Using Binary Indicator Variables in MPL

A Technical White Paper

Maximal Software, Inc.
2111 Wilson Boulevard, Suite 700
Arlington, VA 22201 USA
Telephone: (703) 522-7900
Fax: (703) 522-7902
www.maximalsoftware.com

Using Binary Indicator Variables in MPL

When formulating mixed integer (MIP) models in **MPL**, binary (0/1) variables are often used to indicate whether certain constraints hold true or not or whether a variable is nonzero. In this white paper, we will go over each of these cases and explain how to formulate indicator variables for them.

Variable is nonzero (x > 0)

The simplest case of using indicator variables is when a binary variable is used to represent whether a certain continuous variable is nonzero or not. Let's assume we have a continuous variable *x* that can take any value between zero and ten *(0 <= x <= 10).*

We can now define a binary indicator variable *i* such that

```
x > 0  ->   i = 1
x = 0  ->   i = 0
```

where - is used to mean "implies". To formulate the first condition we note that the constraint

```
x <= 10 i;
```

forces the binary variable *i* to take the value *1* if *(x > 0)* since the right-hand side must be greater than the left-hand side. When *(x = 0)* the constraint becomes (*0 <= 10 i)* which still allows the *i* variable to be either *0* or *1*.

Now we will formulate the second condition. Here we can enter the constraint

```
x >= 0.0001 i
```

This constraint will force the binary variable *i* to take the value *0* if *(x < 0.0001)*. We have to use small tolerance value such as *0.0001* in order to make sure *i* will be zero when (*x = 0)*. The tolerance can be any number, small enough for us to treat *x* values that are below it as zero.

To specify the general case for nonzero indicator variables let define *x* such that

```
x = 0  or  eps <= x <= M
```

Where *eps* is the zero tolerance value and *M* is the upper bound. The constraints for this general case will then be as follows:

```
x <= M * i;                                  [1]
x >= eps * i;                                [2]
```

In some cases you do not have to enter the second constraint. For example if *i* is used in the objective function that is minimized it might already take the lowest possible value automatically.

Less Than constraints (ax <= b)

Binary variables can also be used to indicate whether a certain constraint holds true. Let's say we have a continuous variable *x* that has an upper bound of *5* and that there is a constraint that says *(2x <= 5)*. Further more we will assume the zero tolerance *eps* has the value of *0.0001*. The upper and lower bounds for the *x* variable can be represented as follows:

```
x  >=  0;
x  <=  5;
```

The constraint *(2x <= 5)* can be rewritten as (*2x - 5 <= 0)*. We can now calculate the lower and upper bonds for the left-hand side of that formula as:

```
m = 2 * 0 - 5 = -5
and
M = 2 * 5 - 5 = 5
```

We now define a binary indicator variable *i* such that

```
2x - 5 >= eps  ->  i = 0
2x - 5 <= 0    ->  i = 1
```

We are using the zero tolerance *eps* to allow us to write the condition as *Greater Than Or Equal* instead of just *Greater Than*.

To formulate the conditions we use the formulas:

```
ax - b  >=  eps + (m - eps)*i;              [3]
ax - b  <=  M*(1 - i);                      [4]
```

Entering the value for *a, b, M, m,* and *eps* we get the following two constraints:

```
2x - 5  >=  0.0001 + (-5 - 0.0001)*i;
2x - 5  <=  5*(1 - i);
```

It easy to verify that these constraints are correct by entering the values for *i=1* and *i=0* for each of the constraint:

```
i = 1:   2x - 5 >= -5;      which becomes  x >= 0;
i = 1:   2x - 5 <= 0;       which becomes  x <= 2.5;

i = 0:   2x - 5 >= 0.0001;  which becomes  x >= 2.50005
i = 0:   2x - 5 <= 5;       which becomes  x <= 10;
```

A quick look at these constraints shows that (*i = 1)* forces *x* to take a value between *0* and *2.5* for which the constraint (*2x <= 5)* holds and when (*i = 0), x* takes a value between *2.50005* and *10* which does not fulfill the constraint.

Greater Than constraints (ax >= b)

Binary indicator variables for greater-than constraints are implemented in a very similar fashion to less-than constraints. Let's say we have the same continuous variable *x* as before with upper bound of *6*, but now the constraint is *(3x >= 6)*.

The constraint *(3x >= 6)* can be rewritten as (*3x - 6 >= 0)*. We can now calculate the lower and upper bounds for the left-hand side of that formula as:

```
m = 3 * 0 - 6 = -6
and
M = 3 * 6 - 6 = 12
```

We now define a binary indicator variable *i* such that

```
3x - 6 >= 0     ->  i = 1
3x - 6 <= -eps  ->  i = 0
```

As before we are using the zero tolerance *eps* to allow us to write the condition as *Less Than or Equal* instead of just *Less Than*.

To formulate the conditions we use the formulas:

```
ax - b  >=  m*(1 - i);                     [5]
ax - b  <=  -eps + (M + eps)*i;            [6]
```

Entering the value for *a, b, M, m,* and *eps* we get the following two constraints:

```
3x - 6  >=  -6*(1 - i);
3x - 6  <=  -0.0001 + (12 + 0.0001)*i;
```

It is easy to verify that these constraints are correct by entering the values for *i=1* and *i=0* for each of the constraint:

```
i = 1:   3x - 6 >= 0;        which becomes  x >= 2.0;
i = 1:   3x - 6 <= 12;       which becomes  x <= 6;

i = 0:   3x - 6 >= -6;       which becomes  x >= 0;
i = 0:   3x - 6 <= -0.0001;  which becomes  x <= 1.99997
```

A quick look at these constraints shows that *(i = 1)* forces *x* to take a value between *2.0* and *6.0* for which the constraint (*3x >= 6)* holds and when (*i = 0), x* takes a value between *0* and *1.99997* which does not fulfill the constraint.

Equal constraints (ax = b)

Equal constraints are done in similar fashion as both *Less Than* and *Greater Than* constraints, but you have to include two separate constraints for each case and define three different indicator variables.

When (*i = 1)* both the greater than and the less than constraints are needed in order to force the *(ax = b)* to be true.

Enforce the equal constraint *(ax = b)*:

```
ax - b  >=  m*(1 - i);                        [7]
ax - b  <=  M*(1 - i);                        [8]
```

When (*i = 0),* either the less than constraint *(ax < b)* or the greater than constraint *(ax > b)* must be true. Therefore we need to create two more binary variables *i1* and *i2* to represent each:

Enforce the less than constraint *(ax < b) => (ax <= b - eps)*:

```
ax - b  <=  M - (M + eps)*i1;                 [9]
```

Enforce the greater than constraint *(ax > b) => (ax >= b + eps)*:

```
ax - b  >=  m - (m - eps)*i2;                [10]
```

You can now force at least one of *i1* or *i2* to be one when (*i = 0)* by entering the following constraint:

```
i1 + i2 >= 1 - i;                             [11]
```

Summary of Formulas

Here is a summary of all the formulas we defined above:

Variable is nonzero *(x >= 0)*:

```
x <= M * i;                                   [1]
x >= eps * i;                                 [2]
```

Less Than constraints *(ax <= b)*:

```
ax - b  >=  eps + (m - eps)*i;                [3]
ax - b  <=  M*(1 - i);                        [4]
```

Greater Than constraints *(ax >= b)*:

```
ax - b  >=  m*(1 - i);                        [5]
ax - b  <=  -eps + (M + eps)*i;               [6]
```

Equal constraints *(ax = b)*:

```
ax - b  >=  m*(1 - i);                        [7]
ax - b  <=  M*(1 - i);                        [8]
ax - b  <=  M - (M + eps)*i1;                 [9]
ax - b  >=  m - (m - eps)*i2;                [10]
i1 + i2 >= 1 - i;                            [11]
```

These also work *(ax = b)*:

```
ax - b  >=  m*(1 - i);                        [7]
ax - b  <=  M*(1 - i);                        [8]
ax - b  <=  -eps + (M + eps)*i1;              [9]
ax - b  >=  eps + (m - eps)*i2;              [10]
i1 + i2 <= 1 + i;                            [11]
```

Boolean Conditions

After you have defined binary indicator variables for your constraints, you can use them to create all kinds of boolean conditions in your models. Here are few examples with the corresponding equations on the binary variables:

```
not P1                          i1 = 0
P1 or P2                        i1 + i2 >= 1
P1 xor P2                       i1 + i2 = 1
P1 and P2                       i1 = 1,  i2 = 1

NOT (P1 or P2)                  i1 = 0,  i2 = 0
NOT (P1 and P2)                 i1 + i2 <= 1

P1 -> NOT P2                    i1 + i2 <= 1
P1 -> P2                        i1 <= i2
P1 <-> P2                       i1 = i2

P1 -> (P2 and P3)               i1 <= i2,  i1 <= i3
P1 -> (P2 or P3)                i1 <= i2 + i3
(P1 and P2) -> p                i1 + i2 <= 2 - 1 + i
(P1 and P2 and P3) -> p         i1 + i2 + i3 <= 3 - 1 + i

(P1 or P2) -> p                 i1 <= i,  i2 <= i

P1 and (P2 or P3)               i1 = 1,  i2 + i3 >= 1
P1 or (P2 and P3)               i1 + i2 >= 1,  i1 + i3 >= 1

P1 or P2 or P3                  i1 + i2 + i3 >= 1
P1 xor P2 xor P3                i1 + i2 + i3 = 1

ATLEAST[k]                      i1 + i2 + i3 >= k
EXACTLY[k]                      i1 + i2 + i3 = k
ATMOST[k]                       i1 + i2 + i3 <= k

p <-> P1 or P2 or P3            i1 + i2 + i3 >= i,
                                i >= i1,  i >= i2,  i >= i3
p <-> P1 and P2 and P3          i1 + i2 + i3 <= 3 - 1 + i,
                                i <= i1,  i <= i2,  i <= i3
```