COIN-OR Tools for Stochastic Programming

Michal Kaut

This note has originally been written for a collection "*On Selected Software for Stochastic Programming*", published by MatfyzPress, the publishing house of the Faculty of Mathematics and Physics of the Charles University in Prague. The booklet is aimed for students some familiarity with OR and presents in each chapter one piece of software that can be used for modelling and/or solving of stochastic-programming problems.
The only difference between this note and the published version (apart from a couple of corrected typos) is that it has been made stand-alone, by removing all references to other chapters. In particular, a complete formulation of the example used throughout the note has been added, since it originally resided in Chapter 1 of Kopa (2008)—see Appendix A.

An updated version of the example combining FlopC++ and SMI libraries, presented in Section 3.4, can be found in Kaut, King, and Hultberg (2008). The updated C++ source is also included with the COIN-OR SMI library as example `investment.cpp`.

This note should be cited as: Michal Kaut. COIN-OR Tools for Stochastic Programming. In Miloš Kopa, editor, *On Selected Software for Stochastic Programming*, Chapter 6, pages 88–116, MatfyzPress, Prague, 2008. ISBN 978-80-7378-069-2.

COIN-OR Tools for Stochastic Programming

Michal Kaut*

December 2007

The **Co**mputational **In**frastructure for **O**perations **R**esearch (COIN-OR, or simply COIN) is a collection of open-source projects developing software for the operations research (OR) community. According to the COIN-OR web page[1], the goal of the project is "*to create for mathematical software what the open literature is for mathematical theory*". Most of the projects are released until the *Common Public License* (CPL), an open-source software license published by IBM and approved by the Open Source Initiative and Free Software Foundation (but not compatible with GPLv2). The COIN-OR project is managed by a non-profit educational and scientific foundation called *The COIN-OR Foundation, Inc.*

At the moment of writing, the COIN-OR web site lists 29 projects covering many different aspects of OR – and there are several more projects that are not listed, because they are in an early stage of development. The current projects include solvers for most of the common type of problems: LP, (M)IP, NLP (convex and global) and even MINLP (Mixed Integer NonLinear Programming), all freely available. Many of the solvers can be used from withing AMPL and GAMS (the latter using the *GAMSlinks* project[2]).

In the rest of this note, we will focus on the following two projects: SMI (a **s**tochastic **m**odeling **i**nterface) and FlopC++ (an algebraic modeling language embedded in C++). Like most of the COIN-OR projects, also SMI and FlopC++ are written in C++. While it is fully possible to use the above mentioned stand-alone solvers without any knowledge about C++, some level of programming experience is required to understand the examples presented in this note.

It is also important to realize that all the presented example show just one possible way of doing things; there are other ways, some of which are likely to be better than the presented one. This is especially true in the cases where we mix several approaches in one code, to show the different possibilities, something one would never do in a real code. In addition, we would normally have the object descriptions in separate files and use header files, something we omitted here for the sake of brevity.

1 General concepts

Before we can proceed to the package description, we need to explain some underlying COIN-OR concepts, that are used throughout the COIN-OR collection.

1.1 Obtaining and installation

The easiest way to download and test some of the projects is to use a precompiled binaries, provided by the *CoinBinary* project[3]. Unfortunately, the two projects we are interested in are not (yet) supported,

*Molde University College, P.O. Box 2110, NO-6402 Molde, Norway; email: `michal.kaut@himolde.no`.

[1]See `http://www.coin-or.org/`

[2]See `https://projects.coin-or.org/GAMSlinks`

[3]See `https://projects.coin-or.org/CoinBinary`

so we have to compile the libraries manually.

While it is possible to directly download all the source files[4], the recommended way of obtaining the sources is to use *Subversion*[5] (SVN). This way, one always gets all the dependencies of the package as well. For Windows users, the easiest way is to use *TortoiseSVN*[6], which integrates into the Windows Explorer so there is no need to know the SVN commands. More info about obtaining the source code via SVN can be found in the COIN-OR FAQ page[7].

There are two ways of compiling and installing the COIN-OR projects, depending on the platform used. On POSIX-compatible systems (UNIX, Linux and Windows using Cygwin[8] or MSYS[9]), most of the projects are built and installed by issuing the following sequence of commands from the root directory:

1. `./configure`
2. `make`
3. `maketest`
4. `makeinstall`

Eventual deviations from the procedure are given at the projects' web pages. The projects can be further configured, for example to use external solvers (like CPLEX or GLPK), to specify a build directory, or to build a debug version of the libraries. More information can be found in the "user configure" section of the BuildTools project[10].

For users of Microsoft Visual Studio, many projects—including the two we are interested in—provide project/solution files in their distributions. For more information, see the MSVisualStudio project[11].

All the presented procedures and results have been tested on Windows using MSYS and MinGW[12], a Windows port of the GNU Compiler Collection (gcc), with gcc versions 3.4.5 and 4.2.1. Note that MSYS was used only to configure and build the COIN-OR projects. All the presented codes have been written in an IDE (integrated development environment), in our case Code::Blocks[13], an open source, cross platform IDE for C++.

1.2 Documentation

Most—if not all—COIN-OR packages are configured to generate Doxygen[14] HTML documentation, by issuing `make doxydoc` from the installation directory. Many projects have the Doxygen documentation also available at their project web pages. Since the Doxygen documentation includes all the definitions and syntax, often with additional comments, we will not explain the syntax here.

In addition to the Doxygen documentation, some projects have some kind of user documentation and/or tutorials at their web pages. Most projects also include some examples, which might be the best way of learning the basic usage.

[4]See `http://www.coin-or.org/download/source/`
[5]See `http://subversion.tigris.org/`
[6]See `http://tortoisesvn.tigris.org/`
[7]See `http://www.coin-or.org/faqs.html#ObtainSrcCode`
[8]See `http://www.cygwin.com/`
[9]See `http://www.mingw.org/msys.shtml`
[10]See `https://projects.coin-or.org/BuildTools/wiki/user-configure`
[11]See `https://projects.coin-or.org/MSVisualStudio`
[12]See `http://www.mingw.org/`
[13]See `http://www.codeblocks.org/`
[14]See `http://www.doxygen.org/`

1.3 COIN-OR Open Solver Interface (OSI)

OSI[15] is a “uniform API for interacting with callable solver libraries” for LP and (M)IP solvers. This allows the user to write a solver-independent code, where changing the solver can be as easy as changing two lines in the code. In addition to the COIN-OR solvers, OSI includes interface for the following solvers:

CPLEX	`http://www.ilog.com/`
FortMP	`http://www.optirisk-systems.com/`
GLPK	`http://www.gnu.org/software/glpk/`[16]
MOSEK	`http://www.mosek.com/`
OSL	no longer developed
SoPlex	`http://www.zib.de/Optimization/Software/Soplex/`
Xpress-MP	`http://www.dash.co.uk/`

For each of the above solvers, there is an associated OSI class derived from an abstract base class `OsiSolverInterface`. Hence, to use CPLEX, we create an object of a derived class called `OsiCpxSolverInterface`. If we later decide to switch to, for example, CLP (the **C**OIN-OR **L**inear **P**rogramming solver), all we need to do is to change the class to `OsiClpSolverInterface` and include an appropriate header file.

The `OsiSolverInterface` class provides all the methods needed for building an LP/MIP model and communicating with solvers: adding and deleting of rows and columns, setting bounds, solving and resolving, accessing the solution, etc.

2 Introducing the libraries

2.1 SMI

SMI stands for **S**tochastic **M**odeling **I**nterface and is meant to be an interface for stochastic-programming models. According to the project web page[17], the implemented features include

- a scenario tree structure for multiperiod stochastic data
- an implementation of a Stochastic MPS (SMPS) reader
- interfaces for generating scenario trees from paths and from discrete random variables
- generating an OSI object with the deterministic equivalent problem
- parsing the solutions by stage and scenario.

The most important missing feature is a stochastic solver; so far, the only way to solve the stochastic program represented by an SMI object is to create an OSI object with its deterministic equivalent and use some of the OSI solvers to solve it. Despite this obvious drawback, there are several scenarios where using SMI brings some advantage over using deterministic tools:

- Since SMI contains an SMPS reader, we can easily write a solver for stochastic programs in SMPS format.
- SMI provides a very natural way of building a stochastic program, since it operates with stages and scenarios.
- Even if we can only solve a deterministic equivalent problem, we can still use SMI to give us access to the solution by stage and scenario.

[15]See `https://projects.coin-or.org/Osi/`
[17]See `http://www.coin-or.org/projects/Smi.xml`

The structure of an SMI object is closely related to the SMPS format: we start by creating a core model and then associate its variables and constraints to stages. The scenario tree is then created in an SMPS-like fashion by adding scenarios one by one, specifying their parent scenario, the stage they branch from the parent scenario and the data in which they differ from it. Note that there SMI provides also different ways of creating the scenario tree, by they are not covered here.

Main classes

The most important classes in SMI are:

`SmiCoreData` Class representing the deterministic core problem, including the stage information on variables and constraints.

`SmiScnModel` "Scenario Model Class" - the main class representing the stochastic model. The most important methods are:

- **`readSmps`** Reads the model from three SMPS files (core, time, stoch).
- **`generateScenario`** Adds one scenario, using a previously created `SmiCoreData` object.
- **`processDiscreteDistributionIntoScenarios`** Generate scenarios from a discrete distribution.
- **`getObjectiveValue`** Gets the current objective function value in a given scenario.
- **`getScenarioProb`** Gets the probability of a given scenario.
- **`getLeafNode`** Gets the leaf node of a given scenario.
- **`getRootNode`** Gets the root node of the scenario tree.

`SmiScnNode` Part of the model represented by one node of the scenario tree. The `getLeafNode` and `getRootNode` methods of `SmiScnModel` return a reference to `SmiScnNode`.

For a complete list and syntax, please consult the Doxygen documentation[18].

2.2 FlopC++

FlopC++ stands for ***F**ormulation **o**f **L**inear **O**ptimization **P**roblems in **C++*** and is an algebraic modelling language implemented as a C++ library. In other words, it makes it possible to specify a linear model in C++ in a way which is very close to the modelling languages like AMPL or GAMS, with the additional advantage of having the full strength of the C++ language for manipulating (solving, updating, resolving) the model once it has been created. Or, as the FlopC++ web page[19] puts it, "*FLOPC++ can be used as a substitute for traditional modelling languages, when modelling linear optimization problems, but its principal strength lies in the fact that the modelling facilities are combined with a powerful general purpose programming language. This combination is essential for implementing efficient algorithms (using linear optimization for subproblems), integrating optimization models in user applications, etc.*"

[18]See `http://www.coin-or.org/Doxygen/Smi/annotated.html`

[19]See `https://projects.coin-or.org/FlopC++`

Main classes

In the case of FlopC++, we can afford to be quite brief in the description of the classes, as there is more documentation than in the case of SMI: the Doxygen[20] documentation includes quite a lot of information about all the main classes and there is also a "user documentation" page[21] explaining the main concepts. As a result, we only list the main classes with a selection of methods here:

`MP_model` with methods `minimize`, `getStatus`, etc.

`MP_set` with methods `cyclic`, `size`, etc.

`MP_subset` with methods `insert`, `size`, etc.

`MP_index` , a class representing an index;

`MP_data` with methods `value`, `initialize`, etc.

`MP_variable` with methods `integer`, `binary`, `level`, etc. and attributes `upperLimit` and `lowerLimit`.

`MP_constraint` with attributes `left`, `right`, `sense`, etc.

`MP_expression` , a class representing a linear expression.

A careful reader may notice that the class `MP_subset` is not listed in the "Public Interface" group. Indeed, this class is marked as "*for internal use*", but we list it here because we find it very useful. However, there is a good reason the class is not meant for public use: one has to very careful about indexing. We will thus demonstrate its use on a small example, presented in Figure 1. There, we construct a set

```
 9  int i;
10  MP_set A(5);                       // sets A = {0,...,4}
11  MP_index a, b;
12  MP_subset<1> B(A);
13  B.insert(2); B.insert(3);          // sets B = {2, 3}
14  B.display("set B");
15  MP_data D(B);                      // data indexed on B
16  for (i = 0; i < B.size(); i++)
17    D(i) = 5*(i+1);                  // sets D to 5 and 10
18  D.display("data D");
19
20  for (i = 0; i < A.size(); i++) // print indices of elements of A in B
21    cout << "B(" << i << ") = " << B(i) << endl;
22
23  MP_variable x(A);            // variable defined on set A
24
25  MP_expression objF;          // linear expression (later used as objective)
26  objF = sum(B, D(B) * x(B));
27  objF = sum(B(A), D(B) * x(A));
28  objF = sum(B(a), D(a) * x(a));
29  objF = sum(B(a), D(B(a)) * x(a));
30  objF = sum(B(a), D(B) * x(a));
```

Figure 1: Example demonstrating the use of subsets in FlopC++.

`A` with 5 elements $\{0,\ldots,4\}$ and its subset `B` $= \{2,3\}$. This is confirmed by the output from line 14:

[20] Available at `http://www.coin-or.org/Doxygen/FlopC++/`; The best starting place is probably the "Public Interface" modul at `http://www.coin-or.org/Doxygen/FlopC++/group__PublicInterface.html`

[21] See `https://projects.coin-or.org/FlopC++/wiki/DocumentationPage`

```
set B
2
3
```

Then we define a data object D indexed on the subset B and set its values to $\{5, 10\}$. The output from line 18 is then

```
data D
0   5
1   10
```

Note that the indices are *not* 2 and 3, i.e. the data object "does not know" about the elements of B. Now, let us have a look at lines 20–21 and their output:

```
B(0) = -2
B(1) = -2
B(2) = 0
B(3) = 1
B(4) = -2
```

This demonstrates nicely the meaning of `B(i)`: it is the index of element $\mathtt{i} \in \mathtt{A}$ in the set B, with "-2" meaning $\mathtt{i} \notin \mathtt{B}$. In other words, the output means that B has two elements, 2 and 3.

The rest of the example illustrates the importance of correct indexing an `MP_expression`. We start by defining a variable $\mathtt{x} \in \mathtt{A}$ and an `MP_expression` object `objF`. The goal is to set this to a sum of `D(a)*x(a)` for all $\mathtt{a} \in \mathtt{B}$, i.e. `5 x(2)+10 x(3)`. The example includes five different ways of creating the formula, two of which are wrong (but they do, unfortunately, compile and even run, they just produce wrong results): the code at line 26 produces `5 x(0)+10 x(1)`. The reason is that we index x on the set B, while it is defined on A. This is correct at l. 27, where we connect the two sets in the first arguments of the `sum` and then index the data and the variable on the appropriate sets.

A possibly more readable alternative is to use an `MP_index` variable, but one has to be careful also in this case. For example, the version on line 28 does not create any expression at all, because we try to index D on an index a running over set A. There are (at least) two possible remedies, we can either re-cast the index into set B as on l. 29, or simply use the index only for the set A, as on line 30.

3 Formulation of the investment example

In this section, we formulate the investment example presented in Appendix **??** using the COIN-OR libraries and check that we get the same solution. We start by using each of the libraries separately and then show what can be gained by joining their powers.

3.1 Investment example using Coin-OR SMI

We will present the code piece-by-piece, introducing the objects and methods as we need them. The first thing we need to do is to create an OSI object with the core model. This is implemented in method `create_det_model`, presented in Figure 2. The OSI object is built step-by-step, starting with the LP matrix A (lines 179–202), then right-hand side b (l. 204–207) and finally the objective function c (l. 209–210). Note that since the values of stochastic variables in the core do not really matter, we have used the expected values, so we in fact get an OSI object representing the expected-value problem.

With this method, we can start building the main code, as shown in Figure 3. At the start of the code, line 3 includes the main SMI header file, line 6 includes the header for OSI interface to the COIN-OR solver CLP and line 7 defines a macro for the OSI object we are going to use in the code. Note that lines 6 and 7 are the only ones that have to be changed if we want to use another solver.

Once we have the OSI representation of the core model, we only need the stage information for the variables and constraints (l. 29–31), before we can create the `SmiCoreData` object representing the core at line 34.

Once we have the core model, we can start building the stochastic model. All the data needed for this are at lines 44–58 – note that this would normally be in a separate data file. The `scenDataRet` matrix includes the scenario returns, where we only store returns at the nodes that differ from the parent (in our case previous) scenario. The `branchStage` vector then includes the stages at which each scenario branches from its parent; Note that this has to be equal to one for the first scenario. Once we have all the data in place, we can build the `SmiScnModel` object with the complete stochastic model, by adding scenarios one-by-one (l. 64–74). The only missing piece is the `add_scenario` method, presented in Figure 4. The most important part of the method is the construction of a matrix including all the elements that differ from the parent scenario. Just as in the core model, we specify the matrix elements, using the `[row,column,value]` triplets. The row and column indices for branching at stage 2 are specified by the arrays at lines 238–239. For branching at later stages, we just use part of these arrays, with the starting index computed at line 240. The matrix of differences is then created at lines 243–248.

The scenario is then added using the `generateScenario` method at lines 250–255. The four null pointers represent respectively the vectors of column lower- and upper bounds, objective function coefficients and row lower- and upper bounds that differ from the parent scenario. Since the scenarios in our case differ only in the matrix A, these are empty vectors.

After the end of the code in Figure 3, we have a complete SMI object, so the only thing that remains is to generate an OSI object with the deterministic equivalent, solve it and report some results. This is quite easy to do, as illustrated in Figure 5.

Even if we use a deterministic solver, we can still get information about the solution on the scenario tree, using the SMI object. This is illustrated in Figure 6, with a code that traverses the scenario trees by scenarios and report scenario probabilities and objective values, plus the development of wealth from the root to the leaf of each scenario.

This completes the code. Note that to build the code we need to link the following COIN-OR libraries: Smi, OsiClp, Clp, Osi, CoinUtils. The content of the first four libraries should be clear

```
177 void create_det_model(OsiSolverInterface &modelOSI)
178 {
179   /* Matrix A, using the CoinPackedMatrix (sparse matrix) format
180   Non-zero elements given by three vectors: row and column indices + values
181   (not the most efficient way, but easy to read) */
182   int rowIndx[] = { // Row indices of the elements
183     0, 0,
184     1, 1, 1, 1,
185           2, 2, 2, 2,
186                 3, 3, 3, 3
187   };
188   int colIndx[] = { // Column indices of the elements
189     0, 1,
190     0, 1, 2, 3,
191           2, 3, 4, 5,
192                 4, 5, 6, 7
193   };
194   double elem[] = { // Values of the non-zero elements (expected value problem)
195      1,       1,
196     -1.155, -1.150,  1,       1,
197                     -1.123, -1.148,  1,       1,
198                                     -1.108, -1.140, -1,  1
199   };
200   CoinBigIndex nmbElem = 14; // Number of non-zero elements
201   bool storeByCols = false;   // Internal parameter, should be set to false
202   CoinPackedMatrix A(storeByCols, rowIndx, colIndx, elem, nmbElem);
203
204   // Bounds for variables are not needed, as the default is [0,inf]
205   // Bounds on the rows, i.e. RHS (equality constraints, so LB=UB)
206   double rowLB[] = {55, 0, 0, -80};
207   double rowUB[] = {55, 0, 0, -80};
208
209   // Objective function
210   double objCoef[] = {0, 0, 0, 0, 0, 0, 1.3, -1.1};
211
212   // Load the problem to the provided OSI handle
215   modelOSI.loadProblem(A, NULL, NULL, objCoef, rowLB, rowUB);
223 }
```

Figure 2: Method `create_det_model` for the SMI-based approach. Line numbers correspond to the complete source file, the skipped line numbers are lines with extra comments, tests or output.

by now, the last one includes many general COIN-OR objects, for example the sparse matrix object `CoinPackedMatrix`. If we want to use another solver, the OsiClp and Clp libraries have to be replaced accordingly.

```cpp
#include "SmiScnModel.hpp"

// Change these two lines to use a different solver
#include "OsiClpSolverInterface.hpp"
#define OSI_SOLVER_INTERFACE OsiClpSolverInterface

using namespace std;

int main()
{
  // First, we build the CORE problem, i.e. the deterministic version
  OSI_SOLVER_INTERFACE detModel;
  create_det_model(detModel); // Create the deterministic model

  // Next, add information about stages and create the SMI core data object
  int nmbStages = 4;                                 // Number of stages
  int colStages[] = {0, 0, 1, 1, 2, 2, 3, 3}; // Stages of columns/variables
  int rowStages[] = {0, 1, 2, 3};                // Stages of rows/constraints

  SmiCoreData stochCore(&detModel, nmbStages, colStages, rowStages);

  // Vector of the matrix elements that differ from the prev. scenario
  double scenDataRet[][6] = {
    { -1.25, -1.14, -1.21, -1.17, -1.26, -1.13 }, // scen. ending at node 7
    {                             -1.07, -1.14 }, // scen. ending at node 8
    {               -1.07, -1.12, -1.25, -1.15 }, // scen. ending at node 9
    {                             -1.06, -1.12 }, // scen. ending at node 10
    { -1.06, -1.16, -1.15, -1.18, -1.05, -1.17 }, // scen. ending at node 11
    {                             -1.06, -1.15 }, // scen. ending at node 12
    {               -1.06, -1.12, -1.05, -1.14 }, // scen. ending at node 13
    {                             -1.06, -1.12 }  // scen. ending at node 14
  };
  int branchStage[] = { 1, 3, 2, 3, 1, 3, 2, 3 }; // Branching stage
  int nmbScen = 8;                                 // Number of scenarios
  double scenProb = 1.0 / nmbScen;                 // Scenarios' probability

  // Create the SMI stochastic model object
  SmiScnModel stochModel;

  for (SmiScenarioIndex sc = 0; sc < nmbScen; sc++) {
    int parentScen = (sc == 0 ? 0 : sc - 1); // Parent scenario
    add_scenario(&stochModel, &stochCore, scenDataRet[sc],
                 branchStage[sc], parentScen, scenProb);
  }
```

Figure 3: Start of the `main` code of the SMI-based approach.

```
int add_scenario(SmiScnModel *ptStochModel, SmiCoreData *ptCoreModel,
                 double *matrixElem, int brStage, int parScen, double prob)
{
  // Row and column indices, for branching at stage 2
  int rowDifIndx[] = {1, 1, 2, 2, 3, 3};
  int colDifIndx[] = {0, 1, 2, 3, 4, 5};
  int indxStart = 2 * (brStage-1); // Where to start reading the index vectors
  int nmbElem = 6 - indxStart;     // Number of elements in the matrix

  // Now create the matrix containing the elements that differ from the parent
  CoinPackedMatrix ADiff(false,
                         &(rowDifIndx[indxStart]), &(colDifIndx[indxStart]),
                         matrixElem, nmbElem);

  // Add the scenario to the SMI object.
  int scenIndx = ptStochModel->generateScenario(ptCoreModel, &ADiff,
                                                NULL, NULL, NULL, NULL, NULL,
                                                brStage, parScen, prob);
  return scenIndx; // Return the index of the new scenario
}
```

Figure 4: Method `add_scenario` for the SMI-based approach.

```
  // Attach a solver to the stochastic model
  stochModel.setOsiSolverHandle(*new OSI_SOLVER_INTERFACE());

  // 'loadOsiSolverData()' loads the deterministic equivalent into an
  // internal OSI data structure and returns a handle (pointer) to it.
  OsiSolverInterface *ptDetEqModel = stochModel.loadOsiSolverData();

  // 'ptDetEqModel' now includes the deterministic equivalent
  ptDetEqModel->writeMps("investment.det-equiv");
  cout << endl << "Solving the deterministic equivalent:" << endl;
  ptDetEqModel->initialSolve();
  cout << endl << "The deterministic equivalent model has "
       << ptDetEqModel->getNumCols() << " variables and "
       << ptDetEqModel->getNumRows() << " constraints." << endl;
  cout << "Optimal objective value = " << ptDetEqModel->getObjValue()
       << endl;
```

Figure 5: `main` code of the SMI-based approach – solving the deterministic equivalent.

```
  cout << endl << "The stochastic model has " << stochModel.getNumScenarios()
       << " scenarios." << endl;

  // We will calculate the wealth at each node of the tree, plus the obj. value
  vector<double> nodeWealth(nmbStages, 0);
  double objValue = 0.0;

  // Get vectors of variable values and their objective coefficients
  const double *ptOsiSolution = ptDetEqModel->getColSolution();

  // Compute the wealth at each node, by traversing the tree from leafs up
  for (SmiScenarioIndex sc = 0; sc < nmbScen; sc ++) {
    // Get the leaf node of scenario sc:
    SmiScnNode *ptNode = stochModel.getLeafNode(sc);
    int nodeStage = nmbStages;

    double scProb = ptNode->getModelProb(); // probability of the leaf
    double scenObjVal = stochModel.getObjectiveValue(sc); // objective value
    objValue += scProb * scenObjVal;
    printf ("scen %d: prob = %.3f  obj =%7.2f", sc+1, scProb, scenObjVal);

    // This loop traverses the tree, from the leaf to the root
    while (ptNode != NULL) {
      // info about columns of ptNode in the OSI model (ptDetEqModel)
      int startColInOsi = ptNode->getColStart();
      int nmbColsInOsi = ptNode->getNumCols();

      // Loop through the variables corresponding to this node
      double wealth = 0.0;
      if (nodeStage < nmbStages) {
        // The node is not a leaf -> just sum up the variables
        for (int j = startColInOsi; j < startColInOsi + nmbColsInOsi; j++) {
          wealth += ptOsiSolution[j];
        }
      } else { // Special rules for the leaves
        // get RHS of the only constraint for this node (get LB, check LB=UB)
        double capTarget = -ptDetEqModel->getRowLower()[ptNode->getRowStart()];

        // In the leaves, the wealth is computed as: -w + y + capTarget
        wealth = -ptOsiSolution[startColInOsi]
                 + ptOsiSolution[startColInOsi+1]
                 + capTarget;
      }
      nodeWealth[nodeStage-1] = wealth;

      // Get the parent node (Root will return NULL, stopping the loop)
      ptNode = ptNode->getParent();
      nodeStage--;
    }

    printf (";  wealth:");
    for (int t = 0; t<nmbStages-1; t++)
      printf ("%6.2f ->", nodeWealth[t]);
    printf ("%6.2f\n", nodeWealth[nmbStages-1]);
  }

  printf ("%15s Total obj = %7.3f\n", "", objValue);
  return 0;
}
```

Figure 6: `main` code of the SMI-based approach – reporting the scenario solutions.

3.2 Investment example using FlopC++ – A node-based approach

In this section, we present a node-based formulation of the problem using the FlopC++ library. This is basically a FlopC++ reformulation of the the AMPL model from Appendix A. Just as in the previous section, we will build the code step-by-step, starting with the initialization part presented in Figure 7. In addition to the standard `include` and `using` parts, we see also two versions of a function called `get_pred`. This function calculate the predecessor of a given node, using the fact that we have a regular binary tree. The version at lines 15–18 operates on numbers, while the version at lines 19–23 operates on FlopC++ class `MP_index`. We really need only one of them, the point is to illustrate different ways of doing things: while the latter formulation has an advantage that it can be used directly in indexing expressions, it is also limited in the way that one can only use operators and functions overloaded in the `MP_index` class (such as `floor`).

```
 5 #include "flopc.hpp"
 6
 7 // Change these two lines to use a different solver
 8 #include "OsiClpSolverInterface.hpp"
 9 #define OSI_SOLVER_INTERFACE OsiClpSolverInterface
10
11 using namespace std;
12 using namespace flopc;
14
15 // Predecessor of a given node; input and output as a number
16 int get_pred(const int node) {
17   return floor((node-1) / 2);
18 }
19 // Predecessor of a given node; input and output as an MP_index
21 MP_index_exp get_pred(MP_index &node) {
22   return floor((node-1) / 2);
23 }
```

Figure 7: Initialization part for the FlopC++ version.

In a modelling language like AMPL, one usually starts with the model and then provides the data. In FlopC++, however, we must have the data when we create the model's objects. We thus start the `main` part of the code by providing the input data, see Figure 8. Note that we include the data only to make the example shorter; In a real-life situation the data would be read from an external data file. In other words, the fact that we need data to initialize the model's objects does *not* mean that we need the data at compile time, so it is fully possible to write data-independent models.

The next step is to initialize the `MP_model` object and attach the sets and data, see Figure 9. At lines 53–55, we define the `MP_model` object and attach the OSI solver defined at line 9. At lines 62–70 we define all the sets, indices and subsets, which are then filled at lines 72–94. Note that the tree structure is described by set `PREDS`, which is a subset of `NODES`×`NODES`, specifying a set of predecessors for each node $n \in$ `NODES`. This might seem overly complicated, since we know that a node has at most one predecessor, but it is nevertheless the easiest way we know of. It is also a very general way, capable of model even any trees – all that would be needed is to replace the line 94 by an appropriate description of the tree structure.

At lines 97–99, we define the probabilities as a parameter indexed on set `LEAVES`. Note how we use the method `size()` to get the cardinality of the set `LEAVES`. Finally, at lines 123–125 we define the parameter `Return` and initialize its values by providing a reference to the data defined at lines 36–51.

```
25 int main()
26 {
27   // DATA - This would be normally read from some external file
28   enum {stocks, bonds, nmbAssets}; // assets to invest to; sets nmbAssets = 2
29   int Root = 0;         // root node
30   int nmbNodes = 15;  // nodes are 0..14
31   int firstLeaf = 7;  // leaves are 7..14
32
33   double InitBudget = 55;
34   double CapTarget = 80;
35
36   double retData[][nmbAssets] = {
37     {1.25, 1.14},
38     {1.06, 1.16},
39     {1.21, 1.17},
40     {1.07, 1.12},
41     {1.15, 1.18},
42     {1.06, 1.12},
43     {1.26, 1.13},
44     {1.07, 1.14},
45     {1.25, 1.15},
46     {1.06, 1.12},
47     {1.05, 1.17},
48     {1.06, 1.15},
49     {1.05, 1.14},
50     {1.06, 1.12}
51   };
```

Figure 8: Start of the `main` code of the FlopC++ approach.

Now we are ready to start defining the main model objects, i.e. variables, constraints and the objective, as shown in Figure 10. We start with the easy part, variables (l. 127–135) and the objective function (l. 137–139). Since the default bounds on variables are $[0,\infty]$, we do not have to use the `lowerLimit` and `upperLimit` methods. Note the syntax of the `sum` function used for the objective function, illustrating indexing with set names – this might be a bit strange for AMPL users, while GAMS users should feel at home.

Finally, we have to define the constraints, see lines 141–166. There are three types of constraints: the `initial_budget` constraint in the root, `final_balance` in the leaves and `cashflow_balance` in all the nodes between. The easiest of them is `initial_budget`, given by a simple sum. Here we use an alternative syntax, using an index `a` inside the `sum`. Note that the index has to be a previously defined object of type `MP_index`, see line 70 in Figure 9.

The `cashflow_balance` are more complicated, since they illustrate a few things we have not seen before:

- To get the values of `Return` from the predecessor node of node `n`, we sum over all the predecessors: since `n` is an index of the constraint, `Pred(n,pr)` inside a `sum` will iterate over all values of `pr` from `NODES`, such that `(n,pr)` is in the set `PREDS` – so we would better be sure there is only one for each node (except for the root).
- We use the multiplication symbol "`*`" to index a `sum` on a product of two sets.
- We have to use `FUT_NODES(n)` as an index for `Return`, instead of using only `n`. The reason is that `n` is defined as an index of the node inside the set `MID_NODES`, which might be different

```
// initialize the object for the model
MP_model &model = MP_model::getDefaultModel();
model.setSolver(new OSI_SOLVER_INTERFACE);

// Sets
MP_set NODES(nmbNodes);
MP_subset<2> PREDS(NODES,NODES);  // A 'set of predecessors' for each node
MP_subset<1> FUT_NODES(NODES);    // All nodes except Root
MP_subset<1> LEAVES(NODES);        // Leaves of the tree
MP_subset<1> BRANCH_NODES(NODES); // All nodes except leaves
MP_subset<1> MID_NODES(NODES);    // All nodes except Root and leaves
MP_set ASSETS(nmbAssets);
MP_index n, a, pr; // for use with other sets, see bellow

// Fill the subsets; shows both the direct syntax and using an MP_index
forall (NODES.such_that(NODES != Root), FUT_NODES.insert(NODES));
forall (NODES(n).such_that(n >= firstLeaf), LEAVES.insert(n));
forall (NODES(n).such_that(n < firstLeaf), BRANCH_NODES.insert(n));
forall (NODES(n).such_that(n != Root && n < firstLeaf),
        MID_NODES.insert(n));

// Data - tree structure
forall (NODES(n).such_that(n != Root), PREDS.insert(n, get_pred(n)));

// Probabilities of the leaves
MP_data Prob(LEAVES);
Prob(LEAVES) = 1.0 / LEAVES.size();

// Init data object and read data; alt.: MP_data Return(&retData[0][0]);
MP_data Return(FUT_NODES, ASSETS);
Return.value(&retData[0][0]); // use table retData, startin from el.[0][0]
```

Figure 9: `main` code of the FlopC++ approach – making the `MP_model` object

from the index of `n` inside the set `FUT_NODES` – which is what `FUT_NODES(n)` means. The same is valid for the index `pr` and variable `x`, since `pr` runs on the complete set `NODES` and `x` is defined on the subset `BRANCH_NODES`.

- Finally, note that in this case we have to use at least one `MP_index`, since we have two indices (`x` and `pr`) running on the same set (`NODES`). In other words, we could replace one of them by `NODES` and replace `a` by `ASSETS`, but it would make the constraint more difficult to understand.

The `final_balance` constraints are very similar, except that in this case we use the `MP_index` version of the `get_pred` function to get the predecessor directly, without the `PREDS` set. We repeat that this approach would not work for general unbalanced trees.

The model is now ready, so we need only to build and solve it, as shown in Figure 11. First, we set the `MP_expressionobjFunc` as an objective function and then call the `attach()` method. This creates an OSI object with the model and attaches it to the algebraic representation in the `MP_model` object `model`. The OSI object can be accessed directly using the overloaded "`->`" operator. This is illustrated on line 173, where we call the OSI method `writeMps` to write an MPS representation of the current model. Finally, we solve the model at line 176. It should be noted that the lines 169, 170 and 176 can be replaced by a single command "`model.minimize(objFunc);`", which would do exactly the same.

```
// Variables
MP_variable
  x(BRANCH_NODES, ASSETS), // investment
  w(LEAVES),                // shortage
  y(LEAVES);                // surplus

// Objective - define it as an expression; later say it should be minimized
MP_expression objFunc
  = sum(LEAVES, Prob(LEAVES) * (1.3 * w(LEAVES) - 1.1 * y(LEAVES)));

// Constraints
MP_constraint
  initial_budget,
  cashflow_balance(MID_NODES),
  final_balance(LEAVES);

initial_budget = sum(ASSETS(a), x(Root, a)) == InitBudget;

cashflow_balance(MID_NODES(n))
  = sum(ASSETS(a)*PREDS(n,pr), Return(FUT_NODES(n), a)
                               * x(BRANCH_NODES(pr), a))
    == sum(ASSETS(a), x(BRANCH_NODES(n), a));

final_balance(LEAVES(n))
  = sum(ASSETS(a), Return(FUT_NODES(n), a)
                   * x(BRANCH_NODES(get_pred(n)), a))
    + w(LEAVES(n)) - y(LEAVES(n)) == CapTarget;
```

Figure 10: `main` code of the FlopC++ approach – defining the variables, objective function and constraints.

```
  // Set the objective; attach the model;
  model.setObjective(objFunc);
  model.attach();

  // Write MPS file
  model->writeMps("investment");

  // Minimize the objective, ie. solve the model;
  model.minimize();

  return 0;
}
```

Figure 11: `main` code of the FlopC++ approach – building and solving the model.

This completes the code. Note that we need to link the following COIN-OR libraries in order to build the code: FlopCpp, OsiClp, Clp, Osi, and CoinUtils. Again, if we want to use another solver, the OsiClp and Clp libraries have to be replaced accordingly.

3.3 Investment example using FlopC++ – A stage-based approach

While the FlopC++ model presented above is, hopefully, reasonably easy to understand, it is only a deterministic formulation with very little information about the stochastic structure of the problem. In this section, we will present a different way of modelling the same problem, using an implicit separation of the problem into stages. The formulation is based on the `stampl.cpp` example from FlopC++ distribution, which has been tailored to fit our problem.

The basic idea of this reformulation is to introduce a `StageNode` object that includes the model at one node of the scenario tree, introducing thus an implicit stage structure into the model. Unfortunately, it also makes the code look less like an mathematical model and more like a C++ code – it would be better to hide as much of the "C++ machinery" from the end user as possible, but this is out of the scope of this text.

Defining the stage and scenario-tree objects

Since there is a difference between the models in the root-, leaf- and middle-nodes, we first define a common ancestor class `StageNode` and then derive the stage-specific classes from it. The definition of the `StageNode` base class is presented in Figure 12. The only model-dependent part of the class are the members[22] at lines 35–38, with the same meaning as before. The tree-structure members at lines 40–43 should be self-explanatory, so we move to the class constructor (l. 45–54). There, if we provide a pointer to the parent node (i.e. if the node is not the root), we have to do two things: we compute the node's total (unconditional) probability as the product of the parent's total probability and the specified conditional probability and then we add the node to the parent's set of children.

The `make_obj_function_` method (lines 58–73) creates the objective function for the node and all this children. It is declared `protected`, because the user should only call it from the root, to create the complete objective function. We see (l. 67–72) that the objective function is defined as the sum of children's objective functions. Generally, this should include the node's objective function as well, but we have objective coefficients only in the leaves. This means that we will have to override this function in the leaf-node object—which is why it is declared `virtual`. The children's objective functions must be defined before we can call the code at line 71, so we have to call `make_obj_function_` on the children first (l. 61–66).

The `RootNode` class, presented in Figure 13, adds the following to the base class `StageNode`:

- constraint `initialBudget`, defined on line 79 and initialized in the constructor of the object (line 84);
- method `make_objective_function`, which is a `public` interface for the `protected` method `make_objective_function_`.

The `MidStageNode` class, presented in Figure 14, is a class for all the nodes between the root and leaves. It adds the following two members to the base class:

- constraint `cashFlowBalance`, again initialized in the constructor of the object;
- data object `Return` – we will come back to it later.

Finally, the `LeafNode` class (Figure 15) for the leaves of the scenario tree, which adds:

- variables `w` and `y`;

[22] Note that it is generally considered a bad programming practice to declare members of a class `public`. Instead, the members should be `private` or `protected` and we should provide get and set methods to access them. This, however, would make the example code significantly longer, which is why we have decided to leave all the members `public`.

```
class StageNode {
  public:
    MP_set ASSETS;                     ///< set of assets
    MP_index a;                        ///< index used in formulas
    MP_variable x;                     ///< The "buy" variable, defined on ASSETS
    MP_expression objFunction;         ///< objective function at this node

    StageNode *ptParent;               ///< pointer to parent node
    vector<StageNode *> children;      ///< pointers to children of this node
    int nodeNmb;                       ///< node number - used only for reporting
    double prob;                       ///< probability (unconditional) of the node

    StageNode(StageNode *ptPred, const int nmbAssets, const int nodeN,
              const double condProb)
    : ASSETS(nmbAssets), x(ASSETS), ptParent(ptPred), nodeNmb(nodeN),
      prob(condProb)
    {
      if (ptParent != NULL) {
        prob *= ptParent->prob;                   // Compute the total probability
        ptParent->children.push_back(this);  // Register with the parent
      }
    }
    virtual ~StageNode(){}

  protected:
    /// Create the objective function expression, recursively for all children
    virtual void make_obj_function_() {
      // Create the objective recursively for all descendants of the node
      for (int i=0; i < (int) children.size(); i++) {
        children[i]->make_obj_function_();
      }
      // No objective at non-leaf nodes, so the objective value at this node
      // and all its descendants is a sum of obj. values of the children.
      objFunction = children[0]->objFunction;
      for (int i=1; i < (int) children.size(); i++) {
        objFunction = objFunction + children[i]->objFunction;
      }
    }
};
```

Figure 12: The `StageNode` class for the stage-based FlopC++ approach.

- constraint `finalBalance`;
- data object `Return`;
- its own version of the `make_obj_function_` method, computing the objective function at the node.

There is an important difference in the implementation of the `Return` member in the two above classes: in `MidStageNode`, the `Return` object is constructed as `Return(ptRetVect,ASSETS)` (line 111). This constructor uses a *shallow copy* of the input array `ptRetVect`, i.e. it only stores the pointer (reference) and does **not** copy the values. This means that if we later change the values to which `ptRetVect` points, the `Return` will change as well—and if we delete the array, the code will most likely crash on calling the `attach` method. The constructor of the `LeafNode` class, on the

```
77 class RootNode : public StageNode {
78   public:
79     MP_constraint initialBudget; ///< initial budget constraint
80
81     RootNode(const int nmbAssets, const double initWealth)
82     : StageNode(NULL, nmbAssets, 0, 1.0)
83     {
84       initialBudget() = sum(ASSETS, x(ASSETS)) == initWealth;
85     }
86
87     /// This is the public interface to the protected \c make_obj_function_()
90     void make_objective_function() {
91       make_obj_function_();
92     }
93 };
```

Figure 13: The `RootNode` class for the stage-based FlopC++ approach.

```
 96 class MidStageNode : public StageNode {
 97   public:
 98     MP_constraint cashFlowBalance; ///< cash-flow balance constraint
 99     MP_data Return;                ///< returns of the assets at this node
100
101     /// Constructs a \c MidStageNode object
108     MidStageNode(StageNode *ptPred, const int nodeN,
109                  const double condProb, double *ptRetVect)
110     : StageNode(ptPred, ptPred->ASSETS.size(), nodeN, condProb),
111       Return(ptRetVect, ASSETS)
112     {
114       cashFlowBalance = sum(ASSETS(a), ptParent->x(a) * Return(a))
115                         == sum(ASSETS(a), x(a));
116     }
117     virtual ~MidStageNode(){}
118 };
```

Figure 14: The `MidStageNode` class for the stage-based FlopC++ approach.

other hand, constructs the `Return` object only as `Return(ASSETS)`, see line 135, and then calls the `value` method to initialize it with the provided value. The `value` method uses a *deep copy* of the input data, i.e. it copies the values `ptRetVect` points at. This is slower than doing the shallow copy, but the `Return` will not be affected by any later change to `ptRetVect`.

Note that the only reason to use both approaches in one code is to show two different ways of initializing an `MP_data` object. In a normal situation, one would choose the options that fits best for the given situation and use it consistently throughout the code.

In addition to the stage-node objects, we have decided to add a simple object describing the shape of the scenario tree, see Figure 16. The class provides the following information:

- the number of nodes `nmbNodes`;
- index of the first leaf `firstLeaf`;
- function `get_parent`, returning an index of the parent of a given node.

```
121 class LeafNode : public StageNode {
122   public:
123     MP_variable w;                ///< shortage variable
124     MP_variable y;                ///< surplus variable
125     MP_constraint finalBalance;   ///< constraint for the final balance
126     MP_data Return;               ///< returns of the assets at this node
127 
128     /// Constructs a \a LeafNode object
133     LeafNode(StageNode *ptPred, const int nodeN, const double condProb,
134              const double *ptRetVect, const double capTarget)
135     : StageNode(ptPred, ptPred->ASSETS.size(), nodeN, condProb),
                Return(ASSETS)
136     {
137       Return.value(ptRetVect); // Copy values from retVect to Return
138       // This shows a formula without using an additional MP_index
139       finalBalance = sum(ASSETS, ptParent->x(ASSETS) * Return(ASSETS))
140                      + w() - y() == capTarget;
141     }
142 
143   protected:
144     /// version of \a make_obj_function_() for the leaves - no recursion
145     void make_obj_function_() {
146       objFunction = prob * (1.3 * w() - 1.1 * y());
147     }
148 };
```

Figure 15: The `LeafNode` class for the stage-based FlopC++ approach.

```
154 class ScenTreeStruct {
155   public:
156     int nmbNodes;     ///< nodes are 0..nmbNodes-1, where 0 is root
157     int firstLeaf;    ///< nodes firstLeaf..nmbNodes-1 are leaves
158 
159     /// Constructs the object
160     ScenTreeStruct(const int nNodes, const int firstL)
161     : nmbNodes(nNodes), firstLeaf(firstL)
162     {}
163     virtual ~ScenTreeStruct(){}
164 
165     /// This function returns the parent of a given node (and 0 for the root)
169     virtual int get_parent(int n) const = 0;
170 };
```

Figure 16: The `ScenTreeStruct` class for the stage-based FlopC++ approach.

Function `get_parent` is kept undefined for the general `ScenTreeStruct` class. In fact, the function is declared `virtual` and set equal to zero, making the `ScenTreeStruct` class an *abstract base class* and forcing any derived class to come with its own implementation of the `get_parent` function.

The only derived class we need in our example is a class for regular binary trees, presented in Figure 17. Note that the formula for `get_parent` (line 180) uses the fact that in C++ (and C as well), division of two integers is again an integer, with the result rounded down.

```
/// Class for balanced binary trees
class BinTreeStruct : public ScenTreeStruct {
  public:
    /// Constructs the object - 2^T-1 nodes, first leaf is 2^(T-1)-1
    BinTreeStruct(const int nmbStages)
    : ScenTreeStruct(pow(2.0, nmbStages) - 1, pow(2.0, nmbStages-1) - 1) {}

    int get_parent(int n) const {
      return (n-1) / 2;      // This gives: get_parent(0) = 0
    }
};
```

Figure 17: The `BinTreeStruct` class for the stage-based FlopC++ approach.

The main code

Once we have defined all the objects, the main code is quite short. The initialization of the libraries is the same as before, see lines 5–12 in Figure 7. Also start of the `main` function is the same, as we have start by providing the input data. The only difference from the code in Figure 8 is that we no longer need the `Root`, `nmbNodes` and `firstLeaf` values, as they are all provided by the `BinTreeStruct` class.

The rest of the `main` is then presented in Figure 18. All the scenario tree nodes are created in the loop on lines 225–244, using the tree-structure information from the `BinTreeStruct`-class object `scTree`. Once this is done, we only need to initialize the `objFunction` objects at all nodes by calling `make_objective_function` method from the root (line 263), set the root's `objFunction` expression as the objective function (line 264) and attach the model to the OSI interface (line 265). Then we are ready to solve the model (line 269) and report some results (lines 271–244). We can also save the model as an MPS file, as shown on line 266 – note again that we use the overloaded "`->`" operator to get access to the OSI object attached to our `MP_model`-class object `model`.

```cpp
int main()
{
  // binary scenario tree with 4 stages: 15 nodes, firstLeaf = 7
  BinTreeStruct scTree(4);

  // Initialize the object for the model
  MP_model &model = MP_model::getDefaultModel();
  model.setSolver(new OSI_SOLVER_INTERFACE);

  // Create the scenario tree
  vector<StageNode *> treeNodes;
  treeNodes.push_back(new RootNode(nmbAssets, InitBudget));
  for (int i = 1; i < scTree.nmbNodes; i++) {
    int pred = scTree.get_parent(i);
    if (i < scTree.firstLeaf) {
      // Adding a mid-stage node
      treeNodes.push_back(
        new MidStageNode(treeNodes[pred], i, 1.0 / 2, retData[i-1])
      );
    } else {
      // Adding a leaf
      treeNodes.push_back(
        new LeafNode(treeNodes[pred], i, 1.0 / 2, retData[i-1], CapTarget)
      );
    }
  }

  // create a "shortcut object" for the root
  RootNode *ptRoot = dynamic_cast<RootNode *>(treeNodes[0]);

  ptRoot->make_objective_function();          // Create the objective func.
  model.setObjective(ptRoot->objFunction);  // Set the objective
  model.attach();                             // Attach the model
  model->writeMps("investment");              // Write an MPS file

  // Solve the model
  model.minimize();

  // Report some results
  cout << endl << "Optimal objective value = " << model->getObjValue()
       << endl << "First-stage x:";
  ptRoot->x.display();

  for (int n = 0; n < (int) treeNodes.size(); n++)
    delete treeNodes[n]; // cleaning

  return 0;
}
```

Figure 18: The `main` function of the stage-based FlopC++ approach. Node that we have omitted the input data, since they are the same as in Figure 8, lines 33–51.

3.4 Investment example combining SMI and FlopC++

In Section 3.1, we have seen that the SMI library is a good tool for generating a stochastic model, given the core (deterministic) model and the stochastic information. The major inconvenience of this approach is in the construction of the core model: we had to manually construct all the elements of the LP model. This can easily become an tedious and error-prone procedure for bigger and more complex models. The FlopC++ library, on the other hand, allows for a more natural way of specifying LP models, but does not—at least at the moment[23]—include any stochastic information.

It seems therefore natural to combine the two approaches and use FlopC++ to create the core model and use it to build a stochastic model with SMI. In this section, we present an example of such a combination, using the stage-based FlopC++ model from Section 3.3. We would like to stress that the example's main purpose is to demonstrate that it is possible to combine the two packages. To make this approach practical, one should hide most of the extra objects from the end user by making them a part of the libraries instead – a process that has already begun, but will most likely take some time to finish.

Updating the stage- and scenario-tree objects

Since the code is basically a combination of the codes from Sections 3.1 and 3.3, we will only list the parts of the code that differ. We start by definition of the `StageNode` class, see Figure 19. The major change is addition of the `all_variables` and `all_constraints` objects, that will be later used to assign stages to variables and constraints of the core model. Since adding references to the `all_variables` vector involves constructing new objects (as we will see later), we need to update the destructor of the class, see lines 77–81. We have also added a reference to a constraint called `pt_balance_constraint`, which we will use later as a common way to access the stage-objects' constraints – since the constraints have different names in the derived stage objects, we would otherwise have to access the constraints in the derived classes separately. In other words, this object is not needed, but it will simplify the code later.

The last addition to the `StageNode` class is the `get_wealth` method (lines 83–97), use to report the wealth at the node, after the stochastic model has been solved. Unfortunately, we have not found any easy way to do this using the FlopC++ objects, for reasons that will become apparent later. This is not an issue in our small example, since the wealth in all the nodes except leaves is simply the sum of all the nodes' variables. In a more complex model, though, we would probably need to find a better way.

The only differences in classes `RootNode` and `MidStageNode` are then in their constructors, where we have to add all node's the variables and constraints to the vectors `all_variables` and `all_constraints` object and initialize the `pt_balance_constraint` pointer—see Figures 20 and 21, respectively. Note also that adding variables to the `all_variables` object involves creating new instances of type `VariableRef*`, which is why we had to add the destructor to the `StageNode` class.

There are more changes in the `LeafNode` class, where we also have to override the `get_wealth` function, since the wealth at a leaf node is computed as `-w+y+capTarget`, where `capTarget` is a new class member for storing the capital target. The changes are shown in Figure 22.

The `ScenTreeStruct` and `BinTreeStruct` can remain the same as in the previous section.

[23] There is an ongoing effort to add stochastic elements to the FlopC++ syntax, but it has not been finished by the time of writing. The source code can be found in the FlopC++ SVN repository under `branches/stochastic`.

```
class StageNode {
  public:
    /// \name References to variables and constraints
    vector<VariableRef *> all_variables;       ///< references to all variables
    vector<MP_constraint *> all_constraints;  ///< references to all constraints
    /// A common way to access the balance constraints in the derived classes
    MP_constraint *pt_balance_constraint;

    virtual ~StageNode() {
      for (int a = 0; a < (int) all_variables.size(); a++) {
        delete all_variables[a];
      }
    }

    /// get the wealth at the nodes
    virtual double get_wealth(const double *variableValues,
                              const int nmbVars) {
      double wealth = 0;
      for (int i = 0; i < nmbVars; i++)
        wealth += variableValues[i];
      return wealth;
    }

};
```

Figure 19: The `StageNode` class for the combined Smi & FlopC++ approach. It only lists the parts that have been changed or added, compared to the code in Figure 12.

```
class RootNode : public StageNode {
  public:
    RootNode(const int nmbAssets, const double initWealth)
    : StageNode(NULL, nmbAssets, 0, 1.0)
    {
      initialBudget() = sum(ASSETS, x(ASSETS)) == initWealth;

      for (int a = 0; a < nmbAssets; a++) {
        all_variables.push_back(new VariableRef(x(a)));
      }
      all_constraints.push_back(&initialBudget);
      pt_balance_constraint = NULL;
    }
};
```

Figure 20: Updated constructor of the `RootNode` class for the combined Smi & FlopC++ approach. The rest of the `RootNode` class is the same as in Figure 13

The main code

The main code starts similarly to the previous section, i.e. with the data, so we again skip them. The code then follows the structure of the SMI code, so we start with creating the core model, then create the stochastic model by adding scenarios to the core, solve it, and finally do some reporting.

The core model is created in a manner similar to the previous section, i.e. using the FlopC++

```
class MidStageNode : public StageNode {
  public:
    MidStageNode(StageNode *ptPred, const int nodeN,
                 const double condProb, double *ptRetVect)
    : StageNode(ptPred, ptPred->ASSETS.size(), nodeN, condProb),
      Return(ptRetVect, ASSETS)
    {
      cashFlowBalance = sum(ASSETS(a), ptParent->x(a) * Return(a))
                         == sum(ASSETS(a), x(a));

      for (int a = 0; a < ASSETS.size(); a++) {
        all_variables.push_back(new VariableRef(x(a)));
      }
      all_constraints.push_back(&cashFlowBalance);
      pt_balance_constraint = &cashFlowBalance;
    }
};
```

Figure 21: Updated constructor of the `MidStageNode` class for the combined Smi & FlopC++ approach. The rest of the `MidStageNode` class is the same as in Figure 14

```
class LeafNode : public StageNode {
  public:
    double capTarget;              ///< the capital target parameter

    /// Constructs a \a LeafNode object
    LeafNode(StageNode *ptPred, const int nodeN, const double condProb,
             const double *ptRetVect, const double capTg)
    : StageNode(ptPred, ptPred->ASSETS.size(), nodeN, condProb),
      Return(ASSETS), capTarget(capTg)
    {
      Return.value(ptRetVect); // Copy values from retVect to Return
      finalBalance = sum(ASSETS, ptParent->x(ASSETS) * Return(ASSETS))
                     + w() - y() == capTarget;

      all_variables.push_back(new VariableRef(w()));
      all_variables.push_back(new VariableRef(y()));
      all_constraints.push_back(&finalBalance);
      pt_balance_constraint = &finalBalance;
    }

    /// In the leaves, the wealth is computed as: -w + y + capTarget
    double get_wealth(const double *variableValues,
                      const int nmbVars) {
      return -variableValues[0] + variableValues[1] + capTarget;
    }
};
```

Figure 22: Updated constructor of the `LeafNode` class for the combined Smi & FlopC++ approach. The rest of the `LeafNode` class is the same as in Figure 15.

objects, see Figure 23. The major new features are:

- The `scenNodeNmb` vector. This vector includes the indices of nodes in the currently processed scenario. It is initialized on lines 297–301 to hold the first scenario—note how we start in the leaf (line 297) and then move up the scenario tree.

- The `coreNodes` vector corresponds to the `treeNodes` vector from Figure 18, but includes only one node per stage, while `treeNodes` represented the whole scenario tree. This is because we now operate on scenarios: first create the core model, i.e. a model with only once scenario, and then add the scenarios to the stochastic model one by one.

```
254 int main()
259   const int nmbStages = 4;
289   // Initialize the object for the core (deterministic) model
290   MP_model &coreModel = MP_model::getDefaultModel();
291   coreModel.setSolver(new OSI_SOLVER_INTERFACE);
292   coreModel.silent(); // less output
293
294   int a, i, j, n, t;
295
296   vector<int> scenNodeNmb(nmbStages); // nodes (indices) in a scenario
297   n = scTree.firstLeaf;               // leaf of the first scenario
298   for (t = nmbStages; t > 0; t--) {
299     scenNodeNmb[t-1] = n;
300     n = scTree.get_parent(n);
301   }
302
303   // Create scenario tree for the core model, using data for the 1st scenario
304   vector<StageNode *> coreNodes(nmbStages);
305   coreNodes[0] = new RootNode(nmbAssets, InitBudget);
306   for (t = 1; t < nmbStages-1; t++) {
307     coreNodes[t] = new MidStageNode(coreNodes[t-1], t, 1.0,
308                                      retData[scenNodeNmb[t]-1]);
309   }
311   coreNodes[t] = new LeafNode(coreNodes[t-1], t, 1.0,
312                               retData[scenNodeNmb[t]-1], CapTarget);
313
314   // create a "shortcut object" for the root
315   RootNode *ptRoot = dynamic_cast<RootNode *>(coreNodes[0]);
316
317   ptRoot->make_objective_function();            // Create the objective func.
318   coreModel.setObjective(ptRoot->objFunction);  // Set the objective
319   coreModel.attach();                           // Attach the model
```

Figure 23: First part of the `main` function of the combined Smi & FlopC++ approach. This shows the creation of the core objects, using the FlopC++ objects.

At the end of of the code in Figure 23, we have an `MP_model`-class object `coreModel` with an attached OSI representation of the core model. The last thing we need to create the SMI `SmiCoreData` object is the information about stages of variables and constraints. For this we use the new `all_variables` and `all_constraints` objects, i.e. the lists of all variables and constraints at each node, combined with the fact that we know the stage each node belongs to. In the code in Figure 24, lines 331–345 show the construction of the vector `colStages` with the stage numbers for all variables and lines 350–363 show the corresponding code for constraints. The `SmiCoreData` is then build on lines 368–370. Note how we again use the fact that the operation `->` of an `MP_model` points to the attached OSI object.

Once we have the `SmiCodeData` object, we can start building the stochastic model, i.e. an

```
// Get number of variables and constraints from the OSI model
int nmbCoreCols = coreModel->getNumCols();
int nmbCoreRows = coreModel->getNumRows();

// Now, we have to get the stage number for all variables and constraints
int *colStages = new int[nmbCoreCols];
for (t = 0; t < nmbStages; t++) {
  for (j = 0; j < (int) coreNodes[t]->all_variables.size(); j++) {
    int colIndx = coreNodes[t]->all_variables[j]->getColumn();
    colStages[colIndx] = t;
  }
}

// Now do the same for the constraints
int *rowStages = new int[nmbCoreRows];
for (t = 0; t < nmbStages; t++) {
  for (i = 0; i < (int) coreNodes[t]->all_constraints.size(); i++) {
    int rowIndx = *coreNodes[t]->all_constraints[i];
    rowStages[rowIndx] = t;
  }
}

// Now, we can build the CORE problem, i.e. the deterministic version
SmiCoreData stochCore(coreModel.operator->(), nmbStages,
                      colStages, rowStages);
delete[] colStages;
delete[] rowStages;
```

Figure 24: Second part of the `main` function of the combined Smi & FlopC++ approach. This shows the association of stages to the core model.

`SmiScnModel` object. This means that the rest of the code will be based mostly on Section 3.1. The creation of the `SmiScnModel` object `stochModel` is presented in Figure 25. Remember that in our model, the scenarios differ only in the LP matrix A, so the only data we need to add scenarios is the matrix `ADiff` including the elements that differ from the previous scenario. Note the use of the `reverseOrdering` method on line 389, which changes the sparse-matrix storage from the default column-based to a row-based model, expected by some internal SMI routines[24]

The scenarios are added in the loop on lines 394–437. Note that the loop is actually on the leaves of the tree, using the fact that there is a one-to-one relation between scenarios and leaves. For each leaf, we then move up the tree as long as the nodes are different from the previous scenario (whose indices are stored in the vector `scenNodeNmb`). This is modelled using the `while` loop on lines 400–420. Since `scenNodeNmb` has been initialized to point to the first scenario (see Figure 23, lines 297–301), the first scenario will never enter the `while` loop, leaving the `ADiff` matrix empty. This is exactly what we need, since the first scenario is specified by its difference from the core and the core had been created as a one-scenario model with data from the first scenario—so the matrix of differences is indeed empty.

The matrix `ADiff` for a given scenario is then created on lines 409–415. Remember than we need one matrix element for each return value in the model. For each return value, the matrix coordinates `(i,j)` of the element are such that `j` is the index of the constraint (row) with the value and `i` is

[24] The code would work also without this line, since SMI can reverse the order when needed—but doing so on full matrices is slower than doing it on an empty matrix, as in our case.

```
383   SmiScnModel stochModel;
384
385   // the matrix of differences wrt. the previous (parent) scenario
386   CoinPackedMatrix ADiff;
389   ADiff.reverseOrdering();
390
391   int nmbScen = scTree.nmbNodes - scTree.firstLeaf; // number of scenarios
392   double scenProb = 1.0 / nmbScen;                  // equiprobable scen.
393
394   // Add scenarios, one by one.
397   for (int leaf = scTree.firstLeaf; leaf < scTree.nmbNodes; leaf++) {
398     int scen = leaf - scTree.firstLeaf; // scenario index
399     n = leaf;                           // the current node to be added
400     t = nmbStages-1;                    // stage of node n
402     ADiff.clear(); // clean the matrix of differences - must reset dimensions!
403     ADiff.setDimensions(nmbCoreRows, nmbCoreCols);
404     while (n != scenNodeNmb[t]) {
407       scenNodeNmb[t] = n; // add the current node to the list
408
409       // Find row and column number of the matrix elements with Return values
410       i = *coreNodes[t]->pt_balance_constraint;
411       for (a=0; a<nmbAssets; a++) {
412         // Returns are on 'x' variables from the parent!
413         j = coreNodes[t-1]->x(a).getColumn();
414         ADiff.modifyCoefficient(i, j, retData[n-1][a]);
415       }
416
417       // Move one node up in the scenario tree
418       n = scTree.get_parent(n);
419       t--;
420     }
422
423     // Add the scenario to the Smi model
424     int branchStage = (scen == 0 ? 1 : t+1);
425     int parentScen = (scen == 0 ? 0 : scen - 1); // parent scenario
427     stochModel.generateScenario(&stochCore, &ADiff,
428                                 NULL, NULL, NULL, NULL, NULL,
429                                 branchStage, parentScen, scenProb);
437   }
```

Figure 25: Third part of the `main` function of the combined Smi & FlopC++ approach. This shows the creation of the stochastic-model object `stochModel`.

the index of the involved variable (column). Here we can see the usefulness of the common `pt_balance_constraint` pointer: without it, we would have to replace line 410 by two different lines, depending whether we were in the last stage or not. Since the returns in the balance constraints always correspond to variables from the parent node, we use the `coreNodes` vector to get the pointer to the parent on line 413. Note the use of the `getColumn` method to get a column number corresponding to a given `MP_variable` object[25]. Once we have the `ADiff` matrix, we add the scenario to the stochastic model in the same way as we did in the `add_scenario` method in Section 3.1.

[25] Actually, getColumn is a method of the `VariableRef` class, since indexing an `MP_variable` object a `VariableRef` object. (Technically, the `operator()` of class `MP_variable` is overloaded by a function returning `const VariableRef &`).

The stochastic model is now finished and ready to be used. We can thus attach a solver and solve the deterministic equivalent, using exactly the same code as in Figure 5. Also the detailed reporting code from Figure 6 remains unchanged, with the following two exceptions:

- The `while` loop that traverses the tree and calculates the wealth at the nodes (lines 124–157) can be simplified by using the `get_wealth` method of the `StageNode` class—see Figure 26. Note how the whole loop at lines 130–152 in the old code has been replaced by a single command in the new code (lines 492–495).
- In addition, we should delete the `StageNode` objects we had created while building the core model (see Figure 23):
```
511  for (t = 0; t < nmbStages; t++)
512    delete coreNodes[t]; // cleaning
```

```
486  // This loop traverses the tree, from the leaf to the root
487  while (ptNode != NULL) {
488    // info about columns of ptNode in the OSI model (ptDetEqModel)
489    int startColInOsi = ptNode->getColStart();
490    int nmbColsInOsi = ptNode->getNumCols();
491
492    // get the wealth, using nodes of the core model
493    nodeWealth[nodeStage-1]
494      = coreNodes[nodeStage-1]->get_wealth(&(ptOsiSolution[startColInOsi]),
495                                           nmbColsInOsi);
496
497    // Get the parent node (Root will return NULL, stopping the loop)
498    ptNode = ptNode->getParent();
499    nodeStage--;
500  }
```

Figure 26: The `while` loop for reporting the nodes' wealth, from the final part of the `main` function of the combined Smi & FlopC++ approach. It is significantly shorter than the corresponding code from Section 3.1, see Figure 6, lines 124–157.

4 Conclusions

We have presented the SMI and FlopC++ projects from the COIN-OR collection and shown how they can be used for modelling stochastic programming problems in C++. Each of the projects brings something useful: SMI handles the stochastic aspects of the model, while FlopC++ allows for a more natural modelling of the problem. While they can be used on their own, the real advantage comes from combining the two approaches, as illustrated by the last example. Even if the last example is the longest one if count the lines of the source code, it is probably the best (easiest) to scale to bigger models, both in sense of bigger core model and bigger scenario trees.

We would like to point out there is an ongoing process to simplify the combined usage of SMI and FlopC++, by including some of the extra objects in the projects' distribution. Once this has been finished, the presented examples should become obsolete as there will be an easier way of writing stochastic models.

A Extra material added for the stand-alone version

A.1 The investment example

The is basically the “Financial Planning and Control example from (Birge and Louveaux, 1997, Section 1.2), with slightly different numbers. It is a very simple investment model: we invest in two assets (stocks and bonds) and want to maximize the expected utility of our wealth after a couple of periods. The utility is a piece-wise linear function of the final wealth, with slope 1.3 values below a pre-specified capital target (shortage) and 1.1 for values above the target (surplus). There are no transaction costs.

Instead of showing the standard mathematical formulation, Figure 27 presents the model in the AMPL format. Since the model does not use any advanced features of AMPL, it is a valid GNU MathProg model as well. The corresponding data file is shown in Figure 28.

```
# The scenario tree
param LastNode;                              # index of the last node
set NODES := 0 .. LastNode;                  # all nodes in the scenario tree
param Root in NODES default 0;               # root of the tree
set FUT_NODES := NODES diff {Root};          # all nodes except the root
param ChildPerNode default 2;                # number of children nodes of each node
param Pred{n in FUT_NODES} default (n-1) div ChildPerNode; # predecessors
param FirstLeaf;                             # index of the 1st leaf (last-stage node)
set LEAVES := FirstLeaf .. LastNode;         # set of leaves
param Prob{LEAVES} default 1/(LastNode - FirstLeaf + 1); # probabilities

# Deterministic model and parameters
set ASSETS;          # set of assets
param InitBudget;    # initial budget
param CapTarget;     # capital target

# Stochastic parameters
param Return{FUT_NODES, ASSETS};

# Stochastic model variables
var x{NODES diff LEAVES, ASSETS} >= 0; # investment
var w{LEAVES} >= 0;                    # shortage
var y{LEAVES} >= 0;                    # surplus

# Objective function
minimize shortage_minus_surplus:
  sum{n in LEAVES} Prob[n] * (1.3*w[n] - 1.1*y[n]);

## CONSTRAINTS
subject to initial_budget:
  sum{a in ASSETS} x[Root, a] = InitBudget;

subject to cashflow_balance{n in FUT_NODES diff LEAVES}:
  sum{a in ASSETS} Return[n, a] * x[Pred[n], a]
  = sum{a in ASSETS} x[n, a];

subject to final_balanace{n in LEAVES}:
  sum{a in ASSETS} Return[n, a] * x[Pred[n], a] + w[n] - y[n]
  = CapTarget;
```

Figure 27: The AMPL/GNU MathProg formulation of the investment example.

```
# Scenario tree
param LastNode := 14;
param FirstLeaf := 7;

# Model data
set ASSETS := stocks bonds;

param InitBudget := 55;
param CapTarget := 80;

param Return:
     stocks bonds :=
   1  1.25  1.14
   2  1.06  1.16
   3  1.21  1.17
   4  1.07  1.12
   5  1.15  1.18
   6  1.06  1.12
   7  1.26  1.13
   8  1.07  1.14
   9  1.25  1.15
  10  1.06  1.12
  11  1.05  1.17
  12  1.06  1.15
  13  1.05  1.14
  14  1.06  1.12
;
```

Figure 28: Data for the AMPL/GNU MathProg formulation of the investment example.

Note that the model is very different from the one presented in Birge and Louveaux (1997). The reason is that the book uses a scenario-based formulation, with explicit non-anticipativity constraints, while our formulation is node-based. The advantage of the node-based formulation is that it avoids duplication of the decision variables. In addition, the model formulation is completely independent on the scenario-tree structure: the tree is specified by pointers to the parent/predecessor nodes (parameter `Pred`), together with the root node and the set of leaf (last-stage) nodes and their probabilities. (Actually, both the root and the set of leaves can be derived automatically from the predecessor pointers as the nodes with no parent and no children, respectively.)

The node-based formulation allows naturally divides the nodes of the scenario tree into three subsets, each with different properties:

The root is the first-stage node, representing "now" and thus having a probability of one.

The leaves are the last-stage nodes. Each leaf corresponds to one scenario, so their probabilities must sum-up to one.

All the other nodes of the scenario tree are of the same type, with the same variables and constraints, regardless the stage they are in. If we needed their probabilities (we do not in this model), they could be computed as the sum of the probabilities of their children, going backwards from the penultimate stage. (This is easily done on one line of AMPL code, as long as the nodes are ordered by the stage number.)

```
Optimal objective value = -4.14214

x :=    stocks  bonds
    0    48.46    6.54
    1     0.00   68.03
    2     0.00   58.95
    3    79.60    0.00
    4    76.19    0.00
    5     0.00   69.57
    6     0.00   66.03

Wealth as a function of time (target is 80):
sc. 1: 55.0 -> 68.0 -> 79.6 -> 100.3
sc. 2: 55.0 -> 68.0 -> 79.6 ->  85.2
sc. 3: 55.0 -> 68.0 -> 76.2 ->  95.2
sc. 4: 55.0 -> 68.0 -> 76.2 ->  80.8
sc. 5: 55.0 -> 59.0 -> 69.6 ->  81.4
sc. 6: 55.0 -> 59.0 -> 69.6 ->  80.0
sc. 7: 55.0 -> 59.0 -> 66.0 ->  75.3
sc. 8: 55.0 -> 59.0 -> 66.0 ->  74.0
```

Figure 29: Solution to the investment example (glpsol output).

Using these three types of nodes, we see that the investment variables `x` exist in all nodes except the leaves. The leaves, on the other hand, have extra variables `w` and `y` for computing the final shortage and surplus, respectively. All the nodes need some type of a cash-flow-balance constraint: in the root, the investment is bound by the initial budget (lines 37–38 in Figure 27), in the leaves we compare the final capital to the given target (l. 43–44), and in the other nodes we have the standard "in=out"-type constraint (l. 40–41). In addition, the stochastic asset returns (parameter `Return`) have to be provided for all the nodes except the root.

Finally, Figure 29 presents the result of running glpsol on the above files, with some additional post-processing code not presented here. The complete source files can be obtained from the author's web page `http://work.michalkaut.net`.

References

John R. Birge and François Louveaux. *Introduction to stochastic programming*. Springer-Verlag, New York, 1997. ISBN 0-387-98217-5.

Michal Kaut, Alan J. King, and Tim H. Hultberg. A C++ modelling environment for stochastic programming. Technical Report RC24662, IBM Watson Research Center, 2008. URL `http://domino.watson.ibm.com/library/cyberdig.nsf/papers/3E80629707DD1782852574E300592E33`.

Miloš Kopa, editor. *On Selected Software for Stochastic Programming*. MatfyzPress, Prague, 2008. ISBN 978-80-7378-069-2.