SMPL - A Simplified Modeling Language for Mathematical Programming

Mihály Csaba Markót

November 3, 2008

1 Purpose and Scope

This working paper describes SMPL, an initiative of a Simplified Modeling Language for Mathematical Programming. The original purpose of SMPL is to make easier the process of inputting optimization problems to the GloptLab software, under both the Windows and the Linux operating systems. This document is intended to support the developers of GloptLab and related tools.

2 The SMPL syntax

2.1 General properties of SMPL input files

SMPL input files consist of a sequence of statements. Statements are ended by a semicolon. Spaces, tabs, and line feeds can be freely used between the lexical elements; the only exception is that if a variable or a parameter is referred through indexing (see Sections 2.10 and 2.11), then the opening indexing operator ('[') must immediately follow the variable/parameter name. (This is in order to resolve the disambiguity with constant matrices and intervals.)

SMPL allows single line comments, using the comment character '`#`'. That is, anything following '`#`' in a line will be treated as comment.

2.2 SMPL sections

An SMPL file consist of the following sections:

– the declaration section (both for variables and for parameters),

– the input variable section,

– the objective function section (zero or one statement), and the

– constraint section.

The sections must occur in the input file in the above order. Variable and parameter declaration statements may occur in mixed order within the declaration section. All four sections are allowed to be empty. An empty objective function section indicates a constraint satisfaction problem.

2.3 Numbers

SMPL numbers are either point, uncertain or interval constants.

2.3.1 Point numbers

Unsigned point numbers are allowed to be given in the following ways:

– integer constants in the standard format (leading zeros are allowed),

– real-type constants given as either

- a fixed point number (using decimal point) with an optional exponent part, or
- an integer number with a mandatory exponent part.

Formally, unsigned integer and real numbers are recognized by the following regular expressions:

```
INT   [0-9]+                           # integer number
EPART [deDE][+-]?[0-9]+                # exponent part
FPONT [0-9]+"."[0-9]*|[0-9]*"."[0-9]+  # fixed point notation
REAL  {FPONT}{EPART}?|{INT}{EPART}     # real number
```

Note that in a fixed point notation either the integer or the fractional part must be nonempty. (In contrast to C, where for example `.e+10` is an allowed, but rather ambiguous token.)

Examples:

```
012          # allowed integer
12.          # allowed real
.23          # allowed real
12.e4        # allowed real
12.34e+005   # allowed real
12E+005      # allowed real (although represents an integer)
.e-01        # NOT allowed in SMPL
```

Signed point numbers can be given in the usual way by applying the unary minus operator (cf. Section 2.8).

In most cases SMPL makes no difference between integers and reals: all constants (both numbers and compound expressions) are represented by a string, with the exception of indices and exponents, which must be in integer format, and are handled as numbers.

2.3.2 Uncertain numbers (the 'question mark' notation)

In SMPL, uncertainty can be represented by a new notation syntax called the 'question mark' notation, suggested by Arnold Neumaier.

Formally, 'question mark' constants are recognized through the following regular expressions:

```
PNUM    {REAL}|{INT}                     # point numbers for short
QMCONST {PNUM}\?|{PNUM}\?{INT}|{PNUM}\?u{INT}|{PNUM}\?d{INT}|
        {PNUM}\?a{PNUM}|{PNUM}\?r{PNUM}  # 'question mark' numbers
```

In the above syntax, the point numbers following the question mark sign can be positive or zero. (In contrast to the original initiative, which allows only positive numbers after the question mark.)

Examples:

(In these examples ulp is one unit in the last place of the decimal number displayed.)

```
12.3?      # represents 12.3 +- 1/2 ulp, i.e., [12.25,12.35]
           # Note: An earlier version had here erroneously 1 ulp
12.3?5     # represents 12.3 +- 5 ulp, i.e., [11.8,12.8]
.23?u2     # represents .23 + <=2 ulp, i.e., [.23,.25]
.23?d2     # represents .23 - <=2 ulp, i.e., [.21,.23]
.23?a0.1   # represents .23 +- 0.1, i.e., [.22,.24]
.23?r0.1   # represents .23 * (1+-0.1), i.e., [.207,.253]
```

2.3.3 Interval numbers

The syntax of an unsigned SMPL interval number is

```
[ <const-scalar-expr1> , <const-scalar-expr2> ]
```

where `<const-scalar-expr1>` and `<const-scalar-expr2>` are constant type scalar expressions. (see Section 2.13, Expressions). Note that the arguments can be intervals, that is, nested intervals are also allowed.

Signed interval numbers can be given in the usual way by applying the unary minus operator (cf. Section 2.8).

Interval numbers are also stored as strings. Since expressions are allowed in bounds, unsigned interval numbers are defined as nonterminals of SMPL, while unsigned point numbers are tokens of the language.

2.4 Constant matrix notation

Constant scalar expressions (Section 2.13), that is, scalars composed of point/interval numbers, arithmetic operators and functions can be put together in constant matrices. Matrices should be defined row-by-row: row entries are separated by an '`&`' sign, while subsequent rows are separated by a semicolon. Lower and upper triangular matrices can be defined by skipping the obvious zero entries. Those entries will be automatically substituted by the literals '`0`' or '`[0,0]`', respectively. Examples:

```
[ (1+2)*[3,4] ]         # 1-by-1 constant matrix

[ 1.0 & 2.0 & 3.0;
  4.0 & 5.0 & 6.0  ]    # 2-by-3 constant matrix

[ [1,2]^2 & 0;
  0       & [1,2]^2 ]   # 2-by-2 constant matrix

[ 1 & 2 & 5; 3 & 4; 7 ] # 3-by-3 upper triangular constant matrix
```

2.5 Tables

SMPL allows the usage of *tables* describing piecewise approximations or splines. An SMPL table is specified by

```
<table-type> ( <const-mat-expr> )
```

where `<table-type>` is one of the key words `pcon`, `plin`, `hspline`, `cspline`, `rspline`, and `<const-mat-expr>` is a constant matrix expression with column size 3 for `hspline` and column size 2 for all other table types. All rows of the matrix encodes the triples $(x, f(x), f'(x))$ or the pairs $(x, f(x))$, respectively. For details on defining tables see the GloptLab documentation.

Note, that the clearest and most straightforward way to specify the table matrix is the use of constant matrices (Section 2.4). However, compound expressions resulting in a matrix of the prescribed size are also allowed in the definition.

2.6 Identifiers

Identifiers in SMPL are used for variable and parameter names (and as their references), and for labels. Identifiers must be composed of zero or more digits and one or more letters, and must always start with a letter.

2.7 Labels

SMPL labels are composed of a valid identifier followed by a semicolon (`:`). Note, that labels are currently simply ignored by the SMPL parser, i.e. no semantic information is stored and passed in relation to them.

2.8 Operators and functions

2.8.1 Arithmetic and matrix operators

The binary arithmetic operators are interpreted in the infix fashion

```
<lhs-expr> <op> <rhs-expr>
```

The arithmetic and matrix operators of SMPL are the following, in *increasing order of precedence*:

– **Addition** (‘`+`’) and **subtraction** (‘`-`’). The operator is interpreted componentwise; the dimensionality of the lhs and rhs must match.
– **Multiplication** (‘`*`’). Both scalar-matrix, and matrix-matrix multiplication is allowed. In case of scalar matrix multiplication, the scalar must be a constant expression. In case of matrix-matrix multiplication, the respective dimensions of the arguments must match. Note that variable matrices are not interpreted in SMPL. This implies that variable expressions (vectors) can be multiplied as an inner product. Examples:

```
var x, y { 1..5 };
param p = [ 1.0; 2.0; 3.0 ]    # column vector
param q = [  1 & -1 &  0;
            -1 &  1 & -1;
             0 & -1 &  1  ]

p[1] * y      # scalar-matrix mult., result is a 5-by-1 variable vector
q * p         # matrix-matrix mult., result is a 3-by-1 const.type vector
q * y[1 3 5]  # matrix-matrix mult., result is a 3-by-1 variable vector
y' * y        # inner product, result is a scalar variable-type expr.
```

```
y * y'         # NOT allowed, variable outer product
y[1 3 5] * p' # NOT allowed, result is a variable-type matrix
```

– **Division** (‘/’, with the *same precedence as* ‘*’). The rhs expression must always be a scalar. If the rhs is of constant type, then the lhs can be any kind of expression, if it is of variable type, then the lhs must be scalar. The division is interpreted componentwise.
– **Unary minus (‘-’)**. This prefix operator is allowed on expressions with arbitrary dimensions. In case of vector and matrix arguments, the operator is interpreted componentwise. Note that signed point constants are defined through this operator, thus, white spaces or newlines are allowed between the sign and the constant part of a number.
– **Power** (‘^’). The rhs (exponent) must be a single integer constant (2.3.1), the lhs is allowed to be a scalar expression.
– **Matrix transpose** (‘'’). This operator is used as usual in a postfix fashion, and is allowed on all expressions (Section 2.13).

2.8.2 Relational operators

Relational operators are interpreted in the infix fashion

```
<lhs> <op> <rhs>
```

Either `<lhs>` or `<rhs>` must be of variable type. The relational operators are defined both for scalars and for vectors. If either side is a constant scalar, then each component of the other side (perhaps a vector) is compared with this scalar. In all other cases the relations are applied componentwise.

SMPL includes the following relational operators: ‘`<=`’, ‘`>=`’, ’`=`’. For the operators both point and interval arguments are allowed.

2.8.3 The interval inclusion operator

This operator is denoted by ‘`<i>`’, and is used in the form

```
<lhs> <i> <interval>
```

where `<interval>` is an interval number (and thus, a scalar expression).

The `<lhs>` must be of variable type. The ‘`<i>`’ operator is defined both for scalar and for vector left-hand sides. If the left-hand side is a vector, then each component of it is compared with this scalar, just like for the relational operators.

2.8.4 The set containment operator

This operator is denoted by ‘`in`’, and is used in the form

```
<varref> in { <expr-list> }
```

where `<varref>` is a variable reference (cf. Section 2.10) and `<expr-list>` is a comma-separated list of constant scalar expressions.

If `<varref>` is vector-valued (i.e., it refers to a vector variable or is given as a variable subvector), then the relation is interpreted to all of its elements using the right-hand side set.

2.8.5 Mathematical functions

SMPL math functions are interpreted on arguments with arbitrary dimensions. In case of vector and matrix arguments, the functions are applied componentwise. For binary functions, the row and column dimensions of the operands must match. The arguments must follow the function name in parentheses. SMPL recognizes the following functions:
– **Unary functions:** '`exp`', '`log`', '`xlog`', '`sin`', '`cos`', '`atan`', '`abs`', '`sign`', '`pos`', '`neg`', '`sqrt`'.
– **Binary functions:** '`min`', '`max`'.

2.9 The declaration section

2.9.1 Variable declarations

Variables are defined by using the `var` keyword, following one of the syntax lines below:

```
var <varlist> ;                         # single declaration
var <varlist> <rel_op> <const-expr> ; # declaration with initial bounds
var <varlist> >= <const-expr> , <= <const-expr> ;
                                      # declaration with initial bounds
var <varlist> <i> <interval> ;        # declaration with initial bounds
```

A variable list (`<varlist>`) is a comma-separated sequence of `<vardecl>` elements followed by the optional `integer` keyword, where `<vardecl>` is defined as

```
<var-id>
```

or

```
<var-id> { <index1> .. <index2> }
```

The latter notation is used to declare variable vectors, otherwise `<var-id>` is treated as a single (scalar) variable. `<index1>` and `<index2>` defines the index range of the variable, with `<index1>` $\leq$ `<index2>`. In the present version of SMPL, `<index1>` must be equal to 1. **We may not even need different starting indices.** Variable vectors are always declared as column vectors.

Optionally, a list of variables can be declared as integer. Note, that the `integer` keyword applies to *all variables* of the current declaration statement.

Also optionally, identical initial bounds for *all variables* of a declaration statement can be defined

– by a relational operator,

– by using the '`<=`' and '`>=`' operators at the same time with two *constant* expressions, or

– by the interval inclusion operator (Section 2.8) and a *constant* expression.

The third way of specifying initial bounds is a more compact equivalent notation to the second way in case of scalar bounds.

All of the above initial bounds implicitly encode the constraints

```
<varref> <rel_op> <const-expr>
```

```
<const-expr> <= <varref> <= <const-expr>
```

and

```
<varref> <i> <interval>
```

respectively, for the `<varref>` reference of each variable of the declaration statement. Thus, the type and dimensionality of the constant type expressions are allowed according to Section 2.8.

Remark. If the inital bounds are defined by using '`<=`' and '`>=`' at the same time, then the same dimensionality match condition should occur to both constant bounds. That is, the dimensions of the constant expressions must either match and be the same as the dimensionality of the variable, or they both must be of scalar type.

Examples:

```
var x;

var x1, x2, y {1..2} integer <= 5.0; # all vars are declared as integers

var y {1..5} <i> [1.0,2.0];  # components bounded with the same interval

var z1 {1..2}, z2 {1..2} >= [5;
                             7  ]; # componentwise bounds with 1-by-2 rhs

var z >= 3, <= 7;  # equivalent to "var z <i> [3,7]";
```

2.9.2 Parameter declarations

Parameters can be defined by using the `param` keyword, following one of the syntax lines

```
param <param-id> = <const-scalar-expr> ;
param <paramid> = <const-mat-expr> ;
```

where `<const-mat-expr>` is a constant matrix given in the form described in Section 2.4. The dimensionality of `<param-id>` is automatically determined by the size of the rhs in both cases.

Note that the parameter initializers (the rhs expressions) may contain references to previously declared parameters.

2.10 Variable references

SMPL variables can be used in the objective function and constraint sections. All used variables must be previously declared in the declaration section. An SMPL variable (scalar or vector) can be referenced in two different ways:

- Using the name of the variable only. This refers to the respective variable of the same size as it is declared.
- Subvector with indexing. The variable name is followed by bracketed integer list of the variable indices. The elements of the index list are separated by white spaces/newlines.

This syntax refers to a variable vector of the size of the index list, where each component of the variable vector corresponds to the referred component. Note, that the index list need not be in increasing order and it may contain repeated indices, i.e., repeated references to the same vector component. Examples:

```
var x, y {1..5};

y               # refers to a 5-by-1 variable vector

y[1 2 3]        # refers to the column subvector ( y[1] y[3] y[5] )

y[2 2 2]        # refers to the column subvector ( y[2] y[2] y[2] )

x[1]            # subvector indexing is allowed for scalar variables
```

2.11 Parameter references

SMPL parameters references are similar to variables references, extended by some additional functionalities for matrix parameters. All used parameters must be previously declared in the declaration section. An SMPL parameter (scalar, vector, or matrix) can be referenced in three different ways:

– Using the name of the parameter only. This refers to the respective parameter of the same size as it is declared.

– Subvector with indexing. This way of reference is applicable for vector and scalar parameters only, using the syntax described for variables in Section 2.10.

– Direct indexing. This way of reference is applicable for scalars, vectors, and matrices, and follow the standard way of specifying the row and column index of the respective entry as a bracketed pair of integers, separated by a comma. Examples:

```
param p = 1.5;

param q = [ 1.0 & 2.0 & 3.0;
            4.0 & 5.0 & 6.0 ];

q                           # refers to the whole q matrix

q[1,2]                      # refers to the resp. entry of q (i.e., 2.0)

p[1 1]                      # refers to the column vector ( 1.5 1.5 )

p[1,1]                      # refers to p itself
```

2.12 The input variable section

The section is used to list the original variable of the optimization model, that is, those variables, which are used not only for substitutions/simplifications. The section consist of at most one statement in the form

```
input <varname-list> ;
```

where `<varname-list>` is a comma-separated list of a subset of the *names* of the previously declared variables. (That is, indexing is not allowed, and variables declared as vectors must appear without their range declaration.)

If the input variable section is empty, then it is assumed that *all* declared variables are input variables.

2.13 Expressions

SMPL expressions are, in general, considered as matrices. However, the terms 'scalar' and 'vector' expressions are often used to describe special matrix expressions.

There are two different types of SMPL expressions:

- Constant type expressions. These are expressions built from point and interval numbers and constant matrices using the SMPL operators (except the relational ones) and mathematical functions. Note that point and interval numbers can be mixed within one expression (SMPL does not make the distinction between constant expressions which are evaluated to points or to intervals.)
- Variable type expressions. These are expressions involving one or more variable.

SMPL expressions are either **primitive** or **compound** expressions. The family of **primitive expressions** is the following:

- point numbers (Section 2.3.1),
- interval numbers, composed of two constant scalar expressions (Section 2.3.3),
- constant matrices (Section 2.4),
- variable references (Section 2.10), and
- parameter references (Section 2.11).

Compound expressions are built from primitive expressions using the SMPL operators (except the relational ones) and mathematical functions of Section 2.8.

Parentheses are allowed in the usual form to overwrite operator precedence and to clarify the evaluation order of SMPL expressions.

The central property of SMPL is that since the components of primitive vector/matrix expressions must be set to the same type, all components of a vector/matrix SMPL expression has the same type as well.

2.14 The objective function section

The syntax of the objective function definition (if exists) is as follows:

```
<obj-hdr> <expr> ;
```

where `<obj-hdr>` is one of the strings 'minimize', 'min.', 'maximize', or 'max.', followed by an optional label, and `<expr>` is a scalar expression. `<expr>` can be either variable or constant type.

2.15 The constraint section

An SMPL constraint statement must follow one of the syntax lines

```
<cstr-hdr> <lhs-expr> <rel-op> <rhs-expr> ;
<cstr-hdr> <left-expr> <= <mid-expr> <= <right-expr> ;
<cstr-hdr> <lhs-expr> <i> <interval> ;
<cstr-hdr> <varref> in { <expr-list> } ;
<cstr-hdr> <varref> = <table> ;
```

where `<cstr-hdr>` is the string 'subject to' or 's. t.' followed by an optional label (there are one or more spaces or tabs allowed between 'subject' and 'to', and zero or more spaces or tabs between 's.' and 't.'). `<rel-op>` is a relational operator. According to the GloptLab philosophy, the following four constraint types are allowed:

- `<rel-op>` is '`=`', one side is a variable reference, the other one is a univariate expression. (In a univariate expression each component is an expression involving one scalar variable, or one component of a vector variable only.)
- `<rel-op>` is '`=`', one side is a quadratic variable expression, the other is a constant expression.
- *Either*
  - `<rel-op>` is '`<=`' or '`>=`', and one side is a quadratic variable expression, the other is a constant expression, *or*
  - two '`<=`' operators are used in a chain, `<mid-expr>` is quadratic variable expression, `<left-expr>` and `<right-expr>` are constant expressions, *or*
  - the operator is `<i>` and `<lhs-expr>` is a quadratic variable expression. (In case of scalar bounds, this is equivalent to the previous specification.)
- The constraint is a set constraint using the `in` operator. Remember that if `varref` on the lhs is a vector expression, then the set containment relation is applied to all of its components.
- The constraint is a *table constraint*, where `<table>` is a table described in Section 2.5. Note that if `varref` on the lhs is a vector expression, then the table constraint is applied to all of its components.

As an additional dimensionality condition required when using the chain inequality syntax, see the remark in 2.9.1.

3 An example SMPL file

Below is a meaningless SMPL file listed to demonstrate some current features of the language.

```
# test file for glparser

# variable declarations
var x {1..5} <i> [1.0,2.0];
var y, z{1..3} integer <= 523;
var w;
```

```
# parameter declarations
param p1 = [ 5.0 & 4.0 & 3.0 ; 2.0 & 3.0 ; 1.0 ];
param p5 = [ - (15+7) & 14 & 13 ; 12 & 13 ; p1[3,3] ];
param p3 = 33.0?u6;
param p2 = [ 523/10 & 234/10 & [675/10,685/10] ]; # dummy comment
param p4 = 2*p2[1 3 1]';

# input (original) variables
input x, y, z;

# minimize a quadratic expression
minimize obj: -(z' * p5 * z) + p2 * x[1 2 3];

# constraint: varref (scalar) = univariate function (scalar)
s.t. w = (y - sin(p3)) * sin(cos(y[1]))^2 - min((-y + 2),2*y);

# constraint: univariate function (scalar) = varref (scalar)
s.t. -y + 42 - ( 4*y -2*sin(y) ) * sqrt((sin(y)-cos(y))) = x[1];

# constraint: varref (vector) = univariate function (vector)
s.t. x[2 3 4] = cos(sin(z) + min(z,p2'));

# constraint: quadratic expression (scalar) >= constant expression (scalar)
s.t. z[1]^2 - - z[2]^2 >= 1e+005;

# constraint: quadratic expression (scalar) bounded by constant scalar
# expressions
s.t. 1e+005 <= (x[1 2 3]+x[2 3 4]-x[1 1 1])' * p2' <= 1e+006;

# constraint: quadratic expression (scalar) <i> interval
s.t. x' * z[1 2 1 2 1] + 5 <i> [ p1[1,1] , 2*p1[2,2] ];

# constraint: constant expression (vector) = quadratic expression (vector)
s.t. [ 10 ; 10 ; 10 ] = p1*(-z) - p2'*y;

# constraint: a set constraint (applies to both z[2] and z[3])
s.t. z[2 3] in { 3+p1[1,1], p2[2], 1, 3*4 };

# constraint: a table constraint
s.t. x[1] = pcon ( [ 1   & p2[1] ;
                     4/3 & 1.0   ;
                     5/3 & p2[2] ;
                     2   & 1.0    ] );
```