FLOPC++

Formulation of Linear Optimization Problems in C++

Tim Helge Hultberg

thh@mat.ua.pt

Department of Mathematics, University of Aveiro

FLOPC++

Formulation of Linear Optimization Problems in C++

- Algebraic modelling language.
- Implemented as a C++ class library.
- Declarative optimization modelling (similar to GAMS, AMPL and AIMMS), within a C++ program.
- LP's and MIP's
- Sourcecode downloaded at: www.mat.ua.pt/thh/flopc
- Common Public License version 1.0
- Work in progress.
- New users will be appreciated.

(=[**FLOPC++**.

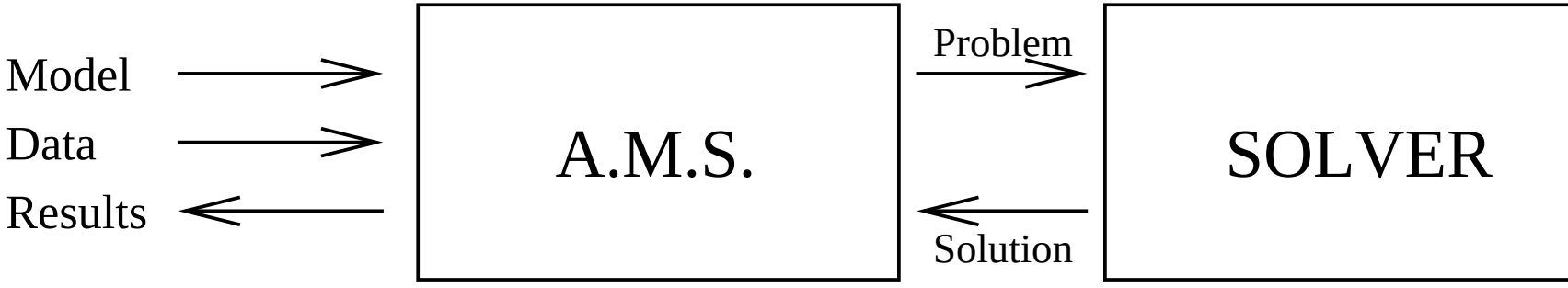
A transportation model

```
main() {
  MP_set S(numS),D(numD);
  MP_data capacity(S), dem(D), d(S,D), c(S,D);
  MP_variable  x(S,D);
  MP_constraint supply(S), demand(D);

  c(S,D) = f * d(S,D) / 1000;

  supply(S) = sum(D, x(S,D)) <= capacity(S);
  demand(D) = sum(S, x(S,D)) >= dem(D);

  minimize( sum(S*D, c(S,D) * x(S,D)) );

  x.display("Optimal transportation plan");
}
```

(=[**FLOPC++**.

Algebraic modelling system/language

- INPUT
  - ♦ Model (generic algebraic representation)
  - ♦ Data
- OUTPUT
  - ♦ A problem instance (LP or MIP)



(=[**FLOPC++**.

Model

minimize $$\sum_{p \in P} (InCost(p) * ip(p) + OutCost(p) * op(p))$$

subject to

$$\forall_{r \in R} : \sum_{p \in P} Consumption(p, r) * ip(p) \leq Capacity(r)$$

$$\forall_{p \in P} : ip(p) + op(p) = Demand(p)$$

$$\forall_{p \in P} : ip(p), op(p) \geq 0$$

(=[**FLOPC++.**

Data

$$R = \{0, 1\}, \quad P = \{0, 1, 2\}$$

	0	1	2
Demand(p)	100	200	300
InCost(p)	0.6	0.8	0.3
OutCost(p)	0.8	0.9	0.4

	Capacity(r)
0	20
1	40

Consumption(r,p)	0	1	2
0	0.5	0.4	0.3
1	0.2	0.4	0.6

(=[**FLOPC++**.

Problem Instance (Solvers format)

- Number of variables and constraints, `n` and `m`
- Optimization sense (minimize or maximize)
- Objective function coefficients, `c`
- Right hand side values, `b`
- Constraint types, `sense`
- Coefficient matrix, `Cst, Clg, Rnr, Elm`
- Bounds on variables, `l` and `u`
- Variable types (continuous or integer), `ctype`

(=[**FLOPC++**.

Problem Instance (Solvers format)

```
int m=5; int n=6;
int c[]={0.6,0.8,0.3,0.8,0.9,0.4};
int b[]={20,40,100,200,300};
char sense[]=('L','L','E','E','E'};
int Cst[]={0,3,6,9,10,11};
int Clg[]={3,3,3,1,1,1};
int Rnr[]={0,1,2,0,1,2,0,1,3,0,1,4,2,3,4};
double Elm[]={0.5,0.2,1,0.4,0.4,1,0.3,0.6,
                1,1,1,1};
```

(=[**FLOPC++**.

Traditional algebraic modelling languages

Strengths .

- Syntax close to the notation used by most modellers
- Instance generation is automized
- Easier to modify problems
- Representation is readable by both humans and computers

Weaknesses .

- Hard to integrate with other software components
- Limited procedural support for algorithm developmer
- Limited model/program structuring facilities

(=[**FLOPC++**.

Embedding an optimization model ...

in a C/C++ application.

- **1. Using the callable library of the solver.**

```
...
// code to generate a problem instance
// in the format of the solver

lp.copylpdata(n, m, minimize, c, b, sense,
              Cst, Clg, Rnr, Elm, l, u);

solstat = lp.primopt();

if (solstat==CPX_OPTIMAL) lp.getX(solution);
...
```

(=[**FLOPC++**.

Embedding an optimization model ...

in a C/C++ application.

- **2. Using an algebraic modelling language.**
  1. Write code to generate a data file.
  2. Write code to spawn the algebraic modelling languag interpreter.
  3. Write code to parse the results from a file.

(=[**FLOPC++**.

Other modelling libraries

'MP' (Søren S. Nielsen) C++

Planner (ILOG) C++

Concert Technology (ILOG) C++

LP Toolkit (Euro-decision) C++

XBSL (Dash) C

AMMO (Drayton Analytics) COM

Component Libraries (ILOG) COM, based on OPL

OptiMax2000 (Maximal Software) COM, based on MPL

EMOSL (Dash) C, based on XPRESS-MP

(=[**FLOPC++**.

Comparison I

ILOG Concert Technology

```
for (IloInt s=0; s<numS; s++)
    model.add( IloSum(x[s]) <=  capacity[s] )
```

FLOPC++

```
supply(S) = sum(D, x(S,D)) <= capacity(S);
```

(=[**FLOPC++**.

Comparison II

ILOG Concert Technology

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

FLOPC++

```
demand(D) = sum(S, x(S,D)) >= dem(D);
```

(=[**FLOPC++**.

Solver independence

FLOPC++ uses OSI (Optimization Solver Interface) from COIN (www.coin-or.org) (a uniform API for calling math programming solvers)

- CLP, CPLEX, dylp, GLPK, OSL, SOPLEX, VOL and XPRESS-MP

(=[**FLOPC++**.

Sparse subsets

```
struct Transport : public MP_model {
    MP_variable x;
    MP_constraint supply, demand;

    Transport(MP_set& S,          // Sources
              MP_set& D,          // Destinations
              MP_subset<2>& Link,// Transportation links (sparse su
              MP_data& SUPPLY,
              MP_data& DEMAND,
              MP_data& DISTANCE) :
        MP_model(new OsiClpSolverInterface), x(Link), supply(S), de

        MP_data COST(Link);
        COST(Link) = 90 * DISTANCE(Link) / 1000.0;

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

        minimize( sum(Link, COST(Link)*x(Link)) );
    }
};
```

(=[FLOPC++.

Example (coex)

```
MP_set I(8);  // size of chess board
MP_subset<4> M(I,I,I,I); // shared positions on the board;

MP_index i,j,ii,jj;

forall(I(i)*I(j)*I(ii)*I(jj).such_that(i==ii||j==jj||abs(i-ii)==abs
       M.insert(i,j,ii,jj) );

MP_binary_variable  b(I,I), // square occupied by a black Queen
                    w(I,I); // square occupied by a white Queen
MP_variable tot;    // total queens in each army;

MP_constraint  eq1(M), // keeps armies at peace
               eq2,    // add up all the black queens
               eq3;    // add up all the white queens;

eq1(M(i,j,ii,jj)) =   b(i,j) + w(ii,jj) <= 1;
eq2() =  tot() == sum(I(i)*I(j), b(i,j));
eq3() =  tot() == sum(I(i)*I(j), w(i,j));

maximize(tot());
```

(=[FLOPC++.

MP_expression

- Can explicitly name a linear expression without adding rows or columns to the generated instance.
- Provides alternative to accounting variables

```
MP_expression production(Products);

production(p) = sum(T(t), xi(p,t) + xo(p,t));

Min_prod(p) = production(p) >= MIN_PROD(p);

E_limit(q) =
    sum(Products(p), E(p,q)*production(p)) <= LIMIT(q);
```

(=[**FLOPC++**.

Extensions

```
void MP_model::minimize_max(MP_set &s, const MP_expression  &obj) {
     MP_variable v;
     MP_constraint c(s);
     add(c);
     c(s) = v() >= obj;
     minimize(v());
}
....
minimize_max( J, s(J)+D(J) );
```

(=[**FLOPC++**.

Problem modification

Routines to change

- the objective function
- lower and upper bounds on variables
- right hand side and sense of constraints
- variable types

or add or delete

- columns (variables)
- rows (constraints)

without regeneration of the problem.

(=[**FLOPC++**.

(Benders decomposition)

```
for ( ; ; ) {
    for (int j=0; j<numDistributionCenters; j++)
        subproblem->setRowBounds( selling(j), received.level(j), re
    double objsub = 0.0;
    for (int s=0; s<numScenarios; s++) {
        for (int j=0; j<numDistributionCenters; j++)
            subproblem->setRowBounds( selmax(j), Demand[s][j], Dema
        subproblem->resolve();
        objsub += prob[s]*subproblem->getObjValue();
        for (int j=0; j<numDistributionCenters; j++) {
            cutCnst += prob[s]*selmax.price(j)*Demand[j][s];
            coeff(j) += prob[s]*selling.price(j);
        }
    }
    upperbound = min(upperbound, objmaster + objsub);
    if ((upperbound-lowerbound) < 0.01*(1+fabs(lowerbound))) break;
    masterproblem.addRow( theta() >= cutCnst + sum(j, coeff(j)*rece
    masterproblem->resolve();
    lowerbound = masterproblem->getObjValue();
    objmaster = lowerbound - theta.level();
}
```

(=[FLOPC++.

MAGIC, Power Scheduling

```
enum {t12pm_6am, t6am_9am, t9am_3pm, t3pm_6pm, t6pm_12pm, numT};
enum {type_1, type_2, type_3, numG};
MP_set T(numT); T.cyclic();
MP_set G(numG);

MP_integer_variable  n(G,T);  // number of generators in use
MP_variable          x(G,T),  // generator output (1000mw)
                     s(G,T);  // number of generators started up

n.upperLimit(j,i) = number(j);

MP_constraint  pow(T),    // demand for power (1000mw)
               res(T),    // spinning reserve requirements (1000mw)
               st(G,T),   // start_up definition
               minu(G,T), // minimum generation level (1000mw)
               maxu(G,T); // maximum generation level (1000mw)

pow(i) =  sum(G(j), x(j,i)) >= dem(i);
res(i) =  sum(G(j), max_pow(j)*n(j,i)) >= 1.15*dem(i);
st(j,i) = s(j,i) >= n(j,i) - n(j,i-1);

minu(j,i) =  x(j,i) >= min_pow(j)*n(j,i);
```

(=[FLOPC++

Implementation

- keep track of correspondence between individual variables (/constraints) and its column (/row) index in th generated problem instance
- expressions evaluate to abstact syntax trees

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

- generation takes place in the leaf nodes, information fro upper level nodes is passed recursively down in the tree
- expressions are reference counted to avoid memory lea

(=[**FLOPC++**.

Conclusion

Advantages in addition to traditional modelling languages.

- Fast problem generation
- Seamless integration with applications
- Efficient and powerfull for customized algorithms (decomposition, column generation, ...)
- Modelling objects are first class C++ types. This allows higher level modelling extensions (such as the minimize max objective) to be implemented.

Disadvantages .

- Hard to make error diagnostics.

```
MP_set I(5), J(2);
MP_variable x(I,J);
MP_constraint c;
c() =  sum(I*J, x(J,I)) <= 100
```

will generate this (without warnings)

```
x(0,0) + x(0,1) + x(1,0) + x(1,1) <= 100
```

(=[**FLOPC++**.