# S2MPJ and CUTEst optimization problems for Matlab, Python and Julia

S. Gratton *and Ph. L. Toint †

11 VII 2025

**Abstract**

A new suite of test problems for optimization is presented, which contains a large fraction of the problems in the CUTEst collection. The problems are supplied in the form of Matlab, Python and Julia files allowing the computation of values and derivatives of the objective function and constraints directly within "native" Matlab, Python or Julia, without any additional installation or interfacing with MEX files or Fortran programs. These files are produced by a new decoder (written in Matlab) for the original SIF descriptions in the CUTEst collection. When used within Matlab, the new problem files optionally support reduced-precision computations.

**Keywords:** nonlinear optimization, test problems, CUTEst, benchmarking, optimization software.

## 1 Introduction

Published nearly thirty years ago [2] and updated [7, 8] since, the CUTEst testing environment and its associated collection of test problems for continuous optimization have been used extensively* by the mathematical optimization community for the design, testing and comparison of unconstrained and constrained optimization software. The test problem collection has indeed grown over time to cover many other proposals such as the Argonne test set [15], Toint's collection [18], Buckley's problems [3], COPS [1], the Maros and Meszaros quadratic programming problems [14], the Hock and Schitkowski collection [11] and the Lukšan and Vlček test set [13], to cite a few of the earlier ones. While the updates have provided improved code, precompiled forms for numerous operating systems and compilers, new tools and new problems, the basic design of the environment has not fundamentally changed since 1995. Each of the test problem is described in a file written in "Standard Input Format" (SIF) [4, 9], an (admittedly somewhat obscure) scheme extending the previous MPS standard [5] for linear programming. To use this file, it was (and still is) necessary to "decode" it using a decoder/compiler written in Fortran, which produces a (possibly large) data file (`OUTSDIF.d`) and a small set of Fortran subroutines encoding the nonlinear parts of the problem. The

*Université de Toulouse, INP, IRIT, Toulouse, France. Email: serge.gratton@enseeiht.fr. Work partially supported by 3IA Artificial and Natural Intelligence Toulouse Institute (ANITI), French "Investing for the Future - PIA3" program under the Grant agreement ANR-19-PI3A-0004"

†NAXYS, University of Namur, Namur, Belgium. Email: philippe.toint@unamur.be

*Google Scholar reports more than 2300 citations, and growing.

user then compiles these Fortran subroutines, loads them with his/her optimization package and calls the Fortran tools provided in the CUTEst environment to evaluate the problem's objective function and constraints, possibly with their first and second derivatives.

While this setting has clearly proved its usefulness, its integration with the evolving computing environment and emerging programming languages has not always been easy. An interface with Matlab was first produced using the "MEX files" mechanism available in that language, but this proved difficult to maintain for all computer architectures. Other tools such as the `pycutest` interface [6] for Python, the `matcutest` [20] interface to Matlab and CUTEst.jl [16] interface to Julia have been proposed to make the test problem collection and associated tools available in today's computing environment, sometimes at the cost of extra complications. For instance, using the `pycutest` interface requires the code to be executed in a well-defined environment running the Fortran decoder, which is typically made easy by using a "docker" container where the original Fortran tools are precompiled. The `matcutest` interface does not currently allow the user to modify the problem parameters (such as dimension or discretization mesh). Maybe more significantly, all these interfaces are interfaces with the Fortran SIF decoder and the Fortran CUTEst evaluation tools. Although this is fine for computing environments where Fortran is available or if working in a docker container is not too restrictive, one must admit that the use of Fortran has globally declined since 1995 while that of Python has significantly increased [17]. Thus reliance on a unique tool written in this former language may make the continued access and use of the test problem collection less widespread.

The purpose of this paper is to introduce new standalone Matlab, Python and Julia versions of a large fraction of the CUTEst problems, allowing the computation of values of the objective function and constraints of CUTEst problems, as well as that of their derivatives, directly within "native" Matlab, Python or Julia environments, without any additional installation or interfacing with MEX files or Fortran programs (or, for Python and Julia users, with Matlab), using at most three lines of code. These new problem instances therefore provide an easier access to the collection at the price of slower execution times. When used in the Matlab environment, an option is provided to use the language variable-precision tools, allowing the user to select a smaller number of digits in all evaluations. The paper also describes S2MPJ[†], the tool that has been used to produce the new problem files.

Our presentation is organized as follows. We first focus, in Section 2, on the description of the newly-decoded problem files and on how to use them in practice, successively considering the Matlab, Python and Julia environments. The way to select classes of test problems is described in Section 3 and the reduced-precision option in Section 4. We then examine our methodology in more detail. After recalling some details of the structure of the CUTEst problems in Section 5.1, we outline, in Section 5.2, the main features and mechanisms allowing S2MPJ to interpret SIF files directly, describe its formal specifications and discuss its limitations. Section 6 presents our tests to ensure coherence of the new Matlab, Python and Julia problem files and evaluation tools with the Fortran-based version. Section 8 describes how the problem files, the S2MPJ decoder and the evaluation tools can be obtained. Finally, Section 9 summarizes our approach and provides some perpectives for future developments.

[†]For SIF to Matlab, Python and Julia.

# 2 The S2MPJ problem files and how to use them

The new Matlab, Python and Julia problem files are intended for the benchmarking of algorithms whose purpose is to solve the problem

$$\min_{x \in \mathbb{R}^n} f(x) \tag{2.1}$$

where $f$ is a function from $\mathbb{R}^n$ into $\mathbb{R}$, possibly subject to bounds constraints of the form

$$x_j^{\rm low} \leq x_j \leq x_j^{\rm upp} \quad (j \in \{1, \ldots, n\}) \tag{2.2}$$

(where $x_j$ is the $j$-th component of $x$) and to linear or general constraints written as

$$c_i^{\rm low} \leq c_i(x) \leq c_i^{\rm upp} \quad (i \in \{1, \ldots, m\}), \tag{2.3}$$

for some function $c$ from $\mathbb{R}^n$ into $\mathbb{R}^m$. In these expressions $x_j^{\rm low}$, $x_j^{\rm upp}$, $c_i^{\rm low}$ and $c_i^{\rm upp}$ may take infinite absolute values. Some problems also specify a lower and/or upper bounds on the objective function's values. The problem files are designed to be called directly from a user's Matlab, Python or Julia program in order to return values of the starting point and bounds, and to compute values of $f(x)$ and $c(x)$ (for a user-supplied $x$) as well as that of their derivatives when available. For problems with general constraints $c(x) = (c_1(x), \ldots, c_m(x))^T$ and given a vector $y \in \mathbb{R}^m$ of multipliers, the values (and derivatives) of the Lagrangian function

$$L(x, y) = f(x) + y^T c(x), \tag{2.4}$$

may also be obtained from the problem files, as well as the products of the objective functions' Hessian, the constraints' Jacobian and the Lagrangian's Hessian times a user-supplied vector. As it may convenient to restrict one's attention to a subset $I = \{i_1, \ldots, i_p\}$ of the constraint's indices $\{1, \ldots, m\}$, values and derivatives of '$I$-restricted' variants of $c(x)$ and $L(x, y)$ given by

$$c_I(x) = (c_{i_1}(x), \ldots, c_{i_p}(x))^T$$

and

$$L_I(x, y) = f(x) + y_I^T c_I(x). \tag{2.5}$$

where $y_I = (y_{i_1}(x), \ldots, y_{i_p}(x))^T)$ may also be computed.

The problems files themselves rely on (supplied) language specific libraries (`s2mpjlib.m`, `s2mpjlib.py` and `s2mpjlib.jl`, respectively).

## 2.1 The Matlab problem files

### 2.1.1 Matlab problem files: how to use them

The use of the Matlab problem files is typically as follows. Suppose we have an optimization code `myoptimizer.m` which we would like to apply to one of the S2MPJ problems, let's say `probname.m`. Suppose also that the library file `s2mpjlib.m` is in the Matlab path. We may then proceed as follows.

1. The first step is to setup the problem data structure, along with the starting points, bounds and other components of the `pb` and `pbm` structs (see below) by issuing the command

```
[ pb, pbm ] = probname( 'setup', args{:} );
```

at the beginning of `myoptimizer.m`, before any call to function/constraint values or derivatives. In this call, `args{:}` is an optional comma-separated list of problem-dependent arguments (such as problem's dimension, for instance) identified by the string `$-PARAMETER` in the SIF file. If `args{:}` is missing, setup is performed using the SIF file defaults.

2. The second step is to insert the calls for the evaluation of the value(s) of the problem functions (objective and constraints) at a vector $x$, together with values of their first and second derivatives (if requested). Every time in `myoptimizer.m` values of the objective function and, possibly its derivatives at the vector $x$ are needed, they are obtained by issuing one of the commands

```
fx = probname( 'fx', x );

[ fx, gx ] = probname( 'fgx', x );

[ fx, gx, Hx ] = probname( 'fgHx', x );
```

The product of the objective function's Hessian (at $x$) times a user-supplied vector $v$ can be obtained by the command

```
Hxv = probname( 'fHxv', x, v );
```

If the problem has constraints other than bounds on the variables, the technique to obtain their values or that of their derivatives is very similar. The values (and that of derivatives, if desired) are obtained by the commands

```
cx = probname( 'cx', x );

[ cx, Jx ] = probname( 'cJx', x );

[ cx, Jx, Hx ] = probname( 'cJHx', x );
```

It is also possible to restrict one's attention to a subset $I$ of the constraints (i.e. using $c_I(x)$, the '$I$-restricted' version of $c(x)$) by using

```
cIx = probname( 'cIx', x, I );

[ cIx, JIx ] = probname( 'cIJx', x, I );

[ cIx, JIx, HIx ] = probname( 'cIJHx', x, I );
```

The product of the (potentially $I$-restricted) constraints' Jacobian (at $x$) times $v$ is computed by issuing one of the commands

```
Jxv = probname( 'cJxv' , x, v );

JIxv = probname( 'cIJxv', x, v, I );
```

and the product of the (potentially $I$-restricted) Jacobian transposed times $v$ is obtained by

```
Jxv = probname( 'cJtxv' , x, v );

JIxv = probname( 'cIJtxv', x, v, I );
```

The value (and derivatives) of the Lagrangian function $L(x, y)$ at $(x, y)$ is obtained by one of the commands

```
Lxy = probname( 'Lxy', x, y );

[ Lxy, Lgxy ] = probname( 'Lgxy', x, y );

[ Lxy, Lgxy, LHxy ] = probname( 'LgHxy', x, y );
```

while the product of the Lagrangian's Hessian times a vector $v$ can be computed with the command

```
HLxyv = probname( 'HLxyv', x, y, v );
```

Finally, and as above for constraints, the Lagrangian may be $I$-restricted in the commands

```
[LIxy, gLIxy, HLIxy ] = probname( 'LIxy', x, y, I );

HLIxyv = probname( 'LIHxyv' , x, y, v, I );
```

Because `probname.m` uses a persistent structure to pass problem structure across its various actions, such calls are valid as long as the Matlab variable space is not cleared and as long as another `otherproblem.m` file is not called.

For a more specific example of use in Matlab, see Appendix B.

### 2.1.2 Matlab problem files: interface specification

Each Matlab problem file `probname.m` is a Matlab function with the following user interface. A call of the form

```
varargout = probname( action, varargin )
```

produces result(s) stored in `varargout` depending on the following choices of the 'action' argument and the number of requested outputs, according to the following rules.

```
[ pb, pbm ] = probname( 'setup', problem_parameters )
```

• The fields of the struct `pb` are defined as follows:

`pb.name` is a string containing the problem's name

`pb.sifpbname` if present, is a string containing the original name of the SIF file before its modification to ensure Matlab/Python/Julia compatibility (see Section 5.2.1)

`pb.n` is an integer giving the problem's number of variables

`pb.nob` is an integer giving the number of objective groups

`pb.nle` if present, is an integer giving the number of `<=` constraints

`pb.neq` if present, is an integer giving the number of `==` constraints

`pb.nge` if present, is an integer giving the number of `>=` constraints

`pb.m` is an integer giving the total number of general constraints

`pb.lincons` if present, is a integer vector containing the indices of the linear constraints

`pb.pbclass` is the problem's extended SIF classification (see Section 7)

`pb.x0` is a real vector giving the problem's starting point

`pb.xlower` is a real vector giving $x_j^{\rm low}$, the problem's lower bounds on the variables (see (2.2))

`pb.xupper` is a real vector giving $x_j^{\rm upp}$, the problem's upper bounds on the variables (see (2.2))

`pb.xtype` is a string whose $i$-th position describes the type of the $i$-th variable:

- `'r'`: the variable is real
- `'i'`: the variable is integer
- `'b'`: the variable is binary (zero-one)

If `xtype` is not a field of `pb`, all variables are assumed to be real.

`pb.xscale` if nonempty, is a real vector containing the scaling factors $\varsigma_j$ to be applied by the user on the occurrences of the variables in the linear terms of the problem's groups (see (5.2)). If empty, the scaling factors (if any) are applied to the relevant terms internally to the decoder without need for further user action (see the `pbxscale` option of S2MPJ in Section 5.2.2)

`pb.y0` if present, is a real vector containing the starting values for the constraint's multipliers

`pb.clower` if present, is the real vector $c_i^{\rm low}$ of lower bounds on the constraints values (see (2.3))

`pb.cupper` if present, is the real vector $c_i^{\rm upp}$ of upper bounds on the constraints values (see (2.3))

`pb.objlower` if present, is a real lower bound of the objective function's value

`pb.objupper` if present, is a real upper bound of the objective function's value

`pb.xnames` if present, is a cell of strings of length `pb.n` containing the name of the variables

`pb.cnames` if present, is a cell of strings of length `pb.m` containing the name of the constraints.

`pbm.objderlvl` is an interger in $\{0, 1, 2\}$ giving the order of the highest derivative of the objective function supplied for the problem. If the field is not present, first and second derivatives are assumed to be available.

`pbm.conderlvl` is an interger in $\{0, 1, 2\}$ giving the order of the highest derivative of the constraints functions supplied for the problem. If of length one, the level is the same for all constraints. Otherwise, it is of length `pb.m` and its $i$-th component gives the derivative level for the $i$-th constraint. If the field is not present, first and second derivatives are assumed to be available for all constraints.

Note that `pb.objderlvl` and `pb.conderlvl` may be checked by the user to obtain the level of available derivatives, which could limit the range of evaluation actions that are meaningful for the problem at hand.

In the 'setup' call to the problem file, `problem_parameters` is a coma-separated list of parameters identified by a `$-PARAMETER` string in the SIF file. They are assigned in the order in which they appear in `varargin`, which is the same as that used in the SIF file as well as in the Matlab, Python or Julia problem files. If `varargin` is too short in that it does not provide a value for a parameter of the SIF file, the SIF default value is used.

If constraints are present, they are ordered[‡] as follows:

| | |
|---|---|
| 1,..., `pb.nle` | : constraints of `<=` type, |
| `pb.nle+1`,..., `pb.nle+pb.neq` | : constraints of `==` type, |
| `pb.nle+pb.neq+1`,..., `pb.nle+pb.neq+pb.nge` | : constraints of `>=` type. |

If the problem has no constraints or bound constraints only, then `lincons`, `y0`, `clower`, `cupper` and `cnames` are not fields of the `pb` struct returned on setup. `pb.m` is still defined in this case, but its value is 0. The fields `objlower` and `objupper` may also be missing if no value is provided in the SIF file. When specific S2MPJ options are used (see Section 5.2.2), the fields `xnames` and/or `cnames` may be missing from the `pb` struct.

• The `pbm` struct is supplied *for information and debugging only*. It contains details of the problem's inner structure and must not be modified by the user. A more detailed description of its content is given in Section 5.2.

A call to `probname( 'setup', ...)` *must precede* any call to `probname` with other actions (in order to setup the problem's data structure).

```
fx = probname( 'fx', x )
```

`fx` is $f(x)$, the value of the objective function at $x$,

```
[ fx, gx ] = probname( 'fgx', x )
```

`fx` is $f(x)$, the value of the objective function at $x$,

`gx` is $\nabla^1_x f(x)$, the value of the objective function's gradient at $x$,

Note that the call `[ fx, gx ] = probname( 'fx', x )` produces the same results.

```
[ fx, gx, Hx ] = probname( 'fgHx', x )
```

`varargout{1}` is $f(x)$, the value of the objective function at $x$,

`varargout{2}` is $\nabla^1_x f(x)$, the value of the objective function's gradient at $x$,

[‡]This ordering may be superseded by setting the `keepcorder` flag in the S2MPJ options (see Section 5.2.2).

`varargout{3}` is $\nabla_x^2 f(x)$, the value of the objective function's Hessian at $x$,

Note that the call `[ fx, gx, Hx ] = probname( 'fx', x )` produces the same results.

```
cx = probname( 'cx', x )
```

`cx` is $c(x)$, the vector of constraint's values at $x$,

```
[ cx, Jx ] = probname( 'cJx', x )
```

`cx` is $c(x)$, the vector of constraint's values at $x$,

`Jx` is the matrix $J(x)$, the constraint's Jacobian at $x$,

Note that the call `[ fx, gx ] = probname( 'cx', x )` produces the same results.

```
[ cx, Jx, Hix ] = probname( 'cJHx', x )
```

`cx` is $c(x)$, the vector of constraint's values at $x$,

`Jx` is the matrix $J(x)$, the constraint's Jacobian at $x$,

`Hix` is a cell whose $i$-th entry is $\nabla_x^2 c_i(x)$, the Hessian of the $i$-th constraint at $x$,

Note that the call `[ fx, gx, Hix ] = probname( 'cx', x )` produces the same results.

```
cIx = probname( 'cIx', x, I )
```

`cIx` is $c_I(x)$, the vector of the $I$-restricted constraint's values at $x$,

```
[ cIx, JIx ] = probname( 'cIJx', x, I )
```

`cIx` is $c_I(x)$, the vector of the $I$-restricted constraint's values at $x$,

`JIx` is the matrix $J_I(x)$, the Jacobian of the $I$-restricted constraints at $x$,

```
[ cIx, JIx, HIx ] = probname( 'cIJHx', x, I )
```

`cIx` is $c_I(x)$, the vector of the $I$-restricted constraint's values at $x$,

`JIx` is the matrix $J_I(x)$, the Jacobian of the $I$-restricted constraints at $x$,

`HIx` is a cell whose $i$-th entry is $\nabla_x^2 c_\ell(x)$, the Hessian of the $\ell$-th constraint at $x$ where $\ell$ is the $i$-th entry in $I$,

```
Hxv = probname( 'fHxv', x, v )
```

`Hxv` is the vector $\nabla_x^2 f(x)v$, the product of the objective function's Hessian at $x$ times the vector $v$,

```
Jxv = probname( 'cJxv', x, v )
```

`Jxv` is the vector $J(x)v$, the product of the constraint's Jacobian at $x$ times the vector $v$,

```
JIxv = probname( 'cIJxv', x, v )
```

`JIxv` is the vector $J_I(x)v$, the product of the constraint's $I$-restricted Jacobian at $x$ times the vector $v$,

```
Jtxv = probname( 'cJtxv', x, v )
```

`Jtxv` is the vector $J(x)^Tv$, the product of the constraint's Jacobian at $x$ transposed times the vector $v$,

```
JItxv = probname( 'cIJtxv', x, v )
```

`JItxv` is the vector $J_I(x)^Tv$, the product of the constraint's $I$-restricted Jacobian at $x$ transposed times the vector $v$,

```
Lxy = probname( 'Lxy', x, y )
```

`Lxy` is $L(x, y)$, the value of the problem's Lagrangian at $(x, y)$,

```
[ Lxy, gLxy ] = probname( 'Lgxy', x, y )
```

`Lxy` is $L(x, y)$, the value of the problem's Lagrangian at $(x, y)$,

`gLxy` is $\nabla_x^1 L(x, y)$, the value of the Lagrangian's gradient with respect to $x$ taken at $(x, y)$,

Note that the call `[ Lxy, Lgxy ] = probname( 'Lxy', x, y )` produces the same results.

```
[ Lxy, gLxy, HLxy ] = probname( 'LgHxy', x, y )
```

`Lxy` is $L(x, y)$, the value of the problem's Lagrangian at $(x, y)$,

`gLxy` is $\nabla_x^1 L(x, y)$, the value of the Lagrangian's gradient with respect to $x$ taken at $(x, y)$,

`HLxy` is $\nabla_x^2 L(x, y)$, the value of the Lagrangian's Hessian with respect to $x$ taken at $(x, y)$,

Note that the call `[ Lxy, gLxy, HLxy ] = probname( 'Lxy', x, y )` produces the same results.

```
LIxy = probname( 'LIxy', x, y, I )
```

`LIxy` is $L_I(x, y)$, the value of the problem's $I$-restricted Lagrangian at $(x, y)$,

```
[ LIxy, gLIxy ] = probname( 'LIgxy', x, y, I )
```

`LIxy` is $L_I(x, y)$, the value of the problem's $I$-restricted Lagrangian at $(x, y)$,

`gLIxy` is $\nabla_x^1 L_I(x,y)$, the value of the $I$-restricted Lagrangian's gradient with respect to $x$ taken at $(x,y)$,

Note that the call `[ LIxy, gLIxy ] = probname( 'Lxy', x, y, I )` produces the same results.

```
[ LIxy, gLIxy, HLIxy ] = probname( 'LIgHxy', x, y, I )
```

`LIxy` is $L_I(x,y)$, the value of the problem's $I$-restricted Lagrangian at $(x,y)$,

`gLxy` is $\nabla_x^1 L_I(x,y)$, the value of the $I$-restricted Lagrangian's gradient with respect to $x$ taken at $(x,y)$,

`HLxy` is $\nabla_x^2 L_I(x,y)$, the value of the $I$-restricted Lagrangian's Hessian with respect to $x$ taken at $(x,y)$,

Note that the call `[ LIxy, gLIxy, HLIxy ] = probname( 'LIxy', x, y )` produces the same results.

```
HLxyv = probname( 'LHxyv', x, y, v )
```

`HLxyv` is the vector $\nabla_x^2 L(x,y)v$, the product the problem's Lagrangian Hessian with respect to $x$ taken at $(x,y)$ times the vector $v$,

```
HLIxyv = probname( 'LIHxyv', x, y, v, I)
```

`HLIxyv` is the vector $\nabla_x^2 L_I(x,y)v$, the product the problem's $I$-restricted Lagrangian Hessian with respect to $x$ taken at $(x,y)$ times the vector $v$.

The actions `cx`, `cIx`, `Jxv`, `JIxv`, `Lxy`, `LIxy`, `HLxyv` or `HLIxyv` are of course meaningless when the problem is unconstrained or only has bound constraints.

## 2.2 The Python problem files

### 2.2.1 Python problem files: how to use them

The use of a Python problem file is very similar to that of a Matlab file. We now assume that the `s2mpjlib.py` is in the system's path. The various steps described above for the the use of the Matlab problem file are now (given the `probname.py` file) as follows. After importing the functions of the `probname` module by

```
from probname import *
```

the setup of the problem is now performed by a call of the form

```
probname = probname( args[:]  )
```

where `args[:]` is a comma separated list of problem-dependent arguments. The subsequent evaluation tasks are then obtained by calling one (or more) of

```
fx = probname.fx( x )
fx, gx = probname.fgx( x )
x, gx, Hx = probname.fgHx( x )
Hxv = probname.fHxv( x, v )
cx = probname.cx( x )
cx, Jx = probname.cJx( x )
cx, Jx, HXs = probname.cJHx( x )
cJxv = probname.cJxv( x, v )
cJtxv = probname.cJtxv( x, v )
cIx = probname.cIx( x, I )
cIx, cIJx = probname.cIJx( x, I )
cIx, cIJx, cIJHx = probname.cIJHx( x, I )
cIJv = probname.cIJxv( x, v, I )
cIJtv = probname.cIJtxv( x, v, I )
Lxy = probname.Lxy( x, y )
Lxy, Lgxy = probname.Lgxy( x, y )
Lxy, gLxy, HLxy = probname.LgHxy( x, y )
LHxyv = probname.LHxyv( x, y, v )
LIxy = probname.LIxy( x, y, I )
LIxy, gLIxy = probname.LIgxy( x, y, I )
LIxy, gLIxy, HLIxy = probname.LIgHxy( x, y, I )
LIHxyv = probname.LIHxyv( x, y, v, I )
```

For a more specific example of use in Python, see Appendix B.

### 2.2.2 Python problem files: interface specification

Each Python problem file describes a Python class whose name is that of the problem and which is derived from the parent `CUTEst_problem` class described in the `s2pmjlib.py` file. It inherits the evaluation methods of this parent class, which are organized in a manner similar to actions for the Matlab output file, except that the Matlab 'setup' action is replaced by a simple call to the class with the problem parameters, the setting up of the problem structure(s) being then performed by the `__init__` method of the class. It also inherits all fields described above for the `pb` and `pbm` structs in Matlab. So, if the problem is given by the `probname` class (in the `probname.py` problem file), setting it up is achieved by the call

```
problem = probname(args[:])
```

where `args[:]` is an optional comma-separated list of problem-dependent arguments. The problem's starting point is then given by `problem.x0`, while the evaluation task corresponding to the Matlab call `[outputs] = probname( action, x, other_args );` is performed by the Python call

```
ouputs = problem.action( x, other_args )
```

for all actions listed above except 'setup'. The real vectors resulting from the various evaluations or being present in the fields of `problem` are column-oriented numpy.ndarrays. Real matrices are sparse.csr matrices, or lists of sparse.csr matrices (for the Hessians of the constraints).

## 2.3 The Julia problem files

### 2.3.1 Julia problem files: how to use them

Finally, the typical use of the Julia problem file is as follows. After including the S2MPJ library and the `probname` functions in the Julia program using

```
include( "s2mpjlib.jl" )
include( "probname.jl" )
```

(possibly giving the full path name of these files), the setup of the problem is performed by a call of the form

```
pb, pbm = probname( "setup", args[:]  )
```

where `args[:]` is an optional comma separated list of problem-dependent arguments. The subsequent evaluation tasks are then obtained by calling one (or more) of

```
fx = probname( "fx", pbm, x )
fx, gx = probname( "fgx", pbm, x )
fx, gx, Hx = probname( "fgHx", pbm, x )
Hxv = probname( "fHxv", pbm x, v )
cx = probname( "cx", pbm, x )
cx, Jx = probname( "cJx", pbm, x )
cx, Jx, HXs = probname( "cJHx", pbm, x )
cJxv = probname( "cJxv", pbm, x, v )
cJtxv = probname( "cJtxv", pbm, x, v )
cIx = probname( "cIx", pbm, x, I )
cIx, cIJx = probname( "cIJx", pbm, x, I )
cIx, cIJx, cIJHx = probname( "cIJHx", pbm, x, I )
cIJv = probname( "cIJxv", pbm, x, v, I )
cIJtv = probname( "cIJtxv", pbm, x, v, I )
Lxy = probname( "Lxy", pbm, x, y )
Lxy, Lgxy = probname( "Lgxy", pbm, x, y )
Lxy, Lgxy, LgHxy = probname( "LgHxy", pbm, x, y )
LHxyv = probname( "LHxyv", pbm, x, y, v )
LIxy = probname( "LIxy", pbm, x, y, I )
LIxy, LIgxy = probname( "LIgxy(", pbm, x, y, I )
LIxy, LIgxy, LIgHxy = probname( "LIgHxy", pbm, x, y, I )
LIHxyv = probname( "LIHxyv", pbm, x, y, v, I )
```

Note that `pbm` always occurs as the second input argument. Also note that `probname` may be replaced by `pbm.call` in the evaluation actions (for instance if one wishes to obtain values within a function without hardcoding the problem name in the function, see the Julia example in Appendix B).

For a more specific example of use in Julia, see Appendix B

### 2.3.2 Julia problem files: interface specification

Each Julia problem file is a cross between the Matlab and Python ones. It is a Julia function whose interface is identical to that of the Matlab files, *except* that the second argument in the

call must be the `pbm` struct. This struct now has an additional field `call` so that `pbm.call` contains the Julia call to the problem file after execution with the 'setup' action.

# 3 Selecting classes of test problems

The libraries `s2mpjlib` also provide, for all three languages, a tool to list test problems belonging to user-specified classes. This tool uses an extension[§] of the SIF problem classification described in [2]. The (extended) classification string is of the form

```
X-XXXXr-XX-n-m
```

and only differs form the standard SIF classification by the addition of the first three characters. In this extended string, the characters `X` describe problem characteristics (such an collection of origin, type of variables, nonlinearity type, constraint type, origin and regularity), while `n` and `m` specify the (fixed or variable) numbers of variables and constraints, respectively (the details are described in Appendix C). For instance, the classification string for the CUTEst Rosenbrock's problem in two variables is `C-CSUR2-AN-2-0`, where the first `C` indicates that problem belongs to the CUTEst collection, the second `C` that the problem has continuous variables only, `S` indicates that the objective function is a sum of squares, `U` that the problem is unconstrained, `R2` that the problem is regular and that derivatives up to order two are available, `A` that the problem is academic (designed for testing algorithms), `N` that is does not feature internal variables in its GPS structure (see Section 5.1), `2` that is has two variables and `0` that it has zero constraints. A class of problems is then specified by replacing by a dot each character of the classification string for which all values are acceptable (the dot plays the role of a wildcard character). For instance, the string `C-CSUR2-..-V-0` defines the class of all unconstrained sum-of-squares CUTEst problems with available gradient and Hessian and with a variable number of continuous variables. The classification strings `unconstrained`, `bound-constrained`, `fixed-variables`, `general-constraints`, `variable-n` and `variable-m` are also allowed.

Let us denote by `class` the string defining the class of problems of interest. After connecting to the parent of the directory containing the Matlab, Python or Julia problems, the list of problems in this class can be obtained in Matlab by the command

```
s2mpjlib( 'select', class )
```

in Python by

```
from s2mpjlib import s2mpjlib_select
s2mpjlib_select( class )
```

and in Julia by

```
include("s2mpjlib.jl")
s2mpjlib_select( class )
```

The list of problems is output on the standard output. If desired, it can be redirected to a file by giving the name of the file as an additional arguments in the above calls.

[§]Necessary to allow the S2MPJ collection to grow independently of CUTEst, while being able to track the origin of each problem and its type of variables (the problems CUTEst collection all have continuous variables only).

# 4 Using reduced precision with S2MPJ in Matlab

When using S2MPJ in the Matlab environment, it is possible to use reduced precision arithmetic for all evaluations (of values, derivatives and Hessian-times-vector products), provided the Matlab Symbolic Math Toolbox is installed in the environment. If this is the case, setting up the problem data-structures may now be done by the call

```
[ pb, pbm ] = probname( 'setup_redprec', problem_parameters, ndigits )
```

Note that the action keyword is now `'setup_redprec'` and that a new input argument `ndigits` is now present at the end of the calling sequence. This new argument `ndigits` specifies the number of digits to use in the computation and it must be less or equal to 15.

If the action keyword `'setup_redprec'` is used, the problem data is converted to the requested number of digits at the end of the setup process. As a consequence, its accuracy remains limited by the double-precision computations performed during the setup operations (and also by the accuracy of the SIF-provided reals, see Section 6), justifiying the bound `ndigits <= 15`. Thus only the *evaluation* of function values and derivatives is affected by the reduced-precision request. Moreover, this is only true if the vector `x` at which evaluation is requested is itself passed by the user to the problem file in reduced-precision format. Also note that, for now, reduced-precision Hessians and Jacobians are no longer computed in sparse format. As can be expected, the speed of reduced-precison computation (performed symbolically in Matlab) is slower than in full precision, in our experience (with Matlab 2024a) by a factor sometimes exceeding 100 for problems of moderate size.

Using reduced precision is of course optional: a call to problem setup using the action keyword `'setup'` as described in Section 2 results in standard double-precision computations and sparse Hessians and Jacobians. It is also possible to disable the reduced-precision option when decoding a SIF problem by setting the `redprec` option of `s2mpj` to 0 (see Section 5.2.2). The resulting `probname.m` file is then marginally shorter but will issue an error message if the keyword `'setup_redprec'` is specified. Disabling the default of allowing reduced-precision requests is automatic when decoding a SIF problem file if the Symbolic Math Toolbox is not installed.

# 5 The CUTEst problems and the S2MPJ decoder

The material described in the above sections is sufficient for the use of the test problems supplied by S2MPJ. However, should one decide to modify them beyond specifying their parameters (dimension, etc...) or decode new SIF files, more detail on their structure and how the problem files are produced using the S2MPJ decoder is necessary.

## 5.1 The group-partially-separable structure of the CUTEst problems

All problems in the CUTEst collection are specified as *group-partially-separable* (GPS) problems [4]. Because this is important for understanding what follows, we remind the reader of what this means. In a GPS problem of the type encoded in SIF files, one is interested in minimizing a function from $\mathbb{R}^n$ into $\mathbb{R}$ of the form

$$\min_{x\in\mathbb{R}^n} f(x) = \sum_{i\in\mathcal{G}_{\rm obj}} \frac{F_i[a_i(x),\omega_i]}{\sigma_i} + \tfrac{1}{2}x^T H x, \tag{5.1}$$

that is the sum of a quadratic term $\frac{1}{2}x^T Hx$ and of one or more *objective-function groups* $F_i[a_i(x), \omega_i]/\sigma_i$ for $i$ belonging to some index set $\mathcal{G}_{\rm obj}$, with $F_i[.]$ a (possibly nonlinear) univariate *group function* with parameter(s) $\omega_i$ and

$$a_i(x) = \sum_{j \in \mathcal{E}_i} w_{ij} f_j(U_j x^{\rm e}_{ij}, \tau_{ij}) - \sum_{j=1}^{n} \alpha_{ij} \frac{x_j}{\varsigma_j} - \beta_i. \tag{5.2}$$

In (5.2), $\mathcal{E}_i$ is the index set of *nonlinear elements* for group $i$, element $j \in \mathcal{E}_i$ involving the *element weight* $w_{ij}$, the (typically nonlinear) *element function* $f_j$, the (typically low rank and rectangular) *range matrix* $U_j$, the subvectors $x^{\rm e}_{ij}$ and $U_j x^{\rm e}_{ij}$ of *internal* and *elemental variables* occurring in this element, the *linear coefficients* $\alpha_{ij}$, the *elemental parameter(s)* $\tau_{ij}$, the *group constant* $\beta_i$, the nonzero *group scaling* $\sigma_i$ and the nonzero *variable scalings* $\{\varsigma_j\}_{j=1}^n$. This objective function is said to have internal variables if the columns of $U_j$ are not a selection of those of the identity. The minimization (5.1) may be subject to (possibly infinite) bound constraints of the form (2.2) and to general constraints, each of them involving a single group and written as (2.3) (with possibly infinite $c_i^{\rm low}$ and $c_i^{\rm upp}$) and

$$c_i(x) = \frac{C_i[a_i(x), \omega_i]}{\sigma_i} \quad (i \in \mathcal{G}_{\rm cons}) \tag{5.3}$$

where $\mathcal{G}_{\rm cons}$ is the index set of the groups associated with constraints, $C_i[\cdot]$ is the univariate constraint's group function, $\omega_i$ gives its parameter(s) and $\sigma_i$ is the constraint's scaling. The term $a_i(x)$ also has the form (5.2) (for $\mathcal{E}_i$ now the set of nonlinear elements of group $i \in \mathcal{G}_{\rm cons}$). Finally, $H$ in (5.1) is a symmetric $n \times n$ matrix. We also consider the associated Lagrangian function (2.4).

Within SIF files, group functions ($F_i$ and $C_i$) and element functions ($f_j$) are classified according to named *element types* and *group types*, each specifying whether and which parameters are defined for the type, and, for element types, the number of elemental variables. When an element $j$ is attached to a group $i$ (that is $j \in \mathcal{E}_i$), the name of its type, its weight $w_{ij}$, its elemental variables $U_j x^{\rm e}_{ij}$ and (optionnally) its elemental parameters $\tau_{ij}$ are specified. Similarly, the type's name and (optionally) the group parameters $\omega_i$ are specified for each group $i$. In addition, SIF allows the definition of *global element parameters* and *global group parameters*, that is parameters whose value is constant across all elements or groups.

The notion of the GPS structure originates in the design of the LANCELOT package [4] where the low rank of the range matrices $U_j$ was successfully exploited in the "partitioned updating" strategy for quasi-Newton Hessian approximations [10].

Now, given an optimization problem and a vector of variables $x \in \mathbb{R}^n$, we may typically be interested in

- computing the value of the objective function $f(x)$, possibly with its gradient $g(x) = \nabla^1_x f(x)$ and even its Hessian $H(x) = \nabla^2_x f(x)$;
- computing the vector of constraints' values $c(x)$, possibly with its Jacobian[¶] matrix $J(x) = \nabla^1_x c(x)$ and even the collection of the constraints' Hessian matrices $\{H_i(x) = \nabla^2_x c_i(x)\}_{i=1}^m$;
- given a vector $y$ of multipliers, computing the value of the Lagrangian function $L(x, y)$, possibly with its gradient (with respect to $x$) $\nabla^1_x L(x, y)$ or even its Hessian $\nabla^2_x L(x, y)$;

[¶]The rows of the Jacobian are the transpose of the constraint's gradients.

- given a vector $v \in \mathbb{R}^n$, computing the product $H(x)v$, $J(x)v$ or, given $y$, $\nabla^2_x L(x,y)v$.

We may also be interested in performing the same evaluations using a user-specified "$I$-restricted" set of constraints, where $I \subseteq \{1,\ldots,m\}$.

It is important to note that the above description differs from that used by the Fortran decoder on three points.

1. By default, S2MPJ numbers the problem's general constraints (in $c(x)$, $J(x)$ or $\{H_i(x)\}_{i=1}^m$) starting with "less or equal" inequality (`<=`) constraints, followed by "equality" (`==`) constraints, followed by "larger or equal" (`>=`) inequality constraints. The order in which constraints appear in the SIF file is preserved within these three subsets. By contrast, the Fortran decoder does not reorder constraints, thus preserving the order of appearance in the SIF file. In S2PMJ, it is possible to adhere strictly to the Fortran decoder format when decoding the problem's SIF file by specifying the `options.keepcorder` flag appropriately (see Section 5.2.2).

2. The format (2.3) in which the constraints are described to the user also differs from that used by the Fortran decoder [9], where constraints are described, depending on their type, by
$$\begin{array}{rcll} r_i & \leq & c_i(x) \leq 0 & \text{(for <= constraints)} \\ 0 & = & c_i(x) & \text{(for == constraints)} \\ 0 & \leq & c_i(x) \leq r_i & \text{(for >= constraints)} \end{array} \tag{5.4}$$
for *constraint ranges* $r_i$ (optionally) specified in the problem's SIF file (see Section 2.1 of the SIF report). Note that $r_i$ must be nonpositive for `<=` constraints, or nonnegative for for `>=` ones. This format is, in the authors' view, less intuitive, but may be best suited for use with an optimization code where inequality constraints are described by transforming them to equality ones by the introduction of slack variables. Maintaining the Fortran decode format is possible by setting the `options.keepcformat` flag appropriately when decoding the problems' SIF file (see Section 5.2.2).

3. Finally, the SIF format allows the specification of the variable scaling factors $\varsigma_j$ in (5.2). At this time, the Fortran decoder ignores them entirely (albeit there are plans to provide a tool to pass their values to the user). In S2MPJ, they are explicitly taken into account in that the coefficients of the linear terms in each group are adapted according to (5.2). Again, it is possible to adhere to the Fortran decoder's choice to ignore them by suitably setting the flag `options.pbxscale` when decoding the problem's SIF file, in which case the values of the factors $\varsigma_j$ are passed back to the user (see Section 5.2.2), but full compatibility with the Fortran decoder then requires to ignore their value.

## 5.2 The S2MPJ framework and its decoder

The S2MPJ software framework is somewhat simpler than that described in Section 1 for the Fortran one. It also starts by decoding each problem's SIF file (using the `s2mpj.m` Matlab script), but now produces a single executable ouput file, in Matlab, Python or Julia, which can then be called for evaluating quantities of interest directly from the native Matlab, Python or Julia environment, that is without further interfacing with Fortran (or, for Python and Julia users, with Matlab). In order to avoid making these files longer than necessary, the problem-independent parts of the process are encapsulated in S2MPJ-supplied language-dependent

libraries (`s2mpjlib.m`, `s2mpjlib.py` and `s2mpjlib.jl`) which are called while running the problem file.

We now turn to describing the first step of this process (decoding) and the `s2mpj` script in more detail.

### 5.2.1 Decoder's overview

Let `sifpbname` be the name of a problem's SIF file. Broadly speaking, this file consists of a "data" section, describing the structure of the problem as given by (5.1)-(2.4), with the exception of the definitions for the element functions $f_j(U_j\cdot)$ and the group functions $F_i(\cdot)$ and $C_i(\cdot)$. These are then specified in the second part of the SIF file, in sections called ELEMENTS and GROUPS.

This `sifpbname` file (which must be in the Matlab path) is read one line at a time by S2MPJ, and each line is then translated, sometimes in a somewhat indirect way, into the corresponding Matlab/Python/Julia commands. The decoder

- first sets up the problem data structure, including the value of its constant parameters, in the 'setup' action[‖];
- then includes other actions defining the involved nonlinear functions, using information supplied by the Fortran statements in the SIF ELEMENTS and GROUPS sections;
- adds a final call to the S2MPJ-supplied libraries `s2mpjlib.m` (for Matlab), `s2mpjlib.py` (for Python) or `s2mpjlib.jl` (for Julia) that access the previously defined actions and data-structures to perform the required evaluation tasks.

As a result, S2PMJ produces an problem file called `probname.m`, `probname.py` or `probname.jl` in the current directory, possibly overwriting an existing file with the same name (see Section 5.2.2 below to modify the directory where the SIF file is found and where the problem file is written). This file is intended for direct calls from Matlab, Python or Julia producing values of the objective function, constraints or Lagrangian (possibly with their derivatives), as well as products of the objective function's or Lagrangian's Hessian or constraints' Jacobian times a user-supplied vector (see below). Importantly, the produced Matlab/Python/Julia file, hereafter called the 'problem file', is not extensive, meaning that loops are not unrolled (thereby maintaining a reasonably compact description).

The name `probname` of the problem file is, in most cases, identical to `sifpbname`, but differs from it when the `sifpbname` string starts with a digit, in which case it is prefixed by `n`, or contains one or more of the characters '+', '-', '*' or '/'; these characters are then replaced by `p`, `m`, `t`, and `d`, respectively. This renaming[**] is necessary to allow the problem file to be used as a Matlab/Julia function or as a Python class.

Within the 'setup'/__init__ action, instructions are written in the problem file to define the various problem parameters, as well as vectors of bounds on the variables, constraints and objective, and the variables'/multipliers' starting values. Loops in the SIF file are directly transformed into loops in the problem file. Structured sets of the SIF files (such as variables, groups, elements) are characterized by the fact that their components may be defined

[‖] That is the 'setup' `case` statement in the Matlab problem file or `__init__` method in the Python problem file or the outer `if-then-else` statement in the Julia problem file.

[**] For instance, problems `C-RELOAD` and `10FOLDTR` are renamed `CmRELOAD` and `n10FOLDTR`.

using different names and multi-indexes. Because most optimization codes only recognize linear structures (a vector of variables, a vector of constraints), S2MPJ transforms entities such as variables, groups and elements into linear 'flat' unidimensional structures. Since this transformation may depend on problem internal parameters such as loop limits, themselves depending on problem input arguments only known at runtime, the problem file uses a function[††] supplied in the `s2mpjlib` libraries to compute the relevant indices at runtime. S2MPJ also assign a Matlab/Python/Julia-compatible name to each of these entities. These names are then used in dictionaries associating names and values or names and index.

S2PMJ uses three different structures to pass information between the various components of the framework:

- `pbs` is a structure internal to `s2mpj.m` (and hence invisible to the user) used by the code to pass information on entities like loops, element's or group's type(s) across its different subfunctions;
- `pb` is the structure used to pass information on the problem to the user once the setup/__init__ action is completed;
- `pbm` is the structure used to pass information between the different actions of the problem file (occurring on successive calls) and the evaluation functions/methods in the `s2mpjlib` librairies. It is visible to the user but should not be interfered with. For problems described in GPS format (as is the case of all CUTEst ones), its fields are defined as follows:

  `pbm.name` is a string containing the name of the problem,

  `pbm.objgrps` if present, is the list of indices of the objective groups, that is $\mathcal{G}_{\rm obj}$ in (5.1),

  `pbm.congrps` if present, is the list of indeces of the contraint groups, that is $\mathcal{G}_{\rm cons}$ in (5.3),

  `pbm.A` if present, is the real sparse matrix whose lines $i$ contains the coefficients $\alpha_{ij}$ of the linear terms in the groups (see (5.2)),

  `pbm.Ashape` if present, is an integer vector of length 2 containing the number of rows of `pbm.A` and its number of columns (for Python only),

  `pbm.gconst` if present, is a real vector containing the group's constants $\beta_i$ of (5.2),

  `pbm.H` if present, is a symmetric sparse real matix containing the Hessian matrix $H$ of (5.1),

  `pbm.enames` if present, is a cell of strings containing the names of the nonlinear elements,

  `pbm.elftype` if present, is a cell of strings containing the names of the element's types,

  `pbm.elvar` if present, is a cell of integer vectors containing the indices of the elemental variables for each element,

  `pbm.elpar` if present, is a cell of real vectors containing the values of the elemental parameters for each element,

  `pbm.gscale` if present, is a real vector containing the group's scaling factors $\sigma_i$ in (5.1),

[††] `s2mpj_ii`.

`pbm.grnames` if present, is a cell of strings containing the group's names,

`pbm.grftype` if present, is a cell of strings containing the group's types,

`pbm.grelt` if present, is a cell of integer vectors containing the indices of the nonlinear elements occurring in each group,

`pbm.grelw` if present, is a cell of real vectors containing the weights of the nonlinear elements occurring in each group, that is the $w_{ij}$ in (5.2),

`pbm.grpar` if present, is a cell of real vectors containing the values of the parameters $\omega_i$ of the group functions,

`pbm.efpar` if present, is a real vector containing the values of the global element parameters,

`pbm.gfpar` if present, is a real vector containing the values of the global group parameters,

`pbm.objderlvl` is an interger in $\{0, 1, 2\}$ giving the order of the highest derivative of the objective function supplied for the problem. If a derivative is not supplied its value in a call requesting it is set to `NaN` in Matlab and Julia, and to `numpy.nan` in Python. If the field is not present fisrt and second derivatives are assumed to be available.

`pbm.conderlvl` is an interger in $\{0, 1, 2\}$ giving the order of the highest derivative of the constraints functions supplied for the problem. If of length = 1, the level is the same for all constraints. Otherwise, it is of length `pb.m` and the derivative level for the $i$-th constraint (that is the constraint whose group number is `pbm.congrps(`$i$`)`) is given by `pbm.conderlvl(`$i$`)`. If a derivative is not supplied, its value[‡‡] a call requesting it is set to `NaN` in Matlab and Julia, and to `numpy.nan` in Python. If the field is not present fisrt and second derivatives are assumed to be available for all constraints.

`pbm.ndigs` if present is an integer giving the number of digits requested for reduced-precision evaluations (see Section 4),

`pbm.call` is only present in the Julia setting and contains the function to be used for invoking the problem.

Only the `name` field is mandatory.

Formally, the `s2mpj` decoder is a Matlab function whose input parameters are

`sifpbname:` a string containing the name of the problem to be decoded (S2MPJ then reads the `probname.SIF` file for input);

`varargin:` if present, `varargin{1}` allows the specification of decoding options (see Section 5.2.2). The use of the suitable option is mandatory for decoding SIF files into a Python or a Julia problem file. Components of `varargin` beyond the first are ignored.

The function `s2mpj.m` has three output values:

`probname:` a string containing the name of the Matlab, Python or Julia problem file (without the `.m`, `.py` or `.jl` suffix), possibly after renaming (see above);

[‡‡] Or, for vectors, the value of its first component in Matlab and Julia or its zero-th component in Python, while, for matrices, the value of its (1,1)/[0,0] or [1,1] component.

`exitc`: the number of errors which occured before termination (or crash). A zero value thus indicates error-free execution.

`errors`: a cell/list of length `exitc`, whose entries contain a brief description of the error(s) found.

Beyond those arguments, the `s2mpj` function produces an problem file called `probname.m`, `probname.py` or `probname.jl` in the current directory.

### 5.2.2 Decoding options

S2MPJ allows the specification of a number of options, typically to control file production during execution, save memory, help debugging or remain as close as possible to the original Fortran SIF decoder. These options are accessible by suitably setting the first (and only) variable input argument `varargin{1}`.

If `varargin{1}` is a struct then its fields have the following meaning.

`varargin{1}.language` is is a string specifying the type of output produced by S2MPJ and can take different values:

- `'matlab'`: a Matlab problem file named `probname.m` is written in the current directory, possibly overwriting an existing file with the same name.
- `'python'`: a Python problem file named `probname.py` is written in the current directory, possibly overwriting an existing file with the same name.
- `'julia'`: a Julia problem file named `probname.jl` is written in the current directory, possibly overwriting an existing file with the same name.
- `'stdma'`: the content of a *potential* Matlab problem file is printed on the standard output (no file is produced);
- `'stdpy'`: the content of a *potential* Python problem file is printed on the standard output (no file is produced);
- `'stdjl'`: the content of a *potential* Julia problem file is printed on the standard output (no file is produced);

(default: `'matlab'`)

`varargin{1}.showsiflines` is a binary flag which is true if the SIF data lines must be printed on the standard output. This is intended for debugging and requires the option `varargin{1}.language` to be set to `stdma`, `stdpy` or `stdjl`.
(default: 0);

`varargin{1}.getxnames` is a binary flag which is true if the names of the variables must be returned in `pb.xnames` on exit of the setup action.
(default: 1)

`varargin{1}.getcnames` is a binary flag which is true if the names of the constraints must be returned in `pb.cnames` on exit of the setup action.
(default: 1)

`varargin{1}.getenames` is a binary flag which is true if the names of the nonlinear elements $f_j$ must be returned in `pbm.elnames` on exit of the setup action.
(default: 0)

`varargin{1}.getgnames` is a binary flag which is true if the names of the groups elements must be returned in `pbm.grnames` on exit of the setup action.
(default: 0)

`varargin{1}.pbxscale` is a binary flag which is true if the variable's scaling $\varsigma_j$ (in (5.2)) must be provided in `pb.xscale`, instead of being applied internally.
(default: 0)

`varargin{1}.keepcorder` is a binary flag which is true iff, on exit of the setup action, the constraints must not be reordered to appear in the order `<=`, followed by `==` and then by `>=`, but should instead appear in the order in which they are defined in the SIF file.
(default: 0)

`varargin{1}.keepcformat` is a binary flag which is true if, on exit of the setup action, the constraints must be specified using the SIF format (i.e. specifying ranges and types, see above) instead of specifying lower and upper bounds (`clower` and `cupper`) on the constraints values. If the flag is set, the fields clower and cupper of `pb` are replaced by the field `ctypes` (a cell whose `i`-th component contains a string (`'<='`, `'=='`, or `'>='` ) defining the type of the `i`-th constraint) and, if ranges are defined in the SIF file, a field `ranges` giving the ranges $r_i$ (see (5.4)) of the constraints.
(default: 0)

`varargin{1}.writealtsets` is a binary flag which is true if alternative sets of constants, ranges, bounds, starting points or objective function bounds must be written (as comments) in the problem file.
(default: 0)

`varargin{1}.sifcomments` is a binary flag which is true if the comments appearing in the SIF file must be repeated in the problem file.
(default: 0)

`varargin{1}.disperrors` is a binary flag which is true if the error messages are to be displayed on the standard output as soon as the error is detected.
(default: 1 )

`varargin{1}.sifdir` is a string giving the path to the directory where the problem's SIF file must be read. (default: in the Matlab path)

`varargin{1}.outdir` is a string giving the path to the directory where the problem file must be written.
(default: '.' )

`varargin1.redprec` is a binary flag which is true iff variable precision support must be provided in the 'setup' section of the Matlab problem file.
(default: 1)

`varargin1.dicttype` is a string indicating the type of dictionary to be used by Matlab:

**'native':** use the dictionary facility included in the Matlab language,

**'custom':** use a dictionary based on containers maps.

(default: 'custom')

Not every of the above fields must be defined, each field being tested for presence and value individually. If `varargin{1}` is not a Matlab struct or is not present, all options take their default values.

All problem files supplied by S2MPJ are decoded from the original CUTEst SIF files by applying the S2MPJ decoder with its default options. Should other options be desired, it is then necessary to re-decode the SIF file. This operation requires Matlab to be available.

- For Matlab problems, this is achieved by issuing the command

  ```
  PROBLEM = s2mpj( 'SIFPROBLEM', options );
  ```

  This assumes that the `SIFPBNAME` file is in the Matlab path. A file `PROBLEM.m` is then produced in the current directory (potentially overwriting an existing one with the same name).

- For Python problems, the decoding of the SIF file and production of the `PROBLEM.py` file containing the `PROBLEM` class must be performed by the Matlab call

  ```
  PROBLEM = s2mpj( 'SIFPBNAME', inpy )
  ```

  where `inpy` is a struct describing the S2MPJ options with `inpy.language = 'python'`.

- For Julia problems, the `PROBLEM.jl` problem file is produced from the `PROBLEM.SIF` file by the Matlab call

  ```
  PROBLEM = s2mpj( 'SIFPBNAME', injl )
  ```

  where `injl` is a struct describing the S2MPJ options with `injl.language = 'julia'`.

A Matlab tool `regenerate.m` is also supplied in S2MPJ in order to regenerate all Matlab/Python/Julia problem files (on a Linux system), should one desire to use different S2MPJ options.

### 5.2.3 Limitations

The S2MPJ version of the CUTEst collection (in its present form) has some clear limitations.

1. The following features are *not* supported by the S2PMJ decoder:
   - the call to external FORTRAN subroutines;
   - the D data lines in the data GROUPS section;
   - the occurrence of blank characters within indices (probably against the standard anyway);
   - the FREE FORM version of the SIF file;
   - ... and probably other unforeseen strange SIF and FORTRAN constructs.

Thus, not all CUTEst problems are supplied by default by S2MPJ. In addition, very large problem files have not been decoded by default, to keep the storage required for the collection within reasonable bounds.

In addition, S2MPJ *does not* provide a comprehensive explanation of possible errors in the SIF file (it is supposed to adhere to the standard): it may thus crash on errors, or the produced problem file may also crash. Should this happen, running S2MPJ with `varargin{1}.showsiflines` = 1, usually helps spotting the SIF error (or detecting the S2MPJ bug).

S2MPJ makes the assumption that RANGES $r_i$ (in (5.4)) are only meaningful for inequality constraints. The SIF standard appears to leave open the possibility of ranges for other types of groups, but fails to mention what they can/could be used for. Thus S2MPJ ignores the ranges of equality constraints or objective groups, should they be supplied in the SIF file.

2. The speed of execution of the problem files crucially depends on that of Matlab, Python and Julia. Although this in turn depends on the computing platform and versions used, it is generally slower than Fortran, sometimes by several orders of magnitude. Thus, if speed in running the test problems is sought, it is often preferable to use the Fortran version directly or one of its wrappers `pycutest` [6], `matcutest` [20] or CUTEst.jl [16], despite their own limitations.

#### 5.2.4 Ensuring full compatibility with the Fortran decoder

If, for the purpose of comparison with preexisting numerical experiments, one desires to run a version of the S2MPJ framework which is fully compatible with the Fortran decoder, one needs to (re)decode the SIF problems of interest with S2MPJ using an `options` struct such that

```
options.keepcformat = 1;
options.keepcorder = 1;
options.pbxscale = 1;
```

(see Section 5.2.2).

## 6 Testing

The validity of the S2MPJ decoder has been tested on 1075 problems of the CUTEst collection (see Section 8), avoiding large SIF files and problems involving external Fortran subroutines (see Section 5.2.3). The list of these problems and the corresponding files are part of the framework (see Section 8)

A first step was being able to decode all 1075 SIF files in the three programming languages of interest. Once this was achieved, we verified the coherence between the evaluations produced at the problem's starting point by the S2MPJ decoder (in standard double precision) using the Python problem files and those produced by the Fortran decoder (as obtained from the `pycutest` wrapper). This turned out to be subject to a significant limitation due to a feature of the Fortran decoder inherited from the MPS format: the `OUTSDIF.d` data file written by the Fortran decoder and subsequently used by the CUTEst tools only stores real

numbers with eight significant digits. Unfortunately, the methods used by Fortran and Matlab (when writing the problem files) to promote low precision numbers to double precision ones differ: Fortran first rounds the low precision number before completing it with zeros, while Matlab does not perform any rounding. One therefore has to live with a coherence between the Fortran and Python results of the order of single precision. When explicit numbers (such as least-squares data values) are explicitly given in the SIF file, this can be viewed as a minor limitation. It remains however an advantage of S2MPJ that, because the problem files are executable, computation of constants not depending on SIF-provided reals (such as meshsizes derived from problem dimension) are conducted in full precision. If, as is the case for a significant number of problems, all constants are of this type, then the relative difference between results obtained with S2MPJ and those obtained by the Fortran decoder typically falls below $10^{-14}$.

The final step was to verify the coherence of the evaluation produced by the Matlab and Julia problem files with those produced by the Python files. This was much more accurate, again typically resulting in relative differences below $10^{-14}$.

All validation tests were run on a Dell 16 cores Precision computer with 64 GiB of memory and running Ubuntu and Matlab R2024a, Python 3.8.10 and Julia 1.10.4.

## 7 Writing your own optimization problem in S2PMJ format

It is of course possible (and simple) to create new test examples directly in Matlab, Python or Julia. For making them compatible with the S2MPJ libraries, it is enough that they adhere to the specifications described above (the "S2MPJ format"). In doing so, the GPS structure may be entirely ignored and only the values of the outputs of calls to the problem file need to be supplied. Appendix A gives such simple versions of the ROSENBR problem in Matlab, Python and Julia, intended as possible templates for other cases.

## 8 Distribution

The S2MPJ decoder `s2mpj.m` and its associated libraries `s2mpjlib.m`, `s2mpjlib.py` and `s2mpjlib.jl` are available on the Github repository

`https://github.com/GrattonToint/S2MPJ`

This repository also provides the 1075 SIF files used in our test, as well as the corresponding Matlab, Python and Julia problems files (generated using the S2MPJ defaults).

## 9 Conclusion and perspectives

We have presented S2PMJ a new suite of Matlab, Python and Julia test problems for optimization, derived from the CUTEst collection. The files allowing evaluation of quantities of interest for each problem have been produced by applying a new decoder to the original CUTEst SIF files. They can be used directly (without further decoding) within the native Matlab, Python or Julia environment. The generated problem files, the code for the decoder and its associated libraries are publicly available on Github.

As it turned out, the design of S2MPJ was useful as an independent check of the collection SIF files (a few were either corrected or completed) and also resulted in an improved Fortran

decoder. But the authors are well aware of the limitations of the S2MPJ framework: not all existing SIF files could be decoded, the format of the Matlab/Python/Julia problem files can be very verbose (it has to cover every possible SIF syntax) and they are intrinsically slower to execute than their Fortran counterparts, as expected for JIT compiled languages.

In particular, the 'setup' action and the first evaluation in Julia file can occasionally be very memory-intensive and slow, to the point of making the approach irrelevant for problems involving a large amount of data or a large number of variables. It is therefore clear that *both the Fortran and S2MPJ frameworks have complementary uses*. Equally clear is the fact that improvements to S2MPJ (such as automatic storage of precompiled problem files, automatic code vectorization, reduced-precision options in Python and Julia) are desirable.

## Acknowledgements

The authors wish to thank Nick Gould, Cunxin Huang, Zaikun Zhang and Dominique Orban for their useful comments, corrections and suggestions. They are also indebted to the Polytechnic University of Hong Kong for supporting a visit dedicated to the S2MPJ project. Philippe Toint finally gratefully acknowledges the partial support of the Institut National Polytechnique (INP) of Toulouse (France).

The authors report there are no competing interests to declare.

## References

[1] A. S. Bondarenko, D. M. Bortz, and J. J. Moré. COPS: Large-scale nonlinearly constrained optimization problems. Technical Report ANL/MCS-TM-237, Mathematics and Computer Science, Argonne National Laboratory, Argonne, Illinois, USA, 1999.

[2] I. Bongartz, A. R. Conn, N. I. M. Gould, and Ph. L. Toint. CUTE: Constrained and Unconstrained Testing Environment. *ACM Transactions on Mathematical Software*, 21(1):123–160, 1995.

[3] A. G. Buckley. Test functions for unconstrained minimization. Technical Report CS-3, Computing Science Division, Dalhousie University, Dalhousie, Canada, 1989.

[4] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. LANCELOT*: a Fortran package for large-scale nonlinear optimization (Release A)*. Number 17 in Springer Series in Computational Mathematics. Springer Verlag, Heidelberg, Berlin, New York, 1992.

[5] International Business Machine Corporation. Mathematical programming system/360 version 2, linear and separable programming-user's manual. Technical Report H20-0476-2, IBM Corporation, 1969. MPS Standard.

[6] J. Fowkes, L. Roberts, and Á. Bürmen. PyCUTEst: an open source Python package of optimization test problems. *Journal of Open Source Software*, 7(78):4377, 2022.

[7] N. I. M. Gould, D. Orban, and Ph. L. Toint. CUTEr, a constrained and unconstrained testing environment, revisited. *ACM Transactions on Mathematical Software*, 29(4):373–394, 2003.

[8] N. I. M. Gould, D. Orban, and Ph. L. Toint. CUTEst: a constrained and unconstrained testing environment with safe threads for mathematical optimization. *Computational Optimization and Applications*, 60(3):545–557, 2015.

[9] A. R. Conn, N. I. M. Gould, D. Orban, and Ph. L. Toint. The SIF Reference Report (revised version). `https://github.com/ralna/SIFDecode/blob/master/doc/pdf/sif.pdf`, 2024.

[10] A. Griewank and Ph. L. Toint. On the unconstrained optimization of partially separable functions. In M. J. D. Powell, editor, *Nonlinear Optimization 1981*, pages 301–312, London, 1982. Academic Press.

[11] W. Hock and K. Schittkowski. *Test Examples for Nonlinear Programming Codes*. Springer Verlag, Heidelberg, Berlin, New York, 1981. Lectures Notes in Economics and Mathematical Systems 187.

[12] I. L. Junkins and I. D. Turner. Optimal continuous torque attitude maneuvers. AIAA/AAS Astrodynamics Conference, Palo Alto, 1978.

[13] L. Lukšan and J. Vlček. Sparse and partially separable test problems for unconstrained and equality constrained optimization. Technical Report Technical Report 767, Inst. Computer Science, Academy of Sciences of the Czech Republic, 182 07 Prague, Czech Republic, 1999.

[14] I. Maros and C. Meszaros. A repository of convex quadratic programming problems. *Optimization Methods and Software*, 11-12:671–681, 1999.

[15] J. J. Moré, B. S. Garbow, and K. E. Hillstrom. Testing unconstrained optimization software. *ACM Transactions on Mathematical Software*, 7(1):17–41, 1981.

[16] A. S. Siqueira and D. Orban. CUTEst.jl (v0.13.2). Zenodo.7776400, 2023.

[17] `https://www.tiobe.com/tiobe-index/`, 2025.

[18] Ph. L. Toint. Test problems for partially separable optimization and results for the routine PSPMIN. Technical Report 83/4, Department of Mathematics, FUNDP - University of Namur, Namur, Belgium, 1983.

[19] E. van den Berg, M. P. Friedlander, G. Hennenfent, F. Herrmann, R. Saab and Ö. Yilmaz. Algorithm 890: Sparco: A Testing Framework for Sparse Reconstruction. *ACM Transactions on Mathematical Software*, 35(19):1–16, 2009.

[20] Z. Zhang. MatCUTEst, 2024. `https://github.com/equipez/matcutest/releases/tag/v1.0`.

# A Examples of simple S2MPJ problem specification

ROSSIMP.m: a simple ROSENBR in Matlab

```
function varargout = ROSSIMP(action,varargin)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%    The ever famous 2 variables Rosenbrock "banana valley" problem
%    This version ignores the GPS structure.
%    classification = 'C-CSUR2-AN-2-0'
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    persistent pbm;
    name = 'ROSSIMP';
    switch(action)
    case 'setup'
       pb.name      = name;
       pbm.name     = name;
       pb.n = 2;
       pb.m = 0;
       pb.xlower    = [ -Inf; -Inf ];
       pb.xupper    = [ +Inf; +Inf ];
       pb.x0        = [ -1.2; 1.0 ];
       pb.objlower  = 0.0;
       pb.pbclass   = 'C-SUR2-AN-2-0';
       pb.objderlvl = 2;
       varargout{1} = pb;
       varargout{2} = pbm;
    case { 'fx', 'fgx', 'fgHx' }
       x  = varargin{1};
       varargout{1} = 100.0*(x(2)-x(1)*x(1))^2+(x(1)-1.0)^2;
       if(nargout>1)
          g(1) = -400.0*(x(2)-x(1)*x(1))*x(1)+2.0*(x(1)-1.0);
          g(2) = 200.0*(x(2)-x(1)*x(1));
          varargout{2} = g;
```

```
          if(nargout>2)
             H(1,1) = 1200.0*x(1)*x(1)+2.0;
             H(1,2) = -400.0;
             H(2,1) = H(1,2);
             H(2,2) =  200.0;
             varargout{3} = H;
          end
       end
    case 'fHxv'
       x      = varargin{1};
       H(1,1) = 1200.0*x(1)*x(1)+2.0;
       H(1,2) = -400.0;
       H(2,1) = H(1,2);
       H(2,2) =  200.0;
       varargout{1} = H * varargin{2};
    end
    return

end
```

ROSSIMP.py: a simple ROSENBR in Python

```
from s2mpjlib import *
class  ROSSIMP(CUTEst_problem):
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#
#    The ever famous 2 variables Rosenbrock "banana valley" problem
#    This version ignores the GPS structure.
#    classification = "C-CSUR2-AN-2-0"
#
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    name = 'ROSSIMP'
    def __init__(self,*args):
        import numpy as np
        self.n = 2
        self.m = 0
        self.xlower = np.full((2,1),-float('Inf'))
        self.xupper = np.full((2,1),+float('Inf'))
        self.x0     = np.zeros((2,1))
        self.x0[0]  = float(-1.2)
        self.x0[1]  = float(1.0)
        self.objlower  = 0.0
        self.pbclass   = "C-CSUR2-AN-2-0"
        self.objderlvl = 2

    def fx(self,x):
        import numpy as np
        fx = 100.0*(x[1]-x[1]*x[1])**2+(x[0]-1.0)**2
        return fx

    def fgx(self,x):
        import numpy as np
        x     = args[0]
        gx    = np.zeros(2)
        gx[0] = -400.0*(x[1]-x[0]*x[0])*x[0]+2.0*(x[0]-1.0)
        gx[1] =  200.0*(x[1]-x[0]*x[0])
```

```
        return fx,gx

    def fgHx(self,x):
        import numpy as np
        fx      = 100.0*(x[1]-x[1]*x[1])**2+(x[0]-1.0)**2
        gx      = np.zeros(2)
        gx[0]   = -400.0*(x[1]-x[0]*x[0])*x[0]+2.0*(x[0]-1.0)
        gx[1]   =  200.0*(x[1]-x[0]*x[0])
        Hx      = np.zeros((2,2))
        Hx[0,0] = 1200.0*x[0]*x[0]+2.0
        Hx[0,1] = -400.0
        Hx[1,0] = Hx[0,1]
        Hx[1,1] =  200.0
        return fx,gx,Hx

    def fHxv(self,x,v):
        Hx      = np.zeros((2,2))
        Hx[0,0] = 1200.0*x[0]*x[0]+2.0
        Hx[0,1] = -400.0
        Hx[1,0] = Hx[0,1]
        Hx[1,1] =  200.0
        return Hx*v
```

ROSSIMP.jl: a simple ROSENBR in Julia

```
function ROSSIMP(action::String,args::Union{PBM,Int,Float64,Vector{Int},Vector{Float64}}...)
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#
#    The ever famous 2 variables Rosenbrock "banana valley" problem
#    This version ignores the GPS structure.
#    classification = "C-CSUR2-AN-2-0"
#
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    name = "ROSSIMP"
    if action == "setup"
        pb            = PB(name)
        pb.n = 2
        pb.m = 0
        pb.xlower     = -1*fill(Inf,pb.n)
        pb.xupper     =    fill(Inf,pb.n)
        pb.x0         = [ Float64(-1.2), Float64(1.0) ]
        pb.objlower   = 0.0
        pb.pbclass    = "C-CSUR2-AN-2-0"
        pb.objderlvl  = 2;
        pbm           = PBM(name)
        pbm.objderlvl = 2
        pbm.call      = getfield( Main, Symbol( name ) )
        return pb, pbm
    elseif action == "fx"
        pbm = args[1]
        x   = args[2]
        fx  = 100.0*(x[2]-x[1]*x[1])^2+(x[1]-1.0)^2
        return fx
    elseif action == "fgx"
        pbm = args[1]
```

```
        x   = args[2]
        fx  = 100.0*(x[2]-x[1]*x[1])^2+(x[1]-1.0)^2
        gx    = zeros(Float64,2)
        gx[1] = -400.0*(x[2]-x[1]*x[1])*x[1]+2.0*(x[1]-1.0)
        gx[2] = 200.0*(x[2]-x[1]*x[1])
        return fx,gx
    elseif action == "fgHx"
        pbm = args[1]
        x   = args[2]
        fx  = 100.0*(x[2]-x[1]*x[1])^2+(x[1]-1.0)^2
        gx    = zeros(Float64,2)
        gx[1] = -400.0*(x[2]-x[1]*x[1])*x[1]+2.0*(x[1]-1.0)
        gx[2] = 200.0*(x[2]-x[1]*x[1])
        Hx    = zeros(Float64,2,2)
        Hx[1,1] = 1200.0*x[1]*x[1]+2.0
        Hx[1,2] = -400.0
        Hx[2,1] = Hx[1,2]
        Hx[2,2] =  200.0
        return fx,gx,Hx
    elseif action == "fHxv"
        pbm = args[1]
        x   = args[2]
        v   = args[3]
        Hx      = zeros(Float64,2,2)
        Hx[1,1] = 1200.0*x[1]*x[1]+2.0
        Hx[1,2] = -400.0
        Hx[2,1] = Hx[1,2]
        Hx[2,2] =  200.0
        return Hx*v
    end

end
```

# B Examples of use in Matlab, Python and Julia

We illustrate (in pseudo-code) the use of a putative optimizer `myoptimizer` on the `JUNKTURN` spacecraft orientation problem by Junkins and Turner [12]. This optimal control problem has a parameter `N` defining the number of discretization points used in its formulation. Given `N`, it then has $10(\mathtt{N}+1)$ bounded variables and $7\mathtt{N}$ nonlinear equality constraints. We assume in our simple illustration that $\mathtt{N} = 10$.

Using `JUNKTURN` in Matlab

We start with a Matlab illustration, where we assume that the files `JUNKTURN.m` and `s2mpjlib.m` are in the Matlab path

```
function main()

%   Problem setup

problem_name      = "JUNKTURN";
problem_parameter = 10;

PROBLEM           = str2func( problem_name );  % A function handle for the problem
```

```
[ pb, pbm ]       = PROBLEM( 'setup', problem_parameter );  % Data structure setup

n      = pb.n;         %  The number of variables
m      = pb.m;         %  The number of (equality) constraints
x0     = pb.x0;        %  The starting point
xlower = pb.xlower;    %  The lower bounds on the variables
xupper = pb.xupper;    %  The upper bounds on the variables
cvalue = pb.clower;    %  The rhs of the 'equality) constraints

%   Optimization

[ x, fx, cx ] = myoptimizer( PROBLEM, n, m, x0, xlower, xupper, cvalue, ... );

return

end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [ x, fx, cx ] = myoptimizer( PROBLEM, n, m, x0, xlower, xupper, cvalue, ... )

%   An optimization method using values and first derivatives of the
%   objective function and constraints.

... some code ...

%   Compute the value and gradient of the objective function
%   and constraints at the vector x of size (n,1)
%   (for instance PROBLEM.x0).
%   (for instance PROBLEM.x0).

    [ f_value, f_gradient ]  = PROBLEM( 'fgx', x );
    [ c_values, c_Jacobian ] = PROBLEM( 'cJx', x );

%   Compute the value and gradient of the Lagrangian at x of size (n,1)
%   and y of size (m,1).

    Lag_value    = f_value + y'*c_values;
    Lag_gradient = f_gradient + c_Jacobian'*y;

... some code ...

return

end
```

Note that, if no access to `f_value`, `f_gradient`, `c_values` or `c_Jacobian` is needed and the values of `Lag_value` and `Lag_gradient` suffice, the sections "Compute the value and gradient of the ojective function and constraints" and "Compute the value and gradient of the Lagrangian" in `myoptimizer` can be replaced by the single call

```
[ Lag_value, Lag_gradient ] = PROBLEM( 'Lgxy'', x, y );
```

Using `JUNKTURN` in Python

We now show the Python version, assuming that the files `JUNKTURN.py` and `s2mpjlib.py` are

in the system path.

```
import numpy as np
from JUNKTURN import *

############################

def myoptimizer( PROBLEM, n, m, x0, xlower, xupper, cvalue, ... ):

#   An optimization method using values and first derivatives of the
#   objective function and constraints.

    ... some code ...

#   Compute the value and gradient of the objective function
#   and constraints at the vector x of size (n,1).
#   (for instance PROBLEM.x0).

    f_value, f_gradient  = PROBLEM.fgx( x )
    c_values, c_Jacobian = PROBLEM.cJx( x )

#   Compute the value and gradient of the Lagrangian at x of size (n,1)
#   and y of size (m,1).

    Lag_value    = f_value + y.T.dot(c_values)
    Lag_gradient = f_gradient + c_Jacobian.T.dot(y)

    ... some code ...

    return x, f_value, c_values

############################

#   Setup

PROBLEM = JUNKTURN( 10 )

n       = PROBLEM.n        #  The number of variables
m       = PROBLEM.m        #  The number of (equality) constraints
x0      = PROBLEM.x0       #  The starting point
xlower  = PROBLEM.xlower   #  The lower bounds on the variables
xupper  = PROBLEM.xupper   #  The upper bounds on the variables
cvalue  = PROBLEM.clower   #  The rhs of the (equality) constraints

#   Optimization

x, fx, cx = myoptimizer( PROBLEM, n, m, x0, xlower, xupper, cvalue, ... );
```

Using `JUNKTURN` in Julia

We finally show the Julia version, assuming that the files `JUNKTURN.jl` and `s2mpjlib.jl` are in the directories `problem_path` and `library_path`, respectively. Note the use of `pbm.call` to invoke `JUNKTURN` with the `myoptimizer` function.

```
include(library_path*"/s2mpjlib.jl")

############################

function myoptimizer( pbm, n, m, x0, xlower, xupper, cvalue, ... )

#   An optimization method using values and first derivatives of the
#   objective function and constraints.

 ... some code ...

#   Compute the value and first derivatives of the objective function
#   and constraints at the vector x.
#   (for instance pb.x0).

    f_value, f_gradient  = pbm.call( "fgx", pbm, x )
    c_values, c_Jacobian = pbm.call( "cJx", pbm, x )

#   Compute the value and gradient of the Lagrangian at the vectors
#   x and y, where y has dimension (m,1).

    Lag_value    = f_value + only( y'*c_values )
    Lag_gradient = f_gradient + c_Jacobian'*y

    ... some code ...

    return x, f_value, c_values

end

#   Problem setup

problem_name = "JUNKTURN"
include( problem_path*"/"*problem_name*".jl" )
PROBLEM = getfield( Main, Symbol( problem_name ) )
pb, pbm = PROBLEM( "setup", 10 )

n       = pb.n        #  The number of variables
m       = pb.m        #  The number of (equality) constraints
x0      = pb.x0       #  The starting point
xlower  = pb.xlower   #  The lower bounds on the variables
xupper  = pb.xupper   #  The upper bounds on the variables
cvalue  = pb.clower   #  The rhs of the (equality) constraints

#   Optimization

x, fx, cx = myoptimizer( pbm, n, m, x0, xlower, xupper, cvalue, ... )
```

# C The extended SIF problem classification

A problem is classified by a string of the form

`X-XXXXr-XX-n-m`

The first character in the string identifies the problem collection from which the problem is extracted. Possible values are

`C` the problem belongs to the CUTEst collection;

`S` the problem belongs to the SPARCO collection [19]; and

`N` none of the above.

The character immediately following the first hyphen specifies the type of variables occurring in the problem. Its possible values are

`C` the problem has continuous variables only;

`I` the problem has integer variables only;

`B` the problem has binary variables only; and

`M` the problem has variables of different types.

The second character after the first hyphen defines the type of the problem's objective function. Its possible values are

`N` no objective function is defined;

`C` the objective function is constant;

`L` the objective function is linear;

`Q` the objective function is quadratic;

`S` the objective function is a sum of squares; and

`O` the objective function is none of the above.

The third character after the first hyphen defines the type of constraints of the problem. Its possible values are

`U` the problem is unconstrained;

`X` the problem's only constraints are fixed variables;

`B` the problem's only constraints are bounds on the variables;

`N` the problem's constraints represent the adjacency matrix of a (linear) network;

`L` the problem's constraints are linear;

`Q` the problem's constraints are quadratic; and

`O` the problem's constraints are more general than any of the above alone.

The fourth character after the first hyphen indicates the smoothness of the problem. There are two possible choices

`R` the problem is regular, that is, its first and second derivatives exist and are continuous everywhere; or

`I` the problem is irregular.

The integer (r) which corresponds to the fourth character of the string is the degree of the highest derivatives provided analytically within the problem description. It is restricted to being one of the single characters 0, 1, or 2. The character immediately following the second hyphen indicates the primary origin and/or interest of the problem. Its possible values are

`A` the problem is academic, that is, has been constructed specifically by researchers to benchmark algorithms;

`M` the problem is part of a modeling exercise where the actual value of the solution is not used in a genuine practical application; and

`R` the problem's solution is (or has been) actually used in a real application for purposes other than testing algorithms.

The next character in the string indicates whether or not the problem description contains explicit internal variables. There are two possible values, namely,

`Y` the problem description contains explicit internal variables (see Section 5.1); or

`N` the problem description does not contain any explicit internal variables.

The symbol(s) between the third and fourth hyphen indicate the number of variables in the problem. Possible values are

`V` the number of variables in the problem is a parameter that can be chosen by the user on setup; or

`n` a positive integer giving the actual (fixed) number of problem variables.

The symbol(s) after the fourth hyphen indicate the number of constraints (other than fixed variables and bounds) in the problem. Note that fixed variables are not considered as general constraints here. The two possible values are

`V` the number of constraints in the problem is a parameter that can be chosen by the user on setup; or

`m` a nonnegative integer giving the actual (fixed) number of constraints.

The comments at the beginning problem files not only specify its classification string, but often give additional information about the problem's nature and origin.