Solver .NET User Guide

45.01

USER GUIDE

FICO® Xpress Optimization

Xpress Optimizer 45.01 (FICO® Xpress 9.7)

Deliverable Version: A

Last Revised: 29 July, 2025

Contents

Introduction

The `XpressProblem` class provides a convenient and efficient interface for stating and working with optimization problems. Using this class, optimization problems can be built easily and all features of the Xpress Solver can be accessed.

The classes for modeling problems are spread across several namespaces. The base namespace for everything related to the Xpress Solver in this API is `Optimizer`. The sub-namespaces are listed below. See the reference documentation for more details.

`Optimizer.Objects` Contains all the classes required for modeling using objects rather than row and column indices. This defines classes like `Variable`, `Expression` and `XpressProblem` that allow convenient and efficient building of problems. This is the main package you will interact with when creating optimization problems.

`Optimizer.Maps` Contains implementations of multi-dimensional maps. These maps build on maps provided by the language itself but also provide some more convenient ways to access data. For example, in order to query an element, you do not need to chain several calls to `[]` but can instead put the arguments for all dimensions into a single call.

`Optimizer.Functional` Provides definitions of functors used in the API. Usually you should not need to use any of these explicitly.

The `XpressProblem` class itself is located in `Optimizer.Objects`. Its fully qualified name is `Optimizer.Objects.XpressProblem`.

To avoid typing the fully qualified name you can do this:

```
// Import a whole package:
using Optimizer.Objects;
// "Import" a single class:
using XpressProblem = Optimizer.Objects.XpressProblem;
// make functions like Sum() available without qualification
using static Optimizer.Objects.Utils;
```

In the following sections we do not list the required `using` statements that are required to make functions/classes/interfaces available without full qualification. We assume that appropriate statements have been written at the beginning of the code.

CHAPTER 1

Getting started with the .NET API

1.1 Instantiating the XpressProblem class

While all low-level APIs to the Xpress Solver describe the problem in terms of a matrix using row and column indices, the `XpressProblem` provides a more abstract view on the problem by using variable and constraint objects.

In order to create an object-oriented problem you need an instance of the `XpressProblem` class. The constructor of that class takes zero, one, or two strings. The first string is the name to be assigned to the problem, the second string points to your license file. Either string may be `null` or may be omitted. If your license can be found without explicit license file information (for example because it is in a default location) then you can omit the license argument to the constructor.

An instance of `XpressProblem` wraps native resources, so it is good practice to release these resources as soon as the object is no longer needed, for example:

```
using (XpressProblem prob = new XpressProblem()) {
  ... // Your code for creating and solving the optimization problem here
  }
```

1.2 Creating variables

The first step in stating an optimization problem is creating the required decision variables. Variables are created following the "builder pattern". This pattern starts with calling the `AddVariables()` function. This function returns a builder that can be modified (see below) and can be turned into an array or a map of variables using its `ToArray()` or `ToMap()` functions, respectively.

For example:

```
prob.AddVariables(5).ToArray();
```

creates an array of five variables, where all variables have default properties (lower bound zero, upper bound infinity, no name, continuous).

Similarly,

```
prob.AddVariables(new string[]{ "a", "b" }).ToMap();
```

creates a map with two variables. The keys in this map are the two strings "a" and "b" (passed as first argument to the function) and the values in the map are the variables that were created for the strings. Again, these variables have all properties at default values.

Builders can be modified before calling `ToArray()` or `ToMap()`. For this purpose, builders provide a number of `With` functions. For example, function `WithLB` changes the lower bound of all the variables the builder will create after this function was called:

```
prob.AddVariables(5).WithLB(1).ToArray();
```

The statment above again creates five variables, but this time all the variables have a lower bound of 1.

Properties cannot only be changed to a constant value. They can also be changed to value that depends on the variable being created. Consider

```
prob.AddVariables(5).WithLB(i => (double)i).ToArray();
```

This creates five variables. The first variable has a lower bound of 0. The second variable has a lower bound of 1. The third has a lower bound of 2 and so on. What happens here is that the function passed to `WithLB` is invoked for every variable created in order to generate the lower for this particular variable.

Similarly, consider

```
prob.AddVariables(new string[]{ "a", "b" }).WithName(s => "x"+s).ToMap();
```

This creates the same map of variables as before, but this time the variables get names. The names are constructed from the strings that were passed to the function. These strings are passed as parameter `s` to the function that was registered with `WithName`.

The functions to modify a variable builder are the following

WithLB to specify the lower bound, the default is zero

WithUB to specify the upper bound, the default is infinity

WithName to specify the name, the default is no name

WithType to specify the variable type (continuous, integer, binary, ...), the default type is continuous

WithLimit to specify the limit for semi-continuous, semi-integer or partial integer variables, the default is zero

For example, to create an array of 5 integer variables with lower bound 1, upper bound 2, and names ”x1”, ”x2”, ... use

```
Variable[] x = prob.AddVariables(5)
                   .WithLB(1)
                   .WithUB(2)
                   .WithType(XpressProblem.ColumnType.Integer)
                   .WithName(i => $"x{i+1}")
                   .ToArray();
```

The `AddVariables()` functions are also overloaded to create multi-dimensional arrays or maps of variables. In order to do that, instead of passing a single dimension count or collection, pass as many counts or collections as there are dimensions:

```
Variable[,,] x = prob.AddVariables(2, 3, 4)
                     .WithName("x[{0}][{1}][{2}]")
                     .ToArray();
IEnumerable<Type1> c1 = ...;
IEnumerable<Type2> c2 = ...;
IEnumerable<Type3> c3 = ...;
HashMap3<Type1,Type2,Type3,Variable> x = prob.AddVariables(c1, c2, c3)
                                             .WithName("x[{0}][{1}][{2}]")
                                             .ToMap();
```

Using function `AddVariable()` it is also posible to create a single variable. This function does not follow the builder pattern but instead takes all the variable properties as arguments:

```
// Unnamed continuous variable in [0,infinity)
x = prob.AddVariable();
// Integer variable in [2,3] with the name "x"
x = prob.AddVariable(2, 3, ColumnType.Integer, "x");
```

There are more overloads for `AddVariable()`, please refer to the reference documentation for further details.

Note that variables are specific to a problem. In other words, an instance of `Variable` can be used only with the `XpressProblem` instance that created it. If you use a variable with a problem that did not create it then this will raise an exception.

1.3 Creating constraints

Once you have created some `Variable` objects, you can start creating constraints on them. The most basic kind of constraints are inequality constraints. They represent constraints of one of the following types:

```
expression1 <= expression2
expression1 >= expression2
expression1 == expression2
expression1 in [lb,ub]
```

Here `expression1` is called the left-hand side (frequently referred to as ”lhs”) and `expression`” is called the right-hand side (frequently referred to as ”rhs”). The operator between the two expressions is called the ”sense” of the constraint. The last constraint above is called a range constraint and requires `expression1` to be between `lb` and `ub`. Note that there are no ”not equal” or strict inequality operators since these are not supported by the theory of mathematical programming on which Xpress is built. Strictily speaking, ”==” and ”in” are not inequalities, but in this document and in the programming API we still call those an inequality.

Examples of expressions are

$$x_1 + 2x_3 \qquad x_1^2 + x_2 + 4 \qquad \sin(x_1)$$

The first type of expression is called a ”linear” expression (it only involves linear terms), the second one is called a ”quadratic” expression (it involves linear and quadratic terms), the third one is called a ”nonlinear” expression (it involves terms that are neither linear nor quadratic).

In order to create an inequality constraint, you first need an expression, so we start by explaining how to create a linear expression.

1.3.1 Creating linear expressions

There are several ways to construct a linear expression:

Use an instance of class `Variable`. The `Variable` class extends the `Expression` class, so it can be used whenever an expression is required.

Multiply a variable by a constant. Class `Variable` provides member function `Mul()` that returns a linear term representing a multiplication of the variable with the given constant.

Construct an expression using `ScalarProduct()`. The `ScalarProduct()` function constructs a linear expression by elementwise multiplying a vector of variables by a vector of numbers.

Construct an expression using `Sum()`. The `Sum()` function takes arbitrary expressions and sums them up to build a new expression.

Construct an expression by accumulating terms in a `LinExpression`. The (abstract) `LinExpression` class provides member functions `AddTerm()` and `SetConstant()` or `AddConstant()` to build up an expression piecemeal. An empty linear expression can be constructed using the static `LinExpression.Create()` function.

There also is a `ConstantExpression` class and a static member function `Constant()` that can be used to represent expressions that only have a constant term.

Examples:

```
Variable x[2] = ...;
Expression t = x[0].Mul(1.5);                          // 1.5*x[0]
Expression p = ScalarProduct(x, new double[]{2,3}); // 2.0*x[0] + 3.0*x[1]
Expression s = Sum(t, p);                              // 3.5*x[0] + 3.0*x[1]
LinExpression e = LinExpression.Create();
e.AddTerm(x[0], 1.5);                                  // 1.5*x[0]
e.AddTerm(x[1], 2.5);                                  // 1.5*x[0] + 2.5*x[1]
e.SetConstant(3.0);                                    // 1.5*x[0] + 2.5*x[1] + 3.0
Constant(5);                                           // 5.0
```

Remember that for using `Sum` or `ScalarProduct` without qualified name you have to import them via

```
using static Optimizer.Objects.Utils;
```

The API also overloads operators +, −, *, / to make it even easier to create expressions:

```
e = 3.5 * x[0] + 2.5 * x[1];
```

Note that `LinExpression` is an abstract class for which two implementations exist: `LinTermMap` and `LinTermList`. `LinExpression.Create()` will create an instance of `LinTermMap` which is the most versatile implementation but not necessarily the most efficient one. The next two paragraphs explain the difference between the two implementations.

The first implementation is class `LinTermMap`. As the name suggests, this implementation is backed up by a map: internally the expression is represented as a map that maps variables to their respective coefficients. This implementation is very flexible and allows for all kind of modifications of the expression, in particular it allows directly setting coefficients for variables by means of the `SetCoefficient()` function. On the downside, accessing coefficients incurs some runtime overhead.

The second implementation is class `LinTermList`. This keeps all coefficients in a simple list. Consequently, you cannot query coeffcients and can also not explicitly set coefficients for certain variables. Another problem is that duplicate terms (multiple terms for the same variable) require explicit handling. The upside is that this implementation is more efficient. When using this class, make sure you understand the implications and assumptions that this class makes. These are stated in the reference documentation.

As a rule of thumb, the `LinTermMap` implementation should be used, unless it turns out to be prohibitively expensive with respect to performance of building expressions.

1.3.2 Creating constraints from expressions

Once you have created expressions, you can start building constraints from them. For this purpose, the `Expression` class provides member functions `Leq`, `Geq`, `Eq` that create a less-than-or-equal, greater-than-or-equal, or equal-to constraint respectively. These functions produce a constraint definition than can then be added to a problem using the `AddConstraint()` function.

For example:

```
Expression lhs = Sum(x[1], x[2].Mul(2.0));
prob.AddConstraint(lhs.Leq(4.0));  // Adds constraint x[1] + 2 * x[2] <= 4
```

The API overloads operators "<=", ">=" and "==" so that you can write the above code as

```
prob.AddConstraint(x[1] + 2 * x[2] <= 4);  // Adds constraint x[1] + 2 * x[2] <= 4
```

In addition to inequality constraints, you can also create ranged constraints that bound an expression from below and above. This is done using the `In()` member function of the `Expression` class:

```
Expression expr = ...;
prob.AddConstraint(expr.In(2, 5)); // Add constraint 2 <= expr <= 5
```

1.3.3 Creating quadratic expressions

Quadratic expressions are created in similar ways as linear expressions.

Starting from a variable. Starting from a `Variable` instance, a quadratic expression can be built using the `Mul` function: `x.Mul(y).Mul(3)` creates the quadratic term 3*x*y.

Using function `ScalarProduct`. `QuadExpression.ScalarProduct()` is for quadratic expressions what `LinExpression.ScalarProduct()` is for linear expressions. This function multiplies three vectors element by element to create a new quadratic expression.

Using the `QuadExpression` class directly. Using `QuadExpression.Create()` an empty instance of `QuadExpression` can be created. This instance can then be modified using `AddTerm()`, `SetConstant()` and `AddConstant()`.

Examples:

```
Variable[2] x = ...;
Expression t = x[0].Mul(x[1]).Mul(1.5);              // 1.5*x[0]
Expression p = ScalarProduct(x, y, new double[]{2,3}); // 2.0*x[0]*y[0]+3.0*x[1]*y[1]
QuadExpression e = QuadExpression.Create();
e.AddTerm(x[0], x[1], 1.5);                          // 1.5*x[0]*x[1]
e.AddTerm(x[1], 2.5);                                // 1.5*x[0]*x[1]+2.5*x[1]
e.SetConstant(3.0);                                  // 1.5*x[0]*x[1]+2.5*x[1]+3.0
```

There also is some limited operator overloading. So for variables `x` and `y` and a quadratic expression `q` you can create quadratic terms like this:

```
Expression term = x * x * 3.5;
q.AddTerm(2.5 * x);
q.AddTerm(3.5 * x * y);
prob.AddConstraint(3.5 * x * y <= 3);
```

1.3.4 Creating nonlinear expressions

Nonlinear expressions are represented by expression trees. Each node in such a tree is an instance of class `FormulaExpression` (or a subclass of it). The full expression is represented by the root node of that tree.

In order to create an expression tree you can either create the nodes directly using their constructor or you can use the static utility functions provided by class `FormulaExpression`:

```
using static Optimizer.Objects.Utils;

Variable x = ...;
Expression nonLinear = Sin(Cos(x));
```

The classes that implement nodes in an expression tree can be found in `Optimizer.Objects` and are

BinaryExpression represents binary operations, such as addition, subtraction, exponentiation, etc.

UnaryExpression represents unary operations such as unary minus (negation)

ColumnReferenceExpression represents a reference to a column/variable

InternalFunctionExpression represents a call to an elementary mathematical function

UserFunctionExpression represents a call to a user function. Note that instances of this class can also be created by calling funcion `Call()` on the corresponding user function object (see `AbstractUserFunction`).

Once a nonlinear expression is created, it can be used like any other expression to build a constraint:

```
prob.AddConstraint(Sin(x).Leq(Cos(x)));      // sin(x) <= cos(x)
prob.AddConstraint(Min(x, y) <= Pow(a, b)); // min(x,y) <= a^b
```

The arguments to nonlinear expressions can be instances of any class in `Optimizer.Objects` that extends the `Expression` class. In particular, they can be instances of `LinExpression`, `QuadExpression`, `Variable` etc. While this is very flexible, it is still recommended that you use `LinExpression`, `QuadExpression` or functions `Sum()`, `ScalarProduct` to create linear or quadratic expressions. This is usually more efficient with respect to memory and time.

1.4 Setting the objective function

The objective function is set using function `SetObjective()`. There are two overloads for this function, one that specifies the objective sense and one that does not:

```
SetObjective(Expression obj, ObjSense sense);
SetObjective(Expression obj);
```

The default optimization sense is ”minimize”.

The expression passed to the function can be a linear or quadratic expression.

1.5 Solving the problem

Once the problem is set up, it can be optimized. This happens by means of the `Optimize()` function. This function will start optimization and return only once the optimal solution is found, the problem is found infeasible, or some resource limit was hit.

Once the function returns, you need to query the `SOLVESTATUS` and `SOLSTATUS` attributes to understand what caused the function to stop. This is done by querying `GetSolveStatus()` and `GetSolStatus()` respectively. The values returned by that indicate why the solution process stopped and what kind of solution is available.

```
XpressProblem prob = ...;
Variable[] x = ...;
... // Build the model here
```

```
prob.Optimize();
if (prob.SolStatus == Optimizer.SolStatus.Optimal) {
  double[] sol = prob.GetSolution();
  foreach (Variable v in x)
    Console.WriteLine("{0} = {1}", v.GetName(), v.GetValue(sol));
}
```

Note that there are overloads of the `Optimize()` function that return the solve and solution status in additional arguments.

1.5.1 Setting solver controls

The Xpress Solver behavior and the solution strategy can be tweaked by setting solver controls.

All controls are available as properties of the `XpressProblem` class (more precisely, the `XPRSprob` superclass). For example, to stop the solver after 30 seconds, set the *timelimit* control:

```
prob.TimeLimit = 30;
Console.WriteLine("Time limit set to " + prob.TimeLimit);
```

1.6 Querying the solution

Once a solution is obtained (remember to check the solution status after calling `Optimize`), the values of that solution can be queried in two different ways. The most efficient way is to call the `GetSolution()` function of the `XpressProblem` instances, store the values in an array and then invoke the `Variable`'s `GetValue()` function on that array:

```
Variable x = ...;
prob.Optimize();
double[] sol = prob.GetSolution();
Console.WriteLine(x.GetValue(sol));
```

Another way to get the solution value for a variable is to directly call the `GetSolution()` function on the `Variable` instance:

```
Variable x = ...;
prob.Optimize();
Console.WriteLine(x.GetSolution());
```

This is less efficient when querying multiple variable values since it performs multiple queries against the underlying optimizer.

Similarly to solution values, you can query slacks of `Inequality` instances as well as reduced costs and dual multipliers in case of purely continuous problems. This is done by using functions `GetSlacks()`, `GetRedCosts()`, `GetDuals()`, respectively. These return an array of values that can then be accessed using `Variable.GetValue()` or `Inequality.GetValue()`, just like in the case of solution values. Similarly, there are `Inequality.GetSlack()`, `Variable.GetRedCost()`, `Inequality.GetDual()` that return the same information for an individual object – and to which the same caveats as for solution values apply.

1.7 A full example

The below code models a very simple Knapsack problem. The objective in this model is to maximize

$$5x_1 + 4x_2 + 3x_3 + 8x_4$$

such that

$$2x_1 + 5x_2 + 7x_3 + 9x_4 \leqslant 12$$

and all variables are binary.

```
using Variable = Optimizer.Objects.Variable;
using XpressProblem = Optimizer.Objects.XpressProblem;
using ObjectiveSense = Optimizer.ObjectiveSense;
using SolStatus = Optimizer.SolStatus;
using ColumnType = Optimizer.ColumnType;
using static Optimizer.Objects.Utils; // for scalarProduct()
namespace Example
    class Example {
        private static double[] weight = new double[]{ 2, 5, 7, 9 };
        private static double[] profit = new double[]{ 5, 4, 3, 8 };
        private static double capacity = 12;

        public static void Main(string[] args) {
            using (XpressProblem prob = new XpressProblem()) {

                // Create one binary variable for each item to model whether the
                // item is selected (variable is one) or not (variable is zero).
                Variable[] x = prob.AddVariables(weight.Length)
                                   .WithType(ColumnType.Binary)
                                   .WithName(i -> $"x{i+1}")
                                   .ToArray();

                // Respect capacity constraint.
                prob.AddConstraint(ScalarProduct(x, weight).Leq(capacity));

                // Maximize profit.
                prob.SetObjective(ScalarProduct(x, profit),
                                  ObjSense.Maximize);

                prob.Optimize();

                if (prob.SolStatus == SolStatus.OPTIMAL) {
                    Console.WriteLine("Maximum profit: {0}", prob.ObjVal);
                    double[] sol = prob.GetSolution();
                    for (int i = 0; i < x.Length; ++i) {
                        if (x[i].GetValue(sol) > 0.5)
                            Console.WriteLine("Item {0} was selected", i+1);
                    }
                }
            }
        }
    }
}
```

CHAPTER 2

Callbacks

Callbacks are a way to interact with the solver *during* the solve. By using callbacks the solver behavior can be changed to some degree while a solve is running and without the need to interrupt it.

Callbacks are represented by delegates, that is, anything that can be called like a function that matches the delegate definition.

Callbacks are handled by member functions of the `XpressProblem.callbacks` field.

Here is an example that uses lambda expressions to register a callback that prints a message every time a feasible solution is found.

```
prob.callbacks.addIntsolCallback((p) => {
  Console.WriteLine("New integer solution with objective {0}",
                    p.ObjVal);
});
prob.Optimize();
```

The same can be achieved by using references to functions with an appropriate signature:

```
public class MyClass {
  private static void MyIntsolCallback(XpressProblem p) {
    Console.WriteLine("New integer solution with objective {0}",
                      p.ObjVal);
  }
  public static void Main(string[] args) {
    using (XpressProblem prob = new XpressProblem()) {
      ...
      prob.callbacks.AddIntSolCallback(MyClass.MyIntsolCallback);
      prob.Optimize();
    }
  }
}
```

As suggested by the examples above, callbacks only take action during a solve, so adding a callback without calling any kind of optimization function is pointless. An exception to this is the message callback because warning and error messages may be printed through this callback even before a solve is started.

To get a list of callbacks that can be registered, refer to the reference documentation of class `XpressProblem.CallbackAPI`.

CHAPTER 3

More advanced features

3.1 Other constraint types

In addition to inequality constraints (which include equations and ranged rows), there are a number of other constraints that Xpress supports:

- Special ordered set constraints restrict the number of variables that can be non-zero.
- Indicator constraints conditionally enable or disable inequality constraints, based on the values of binary variables.
- Piecewise linear constraints set one variable to the result of evaluating a piecewise linear function for another variable.
- General constraints model things like absolute value, maximum or minimum of variables and values, and logical and/or between binary variables.

Creation of these constraints is described in detail below.

3.1.1 Special ordered set (SOS) constraints

Special ordered set (SOS) constraints specify how many and which variables in a set of variables can be non-zero. For an SOS constraint of type 1 only one variable in the set can be non-zero. For an SOS constraint of type 2 at most two variables in the set can be non-zero and non-zero variables must be adjacent.

The ordering of variables within a set is given by weights assigned to variables: variables are ordered by increasing weight values and two different variables cannot have the same weight. In many cases the order in which variables are passed to functions is already the intended one. In that case the `weight` argument to the functions can be `null`. The function will then assume weights 1, 2, ...

Note that weights are not only used for ordering. Instead the actual weight values may be used by the solver algorithm when taking branching or other decisions in the solution process. Refer to the solver documentation for more details.

One way to create SOS constraints is by using the static `Sos()`, `Sos1()`, or `Sos2()` function of the `SOSConstraint` class:

```
// Create an SOS1 for { x1, x2, x3 } with default weights.
using static Optimizer&nssep;Objects.SOS; // for functions sos(), sos1(), sos2()
prob.AddConstraint(Sos(SetType.SOS1, new Variable[]{ x, y, z }, null, "SOS constraint"));
// Create the same SOS1
prob.AddConstraint(Sos1(new Variable[]{ x, y, z }, null, "SOS constraint"));
// Create an SOS2 for { x, y, z } with default weights.
prob.AddConstraint(Sos2(new Variable[]{ x, y, z }, null, "SOS constraint"));
```

3.1.2 Indicator constraints

In Xpress, indicator constraints are not represented as explicit constraints. Instead they are defined by combining an inequality `r` with a binary variable `b`. Once these two are combined, the solver will enforce a condition that if `b` is set then `r` is satisfied. If `b` is not set then `r` may be violated.

Note that the condition on variable `b` may be complemented. In that case `r` is only enforced if `b` is not set.

In order to set indicator variables for certain inequality constraints, use function `XpressProblem.SetIndicatorVariable()`:

```
Variable b = ...;
Inequality r1 = ...;
Inequality r2 = ...;
prob.SetIndicatorVariable(b, true, r1);
prob.SetIndicatorVariable(b, false, r2);
```

The above code will enforce `r1` if variable `b` is set (has value 1) and enforce `r2` if `b` is not set (has value 0).

The `XpressProblem` class also has a `SetIndicatorVariables` function with which multiple indicators can be set with a single function call.

Another way to set indicator variables for inequalities is by using the `IfThen` and `IfNotThen` function of the `Variable` class. These functions create an indicator constraint with the given variable as indicator variable:

```
prob.AddConstraint(b.IfThen(r1)); // enforce r1 if b is set
prob.AddConstraint(b.IfNotThen(r2)); // enforce r2 if b is not set
```

3.1.3 Piecewise linear constraints

A piecewise linear constraint is a constraint of the form

```
resultant = PWL(breakpoints)(argument)
```

Here `resultant` and `argument` are variables and `PWL(breakpoints)` is a piecewise linear function specified by the given breakpoints. See the documentation of function `XPRSaddpwlcons` for a detailed explanation of how breakpoints are specified.

Piecewise linear constraints can be created using the `PwlOf` member function of class `Variable` (assuming that `resultant` and `argument` are instances of `Variable`:

```
double[] breakX = ...;
double[] breakY = ...;
prob.AddConstraint(resultant.PwlOf(argument, breakX, breakY));
Breakpoint[] breaks = ...;
prob.AddConstraint(resultant.PwlOf(argument, breaks));
```

3.1.4 General constraints

A general constraint is a constraint of the form

```
resultant = function(y1, y2, ..., v1, v2, ...)
```

where `resultant` is a variable, `function` is a function like ”min”, ”max”, ”abs”, ”or”, ”and”, `yi` are variables and `vi` are constant values.

General constraints can be created by invoking an appropriate member function on the resultant variable:

```
Variable x, y1, y2, y3;
prob.AddConstraint(x.AbsOf(y));          // add x = |y|
prob.AddConstraint(x.OrOf(y1, y2));      // add x = y1 or y2
prob.AddConstraint(x.AndOf(y1, y2, y3)); // add x = y1 and y2 and y3
prob.AddConstraint(x.MaxOf(y1, y2, 4));  // add x = max(y1, y2, 4.0)
prob.AddConstraint(x.MinOf(y1, 2));      // add x = min(y1, 2.0)
```

3.2 Performance considerations

While building small to medium-sized problems, there is usually no need to consider which way is the fastest to create the problem. The speed of different strategies does not matter.

When building models with hundreds of thousands or even millions of elements, it is important to know that some methods are faster than others:

Building linear expressions The most efficient way to build a linear expression is by creating an instance of `LinExpression` and then add the terms to this instance. In order to be as fast as possible, use the `LinTermList` class rather than the `LinTermMap` class (note that there are some caveats about this class, see above and in the reference documentation).

Building quadratic expressions As with linear expressions, the most efficient way to build a quadratic expression is using `QuadExpression` (`QuadTermList`).

Bulk operators are faster than single operations Consider this loop:

```
for (int i = 0; i < n; ++i) prob.AddConstraint(...);
```

This adds constraints one by one to the model. A more efficient way to do this is to add all constraints in one shot:

```
prob.AddConstraints(n, i => ...);
```

This adds exactly the same constraints in the same order but reduces interaction with the low-level code.

3.3 Accessing the matrix

Class `XpressProblem` is a subclass of `XPRSprob`. The latter provides a direct matrix-based interface to the solver. Any function in `XPRSprob` directly maps to a function call in the C implementation of the solver. Since `XpressProblem` is a subclass of `XPRSprob`, all these functions are also available when modeling with objects. In order to interact with a function in `XPRSprob`, you need the indices of the objects. These can be obtained using the `GetIndex()` function of classes like `Variable`, `Inequality`, `SOS`, `PWL`, `GeneralConstraint`.

CHAPTER 4

Migrating from BCL

As described above, in Xpress 9.4 the .NET API to the Optimizer was extended to allow the creation of an optimization problem in a fully object oriented fashion. Instead of specifying the problem as a matrix with row and column indices, you can use variable and constraint objects to state the problem.

The new API is fully integrated with the low-level Optimizer API, and has been designed for high performance. The new API is a replacement for BCL, which will be deprecated in future Xpress releases.

We will only give very short code example snippets here. Please refer to the reference documentation for more details.

4.1 Replacing the `XPRBprob` factory class

Any BCL application requires an instance of class `XPRBprob` as factory to create variables and/or constraints. For the new API you also need a factory class. The fully qualified name of this class is `Optimizer.Objects.XpressProblem`.

The `XpressProblem` class wraps native solver resources and is thus best used within a using-block in:

```
using (XpressProblem prob = new XpressProblem("problem name")) {
    ...
}
```

All classes required to formulate optimization problems with the new API are located in the `Optimizer.Objects` namespace. In the following we assume that you imported all names from this and will no longer provide fully qualified names.

4.2 Creating variables

In BCL only a single variable can be created at a time. This is done using function `XPRBprob.newVar()`.

With the new API you can also create variables one by one using `XpressProblem.AddVariable()` family of functions. Alternatively, you can create multiple variables with a single statement (this is more efficient than creating variables one by one). The overloads of function `XpressProblem.AddVariables()` achieve this. These functions allow creating single and multidimensional arrays or maps of variables with a single function call.

Creating individual variables in the new API is similar to creating variables in BCL. All properties of the variable (bounds, type, name, ...) are specified as arguments to the function that creates the variable:

```
//Unnamed continous variable in [0,infinity)
Variable x = prob.AddVariable();
// Integer variable in [2,3] named "x"
```

```
Variable x = prob.AddVariable(2, 3, ColumnType.Integer, "x");
```

Please refer to the reference documentation for an exhaustive list of overloads.

When creating multiple variables at once then the builder pattern is applied to modify the default properties of the variables. Each property can be modified by passing a constant value that applies to all variables created or by passing a function that generates the respective property based on the indices/objects it is created for:

```
// Create a 3x4 matrix of binary variables.
// The names of the variables are "x_0_1", "x_0_2", ...
Variable[,] x = prob.AddVariables(3,4)
  .WithType(ColumnType.Binary)
  .WithName((i,j) => "x_" + i + "_" + j)
  .ToArray();
// Create a map of variables that is indexed by pairs of objects from
// the cartesian product of c1 and c2. All variables are integer and
// the upper bounds are computed from the properties of the objects
List<C1Type> c1 = ...;
List<C2Type> c2 = ...;
HashMap2<C1Type,C2Type,Variable> y = prob.AddVariables(c1, c2)
  .WithType(ColumnType.Integer)
  .WithUB((c1object,c2object) => c1object.UB * c2object.GetUB)
  .ToArray();
```

Again, the reference documentation provides more details.

Note that the `ToArray()` function does not return a jagged array `[][]` but instead returns type `[,]`.

4.3 Expressions

In BCL any kind of expression is represented by class `XPRBexpr` and its subclasses `XPRBlinExp` (for linear expressions) and `XPRBquadExp` (for quadratic expressions).

The new API provides analogous classes `Expression`, `LinExpression` and `QuadExpression`. These expressions work in the same way in all APIs:

```
Variable x = ..., y = ..., z = ...
// Create 3*x + 4*y + z
LinExpression l = LinExpression.Create();
l.AddTerm(3, x);
l.AddTerm(y, 4);
l.AddTerm(z);
// Create 3*x^2 + 4*x*y + 5*z
QuadExpression q = QuadExpression.Create();
q.AddTerm(3, x, x);
q.AddTerm(4, x, y);
q.AddTerm(5, z);
```

4.4 Creating constraints

In BCL a constraint is created using `XPRBprob.newCtr()`. This function creates a constraint and returns an object representing the newly created constraint.

Adding constraints with the new API works similarly. Here you use function `AddConstraint()` to create a single constraint and function `AddConstraints()` to create multiple constraints with a single function call (this is more efficient). The latter function has a number of overloads, in particular for creating multi-dimensional constraints, for creating constraints directly from data etc. See the reference documentation for further details.

In order to create a less-than-or-equal, equal, greater-than-or-equal or a range constraint, use functions `Leq()`, `Eq()`, `Geq()`, or `In()`, respectively, of the `Expression` class:

```
// Add constraint 2*x+3*y <= 5
prob.AddConstraint(x.mul(2).plus(y.mul(3)).leq(5));
// Add constraint 6*y >= 7*z
prob.AddConstraint(y.mul(6).geq(z.mul(7)));
// Require x+y to be in [2,3]
prob.AddConstraint(x.plus(y).in(2, 3));
```

You can also directly use operators <=, >= and == instead of functions `Leq()`, `Geq()` or `Eq()`.

4.5 Specifying the objective function

In BCL the objective function is set by calling `XPRBprob.setObj()` which expects as argument a constraint expression without relational operator.

In the new API the objective function is set using function `XpressProblem.SetObjective()` which expects an expression as argument. The objective sense is set by picking an overload for `SetObjective` that also allows setting the sense.

```
// Minimize x+y
prob.SetObjectve(x + y);
// Maximize x+y
prob.SetObjective(x + y, ObjSense.Maximize);
```

4.6 Operators

Like BCL, the new API makes it possible to create linear and quadratic expressions by combining constants, variables and expressions using the `Plus()`, `Minus()`, `Mul()`, and `Div()` member functions. As opposed to BCL, these operations are always guaranteed to work in constant time. None of the operations require deep copies of their arguments.

In addition, there are convenience functions `Sum()` and `ScalarProduct()` in `Optimizer.Objects.Utils`. These functions make creating expressions somewhat easier and more readable in some cases.

```
// Create 3*x[0] + 2*x[1] + 4*x[2] in different ways:
Variable[] x = ...
e = x[0].Mul(3).Plus(x[1].Mul(2)).Plus(x[2].Mul(4))
e = Sum(x[0].Mul(3), x[1].Mul(2), x[2].Mul(4))
e = ScalarProduct(x, new double[]{ 3, 2, 4 })
e = 3*x[0] + 2*x[1] + 4*x[2]
```

As seen above, operators +, −, `*`, / are overloaded. So instead of `x.Plus(y)` you can also write `x+y`.

4.7 Interacting with Optimizer controls and attributes

In order to interact with Optimizer attributes or controls or with special solver features in BCL, you first have to obtain a representation of the solver problem, an instance of class `XPRSprob`. In addition, when using this instance, you have to make sure that data is properly synchronized between BCL and the Optimizer.

None of this is necessary in the new API. The `XpressProblem` is a subclass of `XPRSprob` and it no longer holds a representation of the problem model that is separate from the solver. In the new API there is only a single representation of the problem. Any change you make to the problem formulation via the new API directly changes the problem held in the solver. Also, all solver controls and attributes are directly available as members of the `XpressProblem` class.

```
XpressProblem prob = ...;
// Set maximum number of nodes and query the solution status
prob.MaxNode = 1;
Console.WriteLine(prob.SolStatus);
```

4.8 Solution objects

BCL provides `XPRBsol` objects to capture a solution.

The new API does not provide such a class. Instead a solution is represented as an array of `double` values, one for each variable. You retrieve such an array using `XpressProblem.GetSolution()` and then pass the array to a variable's `GetValue()` function:

```
Variable[] x = ...;
... // build the model here
prob.Optimize();
double[] sol = prob.GetSolution();
foreach (Variable v in x)
Console.WriteLine("{0} = {1}", x.GetName(), x.GetValue(sol));
```

4.9 Special ordered sets

In BCL special ordered set constraints are specified using `XPRBprob.newSos()` which expects the variables and their weights to be passed as an expression (`XPRBexpr`).

In the new API a special ordered set constraint is created by first creating an SOS description, for example using function `SOS.Sos()` and then passing this result to `XpressProblem.AddConstraint()`:

```
Variable[] x = ...;
double[] weigths = ...;
prob.AddConstraint(SOS.Aos(SetType.SOS1, x, weights));
```

4.10 Indicator constraints

In BCL indicator constraints are defined by setting the "indicator" property of an instance of class `XPRBctr`.

In the new API you can still set the indicator variable for a particular constraint but in a slightly different way:

```
// if x == 1 then y == 5
Variable x = ...;
Variable y = ...;
Inequality i = prob.AddConstraint(y == 5);
prob.SetIndicatorVariable(x, true, i);
```

The boolean argument to `SetIndicatorVariable()` specifies whether the constraint should be activated if the indicator variable is non-zero (`true`) or zero (`false`).

The same indicator constraint can also be created in a different way:

```
prob.AddConstraint(x.IfThen(y == 5));
```

There are also functions `IfThen()` and `IfNotThen()` in the `Variable` class to create a constraint that only becomes active if the indicator variable is set to one or zero, respectively.

4.11 Index sets

In the new API there is no equivalent to BCL's index sets (as created by `XPRBprob.newIndexSet()`).

Instead you should use the collections that are provided by the programming language.

4.12 Cuts

In the new API there is no equivalent to BCL's cut data type (as created by `XPRBprob.newCut()`).

A cut in the new API is just an instance of `Inequality` (like any other row constraint) and can be added using function `XpressProblem.AddCut()`.

Note that the `AddCut()` function automatically presolves the cut if necessary. In other words, you can provide the cut as an expression in `Variable` instances and the API will correctly translate this cut to the presolved model. It is no longer necessary to call a function like `XPRBprob.setCutMode()` to enable adding of cuts from callbacks.

4.13 Deletion

As in BCL, the new API allows for deleting rows and SOSs. The functions for this are `XpressProblem.DelInequalities()` and `XpressProblem.DelSOS()`.

As opposed to BCL, the new API also allows for deletion of variables by means of function `DelVariables()`.

4.14 Memory

In BCL almost every object is a thin wrapper around an object that is allocated in native (and not managed) memory. Therefore one must be careful to call `Dispose()` soon enough to make sure native resources are released. BCL also deals with two different kinds of problem representation: one is the representation by BCL objects, the other one is the matrix based representation in the Optimizer. This increases the memory footprint since there are always two copies of the problem.

In the new API the only class wrapping native resources is `XpressProblem`. This wraps the solver engine (which is implemented in C). All objects required to model and state the problem are allocated in managed memory. Hence there is no need to think about those being released properly/soon enough: the garbage collector will handle this for you. Moreover, the new API does not have an additional copy of the problem. Everything is directly created in the solver and all manipulations are applied directly to the solver model.

As a consequence it is not necessary to control the lifetime of expression objects via `using` statements or by classes like `XPRBexprContext`.

4.15 Synchronization

Since BCL keeps two separate copies of the problem (one with BCL objects, one inside the Optimizer) these copies must be kept in sync. This requires calling methods like `XPRBprob.loadMat()` or `XPRBprob.sync()`. It also puts some limitations on what things can be done in parallel in callbacks.

Since the new API only ever has a single copy of the problem, no synchronization is required at all. Moreover, no special considerations for parallel callbacks are required.

In callbacks there is no need to call functions like `XPRBprob.beginCB()` and/or `XPRBprob.endCB()`. The problem instance passed into the callback is always up to date.

4.16 Mapping from BCL to the new API

4.16.1 Objects

BCL	New API
`XPRBvar`	`Optimizer.Objects.Variable`
`XPRBctr`, `XPRBcut`, `XPRBrelation`	`Optimizer.Objects.Inequality`
`XPRBexpr`	`Optimizer.Objects.Expression`
`XPRBbasis`	Not required, use plain arrays of `double` and `int` to interact with the basis get and set methods of the `XpressProblem` class (more precisely, its `XPRSprob` superclass)
`XPRBsol`	Not required, use plain arrays of `double` to interact with the solution query methods of the `XpressProblem` class (more precisely, its `XPRSprob` superclass)
`XPRBexpr`, `XPRBlinExp`, `XPRBquadExp`, `XPRBterm`, `XPRBqterm`	`Expression` and all its subclasses (in most cases the actual concrete subclass should not matter)
`XPRBlinExp`	`Optimizer.Objects.LinExpression`
`XPRBquadExp`	`Optimizer.Objects.QuadExpression`
`XPRBindexSet`	Use collections provided by the programming language
`XPRBsos`	`Optimizer.Objects.SOS`

4.16.2 Functions

BCL	New API
`XPRBprob.newVar` (with overloads)	`XpressProblem.AddVariable` and `XpressProblem.AddVariables` (with overloads)
`XPRBprob.newCtr` (with overloads)	`XpressProblem.AddConstraint` and `XpressProblem.AddConstraints` (with overloads)
`XPRBprob.delCtr`	`XpressProblem.DelInequalities`
`XPRBprob.delSos`	`XpressProblem.DelSOS`
`XPRBprob.getLPStat`, `XPRBprob.getMIPStat`	Directly query the solver's `LPSTATUS` and `MIPSTATUS` attributes
`XPRBprob.loadMat`, `XPRBsync`	Not required since there is only one problem representation
`XPRBprob.isValid`	Not required since the model is always in a valid state
`XPRBprob.readBinSol`, `XPRBprob.readSlxSol`, `XPRBprob.writeBinSol`, `XPRBprob.writeSlxSol`	Use the Optimizer's functions to interact with solution files
`XPRBprob.setOutputStream`	Use the Optimizer's message callback instead
`XPRBprob.solve`, `XPRB.mipOptimize`, `XPRB.lpOptimize`	`XpressProblem.Optimize`, `XpressProblem.MipOptimize`, `XpressProblem.LpOptimize` (which are all in fact members of the `XPRSprob` superclass), preferably use `Optimize()` unless you explicitly need one of the other functions.
`XPRBprob.loadMipSol`	Use `XpressProblem.AddMipSol`
`XPRBprob.getProbStat`	Not required, solution status can be queried directly from the Optimizer, see the `SolStatus` attribute

4.16.3 Constants

BCL	New API
Variable type XPRB.BV, XPRB.PI, XPRB.PL, ...	Optimizer.ColumnType
Row type XPRB.E, XPRB.G, XPRB.L, XPRB.N	Optimizer.RowType
LP solution status LP_*	Use the corresponding constants from the XPRSprob class directly.

APPENDIX A

Contacting FICO

FICO provides clients with support and services for all our products.

FICO Customer Support

FICO Customer Support offers technical support and services ranging from self-help tools to direct assistance with a FICO technical support engineer. Support is available to all clients who have an active maintenance contract.

The FICO Customer Self-Service Portal (support.fico.com) is a secure web portal that allows users to open, review, and update their support cases; manage their organization's portal users; find solutions to common problems in the FICO Knowledge Base; and view the availability of their cloud applications 24 hours a day, 7 days a week.

You can find support contact information and a link to the FICO Customer Self-Service Portal (online support) on the Product Support home page (www.fico.com/en/product-support).

Please include *'Xpress'* in the subject line of your support queries.

Documentation

FICO continually looks for new ways to improve and enhance the value of the products and services we provide.

If you have comments or suggestions regarding how we can improve this documentation, let us know by sending your suggestions to techpubs@fico.com. Please include your contact information (name, company, email address, and optionally, your phone number) so we may reach you if we have questions.

FICO Learning

FICO Learning is the principal provider of product training for our clients and partners. FICO Learning offers instructor-led classroom courses, web-based training, seminars, and training tools for both new user enablement and ongoing performance support.

For additional information, visit the FICO Learning home page at www.fico.com/en/product-training or email producteducation@fico.com.

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

About FICO

FICO (NYSE:FICO) is a leading analytics software company, helping businesses in 90+ countries make better decisions that drive higher levels of growth, profitability, and customer satisfaction. Learn more at www.fico.com or contact us at www.fico.com/en/contact-us.