1 Introduction

FlopC++ is a modelling language for linear and mixed-integer programming written in the general programming language C++. With the stochastic extensions to FlopC++ it is also possible to model stochastic programs with recourse. It supports independent random variable based and scenario based stochastic programs. The website can be found at `https://projects.coin-or.org/FlopC++`. Information about the use of FlopC++ to model linear and mixed-integer programs is available at the website. This document is concerned with the use of FlopC++ to model stochastic programs with recourse. Knowledge about the use of FlopC++ is desirable but not necessary. Basic knowledge about stochastic programming is required.

2 Installation

You have several options to install FlopC++.

2.1 Linux

Under Linux just follow these steps to compile FlopC++ to obtain a library.

1. svn co https://projects.coin-or.org/svn/FlopC++/stable/1.0 coin-FlopCpp
2. cd coin-FlopCpp/ThirdParty
3. ./get.ThirdParty
4. cd ..
5. ./configure
6. make
7. make test
8. make install

Step 1 issues the subversion command to obtain the source code and store it in the directory *coin-FlopCpp*. Besides FLOPC++, the sources of these required Coin projects will also be retrieved.

Step 3 gets the ThirdParty libraries boost, googletest and google-glog from the web and unpacks them.

Step 5 runs a configure script that generates the make file.

Step 6 builds the libraries of all the required Coin-OR projects, the external libraries and the FLOPC++ library.

Step 7 builds and runs the FLOPC++ unit test program.

Step 8 installs libraries, executables and header files in the directories *coin-FlopCpp/lib*, *coin-FlopCpp/bin* and *coin-FlopCpp/include*.

2.2 Windows

If you have Visual Studio 2008 installed, you can use the solution file FlopCpp/MsVisualStudio/v9/FlopCpp.sln to load the solution. Then you can start a new C++ project and set a project dependency on libflopcpp.

2.3 Bleeding Edge

You can download the current development version with mercurial at https://dsor-development.upb.de. You have to enter the username "anonymous". A password is not required.

2.4 Troubleshooting

This section covers some errors that are likely to happen on different architectures.

Linux The compiler stops with an error about long long constants. Solution: Solved. Removed -pedantic-errors switch for FlopC++ project.

make install does not work for googletest. Solution: Remove ThirdParty/gtest from subdirs variable in the Makefile before typing make install.

3 An illustrative example

An investment problem [1] illustrates the use of FlopC++ to model multistage stochastic programs with recourse, that uses scenarios. The program is used to maximize the wealth of an investor after a given number of periods. In each period the investor has to buy assets with his available money. In the next period his investments return some money which must then be reinvested. An initial wealth level and a goal are given up front. An excess of the goal is encouraged in the objective function but a shortage is penalized. In the following all the important statements get explained one by one.

Line 1 defines a MP_model with an associated solver, in this case an instance of Clp. Every program should start with an explicit MP_model so that you can reference it easily later on, instead of relying on the global default model. The parameters for initialWealth and the goal are defined in the lines 2 to 4. The number of stages is defined via the so called enum hack in line 6. The stage set is defined in line 7. The same technique is used to define the scenario set and the set of assets. The array that holds the scenario values for all the eight scenarios and the four stage (three stages actually, as there are never random variables in the first stage) is declared in line 12. The values for the returns of the investments in the second stage are given in line 16 for the first asset and in line 17 for the second asset. This is repeated for the third and fourth stage. The array that holds the stochastic values is then given to the random parameter *returns* in line 28. The MP_random_data expects a pointer to a double array, therefore one has to take the adress of the first element of the array with the & operator. The constraints of the model are declared first in the lines 34 to 38. All the constraints without the stage set T are pure first stage constraints. The allocationConstr in line 41

is a constraint that holds at every stage, it is therefore indexed over T. The constraint returnConstr on the other hand is a constraint that does not hold for the first stage. Therefore it is indexed over T+1, so the first stage is left out. Notice that you have to use the T+1 also as the index on the right side, if you want to index variables with the same stage as the constraint. It makes no sense to adress variables in a constraint with a greater stage than the constraint stage. As you can see from this example you can still access earlier stages by reducing the index expression. Most often you connect variables from two adjacent stages, like it is the case with the returnConstr in line 42.

The shortage and overage variables are connected in constraint goalConstr on line 44. This constraint only holds in the last stage, therefore it is indexed using T.last(). The same holds for the objective in line 45 which is set in line 47. From this statement on the model knows the objective. The direction of the optimization is not known at this point but has to be provided when the solve() method gets called, in line 49. If the model is not attached to the solver when you call the solve() method, it gets attached automatically, so the statement in line 48 is not needed, unless you intend to solve the model again (with a new set of sampled random variates if you use random variables in your formulation).

4 A more advanced example

To be done. Contains usage of random variables.

5 Classes and methods

5.1 Language constructs

The C++ classes necessary for the formulation of stochastic programs are described in the following.

MP_stage A special set that enumerates the available stages of the model.

MP_scenario_set A special set that enumerates all available scenarios. It is only specified in the case of a scenario based problem. It is never used anywhere else by the user expect for indexing an array that holds values for the probabilities for the scenarios.

MP_random_data An indexable entity that contains either RandomVariable's or the result of an algebraic function applied on MP_data and MP_random_data. If the user wants to model a scenario based problem it can be feed directly with an array of double arrays, where the values for each scenario are stored.

5.2 Random variable hierarchy

There are some predefined random variables present in FlopC++ that cover some important distributions like the normal, lognormal, exponential and continuous distributions.

```
1  MP_model investmentModel(new OsiClpSolverInterface());
2  MP_data initialWealth,goal;
3  initialWealth() = 55; //set initial data
4  goal() = 80;
5
6  enum {numStage=4};
7  MP_stage T(numStage);
8  enum {numScen=8};
9  MP_scenario_set scen(numScen);
10 enum {asset1,asset2,numAssets=2};
11 MP_set assets(numAssets);
12 double scenarios[numStage-1][numAssets][numScen]=
13 {//T
14     {//assets
15      // stage 2
16         {1.25,1.25,1.25,1.25,1.06,1.06,1.06,1.06}, //asset1, scen 1 to 8
17         {1.14,1.14,1.14,1.14,1.16,1.16,1.16,1.16}  //asset2, scen 1 to 8
18     },
19     {// stage 3
20         {1.21,1.21,1.07,1.07,1.15,1.15,1.06,1.06},
21         {1.17,1.17,1.12,1.12,1.18,1.18,1.12,1.12}
22     },
23     {// stage 4
24         {1.26,1.07,1.25,1.06,1.05,1.06,1.05,1.06},
25         {1.13,1.14,1.15,1.12,1.17,1.15,1.14,1.12}
26     }
27 };
28 MP_random_data returns(&scenarios[0][0][0],T,assets); //create random
       parameter "returns"
29
30 MP_variable x(T,assets);
31 MP_variable wealth(T);
32 MP_variable shortage(T), overage(T); //only needed in last stage
33
34 MP_constraint
35     initialWealthConstr,
36     returnConstr(T),
37     allocationConstr(T),
38     goalConstr(T);
39
40 initialWealthConstr() = sum(assets,x(0,assets)) == initialWealth();
41 allocationConstr(T) = sum(assets,x(T,assets)) == wealth(T);
42 returnConstr(T+1) = sum(assets,returns(T+1,assets)*x(T,assets) ) ==
       wealth(T+1); // only valid for stage 2 to 4
43
44 goalConstr(T.last()) = wealth(T.last()) == goal() + overage(T.last()) -
       shortage(T.last()); // only valid in last stage
45 MP_expression valueFunction( -1.3*shortage(T.last()) + 1.1*overage(T.last
       ()));
46
47 investmentModel.setObjective( valueFunction );
48 investmentModel.attach(investmentModel.Solver);
49 investmentModel.solve(MP_model::MAXIMIZE);
```

Listing 1: Investment Example in FlopC++

It is fairly easy to add own random distributions, by looking at the implementation of these predefined distributions. You can take a look at the doxygen documentation to see what random variables are already available to you.

5.3 Methods

The following methods can be called on the MP_model to enforce a specific behaviour.

setSampleOnlyScenarioGeneration(bool sampleOnly, int defaultSampleSize) If the user does not want a combine all-against-all approach for independent random variables in the scenario generation process, but instead only wants to sample from every random variable exactly *defaultSampleSize* many variates, then this method should be called with *sampleSize* = **true**. This allows the user to precisely define the number of scenarios, given random variables with distributions.

6 Bibliography

References

[1] Michael Kaut. COIN-OR Tools for Stochastic Programming. In Miloš Kopa, editor, *On Selected Software for Stochastic Programming*, pages 88–116, Prague, 2008. matfyzpress. URL `http://www.springerlink.com/index/10.1007/978-0-387-75714-8`.