\}$
  - Logic formulation:

    $y \geq x_1 - x_2$
    $y \geq x_2 - x_1$
    $y \leq x_1 - x_2$ or $y \leq x_2 - x_1$

- Example implementation with Mosel:

```
 declarations
  x: array(1..2) of mpvar
  y, u, v: mpvar
  C1, C2: linctr
  C3: logctr
 end-declarations

! Formulation of y = min{x(1), x(2)}
 C1:= y <= x(1)
 C2:= y <= x(2)
 C3:= y >= x(1) or y >= x(2)

! Formulation of u = max{x(1), x(2)}
 C1:= u >= x(1)
 C2:= u >= x(2)
 C3:= u <= x(1) or u <= x(2)

! Formulation of v = |x(1) - x(2)|
 C1:= v >= x(1) - x(2)
 C2:= v >= x(2) - x(1)
 C3:= v <= x(1) - x(2) or v <= x(2) - x(1)
```