# Optimization Services 2.4 User's Manual

Horand Gassmann, Jun Ma, Kipp Martin, and Wayne Sheng

November 7, 2011

**Abstract**

This is the User's Manual for the Optimization Services (OS) project. The objective of OS is to provide a general framework consisting of a set of standards for representing optimization instances, results, solver options, and communication between clients and solvers in a distributed environment using Web Services. This COIN-OR project provides C++ and Java source code for libraries and executable programs that implement OS standards. The OS library includes a robust solver and modeling language interface (API) for linear, nonlinear and other types of optimization problems. Also included is the C++ source code for a command line executable `OSSolverService` for reading problem instances (OSiL format, nl format, MPS format) and calling a solver either locally or on a remote server. Finally, both Java source code and a Java `war` file are provided for users who wish to set up a solver service on a server running Apache Tomcat. See the Optimization Services home page `http://www.optimizationservices.org` and the COIN-OR Trac page `http://projects.coin-or.org/OS` for more information.

# Contents

# List of Figures



# List of Tables

# 1 The Optimization Services (OS) Project

The objective of Optimization Services (OS) is to provide a general framework consisting of a set of standards for representing optimization instances, results, solver options, and communication between clients and solvers in a distributed environment using Web Services. This COIN-OR project provides source code for libraries and executable programs that implement OS standards. See the COIN-OR Trac page `http://projects.coin-or.org/OS` or the Optimization Services Home Page `http://www.optimizationservices.org` for more information.

Like other COIN-OR projects, OS has a versioning system that ensures end users some degree of stability and a stable upgrade path as project development continues. The current stable version of OS is 2.4, and the current stable release is 2.4.0, based on trunk version 4340.

The OS project provides the following:

1. A set of XML based standards for representing optimization instances (OSiL), optimization results (OSrL), and optimization solver options (OSoL). There are other standards, but these are the main ones. The schemas for these standards are described in Section **??**.

2. Open source libraries that support and implement many of the standards.

3. A robust solver and modeling language interface (API) for linear and nonlinear optimization problems. Corresponding to the OSiL problem instance representation there is an in-memory object, `OSInstance`, along with a collection of `get()`, `set()`, and `calculate()` methods for accessing and creating problem instances. This is a very general API for linear, integer, and nonlinear programs. Extensions for other major types of optimization problems are also in the works. Any modeling language that can produce OSiL can easily communicate with any solver that uses the OSInstance API. The `OSInstance` object is described in more detail in Section 7. The nonlinear part of the API is based on the COIN-OR project CppAD by Brad Bell (`http://projects.coin-or.org/CppAD`) but is written in a very general manner and could be used with other algorithmic differentiation packages. More detail on algorithmic differentiation is provided in Section 8.

4. A command line executable `OSSolverService` for reading problem instances (OSiL format, AMPL nl format, MPS format) and calling a solver either locally or on a remote server. This is described in Section **??**.

5. Utilities that convert AMPL nl files and MPS files into the OSiL XML format. This is described in Section 6.3.

6. Standards that facilitate the communication between clients and optimization solvers using Web Services. In Section 6.1 we describe the `OSAgent` part of the OS library that is used to create Web Services SOAP packages with OSiL instances and contact a server for solution.

7. An executable program `OSAmplClient` that is designed to work with the AMPL modeling language. The `OSAmplClient` appears as a "solver" to AMPL and, based on options given in AMPL, contacts solvers either remotely or locally to solve instances created in AMPL. This is described in Section **??**.

8. Server software that works with Apache Tomcat and Apache Axis. This software uses Web Services technology and acts as middleware between the client that creates the instance and the solver on the server that optimizes the instance and returns the result. This is illustrated in Section **??**.

9. A lightweight version of the project, `OSCommon`, for modeling language and solver developers who want to use OS API, readers and writers, without the overhead of other COIN-OR projects or any third-party software. For information on how to download `OSCommon` see Section **??**.

# 2 Quick Roadmap

If you want to:

- Download the OS source code or binaries – see Section **??**.
- Download just the OS API, readers and writers – see Section **??**.
- Build the OS project from the source code – see Section 5.1.
- Use the OS library to build model instances or use solver APIs – see Sections 6.3, 6.5 and 7.
- Use the OSSolverService to read files in nl, OSiL, or MPS format and call a solver locally or remotely – see Section **??**.
- Use AMPL to solve problems either locally or remotely with a COIN-OR solver, Cplex, GLPK, or LINDO – see Section **??**.
- Use GAMS to solve problems either locally or remotely – see Section **??**.
- Build a remote solver service using Apache Tomcat – see Section **??**.
- Use MATLAB to generate problem instances in OSiL format and call a solver either remotely or locally – see Section **??**.
- Use the OS library for algorithmic differentiation (in conjunction with COIN-OR CppAD) – see Section 8.
- Use modeling languages to generate model instances in OSiL format – see Section **??**.

# 3 Downloading the OS Binaries

The OS project is an open-source project with source code under the Common Public License (CPL). See `http://www.ibm.com/developerworks/library/os-cpl.html`. This project was initially created by Robert Fourer, Jun Ma, and Kipp Martin. The code has been written primarily by Horand Gassmann, Jun Ma, and Kipp Martin. Horand Gassmann, Jun Ma, and Kipp Martin are the COIN-OR project leaders and active developers for the OS project. Most users will only be interested in obtaining the binaries, which we describe next. It is also possible to obtain the source code for the project, which will be of interest mostly to developers. If binaries are not provided for a particular operating system, it may be possible to build them from the source. For details it is best to start reading the OS web page at `http://projects.coin-or.org/OS/`.

### 3.1 Obtaining the Binaries

If the user does not wish to compile source code, the OS library, OSSolverService executable and Tomcat server software configuration are available in binary format for some operating systems. The repository is at `http://www.coin-or.org/download/binary/OS/`. Unlike the source code described in Section **??**, the binary files are not subject to version control and can be downloaded using an ordinary browser. If binaries are not provided for a particular operating system, it may be possible to build them from the source code. Since the source is under version control, this requires svn. (See Sections **??**, **??** and 5.1.

The binary distribution for the OS library and executables follows the following naming convention:

`OS-version_number-platform-compiler-build_options.tgz (zip)`

For example, OS Release 2.1.0 compiled with the Intel 9.1 compiler on an Intel 32-bit Linux system is:

`OS-2.1.0-linux-x86-icc9.1.tgz`

For more detail on the naming convention and examples see:

`https://projects.coin-or.org/CoinBinary/wiki/ArchiveNamingConventions`

After unpacking the `tgz` or `zip` archives, the following folders are available.

**bin** – this directory has the executables `OSSolverService` and `OSAmplClient`.

**include** – the header files that are necessary in order to link against the OS library.

**lib** – the libraries that are necessary for creating applications that use the OS library.

**share** – license and author information for all the projects used by the OS project.

Files are also provided for an Apache Tomcat Web server along with the associated Web service that can read SOAP envelopes with model instances in OSiL format and/or options in OSoL format, call the `OSSolverService`, and return the optimization result in OSrL format. The naming convention for the server binary is

`OS-server-version_number.tgz (.zip)`

For example, the files associated with OS server release 2.0.0 are in the binary distribution

`OS-server-2.0.0.tgz`

There is no platform information given since the server and related binaries were written in Java. The details and use of this distribution are described in Section **??**.

Finally for Windows users we provide Visual Studio project files (and supporting libraries and header files) for building projects based on the OS library and libraries used by the OS project. The binary for this is named

`OS-version_number-VisualStudio.zip`

For example, the necessary files associated with OS stable 2.4 are in the binary distribution

`OS-2.1-VisualStudio.zip`

The binaries provided are based on Visual Studio Express 2008. See Section **??** for more detail.

# 4 Code samples to illustrate the OS Project

The binary distribution contains makefiles for unix users, respectively MS Visual Studio project files for Windows users that can be used as follows.

Under unix, connect to the appropriate directory for the desired project and run `make`. For instance, the code and makefile for the `osModDemo` example of section 4.4 is in the directory

```
examples/osModDemo
```

Under Windows, connect to the `MSVisualStudio` directory and open `examples.sln` in Visual Studio.

The `Makefile` in each example directory is fairly simple and is designed to be easily modified by the user if necessary. The part of the Makefile to be adjusted, if necessary, is

```
##########################################################################
#    You can modify this example makefile to fit for your own program.   #
#    Usually, you only need to change the five CHANGEME entries below.   #
##########################################################################

# CHANGEME: This should be the name of your executable
EXE = OSModDemo
# CHANGEME: Here is the name of all object files corresponding to the source
#           code that you wrote in order to define the problem statement
OBJS =  OSModDemo.o
# CHANGEME: Additional libraries
ADDLIBS =
# CHANGEME: Additional flags for compilation (e.g., include flags)
ADDINCFLAGS =  -I${prefix}/include
# CHANGEME: SRCDIR is the path to the source code; VPATH is the path to
# the executable. It is assumed that the lib directory is in prefix/lib
# and the header files are in prefix/include
SRCDIR = /Users/kmartin/Documents/files/code/cpp/OScpp/COIN-OS/OS/examples/osModDemo
VPATH = /Users/kmartin/Documents/files/code/cpp/OScpp/COIN-OS/OS/examples/osModDemo
prefix = /Users/kmartin/Documents/files/code/cpp/OScpp/vpath
```

Developers can use the Makefiles as a starting point for building applications that use the OS project libraries.

## 4.1 Algorithmic Differentiation: Using the OS Algorithmic Differentiation Methods

In the `OS/examples/algorithmicDiff` folder is test code `OSAlgorithmicDiffTest.cpp`. This code illustrates the key methods in the `OSInstance` API that are used for algorithmic differentiation. These methods are described in Section 8.

## 4.2 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');
osinstance->addVariable(1, "x1", 0, 1, 'B');
```

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 easily added. 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.

```
osinstance->instanceData->nonlinearExpressions->numberOfNonlinearExpressions = 6;
osinstance->instanceData->nonlinearExpressions->nl = new Nl*[ 6 ];
osinstance->instanceData->nonlinearExpressions->nl[ 1] = new Nl();
osinstance->instanceData->nonlinearExpressions->nl[ 1]->idx = 1;
osinstance->instanceData->nonlinearExpressions->nl[ 1]->osExpressionTree =
new OSExpressionTree();
// the nonlinear expression is stored as a vector of nodes in postfix format
// create a variable nl node for x0
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 the expression tree
osinstance->instanceData->nonlinearExpressions->nl[ 1]->osExpressionTree->m_treeRoot =
nlNodeVec[ 0]->createExpressionTreeFromPostfix( nlNodeVec);
```

### 4.3 branchCutPrice: Using Bcp

This example illustrates the use of the COIN-OR Bcp (Branch-cut-and-price) project. This project offers 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.

### 4.4 OSModificationDemo: Modifying an In-Memory `OSInstance` Object

The `osModificationDemo` folder holds the file `OSModificationDemo.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 dual occurrence of the following code

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

in the `OSModificationDemo.cpp` file (lines 177 and 187). This line is critical, since otherwise changes made to the OSInstance object will not be passed 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 still extremely limited in this release. A better API for problem modification will come with a later release of OS.

### 4.5 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` option 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 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 standard. 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 4.6.

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 Dantzig rule. Finally, we set the print level to be 10.

- **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 of 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 of value of `bonmin` tells the solver interface to tell the Bonmin solver to try three starting points at each node.

### 4.6 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.

### 4.7 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). A file (`p0033.osil`) in OSiL format is used to create an OSInstance object. The linear programming relaxation is solved. Then, Gomory, simple rounding, and knapsack cuts are added using `Cgl`. The model is then optimized using `Cbc`.

### 4.8 OSRemoteTest: Calling a Remote Server

This example illustrates the API for the six service methods described in Section ??. 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.

**Windows users should note** that this project links to `wsock32.lib`, which is not part of the Visual Studio Express Package. It is necessary to also download and install the Windows Platform SDK, which can be found at

`http://www.microsoft.com/downloads/details.aspx?FamilyID=E6E1C3DF-A74F-4207-8586-711EBE331CDC&displaylang=en`. See also Section ??.

### 4.9 OSJavaInstanceDemo: Building an OSiL Instance in Java

In this example we demonstrate how to build an OSiL instance using the Java OSInstance API. The example code also illustrates calling the `OSSolverService` executable from Java. In order to use this example, the user should do an svn checkout:

```
svn co https://projects.coin-or.org/svn/OS/branches/OSjava OSjava
```

The `OSjava` folder contains the file `INSTALL.txt`. Please follow the instructions in `INSTALL.txt` under the heading:

```
== Install Without a Web Server==
```

These instructions assume that the user has installed the Eclipse IDE. See `http://www.eclipse.org/downloads/`. At this link we recommend that the user get `Eclipse Classic`. In addition, the user should also have a copy of the `OSSolverService` executable that is compatible with his or her platform. The `OSSolverService` executable for several different platforms is available at `http://www.coin-or.org/download/binary/OS/OSSolverService/`. The user can also build the executable as described in this Manual. See Section 5.1. The code base for this example is in the folder:

```
OSjava/OSJavaExamples/src/OSJavaInstanceDemo.java
```

The code in the file `OSJavaInstanceDemo.java` demonstrates how the Java OSInstance API that is in `OSCommon` can be used to generate a linear program and then call the C++ `OSSolverService` executable to solve the problem. Running this example in Eclipse will generate in the folder

```
OSjava/OSJavaExamples
```

two files. It will generate `parincLinear.osil` which is a linear program in the OS OSiL format, it will also call the `OSSolverService` executable which generates the result file `result.osrl` in the OS OSrL format.

# 5 Using Dip (Decomposition In Integer Programming)

**Important Note:** This example uses COIN-OR projects that are not part of the OS distribution and assumes you have downloaded the `CoinAll` binary.

We follow the notation of Galati and Ralphs [**?**]. The integer program of interest is:

$$z_{IP} \quad = \quad \min{-c^{\top}x \,|\, A'x \geq b',\ A''x \geq b'',\ x \in \mathbb{Z}^{n}}'' \tag{1}$$

The problem is divided into two constraint sets, $A'x \geq b'$ which we refer to as the *relaxed*, *coupling*, or *block constraints*, and the *core constraints* $A''x \geq b''$. We then define the following polyhedron based on the relaxed constraints.

$$\mathcal{P} \quad = \quad \mathrm{conv}({-x \in \mathbb{Z}^{n} \,|\, A'x \geq b'}'') \tag{2}$$

The LP relaxation of the original problem is:

$$z_{LP} \quad = \quad \min{-c^{\top}x \,|\, A'x \geq b',\ A''x \geq b'',\ x \in \mathbb{R}^{n}}'' \tag{3}$$

We also make use of another, related problem $z_D$, defined by

$$z_D \quad = \quad \min{-c^{\top}x \,|\, A'x \geq b',\ x \in \mathcal{P},\ x \in \mathbb{R}^{n}}''. \tag{4}$$

Ideally, the constraints $A'x \geq b'$ should be selected so that solving $Z_D$ is an easy *hard problem* and provides better bounds than $Z_{LP}$.

A generic block-angular decomposition algorithm is now available. We employ an implementation that uses the Optimization Services (OS) project together with another COIN-OR project, Decomposition in Integer Programming (Dip). We call this the OS Dip solver. It has the following features:

1. All subproblems are solved via an oracle; either the default oracle contained in our distribution (see below) or one provided by the user.

2. The OS Dip Solver code is independent of the oracle used to optimize the subproblems.

3. Variables are assigned to blocks using an OS option file; the block definition and assignment of variables to these blocks has no effect on the OS Dip Solver code.

4. Different blocks can be assigned different solver oracles based on the option values given in the OSoL file.

5. There is a default oracle implemented (called OSDipBlockCoinSolver) that currently uses Cbc.

6. Users can add their own oracles without altering the OS Dip Solver code. This is done via polymorphic factories. The user creates a separate file containing the oracle class. The user-provided Oracle class inherits from the generic OSDipBlockSolver class. The user need only: 1) add the object file name for the new oracle to the Makefile, and 2) add the necessary line to OSDipFactoryInitializer.h indicating that the new oracle is present.

In particular, the implementation of the OS Dip solver provides a virtual class `OSDipBlockSolver` with a pure virtual function `solve()`. The user is expected to provide a class that inherits from `OSDipBlockSolver` and implements the method `solve()`. The `solve()` method should optimize a linear objective function over $\mathcal{P}$. More details are provided in Section 5.2. The implementation is such that the user only has to provide a class with a solve method. The user does not have to edit or alter any of the OS Dip Solver code. By using polymorphic factories the actual solver details are hidden from the OS Solver. A default solver, `OSDipBlockCoinSolver`, is provided. This default solver takes no advantage of special structure and simply calls the COIN-OR solver `Cbc`.

## 5.1 Building and Testing the OS-Dip Example

Currently, the Decomposition in Integer Programming (**Dip**) package is not a dependency of the Optimization Services (**OS**) package – **Dip** is not included in the **OS** Externals file. In order to run the OS Dip solver it is necessary to download both the **OS** and **Dip** projects. Download order is irrelevant. In the discussion that follows we assume that for both **OS** and **Dip** the user has successfully completed a `configure`, `make`, and `make install`. We also assume that the user is working with the trunk version of both **OS** and **Dip.**

The OS Dip solver C++ code is contained in `TemplateApplication/osDip`. The `configure` will create a `Makefile` in the `TemplateApplication/osDip` folder. The `Makefile` must be edited to reflect the location of the **Dip** project. The `Makefile` contains the line

```
DIPPATH = /Users/kmartin/coin/dip-trunk/vpath-debug/
```

This setting assumes that there is a **lib** directory:

```
/Users/kmartin/coin/dip-trunk/vpath-debug/lib
```

with the **Dip** library that results from `make install` and an `include` directory

```
/Users/kmartin/coin/dip-trunk/vpath/include
```

with the **Dip** header files generated by `make install`. The user should adjust

```
/Users/kmartin/coin/dip-trunk/vpath/
```

to a path containing the **Dip** `lib` and `include` directories. After building the executable by executing the `make` command, run the `osdip` application using the command:

```
./osdip --param osdip.parm
```

This should produce the following output.

```
FINISH SOLVE
Status= 0 BestLB= 16.00000   BestUB= 16.00000   Nodes= 1
SetupCPU= 0.01 SolveCPU= 0.10 TotalCPU= 0.11 SetupReal= 0.08
SetupReal= 0.12 TotalReal= 0.16
Optimal Solution
-------------------------
Quality = 16.00
0       1.00
1       1.00
```

```
12      1.00
13      1.00
14      1.00
15      1.00
17      1.00
```

If you see this output, things are working properly.

The file `osdip.parm` is a parameter file. The use of the parameter file is explained in Section 5.7.

## 5.2 The OS Dip Solver – Code Description and Key Classes

The OS Dip Solver uses **Dip** to implement a Dantzig-Wofe decomposition algorithm for block-angular integer programs. Here are some key classes.

**OSDipBlockSolver:** This is a virtual class with a pure virtual function:

```
void solve(double *cost, std::vector<IndexValuePair*> *solIndexValPair,
double *optVal)
```

**OSDipBlockSolverFactory:** This is also virtual class with a pure virtual function:

```
OSDipBlockSolver* create()
```

This class also has the static method

```
OSDipBlockSolver* createOSDipBlockSolver(const string &solverName)
```

and a map

```
std::map<std::string, OSDipBlockSolverFactory*> factories;
```

**Factory:** This class inherits from the class **OSDipBlockSolverFactory**. Every sover class that inherits from the **OSDipBlockSolver** class should have a **Factory** class member and since this **Factory** class member inherits from the **OSDipBlockSolverFactory** class it should implement a `create()` method that creates an object in the class inheriting from **OSDipBlockSolver**.

**OSDipFactoryInitializer:** This class initializes the static map

```
OSDipBlockSolverFactory::factories
```

in the **OSDipBlockSolverFactory** class.

**OSDipApp:** This class inherits from the **Dip** class `DecompApp`. In **OSDipApp** we implement methods for creating the core (coupling) constraints, i.e., the constraints $A''x \geq b''$. This is done by implementing the `createModels()` method. Regardless of the problem, none of the relaxed or block constraints in $A'x \geq b'$ are created. These are treated implicitly in the solver class that inherits from the class **OSDipBlockSolver.** This class also implements a method that defines the variables that appear only in the blocks (**createModelMasterOnlys2**), and a method for generating an initial master (the method **generateInitVars()** ).

Since the constraints $A'x \geq b'$ are treated explicitly by the Dip solver the `solveRelaxed()` method must be implemented. In our implementation we have the **OSDipApp** class data member

```
std::vector<OSDipBlockSolver* > m_osDipBlockSolver;
```

when the `solveRelaxed()` method is called for block `whichBlock` in turn we make the call

```
m_osDipBlockSolver[whichBlock]->solve(cost, &solIndexValPair, &varRedCost);
```

and the appropriate solver in class **OSDipBlockSolver** is called. Finally, the **OSDipApp** class also initiates the reading of the OS option and instance files. How these files are used is discussed in Section 5.6. Based on option input data this class also creates the appropriate solver object for each block, i.e., it populates the `m_osDipBlockSolver` vector.

**OSDipInterface:** This class is used as an interface between the **OSDipApp** class and classes in the **OS** library. This provides a number of get methods to provide information to **OSDipApp** such as the coefficients in the $A''$ matrix, objective function coefficients, number of blocks etc. The **OSDipInterface** class reads the input OSiL and OSoL files and creates in-memory data structures based on these files.

**OSDipBlockCoinSolver:** This class inherits from the **OSDipBlockSolver** class. It is meant to illustrate how to create a solver class. This class solves each block by calling **Cbc**. Use of this class provides a generic block angular decomposition algorithm.

There is also **OSDip˙Main.cpp:** which contains the `main()` routine and is the entry point for the executable. It first creates a new price-branch-and-cut decomposition algorithm and then an `Alps` solver for which the `solve()` method is called.

## 5.3 User Requirements

The **OSDipBlockCoinSolver** class provides a solve method for optimizing a linear objective function over $\mathcal{P}$ given a linear objective function. However, this takes no advantage of the special structure available in the blocks. Therefore, the user may wish to implement his or her own solver class. In this case the user is required to do the following:

1. implement a class that inherits from the **OSDipBlockSolver** class and implements the solve method,
2. implement a class **Factory** that inherits from the class **OSDipBlockSolverFactory** and implements the `create()` method,
3. edit the file **OSDipFactoryInitializer.h** and add a line:

   ```
   OSDipBlockSolverFactory::factories["MyBlockSolver"] = new
   MyBlockSolver::Factory;
   ```

4. alter the Makefile to include the new source code.

**Important – Directory Structure:** In order to keep things clean, there is a directory **solvers** in the **osDip** folder. We suggest using the **solvers** directory for all of the solvers that inherit from **OSDipBlockSolver**.

## 5.4 Simple Plant/Lockbox Location Example

The problem is to minimize the sum of the cost of capital due to float and the cost of operating the lock boxes.

**Parameters:**

$m-$ number of customers to be assigned a lock box

$n-$ number of potential lock box sites

$c_{ij}-$ annual cost of capital associated with serving customer $j$ from lock box $i$

$f_i-$ annual fixed cost of operating a lock box at location $i$

**Variables:**

$x_{ij}-$ a binary variable which is equal to 1 if customer $j$ is assigned to lock box $i$ and 0 if not

$y_i-$ a binary variable which is equal to 1 if the lock box at location $i$ is opened and 0 if not

The integer linear program for the lock box location problem is

$$\min \sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij}x_{ij} + \sum_{i=1}^{n} f_i y_i \tag{5}$$

$$(LB) \qquad x_{ij} - y_i \leq 0, \qquad i = 1, \ldots, n,\ j = 1, \ldots, m \tag{6}$$

$$\text{s.t.} \qquad \sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1, \ldots, m \tag{7}$$

$$x_{ij},\ y_i \in \text{–}0, 1'', \quad i = 1, \ldots, n,\ j = 1, \ldots, m. \tag{8}$$

The objective (5) is to minimize the sum of the cost of capital plus the fixed cost of operating the lock boxes. Constraints (6) are forcing constraints and require that a lock box be open if a customer is served by that lock box. For now, we consider these the $A'x \geq b'$ constraints. The requirement that every customer be assigned a lock box is modeled by constraints (7). For now, we consider these the $A''x \geq b''$ constraints.

**Location Example 1:** A three plant, five customer model.

| | | CUSTOMER | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | FIXED COSTS |
| | 1 | 2 | 3 | 4 | 5 | 7 | 2 |
| PLANT | 2 | 4 | 3 | 1 | 2 | 6 | 3 |
| | 3 | 5 | 4 | 2 | 1 | 3 | 3 |

Table 1: Data for a 3 plant, 5 customer problem

$$\begin{aligned} \min \quad & 2x_{11} + 3x_{12} + 4x_{13} + 5x_{14} + 7x_{15} + 2y_1 + \\ & 4x_{21} + 3x_{22} + x_{23} + 2x_{24} + 6x_{25} + 3y_2 + \\ & 5x_{31} + 4x_{32} + 2x_{33} + x_{34} + 3x_{35} + 3y_3 \end{aligned}$$

$$
\begin{aligned}
&x_{11} \le y_1 \le 1 \\
&x_{12} \le y_1 \le 1 \\
&x_{13} \le y_1 \le 1 \\
&x_{14} \le y_1 \le 1 \\
&x_{15} \le y_1 \le 1 \\
&x_{21} \le y_2 \le 1 \\
&x_{22} \le y_2 \le 1 \\
&x_{23} \le y_2 \le 1 \qquad A'x \ge b' \text{ constraints} \\
&x_{24} \le y_2 \le 1 \\
&x_{25} \le y_2 \le 1 \\
&x_{31} \le y_3 \le 1 \\
&x_{32} \le y_3 \le 1 \\
&x_{33} \le y_3 \le 1 \\
&x_{33} \le y_3 \le 1 \\
&x_{33} \le y_3 \le 1 \\
&x_{ij}, y_i \ge 0,\ i = 1, \ldots, n,\ j = 1, \ldots, m.
\end{aligned}
$$

$$
\begin{aligned}
\text{s.t.}\quad &x_{11} + x_{21} + x_{31} = 1 \\
&x_{12} + x_{22} + x_{32} = 1 \\
&x_{13} + x_{23} + x_{33} = 1 \qquad A''x \ge b'' \text{ constraints} \\
&x_{14} + x_{24} + x_{34} = 1 \\
&x_{15} + x_{25} + x_{35} = 1
\end{aligned}
$$

**Location Example 2 (SPL2):** A three plant, three customer model.

| | | CUSTOMER | | | |
|---|---|---|---|---|---|
| | | 1 | 2 | 3 | FIXED COSTS |
| | 1 | 2 | 1 | 1 | 1 |
| PLANT | 2 | 1 | 2 | 1 | 1 |
| | 3 | 1 | 1 | 2 | 1 |

Table 2: Data for a three plant, three customer problem

$$
\begin{aligned}
\min \quad &2x_{11} + x_{12} + x_{13} + y_1 + \\
&x_{21} + 2x_{22} + x_{23} + y_2 + \\
&x_{31} + x_{32} + 1x_{33} + +y_3
\end{aligned}
$$

$$
\begin{aligned}
& x_{11} \le y_1 \le 1 \\
& x_{12} \le y_1 \le 1 \\
& x_{13} \le y_1 \le 1 \\
& x_{21} \le y_2 \le 1 \\
& x_{22} \le y_2 \le 1 \qquad A'x \ge b' \text{ constraints} \\
& x_{23} \le y_2 \le 1 \\
& x_{31} \le y_3 \le 1 \\
& x_{32} \le y_3 \le 1 \\
& x_{33} \le y_3 \le 1 \\
& x_{ij}, y_i \ge 0,\ i = 1, \ldots, n,\ j = 1, \ldots, m.
\end{aligned}
$$

$$
\begin{aligned}
\text{s.t.} \quad & x_{11} + x_{21} + x_{31} = 1 \\
& x_{12} + x_{22} + x_{32} = 1 \qquad A''x \ge b'' \text{ constraints} \\
& x_{13} + x_{23} + x_{33} = 1
\end{aligned}
$$

### 5.5 Generalized Assignment Problem Example

A problem that plays a prominent role in vehicle routing is the *generalized assignment problem.* The problem is to assign each of $n$ tasks to $m$ servers without exceeding the resource capacity of the servers.

**Parameters:**

$n-$ number of required tasks

$m-$ number of servers

$f_{ij}-$ cost of assigning task $i$ to server $j$

$b_j-$ units of resource available to server $j$

$a_{ij}-$ units of server $j$ resource required to perform task $i$

**Variables:**

$x_{ij}-$ a binary variable which is equal to 1 if task $i$ is assigned to server $j$ and 0 if not

The integer linear program for the generalized assignment problem is

$$
\begin{aligned}
& \min \quad \sum_{i=1}^{n} \sum_{j=1}^{m} f_{ij} x_{ij} && (9) \\
(GAP) \qquad & \text{s.t.} \quad \sum_{j=1}^{m} x_{ij} = 1, \qquad i = 1, \ldots, n && (10) \\
& \sum_{i=1}^{n} a_{ij} x_{ij} \le b_j, \qquad j = 1, \ldots, m && (11) \\
& x_{ij} \in -0, 1'',\ i = 1, \ldots, n,\ j = 1, \ldots, m. && (12)
\end{aligned}
$$

The objective function (9) is to minimize the total assignment cost. Constraint (10) requires that each task is assigned a server. These constraints correspond to the $A''x \ge b''$ constraints. The requirement that the server capacity not be exceeded is given in (11). These corresspond to the $A'x \geq b'$ constraints that are used to define $\mathcal{P}$. The test problem used in the file `genAssign.osil` is:

$$
\begin{aligned}
\min \quad & 2x_{11} + 11x_{12} + 7x_{21} + 7x_{22} \\
& +20x_{31} + 2x_{32} + 5x_{41} + 5x_{42} \\
x_{11} + x_{12} &= 1 \\
x_{21} + x_{22} &= 1 \\
x_{31} + x_{32} &= 1 \\
x_{41} + x_{42} &= 1 \\
3x_{11} + 6x_{21} + 5x_{31} + 7x_{41} &\leq 13 \\
2x_{12} + 4x_{22} + 10x_{32} + 4x_{42} &\leq 10
\end{aligned}
$$

### 5.6 Defining the Problem Instance and Blocks

Here we describe how to use the OSOption and OSInstance formats. We illustrate with a simple plant location problem. Refer back to the example in Table 1 for a three-plant, five-customer problem. We treat the fixed charge constraints as the block constraints, i.e., we treat constraint set (6) as the set $A'x \geq b'$ constraints. These constraints naturally break into a block for each plant, i.e., there is a block of constraints:

$$x_{ij} \leq y_i \tag{13}$$

In order to use the OS Dip solver it is necessary to: 1) define the set of variables in each block and 2) define the set of constraints that constitute the core or coupling constraints. This information is communicated to the OS Dip solver using Optimization Services option Language (OSoL). The OSoL input file for the example in Table 1 appears in Figures 1 and 2. See lines 32-55. There is an `<other>` option with `name="variableBlockSet"` for each block. Each block then lists the variables in the block. For example, the first block consists of the variables indexed by 0, 1, 2, 3, 4, and 15. These correspond to variables $x_{11}$, $x_{12}$, $x_{13}$, $x_{13}$, $x_{14}$, and $y_1$. Likewise the second block corresponds to the variable for the second plant and the third block corresponds to variables for the third plant.

It is also necessary to convey which constraints constitute the core constraints. This is done in lines 58-64. The core constraints are indexed by 15, 16, 17, 18, 19. These constitute the demand constraints given in Equation (7).

Notice also that in lines 32, 40, and 48 there is an attribute `value` in the `<other>` variable element with the attribute `name` equal to `variableBlockSet`. The attribute `value` should be the name of the solver factory that should be assigned to solve that block. For example, if the optimization problem that results from solving a linear objective over the constraints defining the first block is solved using `MySolver1` then this must correspond to a

```
OSDipBlockSolverFactory::factories["MySolver1"] = new
MySolver1::Factory;
```

in the file **OSDipFactoryInitializer.h**. In the test file, `spl1.osol` for the first block we set the solver to a specialized solver for the simple plant location problem (`OSDipBlockSplSolver`) and for the other two blocks we use the generic solver (`OSDipBlockCoinSolver`).

```
<?xml version="1.0" encoding="UTF-8"?>
<osol>
   <general>
      <instanceName>spl1 -- setup constraints are the blocks</instanceName>
   </general>
   <optimization>
      <variables numberOfOtherVariableOptions="6">
         <other name="initialCol" solver="Dip" numberOfVar="6" value="0">
            <var idx="0" value="1"/>
            <var idx="1" value="1"/>
            <var idx="2" value="1"/>
            <var idx="3" value="1"/>
            <var idx="4" value="1"/>
            <var idx="15" value="1"/>
         </other>
         <other name="initialCol" solver="Dip" numberOfVar="6" value="1">
            <var idx="5" value="1"/>
            <var idx="6" value="1"/>
            <var idx="7" value="1"/>
            <var idx="8" value="1"/>
            <var idx="9" value="1"/>
            <var idx="16" value="1"/>
         </other>
         <other name="initialCol" solver="Dip" numberOfVar="6" value="2">
            <var idx="10" value="1"/>
            <var idx="11" value="1"/>
            <var idx="12" value="1"/>
            <var idx="13" value="1"/>
            <var idx="14" value="1"/>
            <var idx="17" value="1"/>
         </other>
         <other name="variableBlockSet" solver="Dip" numberOfVar="6" value="MySolver1">
            <var idx="0"/>
            <var idx="1"/>
            <var idx="2"/>
            <var idx="3"/>
            <var idx="4"/>
            <var idx="15"/>
         </other>
         <other name="variableBlockSet" solver="Dip" numberOfVar="6" value="MySolver2">
            <var idx="5"/>
            <var idx="6"/>
            <var idx="7"/>
            <var idx="8"/>
            <var idx="9"/>
            <var idx="16"/>
         </other>
```

Figure 1: A sample OSoL file – SPL1.osol

One can use the OSoL file to specify a set of starting columns for the initial restricted master. In Figure 1 see lines 8-31. In and OS option file (OSoL) there is `<variables>` element that has `<other>` children. Initial columns are specified using the `<other>` elements. This is done by using the `name` attribute and setting its value to `initialCol`. Then the children of the tag contain index-value pairs that specify the column. For example, the first initial column corresponds to setting:

$$x_{11} = 1, \quad x_{12} = 1, \quad x_{13} = 1, \quad x_{14} = 1, \quad x_{15} = 1, \quad y_1 = 1$$

Finally note that in all of this discussion we know to apply the options to **Dip** because the attribute `solver` always had value `Dip`. It is critical to set this attribute in all of the option tags.

## 5.7 The Dip Parameter File

The **Dip** solver has a utility class **UtilParameters**, for parsing a parameter file. The **UtilParameters** class constructor takes a parameter file as an argument. In the case of the OS Dip solver the name of the parameter file is **osdip.parm** and the parameter file is read in at the command line with the command

```
./osdip -param osdip.parm
```

The **UtilParameters** class has a method **GetSetting()** for reading the parameter values. In the OS Dip implementation there is a class **OSDipParam** that has as data members key parameters such as the name of the input OSiL file and input OSoL file. The **OSDipParam** class has a method **getSettings()** that takes as an argument a pointer to an object in the **UtilParameters** and uses the **GetSetting()** method to return the relevant parameter values. For example:

```
OSiLFile = utilParam.GetSetting("OSiLFile", "", common);
OSoLFile = utilParam.GetSetting("OSoLFile", "", common);
```

In the current **osdip.parm** file we have:

```
#first simple plant location problem
OSiLFile = spl1.osil
#setup constraints as blocks
OSoLFile = spl1.osol
#assignment constraints as blocks
#OSoLFile = spl1-b.osol

#second simple plant location problem
#OSiLFile = spl2.osil
#setup constraints as blocks
#OSoLFile = spl2.osol
#assignment constraints as blocks
#OSoLFile = spl2-b.osol

#third simple plant location problem -- block matrix data not used
#OSiLFile = spl3.osil
#setup constraints as blocks
```

```
#OSoLFile = spl3.osol

#generalized assignment problem
#OSiLFile = genAssign.osil
#OSoLFile = genAssign.osol

#Martin textbook example
#OSiLFile = smallIPBook.osil
#OSoLFile = smallIPBook.osol
```

By commenting and uncommenting you can run one of four problems that are in the **data** directory. The first example, **spl1.osil**, corresponds to the simple plant location model given in Table 1. Using the option file **spl1.osol** treats the setup forcing constraints 6 as the $A'x \geq b'$ constraints. Using the option file **spl1-b.osol** treats the demand constraints 7 as the $A'x \geq b'$ constraints. Likewise for the problem **spl2.osil** which correponds to the simple plant location data given in Table 2.

In both examples **spl1.osil** and **spl2.osil** the $A'x \geq b'$ constraints are explicitly represented in the OSiL file. However, this is not necessary. The solver Factory **OSDipBlockSlpSolver** is a special oracle that only needs the objective function coefficients and pegs variables based on the sign of the objective function coefficients. The **spl3.osil** is the example given in Table 1 but without the setup forcing constraints. Each block uses the **OSDipBlockSlpSolver** oracle.

The **genAssign.osil** file corresponds to the generalized assignment problem given in Section 5.5. The option file **genAssign.osol** treats the capacity constraints 11 as the $A'x \geq b'$ constraints.

The last problem defined in the file **smallIPBook.osil** is based on Example 16.3 on page 567 in *Large Scale Linear and Integer Optimization*. The option file treats the constraints

$$4x_1 + 9x_2 \leq 18, \quad -2x_1 + 4x_2 \leq 4$$

as the $A'x \geq b'$ constraints.

The user should also be aware of the parameter `solverFactory`. This parameter is the name of the default solver Factory. If a solver is not named for a block in the OSoL file this value is used. We have set the value of this string to be `OSDipBlockCoinSolver`.

### 5.8 Issues to Fix

- Enhance solveRelaxed to allow parallel processing of blocks. See ticket 30.
- Does not work when there are 0 integer variables. See ticket 31.
- Be able to set options in C++ code. See ticket 41. It would be nice to be able to read all the options from a generic options file. It seems like right now options for the **DecompAlgo** class cannot be set inside C++.
- Problem with Alps bounds at node 0. See ticket 43
- Figure out how to use BranchEnforceInMaster or BranchEnforceInSubProb so I don't get the large bonds on the variables. See ticket 47.

### 5.9 Miscellaneous Issues

If you want to terminate at the root node and just get the dual value under the `ALPS` option put:

```
[ALPS]
nodeLimit = 1
```

More from Matt:

```
Kipp - the example you sent finds the optimal solution after a few passes of pricing and there

If it prices out and has not yet found optimal, then it will proceed to cuts.

This is parameter driven.


You'll see in the log file (LogDebugLevel = 3),
PRICE_AND_CUT  LimitRoundCutIters       2147483647
PRICE_AND_CUT  LimitRoundPriceIters     2147483647

This is the number of Price/Cut iterations to take before switching off (i.e., MAXINT).

To force it to cut before pricing out, change this parameter in the parm file. For example, if

[DECOMP]
LimitRoundPriceIters = 1
LimitRoundCutIters   = 1

It will then go into your generateCuts after one pricing iteration.

\vskip 12pt

If there is an integer solution at the root node, it may be the case that we are still not opt
``By default, DIP assumes, that if problem is LP feasible to the linear system and IP feasible

For an example of using this see, \url{https://projects.coin-or.org/Dip/browser/trunk/Dip/examp
```

# 6 The OS Library Components

## 6.1 OSAgent

The `OSAgent` part of the library is used to facilitate communication with remote solvers. It is not used if the solver is invoked locally (i.e., on the same machine). There are two key classes in the `OSAgent` component of the OS library. The two classes are `OSSolverAgent` and `WSUtil`.

The `OSSolverAgent` class is used to contact a remote solver service. For example, assume that `sOSiL` is a string with a problem instance and `sOSoL` is a string with solver options. Then the following code will call a solver service and invoke the `solve` method.

```
OSSolverAgent *osagent;
string serviceLocation = http://kipp.chicagobooth.edu/os/OSSolverService.jws
osagent = new OSSolverAgent(  serviceLocation );
string sOSrL = osagent->solve(sOSiL, sOSoL);
```

Other methods in the `OSSolverAgent` class are `send`, `retrieve`, `getJobID`, `knock`, and `kill`. The use of these methods is described in Section ??.

The methods in the `OSSolverAgent` class call methods in the `WSUtil` class that perform such tasks as creating and parsing SOAP messages and making low level socket calls to the server running the solver service. The average user will not use methods in the `WSUtil` class, but they are available to anyone wanting to make socket calls or create SOAP messages.

There is also a method, `OSFileUpload`, in the OSAgentClass that is used to upload files from the hard drive of a client to the server. It is very fast and does not involve SOAP or Web Services. The `OSFileUpload` method is illustrated and described in the example code `OSFileUpload.cpp` described in Section ??.

## 6.2 OSCommonInterfaces

The classes in the OSCommonInterfaces component of the OS library are used to read and write files and strings in the OSiL and OSrL protocols. See Section ?? for more detail on OSiL, OSrL, and other OS protocols. For a complete listing of all of the files in `OSCommonInterfaces` see the Doxygen documentation we deposited at `http://www.doxygen.org`. Users who have Doxygen installed on their system can also create their own version of the documentation (see Section ??). Below we highlight some key classes.

### 6.2.1 The OSInstance Class

The OSInstance class is the in-memory representation of an optimization instance and is a key class for users of the OS project. This class has an API defined by a collection of `get()` methods for extracting various components (such as bounds and coefficients) from a problem instance, a collection of `set()` methods for modifying or generating an optimization instance, and a collection of `calculate()` methods for function, gradient, and Hessian evaluations. See Section 7. We now describe how to create an `OSInstance` object and the close relationship between the OSiL schema and the `OSInstance` class.

### 6.2.2 Creating an `OSInstance` Object

The OSCommonInterfaces component contains an `OSiLReader` class for reading an instance in an OSiL string and creating an in-memory `OSInstance` object. Assume that `sOSiL` is a string that will hold the instance in OSiL format. Creating an `OSInstance` object is illustrated in Figure 3.

### 6.2.3 Mapping Rules

The `OSInstance` class has two members, `instanceHeader` and `instanceData`. These correspond to the XML elements `<instanceHeader>` and `<instanceData>`. They are of type `InstanceHeader` and `InstanceData`, respectively, which in turn correspond to the OSiL schema's complexTypes `InstanceHeader` and `InstanceData`, and in themselves are C++ classes.

Moving down one level, Figure 5 shows that the `InstanceData` class has in turn the members `variables`, `objectives`, `constraints`, `linearConstraintCoefficients`, `quadraticCoefficients`, and `nonlinearExpressions`, corresponding to the respective elements in the OSiL file that have the same name. Each of these are instances of associated classes which correspond to complexTypes in the OSiL schema.

Figure 6 uses the `Variables` class to provide a closer look at the correspondence between schema and class. On the right, the `Variables` class contains the data member `numberOfVariables` and a pointer to the object `var` of class `Variable`. The `Variable` class has data members `lb` (double), `ub` (double), `name` (string), and `type` (char). On the left the corresponding XML `complexType`s are shown, with arrows indicating the correspondences. The following rules describe the mapping between the OSiL schema and the `OSInstance` class. (In order to facilitate the mapping, we insist in the schema construction that every `complexType` be named, even though this is not strictly necessary in XML.)

- Each `complexType` in an OSiL schema corresponds to a class in `OSInstance`. Thus the OSiL schema's `complexType` `Variables` corresponds to `OSInstance`'s class `Variables`. Elements in an actual XML file then correspond to objects in `OSInstance`; for example, the `<variables>` element that is of type `Variables` in an OSiL file corresponds to a `variables` object in `OSInstance`.

- An attribute or element used in the definition of a `complexType` is a member of the corresponding `OSInstance` class, and the type of the attribute or element matches the type of the member. In Figure 6, for example, `lb` is an attribute of the OSiL `complexType` named `Variable`, and `lb` is a member of the `OSInstance` class `Variable`; both have type `double`. Similarly, `<var>` is an element in the definition of the OSiL `complexType` named `Variables`, and `var` is a member of the `OSInstance` class `Variables`; the `<var>` element has type `Variable` and the `var` member is a `Variable` object.

- A schema sequence corresponds to an array. For example, in Figure 6 the complexType `Variables` has a sequence of `<var>` elements that are of type `Variable`, and the corresponding `Variables` class has a member that is an array of type `Variable`.

- XML allows a wide range of data subtypes, which do not always have counterparts in the `OSInstance` object. For instance, the attribute `type` in the `<var>` element forms an enumeration, while the corresponding member of the `Variable` class is declared as `char`.

- XML allows default values for optional attributes; these default values can be set inside of the constructor of the corresponding data member.

General nonlinear terms are stored in the data structure as `OSExpressionTree` objects, which are the subject of the next section.

The `OSInstance` class has a collection of `get()`, `set()`, and `calculate()` methods that act as an API for the optimization instance and are described in Section 7.

### 6.2.4 The OSExpressionTree OSnLNode Classes

The `OSExpressionTree` class provides the in-memory representation of the nonlinear terms. Our design goal is to allow for efficient parsing of OSiL instances, while providing an API that meets the needs of diverse solvers. Conceptually, any nonlinear expression in the objective or constraints is represented by a tree. The expression tree for the nonlinear part of the objective function (**??**), for example, has the form illustrated in Figure 7. The choice of a data structure to store such a tree — along with the associated methods of an API — is a key aspect in the design of the `OSInstance` class.

A base abstract class OSnLNode is defined and all of an OSiL file's operator and operand elements used in defining a nonlinear expression are extensions of the base element type OSnLNode. There is an element type OSnLNodePlus, for example, that extends OSnLNode; then in an OSiL instance file, there are <plus> elements that are of type OSnLNodePlus. Each OSExpressionTree object contains a pointer to an OSnLNode object that is the root of the corresponding expression tree. To every element that extends the OSnLNode type in an OSiL instance file, there corresponds a class that derives from the OSnLNode class in an OSInstance data structure. Thus we can construct an expression tree of homogenous nodes, and methods that operate on the expression tree to calculate function values, derivatives, postfix notation, and the like do not require switches or complicated logic.

The OSInstance class has a variety of calculate() methods, based on two pure virtual functions in the OSInstance class. The first of these, calculateFunction(), takes an array of double values corresponding to decision variables, and evaluates the expression tree for those values. Every class that extends OSnLNode must implement this method. As an example, the calculateFunction method for the OSnLNodePlus class is shown in Figure 8. Because the OSiL instance file must be validated against its schema, and in the schema each <OSnLNodePlus> element is specified to have exactly two child elements, this calculateFunction method can assume that there are exactly two children of the node that it is operating on. The use of polymorphism and recursion makes adding new operator elements easy; it is simply a matter of adding a new class and implementing the calculateFunction() method for it.

Although in the OSnL schema, there are 200+ nonlinear operators, only the following OSnLNode classes are currently supported in our implementation.

- OSnLNodeVariable
- OSnLNodeTimes
- OSnLNodePlus
- OSnLNodeSum
- OSnLNodeMinus
- OSnLNodeNegate
- OSnLNodeDivide
- OSnLNodePower
- OSnLNodeProduct
- OSnLNodeLn
- OSnLNodeSqrt
- OSnLNodeSquare
- OSnLNodeSin
- OSnLNodeCos
- OSnLNodeExp
- OSnLNodeIf

- OSnLNodeAbs
- OSnLNodeMax
- OSnLNodeMin
- OSnLNodeE
- OSnLNodePI
- OSnLNodeAllDiff

#### 6.2.5 The OSOption Class

The `OSOption` class is the in-memory representation of the options associated with a particular optimization task. It is another key class for users of the OS project. This class has an API defined by a collection of `get()` methods for extracting various components (such as initial values for decision variables, solver options, job parameters, etc.), and a collection of `set()` methods for modifying or generating an option instance. The relationship between in-memory classes and objects on one hand and complexTypes and elements of the OSoL schema follow the same mapping rules laid out in Section 6.2.3.

#### 6.2.6 The OSResult Class

Similarly the `OSResult` class is the in-memory representation of the results returned by the solver and other information associated with a particular optimization task. This class has an API defined by a collection of `set()` methods that allow a solver to create a result instance and a collection of `get()` methods for extracting various components (such as optimal values for decision variables, optimal objective function value, optimal dual variables, etc.). The relationship between in-memory classes and objects on one hand and complexTypes and elements of the OSoL schema follow the same mapping rules laid out in Section 6.2.3.

### 6.3 OSModelInterfaces

This part of the OS library is designed to help integrate the OS standards with other standards and modeling systems.

#### 6.3.1 Converting MPS Files

The MPS standard is still a popular format for representing linear and integer programming problems. In `OSModelInterfaces,` there is a class `OSmps2osil` that can be used to convert files in MPS format into the OSiL standard. It is used as follows.

```
OSmps2osil *mps2osil = NULL;
DefaultSolver *solver  = NULL;
solver = new CoinSolver();
solver->sSolverName = "cbc";
mps2osil = new OSmps2osil(  mpsFileName);
mps2osil->createOSInstance() ;
solver->osinstance = mps2osil->osinstance;
solver->solve();
```

The `OSmps2osil` class constructor takes a string which should be the file name of the instance in MPS format. The constructor then uses the `CoinUtils` library to read and parse the MPS file. The class method `createOSInstance` then builds an in-memory `osinstance` object that can be used by a solver.

### 6.3.2 Converting AMPL nl Files

AMPL is a popular modeling language that saves model instances in the AMPL nl format. The `OSModelInterfaces` library provides a class, `OSnl2osil`, for reading an nl file and creating a corresponding in-memory `osinstance` object. It is used as follows.

```
OSnl2osil *nl2osil = NULL;
DefaultSolver *solver  = NULL;
solver = new LindoSolver();
nl2osil = new OSnl2osil( nlFileName);
nl2osil->createOSInstance() ;
solver->osinstance = nl2osil->osinstance;
solver->solve();
```

The `OSnl2osil` class works much like the `OSmps2osil` class. The `OSnl2osil` class constructor takes a string which should be the file name of the instance in nl format. The constructor then uses the AMPL ASL library routines to read and parse the nl file. The class method `createOSInstance` then builds an in-memory `osinstance` object that can be used by a solver.

In Section **??** we describe the `OSAmplClient` executable that acts as a "solver" for AMPL. The `OSAmplClient` uses the `OSnl2osil` class to convert the instance in nl format to OSiL format before calling a solver either locally or remotely.

## 6.4 OSParsers

The OSParsers component of the OS library contains reentrant parsers that read OSiL, OSoL and OSrL strings and build, respectively, in-memory OSInstance, OSOption and OSResult objects.

The OSiL parser is invoked through an `OSiLReader` object as illustrated below. Assume `osil` is a string with the problem instance.

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

The `readOSiL` method has a single argument which is a (pointer to a) string. The `readOSiL` method then calls an underlying method `yygetOSInstance` that parses the OSiL string. The major components of the OSiL schema recognized by the parser are

```
<instanceHeader>
<instanceData>
<variables>
<objectives>
<constraints>
<linearConstraintCoefficients>
<quadraticCoefficients>
<nonlinearExpressions>
```

There are other components in the OSiL schema, but they are not yet implemented. In most large-scale applications the `<variables>`, `<objectives>`, `<constraints>`, and `<linearConstraintCoefficients>` will comprise the bulk of the instance memory. Because of this, we have "hard-coded" the OSiL parser to read these specific elements very efficiently. The parsing of the `<quadraticCoefficients>` and `<nonlinearExpressions>` is done using code generated by `flex` and `bison`. The file `OSParseosil.l` is used by `flex` to generate `OSParseosil.cpp` and the file `OSParseosil.y` is used by `bison` to generate `OSParseosil.tab.cpp`. In `OSParseosil.l` we use the `reentrant` option and in `OSParseosil.y` we use the `pure-parser` option to generate reentrant parsers. The `OSParseosil.y` file contains both our "hard-coded" parser and the grammar rules for the `<quadraticCoefficients>` and `<nonlinearExpressions>` sections. We are currently using GNU `bison` version 3.2 and `flex` 2.5.33.

The typical OS user will have no need to edit either `OSParseosil.l` or `OSParseosil.y` and therefore will not have to worry about running either `flex` or `bison` to generate the parsers. The generated parser code from `flex` and `bison` is distributed with the project and works on all of the platforms listed in Table **??**. If the user does edit either `parseosil.l` or `parseosil.y` then `parseosil.cpp` and `parseosil.tab.cpp` need to be regenerated with `flex` and `bison`. If these programs are present, in the OS directory execute

```
make  run_parsers
```

(This requires Unix or a unix-like environment (Cygwin, MinGW, MSYS, etc.) under Windows.)

The files `OSParseosrl.l` and `OSParseosrl.y` are used by `flex` and `bison` to generate the code `OSParseosrl.cpp` and `OSParseosrl.tab.cpp` for parsing strings in OSrL format. The comments made above about the OSiL parser apply to the OSrL parser. The OSrL parser, like the OSiL parser, is invoked using an `OSrL` reading object. This is illustrated below (`osrl` is a string in OSrL format).

```
OSrLReader *osrlreader = NULL;
osrlreader = new OSrLReader();
OSResult *osresult = NULL;
osresult = osrlreader->readOSrL( osrl);
```

The OSoL parser follows the same layout and rules. The files `OSParseosol.l` and `OSParseosol.y` are used by `flex` and `bison` to generate the code `OSParseosol.cpp` and `OSParseosol.tab.cpp` for parsing strings in OSoL format. The OSoL parser is invoked using an `OSoL` reading object. This is illustrated below (`osol` is a string in OSoL format).

```
OSoLReader *osolreader = NULL;
osolreader = new OSoLReader();
OSOption *osoption = NULL;
osoption = osolreader->readOSoL( osol);
```

There is also a lexer `OSParseosss.l` for tokenizing the command line for the OSSolverService executable described in Section **??**.

## 6.5 OSSolverInterfaces

The `OSSolverInterfaces` library is designed to facilitate linking the OS library with various solver APIs. We first describe how to take a problem instance in OSiL format and connect to a solver that has a COIN-OR OSI interface. See the OSI project `www.projects.coin-or.org/Osi`. We then describe hooking to the COIN-OR nonlinear code `Ipopt`. See `www.projects.coin-or.org/Ipopt`.

Finally we describe hooking to the commercial solver LINDO. The OS library has been tested with the following solvers using the Osi Interface.

- Bonmin
- Cbc
- Clp
- Couenne
- Cplex
- DyLP
- Glpk
- Ipopt
- SYMPHONY
- Vol

In the `OSSolverInterfaces` library there is an abstract class `DefaultSolver` that has the following key members:

```
std::string osil;
std::string osol;
std::string osrl;
OSInstance *osinstance;
OSResult   *osresult;
OSOption   *osoption;
```

and the pure virtual function

```
virtual void solve() = 0 ;
```

In order to use a solver through the COIN-OR `Osi` interface it is necessary to create an object in the `CoinSolver` class which inherits from the `DefaultSolver` class and implements the appropriate `solve()` function. We illustrate with the `Clp` solver.

```
DefaultSolver *solver  = NULL;
solver = new CoinSolver();
solver->m_sSolverName = "clp";
```

Assume that the data file containing the problem has been read into the string `osil` and the solver options are in the string `osol`. Then the `Clp` solver is invoked as follows.

```
solver->osil = osil;
solver->osol = osol;
solver->solve();
```

Finally, get the solution in `OSrL` format as follows

```
cout << solver->osrl << endl;
```

Some commercial solvers, e.g., LINDO, do not have a COIN-OR `Osi` interface, but it is possible to write wrappers so that they can be used in exactly the same manner as a COIN-OR solver. For example, to invoke the LINDO solver we do the following.

```
solver = new LindoSolver();
```

A similar call is used for `Ipopt`. In this case, the `IpoptSolver` class inherits from both the `DefaultSolver` class and the Ipopt `TNLP` class. See

small`https://projects.coin-or.org/Ipopt/browser/stable/3.5/Ipopt/doc/documentation.pdf?format=r`

for more information on the Ipopt solver C++ implementation and the `TNLP` class.

In the examples above, the problem instance was assumed to be read from a file into the string `osil` and then into the class member `solver->osil.` However, everything can be done entirely in memory. For example, it is possible to use the `OSInstance` class to create an in-memory problem representation and give this representation directly to a solver class that inherits from `DefaultSolver`. The class member to use is `osinstance.` This is illustrated in the example given in Section 4.2.

### 6.6 OSUtils

The OSUtils component of the OS library contains utility codes. For example, the `FileUtil` class contains useful methods for reading files into `string` or `char*` and writing files from `string` and `char*`. The `OSDataStructures` class holds other classes for things such as sparse vectors, sparse Jacobians, and sparse Hessians. The `MathUtil` class contains a method for converting between sparse matrices in row and column major form.

## 7 The OSInstance API

The OSInstance API can be used to:

- get information about model parameters, or convert the `OSExpressionTree` into a prefix or postfix representation through a collection of `get()` methods,
- modify, or even create an instance from scratch, using a number of `set()` methods,
- provide information to solvers that require function evaluations, Jacobian and Hessian sparsity patters, function gradient evaluations, and Hessian evaluations.

### 7.1 Get Methods

The `get()` methods are used by other classes to access data in an existing `OSInstance` object or get an expression tree representation of an instance in postfix or prefix format. Assume `osinstance` is an object in the `OSInstance` class created as illustrated in Figure 3. Then, for example,

```
osinstance->getVariableNumber();
```

will return an integer which is the number of variables in the problem,

```
osinstance->getVariableTypes();
```

will return a `char` pointer to the variable types (`C` for continuous, `B` for binary, and `I` for general integer),

```
getVariableLowerBounds();
```

will return a `double` pointer to the lower bound on each variable. There are similar `get()` methods for the constraints. There are numerous `get()` methods for the data in the `<linearConstraintCoefficients>` element, the `<quadraticCoefficients>` element, and the `<nonlinearExpressions>` element.

When an `osinstance` object is created, it is stored as an expression tree in an `OSExpressionTree` object. However, some solver APIs (e.g., LINDO) may take the data in a different format such as postfix and prefix. There are methods to return the data in either postfix or prefix format.

First define a `vector` of pointers to `OSnLNode` objects.

```
std::vector<OSnLNode*> postfixVec;
```

then get the expression tree for the objective function (index = -1) as a postfix vector of nodes.

```
postfixVec = osinstance->getNonlinearExpressionTreeInPostfix( -1);
```

If, for example, the `osinstance` object was the in-memory representation of the instance illustrated in Section **??** and Figure 7 then the code

```
for (i = 0 ; i < n; i++){
    cout << postfixVec[i]->snodeName << endl;
}
```

will produce

```
number
variable
minus
number
power
number
variable
variable
number
power
minus
number
power
times
plus
```

This postfix traversal of the expression tree in Figure 7 lists all the nodes by recursively processing all subtrees, followed by the root node. The method `processNonlinearExpressions()` in the `LindoSolver` class in the `OSSolverInterfaces` library component illustrates the use of a postfix vector of `OSnLNode` objects to build a Lindo model instance.

## 7.2 Set Methods

The `set()` methods can be used to build an in-memory `OSInstance` object. A code example of how to do this is in Section 4.2.

## 7.3 Calculate Methods

The `calculate()` methods are described in Section 8.

## 7.4 Modifying an `OSInstance` Object

The OSInstance API is designed to be used to either build an in-memory `OSInstance` object or provide information about the in-memory object (e.g., the number of variables). This interface is not designed for problem modification. We plan on later providing an `OSModification` object for this task. However, by directly accessing an `OSInstance` object it is possible to modify parameters in the following classes:

- `Variables`
- `Objectives`
- `Constraints`
- `LinearConstraintCoefficients`

For example, to modify the first nonzero objective function coefficient of the first objective function to 10.7 the user would write,

```
osinstance->instanceData->objectives->obj[0]->coef[0]->value = 10.7;
```

If the user wanted to modify the actual number of nonzero coefficients as declared by

```
osinstance->instanceData->objectives->obj[0]->numberOfObjCoef;
```

then the only safe course of action would be to delete the current `OSInstance` object and build a new one with the modified coefficients. It is strongly recommend that no changes are made involving allocated memory – i.e., any kind of `numberOf***`. Modifying an objective function coefficient is illustrated in the OSModDemo example. See Section 4.4.

After modifying an `OSInstance` object, it is necessary to set certain boolean variables to true in order for these changes to get reflected in the OS solver interfaces.

- `Variables` – if any changes are made to a parameter in this class set

  ```
  osinstance->bVariablesModified = true;
  ```

- `Objectives` – if any changes are made to a parameter in this class set

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

- `Constraints` – if any changes are made to a parameter in this class set

```
osinstance->bConstraintsModified = true;
```

- `LinearConstraintCoefficients` – if any changes are made to a parameter in this class set

```
osinstance->bAMatrixModified = true;
```

At this point, if the user desires to modify an `OSInstance` object that contains nonlinear terms, the only safe strategy is to delete the object and build a new object that contains the modifications.

### 7.5 Printing a Model for Debugging

The OSiL representation for the test problem `rosenbrockmod.osil` is given in Appendix **??**. Many users will not find the OSiL representation useful for model debugging purposes. For users who wish to see a model in a standard infix representation we provide a method `printModel()`. Assume that we have an `osinstance` object in the `OSInstance` class that represents the model of interest. The call

```
osinstance->printModel( -1)
```

will result in printing the (first) objective function indexed by -1. In order to print constraint $k$ use

```
osinstance->printModel( k)
```

In order to print the entire model use

```
osinstance->printModel( )
```

Below we give the result of `osintance->printModel( )` for the problem `rosenbrockmod.osil`.

```
Objectives:
min 9*x_1 + (((1 - x_0) ^ 2) + (100*((x_1 - (x_0 ^ 2)) ^ 2)))

Constraints:
(((((10.5*x_0)*x_0) + ((11.7*x_1)*x_1)) + ((3*x_0)*x_1)) + x_0) <= 25
10 <= ((ln( (x_0*x_1)) + (7.5*x_0)) + (5.25*x_1))

Variables:
x_0 Type = C  Lower Bound =  0  Upper Bound =  1.7976931348623157e308
x_1 Type = C  Lower Bound =  0  Upper Bound =  1.7976931348623157e308
```

## 8 The OS Algorithmic Differentiation Implementation

The OS library provides a set of `calculate` methods for calculating function values, gradients, and Hessians. The `calculate` methods are part of the `OSInstance` class and are designed to work with solver APIs. For instance, `Ipopt` requires derivatives but does not provide its own differentiation routines, expecting the user to make them available through callbacks.

## 8.1 Algorithmic Differentiation: Brief Review

First and second derivative calculations are made using *algorithmic differentiation*. Here we provide a brief review of this topic. An excellent reference on algorithmic differentiation is Griewank [3]. The OS package uses the COIN-OR project CppAD (`http://projects.coin-or.org/CppAD`), which is also an excellent resource with extensive documentation and information about algorithmic differentiation. See the documentation written by Brad Bell [1]. The development here is from the CppAD documentation. Consider the function $f : X \rightarrow Y$ from $\mathbb{R}^n$ to $\mathbb{R}^m$. (That is, $Y = f(X)$.) Assume that $f$ is twice continuously differentiable, so that in particular the second order partials

$$\frac{\partial^2 f_k}{\partial x_i \partial x_j} \quad \text{and} \quad \frac{\partial^2 f_k}{\partial x_j \partial x_i} \tag{14}$$

exist and are equal to each other for all $k = 1, \ldots, m$ and $i, j = 1, \ldots, n$. The task is to compute the derivatives of $f$.

First express the input vector as a function of $t$ by

$$X(t) = x^{(0)} + x^{(1)}t + x^{(2)}t^2 \tag{15}$$

where $x^{(0)}$, $x^{(1)}$, and $x^{(2)}$ are vectors in $\mathbb{R}^n$ and $t$ is a scalar. By judiciously choosing $x^{(0)}, x^{(1)}$, and $x^{(2)}$ we will be able to derive many different expressions of interest. Note first that

$$\begin{aligned} X(0) &= x^{(0)}, \\ X'(0) &= x^{(1)}, \\ X''(0) &= 2x^{(2)}. \end{aligned}$$

In general, $x^{(k)}$ corresponds to the $k^{\text{th}}$ order Taylor coefficient, i.e.,

$$x^{(k)} = \frac{1}{k!} X^{(k)}(0), \quad k = 0, 1, 2. \tag{16}$$

Then $Y(t) = f(X(t))$ is a function from $\mathbb{R}^1$ to $\mathbb{R}^m$ and is expressed in terms of its Taylor series expansion as

$$Y(t) = y^{(0)} + y^{(1)}t + y^{(2)}t^2 + o(t^3), \tag{17}$$

where

$$y^{(k)} = \frac{1}{k!} Y^{(k)}(0), \quad k = 0, 1, 2. \tag{18}$$

The following are shown in Bell [1].

$$y^{(0)} = f(x^{(0)}). \tag{19}$$

Let $e^{(i)}$ denote the $i^{\text{th}}$ unit vector. If $x^{(1)} = e^{(i)}$ then $y^{(1)}$ is equal to the $i^{\text{th}}$ column of the Jacobian matrix of $f(x)$ evaluated at $x^{(0)}$. That is

$$y^{(1)} = \frac{\partial f}{\partial x_i}(x^{(0)}). \tag{20}$$

In addition, if $x^{(1)} = e^{(i)}$ and $x^{(2)} = 0$ then for function $f_k(x)$, (the $k^{\text{th}}$ component of $f$)

$$y_k^{(2)} = \frac{1}{2} \frac{\partial^2 f_k(x^{(0)})}{\partial x_i \partial x_i}. \tag{21}$$

In order to evaluate the mixed partial derivatives, one can instead set $x^{(1)} = e^{(i)} + e^{(j)}$ and $x^{(2)} = 0$. This gives for function $f_k(x)$,

$$y_k^{(2)} = \frac{1}{2}\left(\frac{\partial^2 f_k(x^{(0)})}{\partial x_i \partial x_i} + \frac{\partial^2 f_k(x^{(0)})}{\partial x_i \partial x_j} + \frac{\partial^2 f_k(x^{(0)})}{\partial x_j \partial x_i} + \frac{\partial^2 f_k(x^{(0)})}{\partial x_j \partial x_j}\right), \tag{22}$$

or, expressed in terms of the mixed partials,

$$\frac{\partial^2 f_k(x^{(0)})}{\partial x_i \partial x_j} = y_k^{(2)} - \frac{1}{2}\left(\frac{\partial^2 f_k(x^{(0)})}{\partial x_i \partial x_i} + \frac{\partial^2 f_k(x^{(0)})}{\partial x_j \partial x_j}\right). \tag{23}$$

## 8.2 Using OSInstance Methods: Low Level Calls

The code snippets used in this section are from the example code `algorithmicDiffTest.cpp` in the `algorithmicDiffTest` folder in the `examples` folder. The code is based on the following example.

$$\begin{aligned}
&\text{Minimize} && x_0^2 + 9x_1 && (24)\\
&\text{s.t.} && 33 - 105 + 1.37x_1 + 2x_3 + 5x_1 \le 10 && (25)\\
& && \ln(x_0 x_3) + 7x_2 \ge 10 && (26)\\
& && x_0, x_1, x_2, x_3 \ge 0 && (27)
\end{aligned}$$

The OSiL representation of the instance (24)–(27) is given in Appendix **??**. This example is designed to illustrate several features of OSiL. Note that in constraint (25) the constant 33 appears in the `<con>` element corresponding to this constraint and the constant 105 appears as a `<number>` OSnL node in the `<nonlinearExpressions>` section. This distinction is important, as it will lead to different treatment by the code as documented below. Variables $x_1$ and $x_2$ do not appear in any nonlinear terms. The terms $5x_1$ in (25) and $7x_2$ in (26) are expressed in the `<objectives>` and `<linearConstraintCoefficients>` sections, respectively, and will again receive special treatment by the code. However, the term $1.37x_1$ in (25), along with the term $2x_3$, is expressed in the `<nonlinearExpressions>` section, hence $x_1$ is treated as a nonlinear variable for purposes of algorithmic differentiation. Variable $x_2$ never appears in the `<nonlinearExpressions>` section and is therefore treated as a linear variable and not used in any algorithmic differentiation calculations. Variables that do not appear in the `<nonlinearExpressions>` are never part of the algorithmic differentiation calculations.

Ignoring the nonnegativity constraints, instance (24)–(27) defines a mapping from $\mathbb{R}^4$ to $\mathbb{R}^3$:

$$\begin{aligned}
\begin{bmatrix} x_0^2 + 9x_1 \\ 33 - 105 + 1.37x_1 + 2x_3 + 5x_1 \\ \ln(x_0x_3) + 7x_2 \end{bmatrix} &= \begin{bmatrix} 9x_1 \\ 33 + 5x_1 \\ 7x_2 \end{bmatrix} + \begin{bmatrix} x_0^2 \\ -105 + 1.37x_1 + 2x_3 \\ \ln(x_0x_3) \end{bmatrix} \\
&= \begin{bmatrix} 9x_1 \\ 33 + 5x_1 \\ 7x_2 \end{bmatrix} + \begin{bmatrix} f_1(x) \\ f_2(x) \\ f_3(x) \end{bmatrix}, \qquad (28)
\end{aligned}$$

$$\text{where } f(x) := \begin{bmatrix} f_1(x) \\ f_2(x) \\ f_3(x) \end{bmatrix}. \tag{29}$$

The OSiL representation for the instance in (24)–(27) is read into an in-memory OSInstance object as follows (we assume that `osil` is a `string` containing the OSiL instance)

```
osilreader = new OSiLReader();
osinstance = osilreader->readOSiL( &osil);
```

There is a method in the `OSInstance` class, `initForAlgDiff()` that is used to initialize the nonlinear data structures. A call to this method

```
osinstance->initForAlgDiff( );
```

will generate a map of the indices of the nonlinear variables. This is critical because the algorithmic differentiation only operates on variables that appear in the `<nonlinearExpressions>` section. An example of this map follows.

```
std::map<int, int> varIndexMap;
std::map<int, int>::iterator posVarIndexMap;
varIndexMap = osinstance->getAllNonlinearVariablesIndexMap( );
for(posVarIndexMap = varIndexMap.begin(); posVarIndexMap
    != varIndexMap.end(); ++posVarIndexMap){
    std::cout <<  "Variable Index = "   << posVarIndexMap->first  << std::endl ;
}
```

The variable indices listed are 0, 1, and 3. Variable 2 does not appear in the `<nonlinearExpressions>` section and is not included in `varIndexMap`. That is, the function $f$ in (29) will be considered as a map from $\mathbb{R}^3$ to $\mathbb{R}^3$.

Once the nonlinear structures are initialized it is possible to take derivatives using algorithmic differentiation. Algorithmic differentiation is done using either a forward or reverse sweep through an expression tree (or operation sequence) representation of $f$. The two key `public` algorithmic differentiation methods in the `OSInstance` class are `forwardAD` and `reverseAD`. These are actually generic "wrappers" around the corresponding CppAD methods with the same signature. This keeps the OS API public methods independent of any underlying algorithmic differentiation package.

The `forwardAD` signature is

```
std::vector<double> forwardAD(int k, std::vector<double> vdX);
```

where `k` is the highest order Taylor coefficient of $f$ to be returned, `vdX` is a vector of doubles in $\mathbb{R}^n$, and the function return is a vector of doubles in $\mathbb{R}^m$. Thus, `k` corresponds to the $k$ in Equations (16) and (18), where `vdX` corresponds to the $x^{(k)}$ in Equation (16), and the $y^{(k)}$ in Equation (18) is the vector in range space returned by the call to `forwardAD`. For example, by Equation (19) the following call will evaluate each component function defined in (29) corresponding only to the nonlinear part of (28) – the part denoted by $f(x)$.

```
funVals = osinstance->forwardAD(0, x0);
```

Since there are three components in the vector defined by (29), the return value `funVals` will have three components. For an input vector,

```
x0[0] = 1; // the value for variable x0 in function f
x0[1] = 5; // the value for variable x1 in function f
x0[2] = 5; // the value for variable x3 in function f
```

the values returned by `osinstance->forwardAD(0, x0)` are 1, -63.15, and 1.6094, respectively. The Jacobian of the example in (29) is

$$J = \begin{bmatrix} 2x_0 & 9.00 & 0.00 & 0.00 \\ 0.00 & 6.37 & 0.00 & 2.00 \\ 1/x_0 & 0.00 & 7.00 & 1/x_3 \end{bmatrix} \tag{30}$$

and the Jacobian $J_f$ of the nonlinear part is

$$J_f = \begin{bmatrix} 2x_0 & 0.00 & 0.00 \\ 0.00 & 1.37 & 2.00 \\ 1/x_0 & 0.00 & 1/x_3 \end{bmatrix}. \tag{31}$$

When $x_0 = 1$, $x_1 = 5$, $x_2 = 10$, and $x_3 = 5$ the Jacobian $J_f$ is

$$J_f = \begin{bmatrix} 2.00 & 0.00 & 0.00 \\ 0.00 & 1.37 & 2.00 \\ 1.00 & 0.00 & 0.20 \end{bmatrix}. \tag{32}$$

A forward sweep with $k = 1$ will calculate the Jacobian column-wise. See (20). The following code will return column 3 of the Jacobian (32) which corresponds to the nonlinear variable $x_3$.

```
x1[0] = 0;
x1[1] = 0;
x1[2] = 1;
osinstance->forwardAD(1, x1);
```

Now calculate second derivatives. To illustrate we use the results in (21)-(23) and calculate

$$\frac{\partial^2 f_k(x^{(0)})}{\partial x_0 \partial x_3} \quad k = 1, 2, 3.$$

Variables $x_0$ and $x_3$ are the first and third nonlinear variables so by (22) the $x^{(1)}$ should be the sum of the $e^{(1)}$ and $e^{(3)}$ unit vectors and used in the first-order forward sweep calculation.

```
x1[0] = 1;
x1[1] = 0;
x1[2] = 1;
osinstance->forwardAD(1, x1);
```

Next set $x^{(2)} = 0$ and do a second-order forward sweep.

```
std::vector<double> x2( n);
x2[0] = 0;
x2[1] = 0;
x2[2] = 0;
osinstance->forwardAD(2, x2);
```

This call returns the vector of values

$$y_1^{(2)} = 1, \quad y_2^{(2)} = 0, \quad y_3^{(2)} = -0.52.$$

By inspection of (28) (or by appropriate calls to `osinstance->forwardAD` — not shown here),

$$
\begin{aligned}
\frac{\partial^2 f_1(x^{(0)})}{\partial x_0 \partial x_0} &= 2, & \frac{\partial^2 f_1(x^{(0)})}{\partial x_3 \partial x_3} &= 0, \\
\frac{\partial^2 f_2(x^{(0)})}{\partial x_0 \partial x_0} &= 0, & \frac{\partial^2 f_2(x^{(0)})}{\partial x_3 \partial x_3} &= 0, \\
\frac{\partial^2 f_3(x^{(0)})}{\partial x_0 \partial x_0} &= -1, & \frac{\partial^2 f_3(x^{(0)})}{\partial x_3 \partial x_3} &= -0.04.
\end{aligned}
$$

Then by (23),

$$
\begin{aligned}
\frac{\partial^2 f_1(x^{(0)})}{\partial x_0 \partial x_3} &= y_1^{(2)} - \frac{1}{2}\left(\frac{\partial^2 f_1(x^{(0)})}{\partial x_0 \partial x_0} + \frac{\partial^2 f_k(x^{(0)})}{\partial x_3 \partial x_3}\right) = 1 - \frac{1}{2}(2+0) = 0, \\
\frac{\partial^2 f_2(x^{(0)})}{\partial x_0 \partial x_3} &= y_2^{(2)} - \frac{1}{2}\left(\frac{\partial^2 f_2(x^{(0)})}{\partial x_0 \partial x_0} + \frac{\partial^2 f_k(x^{(0)})}{\partial x_3 \partial x_3}\right) = 0 - \frac{1}{2}(0+0) = 0, \\
\frac{\partial^2 f_3(x^{(0)})}{\partial x_0 \partial x_3} &= y_3^{(2)} - \frac{1}{2}\left(\frac{\partial^2 f_3(x^{(0)})}{\partial x_0 \partial x_0} + \frac{\partial^2 f_k(x^{(0)})}{\partial x_3 \partial x_3}\right) = -0.52 - \frac{1}{2}(-1-0.04) = 0.
\end{aligned}
$$

Making all of the first and second derivative calculations using forward sweeps is most effective when the number of rows exceeds the number of variables.

The `reverseAD` signature is

```
std::vector<double> reverseAD(int k, std::vector<double> vdlambda);
```

where `vdlambda` is a vector of Lagrange multipliers. This method returns a vector in the range space. If a reverse sweep of order $k$ is called, a forward sweep of all orders through $k-1$ must have been made prior to the call.

### 8.2.1 First Derivative Reverse Sweep Calculations

In order to calculate first derivatives execute the following sequence of calls.

```
x0[0] = 1;
x0[1] = 5;
x0[2] = 5;
std::vector<double> vlambda(3);
vlambda[0] = 0;
vlambda[1] = 0;
vlambda[2] = 1;
osinstance->forwardAD(0, x0);
osinstance->reverseAD(1, vlambda);
```

Since `vlambda` only includes the third function $f_3$, this sequence of calls will produce the third row of the Jacobian $J_f$, i.e.,

$$\frac{\partial f_3(x^{(0)})}{\partial x_0} = 1, \quad \frac{\partial f_3(x^{(0)})}{\partial x_1} = 0, \quad \frac{\partial f_3(x^{(0)})}{\partial x_3} = 0.2.$$

### 8.2.2 Second Derivative Reverse Sweep Calculations

In order to calculate second derivatives using `reverseAD` forward sweeps of order 0 and 1 must have been completed. The call to `reverseAD(2, vlambda)` will return a vector of dimension $2n$ where $n$ is the number of variables. If the zero-order forward sweep is `forwardAD(0,x0)` and the first-order forward sweep is `forwardAD(1, x1)` where `x1` $= e^{(i)}$, then the return vector `z = reverseAD(2, vlambda)` is

$$z[2j-2] = \frac{\partial L(x^{(0)}, \lambda^{(0)})}{\partial x_j}, \quad j = 1, \ldots, n \tag{33}$$

$$z[2j-1] = \frac{\partial^2 L(x^{(0)}, \lambda^{(0)})}{\partial x_i \partial x_j}, \quad j = 1, \ldots, n \tag{34}$$

where

$$L(x, \lambda) = \sum_{k=1}^{m} \lambda_k f_k(x). \tag{35}$$

For example, the following calls will calculate the third row (column) of the Hessian of the Lagrangian.

```
x0[0] = 1;
x0[1] = 5;
x0[2] = 5;
osinstance->forwardAD(0, x0);
x1[0] = 0;
x1[1] = 0;
x1[2] = 1;
osinstance->forwardAD(1, x1);
vlambda[0] = 1;
vlambda[1] = 2;
vlambda[2] = 1;
osinstance->reverseAD(2, vlambda);
```

This returns

$$\frac{\partial L(x^{(0)}, \lambda^{(0)})}{\partial x_0} = 3, \quad \frac{\partial L(x^{(0)}, \lambda^{(0)})}{\partial x_1} = 2.74, \quad \frac{\partial L(x^{(0)}, \lambda^{(0)})}{\partial x_3} = 4.2,$$

$$\frac{\partial^2 L(x^{(0)}, \lambda^{(0)})}{\partial x_3 \partial x_0} = 0, \quad \frac{\partial^2 L(x^{(0)}, \lambda^{(0)})}{\partial x_3 \partial x_0} = 0, \quad \frac{\partial^2 L(x^{(0)}, \lambda^{(0)})}{\partial x_3 \partial x_3} = -.04.$$

The reason why

$$\frac{\partial L(x^{(0)}, \lambda^{(0)})}{\partial x_1} = 2 \times 1.37 = 2.74$$

and not

$$\frac{\partial L(x^{(0)}, \lambda^{(0)})}{\partial x_1} = 1 \times 9 + 2 \times 6.37 = 9 + 12.74 = 21.74$$

is that the terms $9x_1$ in the objective and $5x_1$ in the first constraint are captured in the linear section of the OSiL input and therefore do not appear as nonlinear terms in `<nonlinearExpressions>`. As noted before, `forwardAD` and `reverseAD` only operate on variables and terms in either the `<quadraticCoefficients>` or `<nonlinearExpressions>` sections.

## 8.3 Using OSInstance Methods: High Level Calls

The methods `forwardAD` and `reverseAD` are low-level calls and are not designed to work directly with solver APIs. The `OSInstance` API has other methods that most users will want to invoke when linking with solver APIs. We describe these now.

### 8.3.1 Sparsity Methods

Many solvers such as `Ipopt` (`projects.coin-or.org/Ipopt`) require the sparsity pattern of the Jacobian of the constraint matrix and the Hessian of the Lagrangian function. Note well that the constraint matrix of the example in Section 8.2 constitutes only the last two rows of (29) but does include the linear terms. The following code illustrates how to get the sparsity pattern of the constraint Jacobian matrix

```
SparseJacobianMatrix *sparseJac;
sparseJac = osinstance->getJacobianSparsityPattern();
for(idx = 0; idx < sparseJac->startSize; idx++){
    std::cout << "number constant terms in constraint "   <<  idx << " is "
    << *(sparseJac->conVals + idx)  << std::endl;
    for(k = *(sparseJac->starts + idx); k < *(sparseJac->starts + idx + 1); k++){
        std::cout << "row idx = " << idx <<  "
        col idx = "<< *(sparseJac->indexes + k) << std::endl;
    }
}
```

For the example problem this will produce

```
JACOBIAN SPARSITY PATTERN
number constant terms in constraint 0 is 0
row idx = 0  col idx = 1
row idx = 0  col idx = 3
number constant terms in constraint 1 is 1
row idx = 1  col idx = 2
row idx = 1  col idx = 0
row idx = 1  col idx = 3
```

The constant term in constraint 1 corresponds to the linear term $7x_2$, which is added after the algorithmic differentiation has taken place. However, the linear term $5x_1$ in constraint 0 does not contribute a nonzero in the Jacobian, as it is combined with the term $1.37x_1$ that is treated as a nonlinear term and therefore accounted for explicitly. The `SparseJacobianMatrix` object has a data member `starts` which is the index of the start of each constraint row. The `int` data member `indexes` gives the variable index of every potentially nonzero derivative. There is also a `double` data member `values` that gives the value of the partial derivative of the corresponding index at each iteration. Finally, there is an `int` data member `conVals` that is the number of constant terms in each gradient. A constant term is a partial derivative that cannot change at an iteration. A variable is considered to have a constant derivative if it appears in the `<linearConstraintCoefficients>` section but not in the `<nonlinearExpressions>`. For a row indexed by `idx` the variable indices are in the `indexes` array between the elements `sparseJac->starts + idx` and `sparseJac->starts + idx + 1`. The first `sparseJac->conVals + idx` variables listed are indices of variables with constant derivatives. In this example, when idx is 1, there is one variable with a constant derivative and it is variable $x_2$. (Actually variable $x_1$ has a constant derivative but the code does not check to see if variables that appear in the <nonlinearExpressions> section have constant derivative.) The variables with constant derivatives never appear in the AD evaluation.

The following code illustrates how to get the sparsity pattern of the Hessian of the Lagrangian.

```
SparseHessianMatrix *sparseHessian;
sparseHessian = osinstance->getLagrangianHessianSparsityPattern( );
for(idx = 0; idx < sparseHessian->hessDimension; idx++){
    std::cout <<  "Row Index = " << *(sparseHessian->hessRowIdx + idx) ;
    std::cout <<  "  Column Index = " << *(sparseHessian->hessColIdx + idx);
}
```

The SparseHessianMatrix class has the int data members hessRowIdx and hessColIdx for indexing potential nonzero elements in the Hessian matrix. The double data member hessValues holds the value of the respective second derivative at each iteration. The data member hessDimension is the number of nonzero elements in the Hessian.

### 8.3.2 Function Evaluation Methods

There are several overloaded methods for calculating objective and constraint values. The method

```
double *calculateAllConstraintFunctionValues(double* x, bool new_x)
```

will return a double pointer to an array of constraint function values evaluated at x. If the value of x has not changed since the last function call, then new_x should be set to false and the most recent function values are returned. When using this method, with this signature, all function values are calculated in double using an OSExpressionTree object.

A second signature for the calculateAllConstraintFunctionValues is

```
double *calculateAllConstraintFunctionValues(double* x, double *objLambda,
    double *conLambda, bool new_x, int highestOrder)
```

In this signature, x is a pointer to the current primal values, objLambda is a vector of dual multipliers, conLambda is a vector of dual multipliers on the constraints, new_x is true if any components of x have changed since the last evaluation, and highestOrder is the highest order of derivative to be calculated at this iteration. The following code snippet illustrates defining a set of variable values for the example we are using and then the function call.

```
double* x = new double[4]; //primal variables
double* z = new double[2]; //Lagrange multipliers on constraints
double* w = new double[1]; //Lagrange multiplier on objective
x[ 0] = 1;    // primal variable 0
x[ 1] = 5;    // primal variable 1
x[ 2] = 10;   // primal variable 2
x[ 3] = 5;    // primal variable 3
z[ 0] = 2;    // Lagrange multiplier on constraint 0
z[ 1] = 1;    // Lagrange multiplier on constraint 1
w[ 0] = 1;    // Lagrange multiplier on the objective function
calculateAllConstraintFunctionValues(x, w, z,  true, 0);
```

When making all high level calls for function, gradient, and Hessian evaluations we pass all the primal variables in the `x` argument, not just the nonlinear variables. Underneath the call, the nonlinear variables are identified and used in AD function calls.

The use of the parameters `new_x` and `highestOrder` is important and requires further explanation. The parameter `highestOrder` is an integer variable that will take on the value 0, 1, or 2 (actually higher values if we want third derivatives etc.). The value of this variable is the highest order derivative that is required of the current iterate. For example, if a callback requires a function evaluation and `highestOrder = 0` then only the function is evaluated at the current iterate. However, if `highestOrder = 2` then the function call

```
calculateAllConstraintFunctionValues(x, w, z, true, 2)
```

will trigger first and second derivative evaluations in addition to the function evaluations.

In the `OSInstance` class code, every time a forward (`forwardAD`) or reverse sweep (`reverseAD`) is executed a private member, `m_iHighestOrderEvaluated` is set to the order of the sweep. For example, `forwardAD(1, x)` will result in `m_iHighestOrderEvaluated = 1`. Just knowing the value of `new_x` alone is not sufficient. It is also necessary to know `highestOrder` and compare it with `m_iHighestOrderEvaluated`. For example, if `new_x` is false, but `m_iHighestOrderEvaluated = 0`, and the callback requires a Hessian calculation, then it is necessary to calculate the first and second derivatives at the current iterate.

There are *exactly two* conditions that require a new function or derivative evaluation. A new evaluation is required if and only if

1. The value of `new_x` is true

–OR–

2. For the callback function the value of the input parameter `highestOrder` is strictly greater than the current value of `m_iHhighestOrderEvaluated`.

For an efficient implementation of AD it is important to be able to get the Lagrange multipliers and highest order derivative that is required from inside *any* callback – not just the Hessian evaluation callback. For example, in `Ipopt`, if `eval_g` or `eval_f` are called, and for the current iterate, `eval_jac` and `eval_hess` are also going to be called, then a more efficient AD implementation is possible if the Lagrange multipliers are available for `eval_g` and `eval_f`.

Currently, whenever `new_x = true` in the underlying AD implementation we do not retape (record into the CppAD data structure) the function. This is because we currently throw an exception if there are any logical operators involved in the AD calculations. This may change in a future implementation.

There are also similar methods for objective function evaluations. The method

```
double calculateFunctionValue(int idx, double* x, bool new_x);
```

will return the value of any constraint or objective function indexed by `idx`. This method works strictly with `double` data using an `OSExpressionTree` object.

There is also a public variable, `bUseExpTreeForFunEval` that, if set to `true`, will cause the method

```
calculateAllConstraintFunctionValues(x, objLambda,  conLambda, true, highestOrder)
```

to also use the OS expression tree for function evaluations when `highestOrder = 0` rather than use the operator overloading in the CppAD tape.

### 8.3.3 Gradient Evaluation Methods

One `OSInstance` method for gradient calculations is

```
SparseJacobianMatrix *calculateAllConstraintFunctionGradients(double* x, double *objLambda,
     double *conLambda, bool new_x, int highestOrder)
```

If a call has been placed to `calculateAllConstraintFunctionValues` with `highestOrder = 0`, then the appropriate call to get gradient evaluations is

```
calculateAllConstraintFunctionGradients( x, NULL, NULL,  false, 1);
```

Note that in this function call `new_x = false`. This prevents a call to `forwardAD()` with order 0 to get the function values.

If, at the current iterate, the Hessian of the Lagrangian function is also desired then an appropriate call is

```
calculateAllConstraintFunctionGradients(x, objLambda, conLambda, false, 2);
```

In this case, if there was a prior call

```
calculateAllConstraintFunctionValues(x, w, z,  true, 0);
```

then only first and second derivatives are calculated, not function values.

When calculating the gradients, if the number of nonlinear variables exceeds or is equal to the number of rows, a `forwardAD(0, x)` sweep is used to get the function values, and a `reverseAD(1,` $e^k$`)` sweep for each unit vector $e^k$ in the row space is used to get the vector of first order partials for each row in the constraint Jacobian. If the number of nonlinear variables is less then the number of rows then a `forwardAD(0, x)` sweep is used to get the function values and a `forwardAD(1,` $e^i$`)` sweep for each unit vector $e^i$ in the column space is used to get the vector of first order partials for each column in the constraint Jacobian.

Two other gradient methods are

```
SparseVector *calculateConstraintFunctionGradient(double* x,
    double *objLambda, double *conLambda,  int idx, bool new_x, int highestOrder);
```

and

```
SparseVector *calculateConstraintFunctionGradient(double* x, int idx,
    bool new_x );
```

Similar methods are available for the objective function; however, the objective function gradient methods treat the gradient of each objective function as a dense vector.

### 8.3.4 Hessian Evaluation Methods

There are two methods for Hessian calculations. The first method has the signature

```
SparseHessianMatrix *calculateLagrangianHessian( double* x,
    double *objLambda, double *conLambda, bool new_x, int highestOrder);
```

so if either function or first derivatives have been calculated an appropriate call is

```
calculateLagrangianHessian( x, w, z, false, 2);
```

If the Hessian of a single row or objective function is desired the following method is available

```
SparseHessianMatrix *calculateHessian( double* x, int idx, bool new_x);
```

## References

[1] Bradley Bell. CppAD Documentation, 2007. `http://www.coin-or.org/CppAD/Doc/cppad.xml`.

[2] R. Fourer, L. Lopes, and K. Martin. LPFML: A W3C XML schema for linear and integer programming. *INFORMS Journal on Computing*, 17:139–158, 2005.

[3] Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, Philadelphia, PA, 2000.

[4] J. Ma. Optimization services (OS), a general framework for optimization modeling systems, 2005. Ph.D. Dissertation, Department of Industrial Engineering & Management Sciences, Northwestern University, Evanston, IL.

[5] H.H. Rosenbrock. An automatic method for finding the greatest or least value of a function. *Comp. J.*, 3:175–184, 1960.

```
         <other name="variableBlockSet" solver="Dip" numberOfVar="6" value="MySolver3">
            <var idx="10"/>
            <var idx="11"/>
            <var idx="12"/>
            <var idx="13"/>
            <var idx="14"/>
            <var idx="17"/>
         </other>
      </variables>
      <constraints numberOfOtherConstraintOptions="1">
         <other name="constraintSet" solver="Dip" numberOfCon="5" type="Core">
            <con idx="15"/>
            <con idx="16"/>
            <con idx="17"/>
            <con idx="18"/>
            <con idx="19"/>
         </other>
      </constraints>
   </optimization>
</osol>
```

Figure 2: A sample OSoL file – SPL1.osol (Continued)

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

Figure 3: Creating an `OSInstance` Object

```
class OSInstance{
public:
     OSInstance();
     InstanceHeader *instanceHeader;
     InstanceData *instanceData;
}; //class OSInstance
```

Figure 4: The `OSInstance` class

```
class InstanceData{
public:
     InstanceData();
     Variables *variables;
     Objectives *objectives;
     Constraints *constraints;
     LinearConstraintCoefficients *linearConstraintCoefficients;
     QuadraticCoefficients *quadraticCoefficients;
     NonlinearExpressions *nonlinearExpressions;
}; // class InstanceData
```

Figure 5: The `InstanceData` class

**Schema complexType** | **In-memory class**

```
<xs:complexType name="Variables">  <-------------------------------------------->  class Variables{
                                                                                  public:
  <xs:sequence>                                                                     Variables();
    <xs:element name="var" type="Variable" maxOccurs="unbounded"/>  <----------->   Variable *var;
  </xs:sequence>
  <xs:attribute name="numberOfVariables" type="xs:nonnegativeInteger"
              use="required"/>  <--------------------------------------------->   int numberOfVariables;
</xs:complexType>                                                                 }; // class Variables


<xs:complexType name="Variable">  <--------------------------------------------->  class Variable{
                                                                                  public:
                                                                                    Variable();
  <xs:attribute name="name" type="xs:string" use="optional"/>  <---------------->   string name;
  <xs:attribute name="type" use="optional" default="C">  <---------------------->   char type;
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="C"/>
        <xs:enumeration value="B"/>
        <xs:enumeration value="I"/>
        <xs:enumeration value="S"/>
        <xs:enumeration value="D"/>
        <xs:enumeration value="J"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="lb" type="xs:double" use="optional" default="0"/>  <------>   double lb;
  <xs:attribute name="ub" type="xs:double" use="optional" default="INF"/>  <---->   double ub;
</xs:complexType>                                                                 }; // class Variable
```

**OSiL elements**

```
<variables numberOfVariables="2">
   <var lb="0" name="x0" type="C"/>
   <var lb="0" name="x1" type="C"/>
</variables>
```

**In-memory objects**

```
OSInstance *osinstance;
osinstance->instanceData->variables->numberOfVariables=2;
osinstance->instanceData->variables->var=new Variable*[2];
osinstance->instanceData->variables->var[0]->lb=0;
osinstance->instanceData->variables->var[0]->name="x0";
osinstance->instanceData->variables->var[0]->type= 'C';
osinstance->instanceData->variables->var[1]->lb=0;
osinstance->instanceData->variables->var[1]->name="x1";
osinstance->instanceData->variables->var[1]->type= 'C';
```

Figure 6: The `<variables>` element as an `OSInstance` object

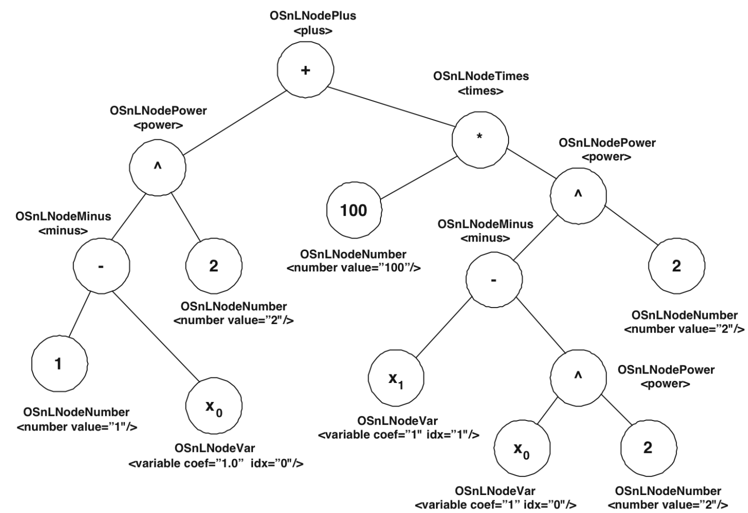


Figure 7: Conceptual expression tree for the nonlinear part of the objective (**??**).

```
double OSnLNodePlus::calculateFunction(double *x){
   m_dFunctionValue =
      m_mChildren[0]->calculateFunction(x) +
      m_mChildren[1]->calculateFunction(x);
   return m_dFunctionValue;
} //calculateFunction
```

Figure 8: The function calculation method for the `plus` node class with polymorphism

# Index