IPX Reference Guide

February 19, 2020

Contents

1 Functionality **1**
1.1 Problem Formulation . . . 1
1.2 Interior Point Method . . . 1
1.3 Crossover . . . 2

2 Usage **2**
2.1 C++ Interface . . . 2
2.2 C Interface . . . 4

3 Parameters **6**

4 Info **9**

1 Functionality

1.1 Problem Formulation

IPX solves linear programming (LP) problems in the form

$$\underset{\mathtt{x}}{\text{minimize}} \quad \mathtt{obj}^T\mathtt{x} \tag{1a}$$

$$\text{subject to} \quad A\mathtt{x}\{\geq,\leq,=\}\mathtt{rhs}, \tag{1b}$$

$$\mathtt{lb} \leq \mathtt{x} \leq \mathtt{ub}. \tag{1c}$$

The matrix A has `num_constr` rows and `num_var` columns. Associated with (1b) are dual variables `y` with the sign convention that

$$\mathtt{y}[i] \geq 0 \quad \text{if constraint is of type } \geq, \tag{2a}$$

$$\mathtt{y}[i] \leq 0 \quad \text{if constraint is of type } \leq, \tag{2b}$$

$$\mathtt{y}[i] \text{ free} \quad \text{if constraint is of type } =. \tag{2c}$$

Associated with $\mathtt{lb} \leq \mathtt{x}$ and $\mathtt{x} \leq \mathtt{ub}$ are dual variable $\mathtt{zl} \geq 0$ and $\mathtt{zu} \geq 0$ respectively. Entries of $-\mathtt{lb}$ and `ub` can be infinity, in which case the dual is fixed at zero.

1.2 Interior Point Method

The interior point method (IPM) computes a primal-dual point $(\mathtt{x}, \mathtt{slack}, \mathtt{xl}, \mathtt{xu}, \mathtt{y}, \mathtt{zl}, \mathtt{zu})$ that approximately satisfies

$$A\mathtt{x} + \mathtt{slack} = \mathtt{rhs}, \quad \mathtt{x} - \mathtt{xl} = \mathtt{lb}, \quad \mathtt{x} + \mathtt{xu} = \mathtt{ub}, \tag{3a}$$

$$A^T\mathtt{y} + \mathtt{zl} - \mathtt{zu} = \mathtt{obj}, \tag{3b}$$

and that is guaranteed to satisfy $\texttt{xl} \geq 0$, $\texttt{xu} \geq 0$, (2) and

$$
\begin{aligned}
\texttt{slack}[i] &\leq 0 \quad \text{if constraint is of type } \geq, && (4a)\\
\texttt{slack}[i] &\geq 0 \quad \text{if constraint is of type } \leq, && (4b)\\
\texttt{slack}[i] &= 0 \quad \text{if constraint is of type } =. && (4c)
\end{aligned}
$$

In theory, the IPM iterates will in the limit satisfy (3a) and (3b), and the primal objective will equal the dual objective

$$\texttt{rhs}^T\texttt{y} + \texttt{lb}^T\texttt{zl} - \texttt{ub}^T\texttt{zu}. \tag{5}$$

(Entries for which `-lb` or `ub` is infinity are understood to be dropped from the sum.)

1.3 Crossover

The crossover method recovers an optimal basis from the interior solution. A basis is defined by variable and constraint statuses

$$\texttt{vbasis}[j] \in \{\texttt{IPX\_basic}, \texttt{IPX\_nonbasic\_lb}, \texttt{IPX\_nonbasic\_ub}, \texttt{IPX\_superbasic}\}, \tag{6}$$

$$\texttt{cbasis}[i] \in \{\texttt{IPX\_basic}, \texttt{IPX\_nonbasic}\}. \tag{7}$$

The columns of A for which $\texttt{vbasis}[j] = \texttt{IPX\_basic}$ and the columns of the identity matrix for which $\texttt{cbasis}[i] = \texttt{IPX\_basic}$ form a square, nonsingular matrix of dimension `num_constr`. The corresponding basic solution $(\texttt{x}, \texttt{slack}, \texttt{y}, \texttt{z})$ is obtained by setting

$$
\begin{aligned}
\texttt{z}[j] &= 0 && \text{if } \texttt{vbasis}[j] = \texttt{IPX\_basic}, && (8a)\\
\texttt{x}[j] &= \texttt{lb}[j] && \text{if } \texttt{vbasis}[j] = \texttt{IPX\_nonbasic\_lb}, && (8b)\\
\texttt{x}[j] &= \texttt{ub}[j] && \text{if } \texttt{vbasis}[j] = \texttt{IPX\_nonbasic\_ub}, && (8c)\\
\texttt{x}[j] &= 0 && \text{if } \texttt{vbasis}[j] = \texttt{IPX\_superbasic}, && (8d)\\
\texttt{y}[i] &= 0 && \text{if } \texttt{cbasis}[i] = \texttt{IPX\_basic}, && (8e)\\
\texttt{slack}[i] &= 0 && \text{if } \texttt{cbasis}[i] = \texttt{IPX\_nonbasic} && (8f)
\end{aligned}
$$

and computing the remaining components such that $A\texttt{x} + \texttt{slack} = \texttt{rhs}$ and $A^T\texttt{y} + \texttt{z} = \texttt{obj}$. The basis is primal feasible if $\texttt{lb} \leq \texttt{x} \leq \texttt{ub}$ and (4) hold; the basis is dual feasible if (2) holds and

$$
\begin{aligned}
\texttt{z}[j] &\geq 0 \quad \text{if } \texttt{vbasis}[j] = \texttt{IPX\_nonbasic\_lb}, && (9a)\\
\texttt{z}[j] &\leq 0 \quad \text{if } \texttt{vbasis}[j] = \texttt{IPX\_nonbasic\_ub}, && (9b)\\
\texttt{z}[j] &= 0 \quad \text{if } \texttt{vbasis}[j] = \texttt{IPX\_superbasic}. && (9c)
\end{aligned}
$$

The IPX crossover consists of a primal and dual push phase. Depending on the accuracy of the interior solution and the numerical stability of the LP problem, the obtained basis may not be primal and/or dual feasible. In this case reoptimization with an external simplex code is required.

2 Usage

2.1 C++ Interface

User code written in C++ must include the header file `src/lp_solver.h`. Both the `src/` and `include/` directories must be in the compiler's search path for header files.

The following code snippet illustrates the use of the C++ interface. The complete example program can be found in `example/afiro.cc` and can be compiled by calling `make` in the `example/` directory.

```
#include <cmath>
#include <iostream>
#include "lp_solver.h"

using Int = ipxint;

constexpr Int num_var = 12;
constexpr Int num_constr = 9;
const double obj[] = { -0.2194, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.32,
                       -0.5564, 0.6, -0.48 };
const double lb[num_var] = { 0.0 };
const double ub[] = { 80.0, 283.303, 283.303, 312.813, 349.187, INFINITY,
                      INFINITY, INFINITY, 57.201, 500.0, 500.501, 357.501};
// Constraint matrix in CSC format with 0-based indexing.
const Int Ap[] = { 0, 2, 6, 10, 14, 18, 20, 22, 24, 26, 28, 30, 32 };
const Int Ai[] = { 0, 5, ... };
const double Ax[] = { -1.0, 0.301, ... };
const double rhs[] = { 0.0, 80.0, 0.0, 0.0, 0.0, 0.0, 0.0, 44.0, 300.0 };
const char constr_type[] = { '<', '<', '=', '<', '<', '=', '<', '<', '<' };

int main() {
    ipx::LpSolver lps;

    // Load the LP model into IPX.
    Int errflag = lps.LoadModel(num_var, obj, lb, ub, num_constr, Ap, Ai, Ax,
                                rhs, constr_type);
    if (errflag) {
        std::cout << " invalid model (errflag = " << errflag << ")\n";
        return 1;
    }

    // Solve the LP.
    Int status = lps.Solve();
    if (status != IPX_STATUS_solved) {
        // no solution
        // (time/iter limit, numerical failure, out of memory)
        std::cout << " status: " << status << ','
                  << " errflag: " << lps.GetInfo().errflag << '\n';
        return 2;
    }

    // Get solver and solution information.
    ipx::Info info = lps.GetInfo();

    // Get the interior solution (available if IPM was started).
    double x[num_var], xl[num_var], xu[num_var], slack[num_constr];
    double y[num_constr], zl[num_var], zu[num_var];
    lps.GetInteriorSolution(x, xl, xu, slack, y, zl, zu);

    // Get the basic solution (available if crossover terminated without error).
    double xbasic[num_var], sbasic[num_constr];
    double ybasic[num_constr], zbasic[num_var];
    Int cbasis[num_constr], vbasis[num_var];
    lps.GetBasicSolution(xbasic, sbasic, ybasic, zbasic, cbasis, vbasis);

    return 0;
}
```

All classes and functions belonging to IPX are declared in namespace `ipx`. There are three classes that are intended for direct use from external code: `ipx::LpSolver`, `ipx::Parameters` and `ipx::Info`. The `Parameters` and `Info` classes have no methods (except for a constructor that initializes them to default values). They are used to pass control parameters to IPX and to return information about the execution of the solver. They are documented in Sections 3 and 4.

The `LpSolver` class provides the user interface to the LP solver. Its methods are documented through source code comments in `src/lp_solver.h`. The method

```
Int LoadModel(Int num_var, const double* obj, const double* lb,
              const double* ub, Int num_constr, const Int* Ap, const Int* Ai,
              const double* Ax, const double* rhs, const char* constr_type);
```

requires the LP problem in the form (1) with the constraint matrix A given in compressed sparse column (CSC) format. That means, `Ap` is an array of size `num_var` $+ 1$ and `Ai` and `Ax` are arrays of size equal to the number of entries in A. They must be set such that `Ai[Ap[j]..Ap[j+1]-1]` and `Ax[Ap[j]..Ap[j+1]-1]` hold the row indices and nonzero values of the entries in column `j` of A. The entries within each column can be in any order, but there must be no duplicates. The input is checked and one of the following error codes is returned if it is invalid:

`IPX_ERROR_argument_null`	A pointer argument was `NULL`.
`IPX_ERROR_invalid_dimension`	`num_constr` < 0 or `num_var` ≤ 0.
`IPX_ERROR_invalid_matrix`	Constraint matrix was invalid (`Ap`$[0] \neq 0$, `Ap`$[j] >$ `Ap`$[j+1]$ for some j, non-finite entry in `Ax`, out-of-range or duplicate index in `Ai`).
`IPX_ERROR_invalid_vector`	One of the vectors `obj`, `lb`, `ub`, `rhs` and `constr_type` contained an invalid entry, or `lb`$[j] >$ `ub`$[j]$ for some j.

2.2 C Interface

C wrapper functions for the methods of class `LpSolver` are provided to allow using the full functionality of IPX from C code. User code written in C must include the header file `include/ipx_c.h`.

The following code snippet illustrates the use of the C interface. The complete example program can be found in `example/afiro_c.c` and can be compiled by calling `make` in the `example/` directory.

```
#include <math.h>
#include <stdio.h>
#include "ipx_c.h"

typedef ipxint Int;

#define NUM_VAR 12
#define NUM_CONSTR 9
const double obj[] = { -0.2194, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.32,
                       -0.5564, 0.6, -0.48 };
const double lb[NUM_VAR] = { 0.0 };
const double ub[] = { 80.0, 283.303, 283.303, 312.813, 349.187, INFINITY,
                      INFINITY, INFINITY, 57.201, 500.0, 500.501, 357.501};
// Constraint matrix in CSC format with 0-based indexing.
const Int Ap[] = { 0, 2, 6, 10, 14, 18, 20, 22, 24, 26, 28, 30, 32 };
const Int Ai[] = { 0, 5, ... };
const double Ax[] = { -1.0, 0.301, ... };
```

```
const double rhs[] = { 0.0, 80.0, 0.0, 0.0, 0.0, 0.0, 0.0, 44.0, 300.0 };
const char constr_type[] = { '<', '<', '=', '<', '<', '=', '<', '<', '<' };

int main() {
    void *lps = NULL;

    // Create new solver instance. This allocates a tiny amount of memory.
    ipx_new(&lps);
    if (!lps) return 1;

    // Load the LP model into IPX.
    Int errflag = ipx_load_model(lps, NUM_VAR, obj, lb, ub, NUM_CONSTR, Ap, Ai,
                                 Ax, rhs, constr_type);
    if (errflag) {
        printf(" invalid model (errflag = %ld)\n", (long) errflag);
        return 2;
    }

    // Solve the LP.
    Int status = ipx_solve(lps);
    if (status != IPX_STATUS_solved) {
        // no solution
        // (time/iter limit, numerical failure, out of memory)
        struct ipx_info info = ipx_get_info(lps);
        printf(" status: %ld, errflag: %ld\n", (long) status,
               (long) info.errflag);
        return 3;
    }

    // Get solver and solution information.
    struct ipx_info info = ipx_get_info(lps);

    // Get the interior solution (available if IPM was started).
    double x[NUM_VAR], xl[NUM_VAR], xu[NUM_VAR], slack[NUM_CONSTR];
    double y[NUM_CONSTR], zl[NUM_VAR], zu[NUM_VAR];
    ipx_get_interior_solution(lps, x, xl, xu, slack, y, zl, zu);

    // Get the basic solution (available if crossover terminated without error).
    double xbasic[NUM_VAR], sbasic[NUM_CONSTR];
    double ybasic[NUM_CONSTR], zbasic[NUM_VAR];
    Int cbasis[NUM_CONSTR], vbasis[NUM_VAR];
    ipx_get_basic_solution(lps, xbasic, sbasic, ybasic, zbasic, cbasis, vbasis);

    // Must call ipx_free() to deallocate memory in solver instance.
    ipx_free(&lps);
    return 0;
}
```

The solver is accessed through a `void` pointer, which must be allocated by `ipx_new` and finally deallocated by `ipx_free` (see `include/ipx_c.h` for documentation of these functions). The remaining functions of the C interface are simply wrappers around the methods of class `LpSolver`, which require the pointer to the solver as their first argument.

Parameters and solver information are passed in and out through data types `struct ipx_parameters` and `struct ipx_info`, which are the same as `ipx::Parameters` and `ipx::Info` in the C++ interface and are documented in Sections 3 and 4. In contrast to the C++ interface, the struct members are uninitialized when a data type is defined. A `struct ipx_parameters` object with default parameter values is returned by `ipx_default_parameters()`.

3 Parameters

display
Type: `ipxint`
Default: `1`
If nonzero, then solver log is printed to standard output.

logfile
Type: `const char*`
Default: `NULL`
Name of file to which solver log is appended. The file is created if it does not exist. If `logfile` is `NULL` or the empty string (`""`), file logging is turned off.

print_interval
Type: `double`
Default: `5.0`
Frequency (in seconds) for printing progress reports during construction of the starting basis and crossover. If negative, progress reports are turned off. If zero, a progress line is printed after each basis update.

time_limit
Type: `double`
Default: `-1.0`
Time limit (in seconds) for the solver. If negative, no time limit is imposed.

dualize
Type: `ipxint`
Default: `-1`
Controls dualization of the LP model by the preprocessor.

0	model is not dualized
≥ 1	model is dualized
< 0	an automatic choice is made

scale
Type: `ipxint`
Default: `1`
Controls the automatic scaling of the LP model by the preprocessor.

≤ 0	no scaling is applied
1	recursive equilibration of rows and columns of the constraint matrix
> 1	currently the same as 1, but reserved for future options

ipm_maxiter
Type: `ipxint`
Default: `300`
Maximum number of interior point iterations.

ipm_feasibility_tol
Type: `double`
Default: `1e-6`
The interior point solver terminates when a feasibility and an optimality criterion are satisfied. The feasibility criterion requires that the relative primal and dual residuals are not larger than `ipm_feasibility_tol`.

ipm_optimality_tol
Type: `double`
Default: `1e-8`
The interior point solver terminates when a feasibility and an optimality criterion are satisfied. The optimality criterion requires that the relative gap between the primal and dual objective values is not larger than `ipm_optimality_tol`.

ipm_drop_primal
Type: `double`
Default: `1e-9`
Controls handling of primal degeneracies in the interior point solve. If a degenerate variable `xl`[j] or `xu`[j] is not larger than `ipm_drop_primal`, then its dual variable is fixed at its current value and eliminated from the optimization; `x`[j] will then be at its bound in the solution. If `ipm_drop_primal` is zero or negative, no primal variables are dropped. If the model was dualized by the preprocessor, then this option affects dual degeneracies in the input model.

ipm_drop_dual
Type: `double`
Default: `1e-9`
Controls handling of dual degeneracies in the interior point solve. If the dual variables `zl`[j] and `zu`[j] are degenerate and not larger than `ipm_drop_dual`, then `x`[j] is fixed at its current value and eliminated from the optimization. The dual variables will then be zero in the solution. If `ipm_drop_dual` is zero or negative, no dual variables are dropped. If the model was dualized by the preprocessor, then this option affects primal degeneracies in the input model.

kkt_tol
Type: `double`
Default: `0.3`
Controls the accuracy to which the KKT linear equation systems are solved by an iterative method. A smaller value reduces the number of interior point iterations but increases the computational cost per iteration. Typical values are within the interval $[0.05, 0.5]$.

precond_dense_cols
Type: `ipxint`
Default: `1`
In combination with diagonal preconditioning, controls handling of "dense" columns (i. e. columns with a relatively large number of entries). If nonzero, dense columns are treated separately by a low rank update.

crash_basis
Type: `ipxint`
Default: `1`
Controls the construction of the starting basis for the preconditioner.

≤ 0	slack basis
1	crash method that prefers variables with a larger interior point scaling factor
> 1	currently the same 1, but reserved for future options

The chosen procedure (slack basis, crash) is followed by a sequence of basis updates that makes free variables basic and fixed (slack) variables nonbasic.

dependency_tol
Type: `double`
Default: `1e-6`
Controls the detection of linearly dependent rows and columns while constructing the starting basis. If possible, columns corresponding to free variables are pivoted into the basis, and slack columns corresponding to equality constraints are pivoted out of the basis. Hereby a nonbasic variable cannot replace a basic variable if the pivot element is less than or equal to `dependency_tol`. A negative value is treated as 0.0.

volume_tol
Type: `double`
Default: `2.0`
Controls the update of the basis matrix from one interior point iteration to the next. An entry of the scaled tableau matrix is used as pivot element if it is larger than `volume_tol` in absolute value. Increasing the parameter usually leads to fewer basis updates but more iterations of the linear solver. Typical values are in the interval $[1.1, 10.0]$. A value smaller than 1.0 is treated as 1.0.

rows_per_slice
Type: `ipxint`
Default: `10000`
Controls the update of the basis matrix from one interior point iteration to the next. The search for pivot elements partitions the tableau matrix into slices, each slice containing approximately `rows_per_slice` rows. A smaller value leads a finer pivot search and possibly a better preconditioner, but makes the update procedure more expensive.

maxskip_updates
Type: `ipxint`
Default: `10`
Controls the update of the basis matrix from one interior point iteration to the next. For each slice of the tableau matrix, the update search is terminated after computing `maxskip_updates` columns of the tableau matrix that do not contain eligible pivot elements. Decreasing the parameter makes the update procedure faster, but may affect the quality of the basis.

lu_kernel
Type: `ipxint`
Default: `0`
Chooses the method/package for computing and updating the *LU* factorization of basis matrices.

≤ 0	BASICLU for factorization and update
1	BASICLU for factorization, Forrest-Tomlin update without hypersparsity
> 1	currently the same as 1, but reserved for future options

lu_pivottol
Type: `double`
Default: `0.0625`
Partial pivoting tolerance for the LU factorization. The tolerance is tightened automatically if a factorization is detected unstable.

crossover
Type: `ipxint`
Default: `1`
If nonzero, crossover is used for recovering an optimal basis.

crossover_start
Type: `double`
Default: `1e-8`
Tightens the IPM termination criterion for crossover. At the beginning of crossover, the final IPM iterate is dropped to complementarity (i. e. for each pair of variables either the primal is set to its bound or the dual is set to zero). In addition to the standard IPM termination criterion, it is required that the relative primal or dual residual caused by dropping any variable is not larger than `crossover_start`. A nonpositive value means that the standard criterion is used. This parameter has no effect if crossover is turned off.

pfeasibility_tol
Type: `double`
Default: `1e-7`
A basic solution is considered primal feasible if the primal variables do not violate their bounds by more than `pfeasibility_tol`.

dfeasibility_tol
Type: `double`
Default: `1e-7`
A basic solution is considered dual feasible if the dual variables do not violate their sign condition by more than `dfeasibility_tol`.

4 Info

Class `ipx::Info` (for C++ users) and struct `ipx_info` (for C users) provide information about the input problem, the solver operations and the computed solution. Most of their members are almost self-explanatory and can be found in `include/ipx_info.h`.

There are four members holding status and error codes that are documented below. Possible values are macro-defined constants that have the prefix `IPX_STATUS_` and `IPX_ERROR_`. They are defined in `include/ipx_status.h`.

status
Termination status of the solver. Possible values are:

`IPX_STATUS_not_run`	Solver has not been called.
`IPX_STATUS_no_model`	Solver was called but no model was loaded.
`IPX_STATUS_solved`	Solver terminated successfully. The solution status (optimal, imprecise, infeasible) is given by **status_ipm** and **status_crossover**.
`IPX_STATUS_stopped`	Solver stopped without having reached its termination criterion (time/iteration limit, numerical failure). **status_ipm** or **status_crossover** hold further information.
`IPX_STATUS_out_of_memory`	Memory allocation failed.
`IPX_STATUS_internal_error`	An internal error occured. This is a bug in IPX.

status_ipm
Status of the interior solution. Possible values are:

`IPX_STATUS_not_run`	IPM has not been started.
`IPX_STATUS_optimal`	IPM terminated successfully and the computed solution satifies the feasibility and optimality tolerances.
`IPX_STATUS_imprecise`	IPM reached its termination criterion, but the solution after postprocessing does not satisfy the feasibility or optimality tolerance.
`IPX_STATUS_primal_infeas`	The LP problem is primal infeasible because equality constraints are inconsistent.
`IPX_STATUS_dual_infeas`	The LP problem is dual infeasible because columns of the constraint matrix corresponding to free variables are linearly dependent and their objective coefficients cause an unbounded ray in primal space.
`IPX_STATUS_time_limit`	IPM stopped because of time limit.
`IPX_STATUS_iter_limit`	IPM stopped because of iteration limit.
`IPX_STATUS_no_progress`	IPM stopped because no progress was achieved over a number of iterations.
`IPX_STATUS_failed`	Linear system solve or building the preconditioner failed. **errflag** holds further information.

status_crossover
Status of the basic solution obtained by crossover. Possible values are:

`IPX_STATUS_not_run`	Crossover has not been started.
`IPX_STATUS_optimal`	Crossover terminated successfully and the computed basic solution satifies the feasibility and optimality tolerances.
`IPX_STATUS_imprecise`	Crossover terminated successfully, but the basic solution does not satisfy the feasibility or optimality tolerance. Reoptimization with a simplex solver is needed.
`IPX_STATUS_time_limit`	Crossover stopped because of time limit.
`IPX_STATUS_failed`	Crossover failed due to linear algebra issues. **errflag** holds further information.

errflag
Possible values when **status_ipm** or **status_crossover** = `IPX_STATUS_failed`:

`IPX_ERROR_cr_iter_limit`
`IPX_ERROR_cr_matrix_not_posdef`
`IPX_ERROR_cr_precond_not_posdef`
`IPX_ERROR_cr_no_progress`
`IPX_ERROR_cr_inf_or_nan`
	The iterative method for solving the linear systems in the IPM did not converge.

`IPX_ERROR_basis_singular`
`IPX_ERROR_basis_almost_singular`
`IPX_ERROR_basis_update_singular`
`IPX_ERROR_basis_repair_overflow`
`IPX_ERROR_basis_repair_search`
`IPX_ERROR_basis_too_ill_conditioned`
	Factorizing or updating the basis matrix failed.

`IPX_ERROR_lapack_chol`	Cholesky factorization of dense Schur complement matrix broke down.

In all cases the failure is caused by numerical difficulties. The specific error codes are meant for developers.