FLOPC++
An algebraic modeling language embedded in C++

Tim Helge Hultberg

EUMETSAT, DE-64295 Darmstadt tim.hultberg@eumetsat.int

Summary. FLOPC++ is an open source algebraic modeling language implemented as a C++ class library. It allows linear optimization problems to be modeled in a declarative style, similar to algebraic modeling languages, such as GAMS and AMPL, within a C++ program. The project is part of COmputational INfrastructure for Operations Research (COIN-OR) and uses its Open Solver Interface (OSI) to achieve solver independence.

1 Introduction

Algebraic modeling languages are important tools in Operations Research, their benefits are well known [2]: They allow a model representation which is readable by both humans and computers, using a syntax close to the notation used by most modelers. They automate the generation of model instances, freeing the modeler from the burden of generating the low level matrix format needed by the solver. In summary, they make implementation and maintenence of optimization models much easier.

However, traditional algebraic modeling languages do also have major weaknesses:

- Limited flexibility for integration with other software components
- Limited procedural support for algorithm development
- Limited model/program structuring facilities
- Limited extendibility

In order to mitigate these weaknesses, most algebraic modeling languages have incorporated procedural constructs, which allow the construction of algorithms around the optimization models. It has also become common to provide some sort of support for embedding the models in applications.

A natural alternative to extending the existing modeling languages is to implement modeling abstractions within an existing general-purpose object-oriented programming language, C++, in order to enable the "Formulation

of Linear Optimization Problems in C++". This is what FLOPC++ is all about.

The work on FLOPC++ began in year 2000 when the author had developed a production model for a Danish poultry butchery in GAMS and was faced with the practical problems of integrating the model in a GUI for collecting data and displaying the results. Discussing these problems with Søren Nielsen, he pointed me to his excellent paper [5], which was the main inspiration for the start of the development of FLOPC++. [4]

The first versions of FLOPC++ used the CPLEX callable library as its solver, but the need for supporting several independent solvers soon became apparent. Luckily, the simultaneous appearance of the Osi (Open Solver Interface) providing an abstract base class to a generic linear programming (LP) solver, along with derived classes for specific solvers, made this an easy task.

The algebraic representation of a model convey structural information that is hidden in the matrix representation required by the solver. Sometimes this information can be used to develop specialized solution algorithms using a general linear optimization solver for subproblems. Examples of such algorithms include decomposition, column generation and model specific cutting plane algorithms. The development of such algorithms is greatly facilitated by working in an environment where the algebraic representation of the model is available. This kind of algorithms can often be implemented in traditional modeling languages, but the implementations are mostly hopelessly inefficient because problems are passed to the solver via file interfaces and most importantly because slightly modified problems get regenerated from scratch. FLOPC++ offers the possibility to modify problem instances and warm start the solver from the last solution.

In Section 2 we give a brief introduction to the implementation of FLOPC++ without going into every aspect, before the conclusion in Section 3.

2 Implementation

Consider the following excerpt from the FLOPC++ model "aircraft.cpp" (the full model is one of the examples in the model library which comes with the distribution):

```
MP_set S(numS);          // Sources
MP_set D(numD);          // Destinations
MP_subset<2> Link(S,D);  // Transp. links (subset of S*D)
MP_data DEMAND(D);
MP_data SUPPLY(S);
MP_variable x(Link);
MP_constraint supply(S);
MP_constraint demand(D);
```

```
supply(S) =  sum(Link(S,D), x(Link)) <= SUPPLY(S);
demand(D) =  sum(Link(S,D), x(Link)) >= DEMAND(D);
```

We see that C++ classes are available to represent the basic algebraic modeling constructs: index sets, parameters, variables and constraints. There is also a class for representing dummy indices (MP_index), but since index sets can also be used as dummy indices, they are not needed for this particular model.

When the last statement is executed several data members of the MP_constraint object supply get instantiated. The dummy index "S" corresponding to the index set "S", over which the constraint is defined, is stored (yes,"S" plays the role of both index set and dummy index). The type of the constraint, >=, is stored and the value of two objects, left and right, of the type MP_expression get assigned.

The objects of the MP_expression class are symbolic representations of linear expressions (including dummy indices) and are implemented as reference counted pointers to MP_expression_base. In general MP_expressions are trees since the classes derived from MP_expression_base have one or two pointers to MP_expression_base as members, except for the two TerminalExpression classes Expression_constant and VariableRef. The complete MP_expression_base class hierarchy is shown below.

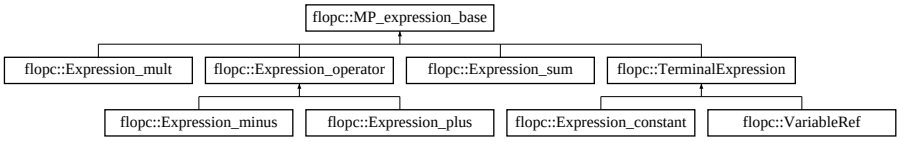

Fig. 1. MP_expression_base class hierarchy

The MP_expression trees are build by a number of small operator overloaded functions and constructors as for example:

```
MP_expression operator+(const MP_expression& e1, const MP_expression& e2) {
    return new Expression_plus(e1, e2);
}
```

and

```
MP_expression::MP_expression(const Constant &c) :
    Handle<MP_expression_base*>(new Expression_constant(c)) {}
```

The symbolic representation can then subsequently be used for generating the problem instances. In fact, it is this two step generation approach which makes the truly declarative notation of FLOPC++ possible in contrast to other modeling libraries such as Concert Technology from ILOG or LP Toolkit from Euro-decision. With Concert Technology, for example, the last constraint would be modeled as:

```
for (IloInt d=0; d<numD; d++) {
    IloExpr total(env);
    for (IloInt s=0; s<numS; s++) {
        total += x[s][d];
    }
    model.add( total >= DEMAND[d] );
    total.end();
}
```

Back to FLOPC++. The generation of the coefficient matrix entries takes place in the leaf nodes of the MP_expressions after informations (constant multipliers and domain) have been passed recursively down in the tree and combined from upper level nodes. When combining the domains, it is realized that D*Link(S,D) is equal to Link(S,D) so the complexity of generating the last constraint is of order |Link(S,D)| and not |D|*|Link(S,D)| as one might expect from a naive implementation.

As the very final step the coefficients matrix entries generated from the individual leaf node TerminalExpressions are assembled, i.e. eventual repeated entries are added.

3 Conclusion

FLOPC++ is available from the home-page `https://projects.coin-or.org/FlopC++`, where detailed download and installation information can be found.

The main goal of FLOPC++ is to be as robust, efficient and easy to use for linear optimization as traditional algebraic modeling languages, such as AMPL and GAMS, while remaining lightweight and reaping the inherent benefits of being embedded in C++.

I hope to see FLOPC++ being increasingly used, for example for modeling slice models [1] and stochastic programs [3], which are otherwise awkward to formulate without extensions to the traditional modeling languages.

Acknowledgments

I would like to thank Philip Walton, Junction Solutions and JP Fasano, IBM who recently implemented several valuable improvements to FLOPC++ (Microsoft Visual C++ compatibility, doxygen comments, use of GNU autotools and full integration into Coin-Or, just to mention the most important) as well as the several other people who have reported bugs and suggested improvements over the last couple of years.

References

1. M. Ferris and M. Voelker. Slice models in general purpose modeling systems. Technical report, Data Mining Institute Technical Report 00-10, Computer Sciences Department, University of Wisconsin, Madison, 2000.
2. R. Fourer. Modeling languages versus matrix generators for linear programming. *ACM Transactions on Mathematical Software*, 1983.
3. R. Fourer and L. Lopes. Stampl: A filtration-oriented modeling tool for stochastic programming. Technical report, Technical report, Department of Industrial Engineering and Management Sciences, Northwestern University, 2003.
4. T. Hultberg. *Topics in computational linear optimization*. PhD thesis, Department of Mathematical Modeling, Technical University of Denmark, 2000.
5. S. Nielsen. A C++ class library for mathematical programming. In *The impact of emerging technologies on computer science and operations research.* Kluwer, 1995.