Using COIN-OR Solvers with Microsoft Windows

Horand Gassmann, Jun Ma, and Kipp Martin

July 3, 2009

1 Overview

This binary distribution is specifically designed for Microsoft Windows users who: 1) want to use COIN-OR solvers to solve optimization problems but do not want to compile any code; or 2) want to write applications using Visual Studio projects that link to COIN-OR solver libraries. This download is plug-and-play, complete with pre-configured Visual Studio project files – *it is not necessary* to build any COIN-OR projects from source code. This document contains the following information. In Section 2 we detail the various components contained in this distribution. In Section 3 we show how to call and use the optimization solvers contained in this distribution. In Section 4 we describe how to call the COIN-OR solvers in this distribution using the AMPL modeling language. In Section 5 we describe the Visual Studio project files. Finally, in Section 6 we detail nine examples (for which there are Visual Studio project files) that illustrate how to use the COIN-OR libraries provided in this distribution. Libraries from the following COIN-OR projects are contained in this distribution.

Bcp (`https://projects.coin-or.org/Bcp`)

Bonmin (`https://projects.coin-or.org/Bonmin`)

Cbc (`https://projects.coin-or.org/Cbc`)

Cgl (`https://projects.coin-or.org/Cgl`)

Clp (`https://projects.coin-or.org/Clp`)

Couenne (`https://projects.coin-or.org/Couenne`)

CoinUtils (`https://projects.coin-or.org/CoinUtils`)

CppAD (`https://projects.coin-or.org/CppAD`)

DyLP (`https://projects.coin-or.org/DyLP`)

Ipopt (`https://projects.coin-or.org/Ipopt`)

Optimization Services (`https://projects.coin-or.org/OS`)

Osi (`https://projects.coin-or.org/Osi`)

SYMPHONY (`https://projects.coin-or.org/SYMPHONY`)

2 The Binary Distribution

When properly installed, the material in this distribution is arranged into a number of folders in hierarchical fashion. The top level of this hierarchy consists of the following folders.

- **bin** – this directory contains solver executables. Using the solver executables is explained in Section 3,
- **data** – problem instances in OSiL, nl, and MPS format,
- **doc** – the directory containing this documentation, for more thorough documentation see `http://www.coin-or.org/OS/doc/osUsersManual_2.0.pdf`,
- **examples** – this directory contains source code illustrating how to build applications that use COIN-OR software,
- **include** – this directory contains the necessary header files if a user wishes to build applications linking to the COIN-OR libraries supplied in the **lib** directory,
- **lib** – this directory contains the solver libraries, the Visual Studio project files are linked to these libraries,
- **MSVisualStudioOSExamples** – this directory contains Visual Studio project files for each of the examples given in Section 6.
- **share** – this directory contains author and license information for each of the COIN-OR projects that are part of this binary download

3 Calling COIN-OR Solvers with Model Instances

The following solvers are contained in the **bin** directory:

- `bonmin.exe` – a solver for mixed-integer nonlinear programs – see `https://projects.coin-or.org/Bonmin`;
- `cbc.exe` – a solver for mixed-integer linear programs – see `https://projects.coin-or.org/Cbc`;
- `clp.exe` – a solver for linear programs – see `https://projects.coin-or.org/Clp`;
- `couenne.exe` – a global optimizer for mixed-integer nonlinear programs – see `https://projects.coin-or.org/Couenne`;
- `ipopt.exe` – an optimizer for continuous nonlinear programs – see `https://projects.coin-or.org/Ipopt`;
- `symphony.exe` – a solver for mixed-integer linear programs – see `https://projects.coin-or.org/SYMPHONY`.

See the respective project pages referenced above for more detail on each of the solvers and which optimization instance formats they take. For the convenience of the user, the **bin** directory also contains the `OSSolverService.exe`. This executable is linked to libraries for all of the solvers that are in the **bin** directory, and can be used in lieu of any of them. One advantage of using the

OSSolverService.exe is its flexibility. Using this executable gives the user access to any of the above solvers with an instance in MPS, nl, or OSiL format. In addition, the OSSolverService.exe returns the solver solution in the OSrL XML format which is easily parsed. We now illustrate several calls to the OSSolverService.exe. Open a command window and change into the **bin** directory. Then execute the following in order to solve an instance (in this case a linear program) in OSiL format:

```
OSSolverService -osil ../../data/osilFiles/parincLinear.osil
```

The following illustrates solving an instance in AMPL **nl** format:

```
OSSolverService -nl ../../data/amplFiles/parinc.nl
```

Likewise, to solve a problem in **mps** format:

```
OSSolverService -mps ../../data/mpsFiles/parinc.mps
```

The result is printed to standard output in OSrL format. For example, the values of the primal variables are expressed as:

```
<values numberOfVar="2">
<var idx="0">539.9999999999999</var>
<var idx="1">252.0000000000001</var>
</values>
```

And the the objective function value is expressed as

```
<objectives>
<values>
<obj idx="-1">7667.941722450357</obj>
</values>
</objectives>
```

You can also print the result to a file by using the **osrl** option. This is done as follows:

```
OSSolverService -osil ../../data/osilFiles/parincLinear.osil
    -osrl result.xml
```

A call to OSSolverService.exe uses the Cbc solver as a default. In order to specify another solver, use the **solver** option to specify the solver you want. For example, to solve a model instance with SYMPHONY type

```
OSSolverService -osil ../../data/osilFiles/p0033.osil
      -solver symphony
```

To solve a nonlinear model with Bonmin type

```
OSSolverService -osil ../../data/osilFiles/bonminEx1.osil
      -solver bonmin
```

The name of the solver should always be given in all lower case. It is possible to build the OSSolverService to work with solvers other than the ones listed in Section 3, but they are not included due to licensing issues. Specifically, OSSolverServices has also been linked to these solvers:

- Glpk
- Cplex
- LINDO

For more detail on using the `OSSolverService.exe` see the documentation for the OS project `http://www.coin-or.org/OS/doc/osUsersManual_2.0.pdf`.

4 Calling COIN-OR Solvers using a Modeling Language

It is also possible to call all of these solvers directly from the modeling language AMPL (see `http://www.ampl.com`. In this discussion we assume the user has already obtained and installed AMPL. In the **bin** directory there is an executable, `OSAmplClient.exe` that is linked to all of the COIN-OR solvers in this distribution. The `OSAmplClient` acts like an AMPL "solver". The `OSAmplClient` is linked with the OS library and can be used to solve problems either locally or remotely. In both cases the `OSAmplClient` uses the `OSnl2osil` class to convert the AMPL generated nl file (which represents the problem instance) into the corresponding instance representation in the OSiL format.

In the following discussion we assume that the AMPL executable `ampl.exe`, the `OSAmplClient`, and the test problem `hs71.mod` are all in the same directory.

The problem instance `hs71.mod` is an AMPL model file included in the `data/amplClient` directory. To solve this problem locally by calling the `OSAmplClient` from AMPL first start AMPL and then execute the following commands. In this case we are testing Ipopt as the local solver.

```
# take in problem 71 in Hock and Schittkowski
# assume the problem is in the AMPL directory
model hs71.mod;
# tell AMPL that the solver is OSAmplClient
option solver OSAmplClient;
# now tell OSAmplClient to use Ipopt
option OSAmplClient_options "solver ipopt";
# now solve the problem
solve;
```

This will invoke Ipopt locally and the result in OSrL format will be displayed on the screen. In addition, the values of the optimal solution, objective, dual variables, etc. are returned to AMPL, so that they can be displayed and manipulated like solutions from any other AMPL solver.

Remote solution is also possible. For details, the user is directed to the OS User's Manual (`http://www.coin-or.org/OS/doc/osUsersManual_2.0.pdf`).

5 Using Visual Studio to Build Applications

In this section we describe the directory **MSVisualStudioOSExamples**. This directory contains nine Visual Studio project files, each in a separate folder. Each of these project files is linked to all of the COIN-OR libraries in the **lib** directory and the necessary header files in the **include** directory. The Visual Studio solution file `osExamplesSolution.sln` contains each of these projects. Building these examples (for instance after making modifications to the code) is therefore very easy: Find the solution file in the Windows Explorer and double-click on it.

This opens up Visual Studio 2008. (If you do not have Visual Studio 2008 available, see the OS user's manual (`http://www.coin-or.org/OS/doc/osUsersManual_2.0.pdf`) for information on how to download a free copy of Visual Studio.) Once inside Visual Studio, push F7 or select Build Solution from the Build menu. To keep things simple, and in order not to have to supply multiple versions of all the libraries, the solution file contains only a single configuration, Release. When the examples are successfully built, the executables will be stored into the folders that contain the project files. For example, `OSSolverDemoTest.exe` will be found in the folder **MSVisualStudioExamples\OSSolverDemoTest**.

The examples are described in more detail in Section 6. Eight of the examples illustrate various aspects of COIN-OR projects. The ninth project is a plug-and-play project. The user can use this project to build his or her own application based on the pre-compiled libraries. Obviously the code in the other projects can be used as a guide to using the OS API.

6 Example Projects

We provide eight examples that demonstrate how to use various aspects of the COIN-OR software. Many users will find the **OSSolverDemo** to be the most useful in that this describes how to write code to hook with the various solvers. See Section 6.5. There is also an empty example **Template** for users to put in their own code.

6.1 Algorithmic Differentiation: Using the OS Algorithmic Differentiation Methods

In the **OS\examples\algorithmicDiff** folder is the test code `OSAlgorithmicDiffTest.cpp`. This code illustrates the key methods in the `OSInstance` API that are used for algorithmic differentiation.

6.2 Instance Generator: Using the OSInstance API to Generate Instances

This example is found in the **instanceGenerator** folder in the **examples** folder. This example illustrates how to build a complete in-memory model instance using the `OSInstance` API. See the code `OSInstanceGenerator.cpp` for the complete example. Here we provide a few highlights to illustrate the power of the API.

The first step is to create an `OSInstance` object.

```
OSInstance *osinstance;
osinstance = new OSInstance();
```

The instance has two variables, x_0 and x_1. Variable x_0 is a continuous variable with lower bound of -100 and upper bound of 100. Variable x_1 is a binary variable. First declare the instance to have two variables.

```
osinstance->setVariableNumber( 2);
```

Next, add each variable. There is an `addVariable` method with the signature

```
addVariable(int index, string name, double lowerBound, double upperBound, char type);
```

Then the calls for these two variables are

```
osinstance->addVariable(0, "x0", -100, 100, 'C', OSNAN, "");
osinstance->addVariable(1, "x1", 0, 1, 'B', OSNAN, "");
```

There is also a method `setVariables` for adding more than one variable simultaneously. The objective function(s) and constraints are added through similar calls.

Nonlinear terms are also added in a straightforward if slightly cumbersome manner. The following code illustrates how to add a nonlinear term $x_0 * x_1$ in the `<nonlinearExpressions>` section of OSiL. This term is part of constraint 1 and is the second of six constraints contained in the instance.

First we set up storage for all six expressions, as follows.

```
osinstance->instanceData->nonlinearExpressions->numberOfNonlinearExpressions = 6;
osinstance->instanceData->nonlinearExpressions->nl = new Nl*[ 6 ];
```

The next code snippet shows how to initial the second of the six expressions.

```
osinstance->instanceData->nonlinearExpressions->nl[ 1] = new Nl();
osinstance->instanceData->nonlinearExpressions->nl[ 1]->idx = 1;
osinstance->instanceData->nonlinearExpressions->nl[ 1]->osExpressionTree =
new OSExpressionTree();
```

Next we build the expression in postfix notation, that is, in the form $(x_0, x_1, *)$.

```
// create a variable nl node for x0 an dput into temporary storage
nlNodeVariablePoint = new OSnLNodeVariable();
nlNodeVariablePoint->idx=0;
nlNodeVec.push_back( nlNodeVariablePoint);
// create the nl node for x1
nlNodeVariablePoint = new OSnLNodeVariable();
nlNodeVariablePoint->idx=1;
nlNodeVec.push_back( nlNodeVariablePoint);
// create the nl node for *
nlNodePoint = new OSnLNodeTimes();
nlNodeVec.push_back( nlNodePoint);
// now move the temporaray storage into the expression tree
osinstance->instanceData->nonlinearExpressions->nl[ 1]->osExpressionTree->m_treeRoot =
nlNodeVec[ 0]->createExpressionTreeFromPostfix( nlNodeVec);
```

6.3 branchCutPrice: Using Bcp

This example illustrates the use of the COIN-OR Bcp (Branch-cut-and-price) project. This project presents the user with the ability to have control over each node in the branch and process. This makes it possible to add user-defined cuts and/or user-defined variables. At each node in the tree, a call is made to the method `process_lp_result()`. In the example problem we illustrate 1) adding COIN-OR Cgl cuts, 2) a user-defined cut, and 3) a user-defined variable.

6.4 OSModDemo: Modifying an In-Memory `OSInstance` Object

The `osModDemo` folder holds the file `OSModDemo.cpp`. This is similar to the `instanceGenerator` example. In this case, a simple linear program is generated. However, this example also illustrates how to modify an in-memory `OSInstance` object. In particular, we illustrate how to modify an objective function coeffient. Note the line of code

```
solver->osinstance->bObjectivesModified = true;
```

which is critical, otherwise changes made to the `OSInstance` object will not be fed to the solver.

This example also illustrates calling a COIN-OR solver, in this case `Clp`.

Important: the ability to modify a problem instance is extremely limited in Release 2.0 of OS. A better API for problem modification will come with a later release of OS.

6.5 OSSolverDemo: Building In-Memory Solver and Option Objects

The code in the example file `OSSolverDemo.cpp` in the folder `osSolverDemo` illustrates how to build solver interfaces and an in-memory `OSOption` object. In this example we illustrate building a solver interface and corresponding `OSOption` object for the solvers Clp, Cbc, SYMPHONY, Ipopt, Bonmin, and Couenne. Each solver class inherits from a virtual `OSDefaultSolver` class. Each solver class has the string data members

- **osil** – this string conforms to the OSiL standard and holds the model instance.
- **osol** – this string conforms to the OSoL standard and holds an instance with the solver options (if there are any); this string can be empty.
- **osrl** – this string conforms to the OSrL standard and holds the solution instance; each solver interface produces an `osrl` string.

Corresponding to each string there is an in-memory object data member, namely

- **osinstance** – an in-memory `OSInstance` object containing the model instance and get() and set() methods to access various parts of the model.
- **osoption** – an in-memory `OSOption` object; solver options can be accessed or set using `get()` and `set()` methods.
- **osresult** – an in-memory `OSResult` object; various parts of the model solution are accessible through `get()` and `set()` methods.

For each solver we detail five steps:

Step 1: Read a model instance from a file and create the corresponding `OSInstance` object. For four of the solvers we read a file with the model instance in OSiL format. For the Clp example we read an MPS file and convert to OSiL. For the Couenne example we read an AMPL nl file and convert to OSiL.

Step 2: Create an `OSOption` object and set options appropriate for the given solver. This is done by defining

```
OSOption* osoption = NULL;
osoption = new OSOption();
```

A key method in the `OSOption` interface is `setAnotherSolverOption()`. This method takes the following arguments in order.

std::string name – the option name;

std::string value – the value of the option;

std::string solver – the name of the solver to which the option applies;

std::string category – options may fall into categories. For example, consider the Couenne solver. This solver is also linked to the Ipopt and Bonmin solvers and it is possible to set options for these solvers through the Couenne API. In order to set an Ipopt option you would set the `solver` argument to `"couenne"` and set the `category` argument to `"ipopt"`.

std::string type – many solvers require knowledge of the data type, so you can set the type to `double`, `integer`, `boolean` or `string`, depending on the solver requirements. Special types defined by the solver, such as the type `numeric` used by the Ipopt solver, can also be accommodated. It is the user's responsibility to verify the type expected by the solver.

std::string description – this argument is used to provide any detail or additional information about the option. An empty string (`""`) can be passed if such additional information is not needed.

For excellent documentation that details solver options for Bonmin, Cbc, and Ipopt we recommend

`http://www.coin-or.org/GAMSlinks/gamscoin.pdf`

Step 3: Create the solver object. In the OS project there is a *virtual* solver that is declared by

```
DefaultSolver *solver  = NULL;
```

The Cbc, Clp and SYMPHONY solvers as well as other solvers of linear and integer linear programs are all invoked by creating a `CoinSolver()`. For example, the following is used to invoke Cbc.

```
solver = new CoinSolver();
solver->sSolverName ="cbc";
```

Other solvers, particularly the nonlinear solvers Ipopt, Bonmin and Couenne are implemented separately. So to declare, for example, an Ipopt solver, one should write

```
solver = new IpoptSolver();
```

The syntax is the same regardless of solver.

Step 4: Import the `OSOption` and `OSInstance` into the solver and solve the model. This process is identical regardless of which solver is used. The syntax is:

```
solver->osinstance = osinstance;
solver->osoption = osoption;
solver->solve();
```

Step 5: After optimizing the instance, each of the OS solver interfaces uses the underlying solver API to get the solution result and write the result to a string named `osrl` which is a string representing the solution instance in the OSrL XML format. This string is accessed by

```
solver->osrl;
```

In the example code `OSSolverDemo.cpp` we have written a method,

```
void getOSResult(std::string osrl);
```

that takes the `osrl` string and creates an `OSResult` object. We then illustrate several of the `OSResult` API methods

```
double getOptimalObjValue(int objIdx, int solIdx);
std::vector<IndexValuePair*>  getOptimalPrimalVariableValues(int solIdx);
```

to get and write out the optimal objective function value, and optimal primal values. See also Section 6.6.

We now highlight some of the features illustrated by each of the solver examples.

- **Clp** – In this example we read in a problem instance in MPS format. The class `OSmps2osil` has a method `mps2osil` that is used to convert the MPS instance contained in a file into an in-memory `OSInstance` object. This example also illustrates how to set options using the Osi interface. In particular we turn on intermediate output which is turned off by default in the Coin Solver Interface.

- **Cbc** – In this example we read a problem instance that is in OSiL format and create an in-memory `OSInstance` object. We then create an `OSOption` object. This is quite trivial. A plain-text XML file conforming to the OSiL schema is read into a string `osil` which is then converted into the in-memory `OSInstance` object by

```
OSiLReader *osilreader = NULL;
OSInstance *osinstance = NULL;
osilreader = new OSiLReader();
osinstance = osilreader->readOSiL( osil);
```

We set the linear programming algorithm to be the primal simplex method and then set the option on the pivot selection to be stepest edge. Finally, we set the print level to be 10.

```
osoption->setAnotherSolverOption("primalS","","cbc","","string","");
osoption->setAnotherSolverOption("primalpivot","steepest","cbc","","string","");
osoption->setAnotherSolverOption("log","10","cbc","","integer","");
```

- **SYMPHONY** – In this example we also read a problem instance that is in OSiL format and create an in-memory `OSInstance` object. We then create an `OSOption` object and illustrate setting the `verbosity` option.

- **Ipopt** – In this example we also read a problem instance that is in OSiL format. However, in this case we do not create an `OSInstance` object. We read the OSiL file into a string `osil`. We then feed the `osil` string directly into the Ipopt solver by

  ```
  solver->osil = osil;
  ```

  The user always has the option of providing the OSiL to the solver as either a string or in-memory object.

  Next we create an `OSOption` object. For Ipopt, we illustrate setting the maximum iteration limit and also provide the name of the output file. In addition, the `OSOption` object can hold initial solution values. We illustrate how to initialize all of the variable to 1.0.

  ```
  numVar = 2; //rosenbrock mod has two variables
  xinitial = new double[numVar];
  for(i = 0; i < numVar; i++){
      xinitial[ i] = 1.0;
  }
  osoption->setInitVarValuesDense(numVar, xinitial);
  ```

- **Bonmin** – In this example we read a problem instance that is in OSiL format and create an in-memory `OSInstance` object just as was done in the Cbc and SYMPHONY examples. We then create an `OSOption` object. In setting the `OSOption` object we intentionally set an option that will cause the Bonmin solver to terminate early. In particular we set the `node_limit` to zero.

  ```
  osoption->setAnotherSolverOption("node_limit","0","bonmin","","integer","");
  ```

  This results in early termination of the algorithm. The `OSResult` class API has a method

  ```
  std::string getSolutionStatusDescription(int solIdx);
  ```

  For this example, invoking

  ```
  osresult->getSolutionStatusDescription( 0)
  ```

  gives the result:

  ```
  LIMIT_EXCEEDED[BONMIN]: A resource limit was exceeded, we provide the current solution.
  ```

- **Couenne** – In this example we read in a problem instance in AMPL nl format. The class `OSnl2osil` has a method `nl2osil` that is used to convert the nl instance contained in a file into an in-memory `OSInstance` object. This is done as follows:

  ```
  // convert to the OS native format
  OSnl2osil *nl2osil = NULL;
  nl2osil = new OSnl2osil( nlFileName);
  // create the first in-memory OSInstance
  nl2osil->createOSInstance() ;
  osinstance =  nl2osil->osinstance;
  ```

This part of the example also illustrates setting options in one solver from another. Couenne uses Bonmin which uses Ipopt. So for example,

```
osoption->setAnotherSolverOption("max_iter","100","couenne","ipopt","integer","");
```

identifies the solver as Couenne, but the category value of `"ipopt"` tells the solver interface to set the iteration limit on the Ipopt algorithm that is solving the continuous relaxation of the problem. Likewise, the setting

```
osoption->setAnotherSolverOption("num_resolve_at_node","3","couenne","bonmin","integer","");
```

identifies the solver as Couenne, but the category value of `"bonmin"` tells the solver interface to tell the Bonmin solver to try three starting points at each node.

6.6 OSResultDemo: Building In-Memory Result Object to Display Solver Result

The OS protocol for representing an optimization result is OSrL. Like the OSiL and OSoL protocol, this protocol has an associated in-memory `OSResult` class with corresponding API. The use of the API is demonstrated in the code `OSResultDemo.cpp` in the folder **OS\examples\OSResultDemo**. In the code we solve a linear program with the Clp solver. The OS solver interface builds an OSrL string that we read into the `OSrLReader` class and create and `OSResult` object. We then use the `OSResult` API to get the optimal primal and dual solution. We also use the API to get the reduced cost values.

6.7 OSCglCuts: Using the OSInstance API to Generate Cutting Planes

In this example, we show how to add cuts to tighten an LP using COIN-OR Cgl (Cut Generation Library).

6.8 OSRemoteTest: Calling a Remote Server

This example illustrates the API for the six service methods that implement the remote solver service. These methods are described in the user's manual (see `http://www.coin-or.org/OS/doc/osUsersManual_2.0.pdf`). The file `osRemoteTest.cpp` in folder **osRemoteTest** first builds a small linear example, solves it remotely in synchronous mode and displays the solution. The asynchronous mode is also tested by submitting the problem to a remote solver, checking the status and either retrieving the answer or killing the process if it has not yet finished.

6.9 Template

The code `template.cpp` is in the **template** directory. This is linked to all of the COIN-OR libraries in **lib** but is an empty example. The user can write his or her own code here and build an application based on the COIN-OR projects.