Submission Notes for `PIPSMILPSolver`

Enrico Calandrini

June 24, 2026

1 Contribution Overview

This submission contributes a new `PIPSMILPSolver` class for the `MILPSolver` module of SMS++. The purpose of the class is to make the interior-point solver PIPS-IPM++ available through the standard SMS++ solver interface. In this way, a block-structured SMS++ model can be loaded, converted into the distributed callback format expected by PIPS-IPM++, solved by PIPS-IPM++, and finally mapped back into the SMS++ representation.

The main implementation is contained in the following files:

- `MILPSolver/include/PIPSMILPSolver.h`;
- `MILPSolver/src/PIPSMILPSolver.cpp`.

The interface currently supports linear continuous problems. Quadratic constraints are explicitly rejected, because PIPS-IPM++ is not used here as a QCP solver. Dynamic modifications of an already-loaded model are not the focus of this first implementation; the intended workflow is to load a complete problem, solve it, retrieve the solution, and then clear or rebuild the solver state if a different instance has to be solved.

2 Architecture of the Interface

The implementation follows the natural hierarchy of an SMS++ block model. When a problem is loaded, `PIPSMILPSolver` first asks the base `MILPSolver` class to construct the linear representation of the model. After that, it builds the PIPS-IPM++ distributed tree.

The adopted mapping is the following:

- the root SMS++ block is mapped to the PIPS root node;
- each direct sub-block of the root block becomes one PIPS leaf;
- deeper nested sub-blocks are collected into the leaf associated with their first-level ancestor;
- variables are assigned to the node containing the block that owns them;
- row constraints are classified as node-local or linking constraints, depending on whether they involve variables from one node only or from multiple leaves.

The solver then provides PIPS-IPM++ with all structural information through callbacks. These callbacks report dimensions, nonzero counts, objective coefficients, row bounds, variable bounds, active-bound flags, and matrix coefficients. The matrices are extracted from the base `MILPSolver` column-wise representation and converted into the row-compressed representation requested by PIPS-IPM++.

This design keeps the ownership of the SMS++ model unchanged: the `Block`, `Variable`, and `Constraint` objects remain SMS++ objects, while PIPS-IPM++ receives only the numerical representation needed for the solve.

3 PIPS-IPM++ Option Mapping

A second part of the integration exposes PIPS-IPM++ options as SMS++ solver parameters. This makes it possible to set PIPS options from ordinary SMS++ configuration files, instead of editing a separate PIPS option file manually.

The option mapping is generated into:

- `MILPSolver/include/PIPS_defs.h`;
- `MILPSolver/include/PIPS_maps.h`.

These files define the SMS++ parameter identifiers corresponding to PIPS integer, double, and string options. At runtime, `PIPSMILPSolver::set_par` translates SMS++ parameter updates into calls to the PIPS-IPM++ option system. This mechanism was also useful for experimental tuning, for example when changing the linear solver, the scaling strategy, the interior-point method, or the iteration and time limits.

Examples of PIPS options that can be relevant in computational tests include:

- the linear solver selection, such as MUMPS, MA57, or Panua PARDISO;
- scaling options, for instance geometric mean scaling;
- interior-point method options;
- symbolic factorization frequency for PARDISO;
- iteration and time limits.

4 Solve Flow and Status Mapping

The actual solve is performed by `PIPSIPMppInterface::run()`. The returned PIPS-IPM++ termination status is translated into the usual SMS++ solver statuses. Successful termination is mapped to `kOK`, detected infeasibility to `kInfeasible`, detected unboundedness to `kUnbounded`, time-limit termination to `kStopTime`, and iteration-limit termination to `kStopIter`. Other abnormal statuses are mapped to `kError`.

Since PIPS-IPM++ solves minimization problems, the objective coefficients are passed with the appropriate sign if the original SMS++ model is a maximization problem. The same convention is also considered when objective bounds and solution information are returned to SMS++.

5 Primal Solution Retrieval

After a successful solve, `PIPSMILPSolver::get_var_solution()` retrieves the primal vector from PIPS-IPM++ by calling:

```
pips_interface->gatherPrimalSolution()
```

Only rank zero writes the solution back into the SMS++ objects. The PIPS solution vector is ordered according to the PIPS distributed tree. Therefore, the implementation iterates over the stored `varNode` structure, retrieves the original `MILPSolver` column index of each `ColVariable`, and fills a dense vector in the original SMS++ column order. Finally, the solution is written with:

```
MILPSolver::write_var_solution( x );
```

This is the central mapping step for primal variables.

6 Dual Solution Retrieval

The current submission also includes an implementation of dual solution retrieval.

PIPS-IPM++ internally stores several dual vectors. The implementation uses:

- equality row duals;
- inequality row duals;
- variable-bound duals, combined as reduced costs.

These are gathered through the PIPS-IPM++ interface methods:

```
gatherDualSolutionEq()
gatherDualSolutionIneq()
gatherDualSolutionVarBounds()
```

The delicate part is again the mapping. PIPS returns equality rows and inequality rows separately, while `MILPSolver::write_dual_solution()` expects one vector `pi` in the original SMS++ row order. The implementation therefore maps:

- local equality rows from `EqConsNode`;
- linking equality rows from `LinkEqCons`;
- local inequality rows from `InEqConsNode`;
- linking inequality rows from `LinkInEqCons`;
- variable-bound duals from `varNode` into the reduced-cost vector.

The final write-back is performed with:

```
MILPSolver::write_dual_solution( pi, rc );
```

This implementation should be regarded as the correct structural starting point.

7 The `test_pips` Driver

For submission and debugging purposes, a small test driver was added under:

```
MILPSolver/test_pips
```

In order to build the test, the intended command is:

```
make release -j2
```

Please, follow these instructions only after SMS++ has been properly installed and MILPSolver follows the PIPSMILPSolver branch (see Section 11). If any trouble is encountered during the process, reach out to me as I will be very happy to help with anything that comes out.

The test is intentionally simple. It deserializes one `.nc4` instance, applies a `BlockSolverConfig`, runs the first solver registered on the root block, prints a compact summary, and writes the solution to text files.

The main files are:

- `test_pips.cpp`, the standalone test driver;
- `makefile`, the build file following the style of the SMS++ tests;
- one or more `BSCfg` files selecting the solver and PIPS options;
- one small `.nc4` instance used for a quick check;
- `README.md`, with practical local instructions.

The executable accepts four optional arguments:

Argument	Meaning	Default
1	instance file	`EC_CO_Test.nc4`
2	solver configuration	project-specific `BSCfg` file
3	primal solution output file	`primal_solution.txt`
4	dual solution output file	`dual_solution.txt`

A typical run is:

```
cd <smspp-project>/MILPSolver/test_pips
./test_pips EC_CO_Test.nc4 BSCfg1-PIPS-MUMPS.txt \
  primal_solution.txt dual_solution.txt
```

The console output reports the solver class, final status, elapsed time, iteration count, lower bound, upper bound, and objective value. If a primal solution is available, the driver calls `get_var_solution()` and writes all `ColVariable` values. If a dual solution is available through the `CDASolver` interface, the driver calls `get_dual_solution()` and writes all `FRowConstraint` dual values.

The primal output file is tab-separated and has the columns:

```
block_path kind group index address value
```

The dual output file is tab-separated and has the columns:

```
block_path kind group index address dual
```

The `block_path` field identifies the position of the variable or constraint in the nested SMS++ block tree. This makes the output useful for checking whether the solution values are written back to the expected block and group.

8 Configuration Files

The `BSCfg` files used by `test_pips` can specify both the SMS++ solver and the PIPS-IPM++ options. In particular, they can be used to select the PIPS linear solver and numerical options. Typical variants are:

- a default PIPS configuration;
- a MUMPS-based configuration;
- an MA57-based configuration;
- a Panua PARDISO-based configuration, when the license and libraries are available.

The same mechanism can be used to set iteration limits, time limits, scaling, linear-system options, and other PIPS parameters exposed through the generated mapping files.

In the test_pips repository, currently several configuration files are already available. Feel free to change them as you please in order to test how the option might affect the computation.

9 Computational Experiments

The solver was tested on several families of block-structured instances. The experiments showed that performance is strongly related to the quality of the block structure exposed to PIPS-IPM++. Instances with a clear two-stage structure, where the root block represents first-stage variables and leaves represent scenarios, are generally well suited to PIPS. In these cases, the Schur complement structure can be exploited effectively.

Other block structures may be much less favorable, even if the absolute problem size is smaller. In particular, instances with many constraints linking multiple leaves, or with a root block that acts mostly as an empty container, can force PIPS-IPM++ to use a denser Schur complement representation. This can dominate both memory usage and solution time. Therefore, the relevant question is not only how many variables, constraints, and nonzeros the model has, but also how those rows and columns are distributed across the root and leaf blocks.

This observation is important for interpreting the computational results: PIPS is expected to be most competitive when the SMS++ block hierarchy matches the decomposition structure that PIPS-IPM++ can exploit.

10 Current Limitations and Future Work

The current implementation should be considered a first working integration. The natural next steps are:

- support of QP models;
- the dual solution mapping should be validated further on small controlled LPs with known signs;
- possibly, specialised Blocks could be built in order to present PIPS-IPM++ with a more natural decomposition structure.

11 Installation Notes

The recommended starting point is the official SMS++ project installation procedure:

https://gitlab.com/smspp/smspp-project

For convenience, the Linux installation commands are reported below. Replace `<your-custom-path>` with the directory where SMS++ should be installed.

Using `curl`:

```
curl -s https://gitlab.com/smspp/smspp-project/-/raw/develop/INSTALL.sh \
  | sudo bash -s -- --install-root=<your-custom-path>
```

Using `wget`:

```
wget -qO- https://gitlab.com/smspp/smspp-project/-/raw/develop/INSTALL.sh \
  | sudo bash -s -- --install-root=<your-custom-path>
```

During this installation step, CMake may open an interactive configuration window asking whether some dependency or installation paths should be changed. If the default paths are acceptable, press c to configure. Once CMake has completed the generation step, press q to quit the interactive screen; the installation script should then continue normally. The installer supports optional dependency switches such as `-without-cplex`, `-without-gurobi`, `-without-scip`, `-without-highs`, `-without-stopt`, `-without-torch`, `-without-lemon`, and `-without-coinor`. These can be appended if some dependencies are not required or are not available on the target machine, but SMS++ will automatically discaard them if they are not available.

After SMS++ has been installed, move to the `MILPSolver` submodule inside the SMS++ installation and switch it to the `PIPSMILPSolver` branch:

```
cd <your-custom-path>/smspp-project/MILPSolver
git fetch origin
git switch PIPSMILPSolver
```

If the branch is only available as a remote-tracking branch, use:

```
git switch -c PIPSMILPSolver --track origin/PIPSMILPSolver
```

Then rebuild SMS++ from the root of the SMS++ project:

```
cd <your-custom-path>/smspp-project
cmake --build build -j
cmake --install build
```

After everything has been completed, you can safely compile and run the test file by following the instructions in Section 7.

12 Notes for Testers

- The commands above assume a Linux system and the standard SMS++ directory layout created by the official installer.
- If SMS++ was installed without `sudo`, the installation root may be under the user's home directory rather than a system directory.
- If PIPS-IPM++ is installed in a non-standard location, the paths in the relevant makefiles must point to the correct headers and libraries.
- If Panua PARDISO is selected, the corresponding library and license must be available at runtime.
- If MA57 is selected, the HSL library must be available at runtime.
- For a quick portability check, the MUMPS configuration is often the most convenient, provided the MPI/MUMPS link line is correct.