Quick reference

FICO

FICO® Xpress Optimization

Guide for evaluators 2

Advanced optimization tasks

Last update 17 May, 2017

www.fico.com | Make every decision count™

FICO® Xpress Optimization

Guide for evaluators 2

Advanced optimization tasks

17 May, 2017

Introduction

Building on the introductory *Guide for evaluators* this document provides a framework for evaluating advanced optimization tasks with FICO® Xpress Optimization. In particular, this guide shows you how to:

1. Parameterize and tune optimization algorithms.
2. Interact with the solvers through callback functions.
3. Obtain multiple solutions.
4. Analyze and handle infeasibility.
5. Exchange data in memory.
6. Work on a distributed architecture.
7. Control the remote execution of Mosel models without any local Xpress installation.

All examples in this guide are provided as Xpress Mosel models (Evaluation Scenario 1). Where applicable, we also explain how to perform the same tasks with Xpress BCL (Evaluation Scenario 2).

Please refer to the *Guide for evaluators* for any questions relating to the installation of Xpress or recommendations for the choice of products from FICO Xpress Optimization.

1 Optimizer parameter settings and tuning

Among the most frequently used controls are the stopping criteria for the optimization algorithms. Stopping criteria controls include

Parameter name	Description
MAXTIME	The maximum time in seconds that the Optimizer will run before it terminates (0: no time limit; n>0: if an integer solution has been found, stop MIP search after n seconds, otherwise continue until an integer solution is finally found; n<0: hard stop after n seconds)
MAXNODES	The maximum number of nodes that will be explored by branch and bound.
MIPRELSTOP	Stop the MIP search when the gap (difference between the best solution's objective function and the current best solution bound) becomes less than the specified percentage.
MIPABSSTOP	Stop the MIP search when the gap (difference between the best solution's objective function and the current best solution bound) becomes less than the specified absolution value.

For the complete list of parameters please refer to the chapter 'Control parameters' of the "Xpress Optimizer Reference Manual".

≫**Scenario 1 (Mosel)**

Open the model file `examples\getting_started\Mosel\foliomip3.mos`. This file contains a version of the portfolio optimization problem from the Mosel part of the 'Getting Started' manual with some additional constraints and a larger data set. Try out the effect of setting different values in the `setparam` commands (just before the call to the optimization).

≫**Scenario 2 (BCL)**

Depending on your choice of a host language open one the BCL files `foliomip3.[c|cxx|java]` located in the corresponding subdirectory `examples\bcl\[C|C++|Java]\UGExpl` of the Xpress distribution. This new version of the portfolio optimization problem from the BCL part of the 'Getting Started' manual has some additional constraints and uses a larger data set. Try out the effect of setting different values for the control parameters (just before the call to the optimization).

Further information

- Mosel: "Xpress Mosel User Guide", Part II 'Advanced language features'
- BCL: see examples documented in the "BCL Reference Manual", Appendix B 'Using BCL with the Optimizer library'

1.1 Tuning the Optimizer

Xpress Optimizer solves optimization problems by applying a number of algorithms and techniques, such as cutting planes, heuristics, branch and bound search, *etc.* These internal algorithms are user customizable through *control parameters*.

Solver *tuning* helps the user to identify a *favorable set of control parameters* that allow the Xpress Optimizer to solve a particular problem (or a set of problems) faster than by using defaults.

Xpress Optimizer has a built-in tuner that can be used through different APIs on all supported platforms.

1.1.1 The Xpress Optimizer built-in tuner

The Xpress Optimizer built-in tuner works with LP and MIP problems, and when Xpress Nonlinear is available it can also work with SLP and MISLP problems. The tuner will solve the problem with its default baseline control settings and then solve the problem mutiple times with each individual control and certain combinations of these controls. As the tuner works by solving a problem mutiple times, it is important and recommended to set time limits. Setting `MAXTIME` will limit the effort spent on each individual solve and setting `TUNERMAXTIME` will limit the overall effort of the tuner.

A tuner run produces detailed log files (by default located under the subdirectory `tuneroutput` of the working directory) and also displays a summary progress log.

The tuner works on an optimization problem loaded into the Optimizer or alternatively it can be launched on a set of matrices in LP or MPS format. The tuning process can be customized by providing pre-defined lists of controls, so-called *factory tuner methods*, and through various output and tuning target settings—the reader is referred to the section *Using the Tuner* of the “Xpress Optimizer Reference Manual” for further detail.

Applying the tuning results

Copy the control parameter settings of the best strategy into your model or program. Note that any solver parameters need to be set *before* starting the optimization.

≫Scenario 1 (Mosel)

The tuner can be launched for a specific optimization problem directly from within a Mosel model by adding the option XPRS_TUNE to the optimization routine call:

```
minimize(XPRS_TUNE, MinCost)
maximize(XPRS_TUNE+XPRS_LIN, TotalProfit)
```

Note that the Optimizer output display (parameter XPRS_VERBOSE) needs to be enabled in order to see the tuner progress log displayed.

Alternatively, within Xpress Workbench select the ’Tune’ button to tune the last optimization problem specified within a model.

Applying the tuning results

Prefix the parameter name with XPRS_ to obtain its Mosel name:

```
setparam("XPRS_PRESOLVE", 0)
```

Alternatively, define parameter settings as a system command (requires module *mmsystem*):

```
command("PRESOLVE=0")
```

≫Scenario 2 (BCL)

After loading the problem from BCL into the solver, you can use the Xpress Optimizer API functions to set solver controls and start the tuning for this problem.

- BCL C++:

```
XPRSprob optprob;
XPRBprob bclprob("MyProb");                 // Initializes BCL and Optimizer

bclprob.loadMat();
optprob = bclprob.getXPRSprob();            // Retrieve the Optimizer problem
XPRSsetintcontrol(optprob, XPRS_MAXTIME, 60);  // Set a time limit
XPRStune(optprob, "");                      // Start the tuning
```

- BCL Java:

```
XPRBprob bclprob;
XPRSprob optprob;

bcl = new XPRB();                           // Initialize BCL
bclprob = bcl.newProb("MyProb");            // Create a BCL problem
XPRS.init();                                // Initialize Xpress Optimizer
optprob = bclprob.getXPRSprob();            // Retrieve the Optimizer problem
optprob.setIntControl(XPRS.MAXTIME, 60);    // Set a time limit
optprob.tune("");                           // Start the tuning
```

Applying the tuning results

You need to retrieve the Xpress Optimizer problem associated with the BCL problem to be able to modify its control parameter settings as shown for the 'MAXTIME' control in the code snippets above.

≫Scenarios 3 and 4 (Xpress Optimizer)

With a loaded problem, the built-in tuner can be started by calling `TUNE` from the Optimizer console, or `XPRStune` from a user application. The Xpress Optimizer API also provides routines for managing tuner methods.

At the command prompt, the following sequence of commands can be used to tune an LP problem that is provided in the form of an MPS matrix in the file foliolp.mps:

```
optimizer
foliolp
readprob
chgobjsense max
maxtime=60
tune -l
quit
```

Applying the tuning results

With the Optimizer console, you can modify control parameter settings as shown for the 'MAXTIME' control in the command listing above. From an application program you need to use one of the Optimizer API routines `XPRSsetdblcontrol` / `XPRSsetintcontrol` / `XPRSsetstrcontrol` depending on the parameter type.

Further information

- Xpress Optimizer: "Xpress Optimizer Reference Manual", Section 5.10 'Using the Tuner'

2 Solver interaction: Setting up solution callbacks

The solvers of FICO® Xpress Optimization offer various entry points for interaction with the solver during optimizations runs. This interaction takes the form of user-defined functions with a fixed format (*callback functions*) that are invoked by the solver at specific points. Callback functions may be used for logging purposes, but some also allow you to modify the optimization algorithms, *e.g.*, by adding your own cutting plane algorithms or even defining new branching objects and strategies for the MIP branch-and-bound search.

Among the most frequently used functions certainly is the integer solution callback of Xpress Optimizer that provides access to MIP solutions at the point where they are found during the branch-and-bound search.

≫Scenario 1 (Mosel)

A unique feature of the Mosel language is the possibility to define subroutines in this high-level modeling language that will be called from the underlying (solver) libraries. It is thus possible to work with Optimizer callback functions directly in your Mosel models.

Open the file `examples\getting_started\Mosel\foliocb.mos` to see an example of how to define an integer solution callback in Mosel.

≫Scenario 2 (BCL)

With BCL you can directly use the callback functionality of the corresponding Xpress Optimizer interface. Depending on your host language, open the file `foliocb.[c|cxx|java]` located in the corresponding subdirectory `examples\bcl\[C|C++|Java]\UGExpl` for an example of how to use the Optimizer integer solution callback with a BCL model formulation.

Further information

- Documentation of Xpress Optimizer callbacks: "Xpress Optimizer Reference Manual", 5.3 'Using the Callbacks'
- See the "Xpress Mosel User Guide", 11.1 'Cut generation', and the Xpress whitepapers "Embedding Optimization Algorithms" and "Hybrid MIP/CP solving" for examples of callback definition in Mosel
- Documentation of Xpress Optimizer callbacks in Mosel: "Mosel Language Reference Manual", Chapter 13: '*mmxprs*'
- Examples of using Xpress Optimizer callbacks with BCL: "BCL Reference Manual", Appendix B 'Using BCL with the Optimizer library'

3 Multiple solution support

As we have seen in the previous section it is possible to retrieve into your model/application all

MIP solutions found by Xpress Optimizer during the branch-and-bound search. Alternatively to defining the integer solution callback and saving the solution in your own structures you can make use of the *MIP solution pool* and *MIP solution enumerator* functionality to have solutions stored and made accessible after search has terminated.

≫Scenario 1 (Mosel)

As shown in the example file `examples\getting_started\Mosel\folioenumsol.mos` the solution pool functionality is enabled by using the option `XPRS_ENUM` of the optimization routines `maximize` or `minimize`. Through the control `XPRS_ENUMMAXSOL` you can set the maximum number of solutions to save (if the search finds more solutions than the specified value, then only the best solutions are retained). The solution enumerator configures the optimization algorithms to generate a large number of feasible solutions, as a consequence, solution times are generally longer than with the default algorithms that are tuned for maximum speed.

≫Scenario 2 (BCL)

The BCL program example `foliosolpool.[c|cxx|java]` in directory `examples\bcl\[C|C++|Java]\UGExpl` shows how to load solutions saved in the MIP solution pool into BCL to use its display functions on the model objects. The solution pool collects all solutions found during the standard search (these are the same solutions as those reported by the MIP solution callback) and its access functions can be configured, for instance, to return solutions in ascending or descending order of the value of the optimization criterion.

If you wish to generate many different feasible solutions you can start the optimization through the solution enumerator, as is shown in the example `folioenumsol.[c|cxx|java]`. In this case, the optimization algorithms are configured to produce a large number of integer feasible solutions; solving times may be longer than with the default algorithm settings that are tuned for maximum speed. The solutions generated by the solution enumerator are stored in a MIP solution pool associated with the enumerator. After the optimization run the solutions saved in the pool are loaded into BCL.

≫Scenario 3 (Optimizer)

The example files `foliomatsolpool.[c|java]` in directory `examples\getting_started\Optimizer` show how to use the solution pool functionality when inputting a problem directly into Xpress Optimizer. The solution pool captures and stores all solutions found during the MIP search. The user can query the stored set of solutions for information such as their objective value and their solution values.

In further example files `foliomatenumsol.[c|java]` the solution pool and solution enumerator functionality is demonstrated showing how to capture the *n*-best solutions of a MIP problem. The example also shows how to access the information about each of the *n* solutions found.

Further information

- “MIP Solution Pool Reference Manual”
- Documentation of solution pool functionality in Mosel: “Mosel Language Reference Manual”, Chapter 13: ‘*mmxprs*’

- Documentation of solution pool functionality in BCL: “BCL Reference Manual”; “BCL Reference Manual Javadoc”

4 Handling infeasibility

A problem is said to be *infeasible* if no solution exists which satisfies all the constraints. The problem status *’infeasible’* generally is an undesirable outcome that should be avoided. In other terms, if an infeasibility arises you need suitable means to analyze what is going wrong and modeling techniques and tools to remedy or prevent this state from happening.

The following sections present different possibilities how to deal with infeasibility in FICO Xpress Optimization models.

4.1 Modeling for infeasibility

By ’modeling for infeasibility’ we mean a model formulation that incorporates some extra freedom to make the problem feasible and minimizes the utilization of the added freedom. In mathematical terms this corresponds to introducing *deviation variables* in constraints to make them feasible and to penalize these deviations in the objective function. Adding deviation variables to a model formulation implies being able to modify the definition of variables or constraints after their creation, a feature well supported in Mosel and BCL.

NB: deviation variables should only be added in constraints that use external data (to make up for infeasibility in data). Do not relax constraints that have no data, only variables, representing relationships that *must* hold, such as inventory balance constraints, physical processes, conversion rates or mass balance.

≫Scenario 1 (Mosel)

The example file `examples\getting_started\Mosel\folioinfeas.mos` solves an LP problem. If the problem instance is found to be infeasible the model retrieves and displays the infeasible variables and constraints, it then adds deviation variables to certain constraints and re-solves the modified problem. The final solution display takes into account the values of the deviation variables, thus providing some insight on the type of constraint violations.

≫Scenario 2 (BCL)

The BCL file `folioinfeas.[c|cxx|java]` in directory `examples\bcl\[C|C++|Java]\UGExpl` implements the same algorithm as the Mosel model above: if a problem instance is found to be infeasible, the program adds deviation variables to certain constraints and re-solves the modified problem. The final solution display takes into account the values of the deviation variables, thus providing some insight on the type of constraint violations.

4.2 Analyzing infeasibility: IIS

A general technique to analyze infeasibility is to find a small portion of the matrix that is itself infeasible. Xpress Optimizer does this by finding *irreducible infeasible sets* (IISs). An IIS is a minimal set of constraints and variable bounds which is infeasible, but becomes feasible if any constraint or bound in it is removed.

≫Scenario 1 (Mosel)

Take a look at the Mosel model in file `examples\getting_started\Mosel\folioiis.mos`. This example retrieves the IIS sets for an LP-infeasible problem and displays their contents.

≫Scenario 2 (BCL)

Similarly, you can use the IIS access functionality of BCL to retrieve and display IIS (see file `folioiis.[c|cxx|java]` in directory `examples\bcl\[C|C++|Java]\UGExpl`).

4.3 Infeasibility repair

In some cases, identifying the cause of infeasibility, even if the search is based on IISs may prove very demanding and time consuming. In such cases, a solution that violates the constraints and bounds minimally can greatly assist modeling. This functionality is provided by the *infeasibility repair* utility of Xpress Optimizer. Based on preferences provided by the user, the Optimizer relaxes the constraints and bounds in the problem by introducing penalized deviation variables associated with selected rows and columns. The preference values reflect the modeler's will to relax the corresponding bound or constraint right-hand-side (the penalty value associated is the reciprocal of the preference), a zero preference means no relaxation.

≫Scenario 1 (Mosel)

The infeasibility repair functionality in Mosel has two forms, the simpler one lets you define a preference per constraint type (=, $\leq$, $\geq$), with its detailed form you can state a preference for every single constraint. The latter is used in the example implementation in file `examples\getting_started\Mosel\foliorep.mos`. This example performs the infeasibility repair algorithm several times, each time with a different value for the parameter `delta` to analyze the effect of extra violations of the constraints and bounds to the underlying model.

≫Scenario 2 (BCL)

The BCL programs `foliorep.[c|cxx|java]` in directory `examples\bcl\[C|C++|Java]\UGExpl` show how to use the infeasibility repair functionality of the Xpress Optimizer Library with a model defined in BCL. In analogy to the Mosel model, this example uses the weighted infeasibility repair algorithm that lets you specify which constraints to relax, and loops over different values for the penalization factor `delta`.

Further information

- "Xpress Optimizer Reference Manual", 3.2 'Infeasibility'
- Examples of algorithms modifying constraint definitions in Mosel: "Xpress Mosel User Guide", Chapter 12: 'Extensions to Linear Programming'
- Functions for modifying constraint definitions in BCL: "BCL Reference Manual", 2.3 'Constraints'

5 Data exchange in memory for Mosel models

The development process of an optimization application typically includes several phases with very different needs in terms of data handling:

1. Feasibility study: development and implementation of the mathematical model, spending the least possible effort on data handling, test data possibly even defined directly in the model
2. Initial testing: running the model with different data sets (*i.e.*, need for parameterization of data instances), possibly producing some performance statistics
3. Prototype for presentation to end users and decision makers: formatted display of results, possibly including graphics, 'hiding' the mathematical model behind a graphical interface that makes accessible only the most important input and output parameters and data
4. Embedding into the company's information systems: incorporate the model into a host application using efficient mechanisms for communicating data and results

With Mosel it is possible to carry through the whole development process using a single model, adding different interfaces on top of this model that all employ the same mechanism for data input and output directly in memory.

The following series of examples works with the model file `foliomemio.mos` located in the directory `examples\getting_started\Mosel` of your Xpress installation. This model file implements the same MIP problem as in `foliomip3.mos` with the difference that all input and output data (arrays, sets, and scalars) have been turned into model parameters and the result display has been removed from the model.

1. **Standalone mode:** run the model `foliomemio.mos` in Xpress Workbench or from the Mosel command line. With all settings at default, this model run reads in data from a text file (`folio10.dat`) and produces an output file listing the result values (file `sol10out.dat`).
2. **Submodel to another Mosel model:** now run the model `runfolio.mos`. This model does not perform any optimization run of its own, it merely serves as an interface to execute the model `foliomemio.mos` from which it retrieves the results and generates text output formatted as an HTML page. Data are exchanged in memory between the two models. The file `runfolio.mos` can easily be extended to run several instances of the optimization model (*e.g.*, with different parameter settings), in sequence or in parallel, and produce the corresponding summary statistics—an example of such parallel solving is shown in `runfoliopar.mos`.
3. **Embedding into a host application:** the C program `runfolio.c` compiles and runs the Mosel model `foliomemio.mos`, initializing its data arrays from data held in C and retrieving result data into the C application. The Java program `runfolio.java` does exactly the same for a Java application, and `runfolio.cs` is the equivalent C# program. In the place of the fixed-size memory blocks used in these examples you can also exchange data dynamically using Mosel's I/O callback functionality—generating data on-the-fly or flexibly (re)sizing output structures (see example files `runfoliod.[c|java|cs]`). And yet another option, in Java or C# it is possible to pass data through buffers (see files `runfoliob.java` and `runfoliob.cs` respectively).

Further information

- Further examples: "Xpress Mosel User Guide", Part III: 'Working with the Mosel Libraries' and Chapter 17: 'Language extensions'; Xpress whitepapers "Generalized file handling in Mosel" and "Multiple models and parallel solving with Mosel".
- Documentation of Mosel C libraries: "Xpress Mosel Library Reference Manual".
- Documentation of Mosel Java libraries: "Xpress Mosel Library Reference Manual JavaDoc".
- Documentation of the Mosel .NET interface: "Xpress Mosel .NET Interface".

6 Working in a distributed architecture

6.1 Remote Mosel instances

When working with a Mosel model it is possible to start new instances of Mosel either locally to the running system or remotely on another machine through the network, and use these instances to run additional models controlled by the model that has started them. This means that the computing capacity of the running model is not restricted to the executing process.

Moving from a single instance implementation running multiple models to a distributed architecture most often requires only few changes to existing models, namely the setup of the remote connections. There are no restictions on the processor or operating system type: any platforms supported by Xpress can be used jointly in a distributed model run provided that they have a suitable version of Xpress installed and licensed.

Please note that before executing models on remote nodes, you need to start the Mosel server 'xprmsrv' on all nodes you wish to use: in a command line interpreter window type:

```
xprmsrv
```

or alternatively, under Windows, double click on the 'xprmsrv' icon.

Two examples show how to run remotely the model file `foliomemio.mos` located in the directory `examples\getting_started\Mosel` of your Xpress installation:
The model `runfoliodistr.mos` is an extension of `runfolio.mos` presented in the preceding section. It serves as an interface to execute remotely the model `foliomemio.mos` from which it retrieves the results and generates text output formatted as an HTML page.
The model version `runfoliopardistr.mos` starts several model instances in parallel, each on a distinct (remote) Mosel instance, and produces the corresponding summary statistics.

Further information

- Examples: Xpress whitepaper "Multiple models and parallel solving with Mosel" (Section 'Working with remote Mosel instances').
- Documentation: Section 'mmjobs' of the "Xpress Mosel Language Reference Manual".

6.2 HTTP

A Mosel model can communicate with external components via HTTP requests. It can act as a *client* sending HTTP queries (GET, POST, PUT or DELETE) or as a *web service* by starting the integrated HTTP server.

HTTP client functionality in Mosel can be configured to work *synchronously* (a request waits for the answer from the server) or *asynchronously* (functions sending requests return immediatly and termination messages are sent via separate event messages). Other configuration settings involve, for example, the maximum number of simultaneous requests accepted by a server or the TCP port to be used by remote connections.

The example file `foliohttpsrv.mos` starts up an HTTP server that accepts HTTP requests triggering the execution of the optimization model in file `folioxml.mos`. The model `foliohttpclient.mos` shows what the corresponding HTTP client might look like, it exchanges input and result data in XML-format with the HTTP server and displays the solution. Both examples are located in the directory `examples\getting_started\Mosel` of your Xpress installation.

Further information

- Documentation: Section 'mmhttp' of the "Xpress Mosel Language Reference Manual".

7 Remote applications without local Xpress installation

Managing Mosel model executions in a distributed architecture does not necessarily imply a need for a local Xpress installation on the machine controling the application: the *Mosel remote invocation library (XPRD)* makes it possible to build applications requiring the Xpress technology that run from environments where Xpress is not installed—including architectures for which Xpress is not available. XPRD is a self-contained library (*i.e.* with no dependency on the usual Xpress libraries) that provides the necessary routines to start Mosel instances either on the local machine or on remote hosts and control them in a similar way as if they were invoked through the Mosel libraries. Besides the standard instance and model handling operations, the XPRD library supports the file handling mechanisms of *mmjobs* as well as its event signaling system.

Several sets of examples implement the counterparts to the Mosel examples `runfoliodistr.mos` and `runfoliopardistr.mos` for the remote execution of the model file `foliomemio.mos` without local installation of Xpress: these C and Java files are located in the directory `examples\mosel\WhitePapers\MoselPar\XPRD`.

The example `runfoliodistr.[c|java]` executes a Mosel model remotely and retrieves the results via streams. Example versions `distfolio.[c|java]` and `distfoliopar.[c|java]` transfer data via files using Mosel's binary format. The file `distfolio.[c|java]` starts a single remote model run and `distfoliopar.[c|java]` shows how to perform several concurrent model executions each using a different (possibly remote) Mosel instance.

Even more advanced is the program version `distfoliocbioev.[c|java]` where result data is communicated during the optimization model run. The interaction of the Mosel model and the host application is coordinated via event messages that are exchanged between the two.

Further information

- Examples: Xpress whitepaper "Multiple models and parallel solving with Mosel" (Section 'XPRD: Remote model execution without local installation').
- Documentation of the XPRD C library: "XPRD: Mosel Remote Invocation Library Reference Manual".
- Documentation of the XPRD Java library: "XPRD: Mosel Remote Invocation Library JavaDoc".