Math. Prog. Comp. (2016) 8:377–391
DOI 10.1007/s12532-015-0094-2

FULL LENGTH PAPER

Customizing the solution process of COIN-OR's linear solvers with Python

Mehdi Towhidi[1,2] · Dominique Orban[1,2]

Received: 9 September 2014 / Accepted: 10 September 2015 / Published online: 5 October 2015
© Springer-Verlag Berlin Heidelberg and The Mathematical Programming Society 2015

Abstract Implementations of the simplex method differ mostly in specific aspects such as the pivot rule. Similarly, most relaxation methods for mixed-integer programming differ mostly in the type of cuts and the exploration of the search tree. We provide a scripting mechanism to easily implement and experiment with primal and dual pivot rules for the simplex method, by building upon COIN-OR's open-source linear programming package CLP, without explicitly interacting with the underlying C++ layers of CLP. In the same manner, users can customize the solution process of mixed-integer linear programs using the CBC and CGL COIN-OR packages by coding branch-and-cut strategies and cut generators in Python. The Cython programming language ensures communication between Python and C++ libraries and activates user-defined customizations as callbacks. Our goal is to emphasize the ease of development in Python while maintaining acceptable performance. The resulting software, named CyLP, has become a part of COIN-OR and is available under open-source terms. For illustration, we provide an implementation of the positive edge rule—a recently proposed rule that is particularly efficient on degenerate problems—and demonstrate how to customize branch-and-cut node selection in the solution of a mixed-integer program.

Research partially supported by NSERC Discovery Grant 299010-04.

✉ Mehdi Towhidi
mehdi.towhidi@gerad.ca

Dominique Orban
dominique.orban@gerad.ca

1 Department of Mathematics and Industrial Engineering, École Polytechnique, Montreal, QC, Canada

2 GERAD, Montreal, QC, Canada

Keywords Linear programming · Mixed-integer programming · Python · Cython · COIN-OR · CLP · CBC · CGL · Simplex pivot

Mathematics Subject Classification 90C05 · 90C10 · 90C11

1 Introduction

The simplex algorithm, considered by many to be among the top ten algorithms of the twentieth century in terms of its scientific and practical impact [7,11,13], is a graceful way of solving linear programs (LP). Although it is efficient in many situations, it has always struggled in the face of degeneracy, whose effects range from adversely impacting performance to cycling. In order to ensure convergence theoretically one needs to modify an essential component of the algorithm—the *pivot selection* [3]. Identifying a pivot selection rule that is efficient across many degenerate LP instances is still an open subject after more than 60 years of research. The Improved Primal simplex [31] and the positive edge pivot rule [30] are recent efforts in that direction.

Commercial implementations of simplex typically do not allow users to plug in customized pivot rules. A promising avenue is to modify an open-source simplex implementation, such as GLPK [24], COIN-OR's CLP [8], or SoPlex [34]. However, such implementations are typically written in a low-level programming language such as C or C++. Delving into large-scale open projects in those languages can be a daunting task even for a seasoned programmer. It does appear however that open-source implementations of simplex are the ideal platform to implement and experiment with pivot rules. Aside from LP, effective customization of solution strategies are also crucial in mixed-integer linear programming (MIP).

In this paper we propose a user-friendly alternative that emphasizes flexibility and ease of use, and promotes fast development and productivity. CyLP is a tool for researchers to implement pivot rules in the dynamic high-level Python programming language [29]. CyLP builds upon CLP and provides flexible and easy to use mechanisms to substitute CLP's built-in primal and dual pivot rules with a user-defined pivot rule written in Python. As for MIPs, CyLP provides facilities to customize the solution process using Python, by allowing users to inject cuts of their own design. In particular, we interface COIN-OR's CBC [5] which provides tools to solve MIPs using branch-and-cut [19,27]. In addition, CyLP can be used as a modeling environment to formulate and solve LPs and MIPs via CLP and CBC. Our main motivation for this research is the design of pivot rules suited to degenerate problems. In follow-up research, we apply such rules to quadratic programs (QP) and mixed-integer QPs.

The goal is to shorten the development phase while minimizing the performance hit due to the usage of an interpreted language. In order to limit such performance hit as much as possible, our choice is to write the communication layer between Python and CLP in the Cython [10] programming language. Cython is a strongly-typed superset of Python whose main design goal is strictly speed and that eases the interfacing of binary code as well as of source code. This feature makes it particularly suitable for facilitating communication between Python and large libraries that users may not wish, or be able, to recompile. CyLP is composed of three layers: a few C++ interface

classes, a thin Cython layer ensuring fast communication between the C++ code and the Python programming language, and convenience Python classes. As our numerical experiments illustrate, not only are we able to maintain competitive execution speeds, but the gains in flexibility and ease of development far outweigh the performance hit. CyLP is available as an open-source package from http://mpy.github.io/CyLP.

The rest of this paper is organized as follows. Section 2 gives a brief description of the simplex method, common pivot rules typically found in solvers and the need to define and examine new pivot rules. In Sect. 3 we present the positive edge method, specifically designed for degenerate problems. Section 4 describes some implementation details of CyLP, provides an implementation of Dantzig's classic pivot rule as an example and shows the essentials of our implementation of the positive edge rule in Python. Section 5 describes how CyLP is used to solve LPs and MIPs. We explain CyLP's MIP customization in Sect. 6. We conclude and look ahead in Sect. 7.

1.1 Related research

Two of the major commercial LP solvers, CPLEX [9] and Gurobi [17], offer a Python API and allow users to interact with the solution process of MIPs using callbacks to customize cut-generation and the branch-and-cut procedure. They do not appear to let users define pivot rules.

PULP [28] is a Python modeler for LP and provides interfaces to existing open-source and commercial LP solvers such as GLPK, CLP and CPLEX. PyCPX [20] is a Cython interface to CPLEX that leverages the power of Numpy [26]—a library defining the standard array type in Python—and provides more convenient modeling facilities than the default CPLEX Python API. Pylpsolve [21] is a similar interface to lpsolve [2] and Pycoin [33] is a Python interface to CLP. None of them appears to allow users to customize the solution process.

There is growing interest in Cython as an interface language for projects written in low-level languages. For example, CyIPOPT [1] is a wrapper for the interior-point optimizer IPOPT [37].

To the best of our knowledge, CyLP is the first toolset that connects with an efficient implementation of simplex and permits experimentation with pivot rules in a high-level language. Like PyCPX, CyLP allows users to exploit the power of Numpy.

1.2 Notation

Throughout this paper we use capital Latin letters for matrices and lowercase Latin letters for vectors. Calligraphic letters are used to denote index sets. For any matrix M, we denote the j-th column of M by M_j and the i-th row of M by M^i. For any vector c, any matrix A and any index set $\mathcal{B}$, $c_{\mathcal{B}}$ is the subvector of c indexed by $\mathcal{B}$ and $A_{\mathcal{B}}$ is the submatrix of A composed of the columns indexed by $\mathcal{B}$. Similarly $A^{\mathcal{B}}$ is the submatrix of A which contains the rows indexed by $\mathcal{B}$. The only norm used in this paper, denoted $\|\cdot\|$, is the Euclidean norm.

2 Implementing simplex pivot rules

This section assumes familiarity with the simplex method [11]. We focus on the primal simplex method because our motivation is to provide a Python implementation of the positive edge rule, which is a primal pivot rule. We stress however that CyLP supports both primal and dual pivot rules.

The linear programming problem in standard form is

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \; c^T x \quad \text{subject to} \quad Ax = b, \quad x \geq 0,$$

where $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times n}$, and the inequality $x \geq 0$ is understood elementwise. Simplex is an iterative method that divides variables into two categories: basic and nonbasic. Let $\mathcal{B}$ be the index set of basic variables, also called the *basis*, and $\mathcal{N}$ be that of the nonbasic variables. The method begins with an initial basis and iteratively swaps an element of $\mathcal{B}$ and one of $\mathcal{N}$, a process called *pivoting*, until optimality is reached [11]. Among the pivot rules that are most often considered in implementations of simplex are Dantzig's pivot rule [11] and the steepest-edge rule and its variations [14,15,18].

Pivoting has the side effect of modifying the right-hand side b at each iteration. Given a basis $\mathcal{B}$, simplex works with the right-hand side $\bar{b} := A_{\mathcal{B}}^{-1} b$. We say that $\mathcal{B}$ is degenerate if it contains at least one k for which $\bar{b}_k = 0$. A pivot on such a k is called a degenerate pivot and an LP for which such pivots occur is said to be degenerate [16]. Such pivots may cause no improvement in the objective function. In real-world large-scale LPs, which are often sparse and degenerate, it is possible to spend the majority of the solution time performing degenerate pivots.

Degeneracy occurs frequently in real-world LPs, including but not limited to large-scale set partitioning problems. Raymond et al. [30] report manpower planning problems with a degeneracy level of 80 %, i.e. at each iteration of simplex, $\bar{b}_i = 0$ for typically 80 % of $i \in \mathcal{B}$ [30]. Likewise, the patient distribution system (pds) instances of [4] have a 80 % degeneracy level on average.

In a recent effort to solve large-scale problems with high occurrence of degenerate pivots efficiently, Raymond et al. [30] introduce the positive edge rule. One of our initial motivations for developing CyLP was to implement the positive edge rule for benchmarking purposes and for application to quadratic and other classes of optimization problems.

CLP implements both Dantzig's pivot selection and two variants of the steepest edge method with optional partial pricing. In CLP, the execution of the simplex method on a given problem is abstracted as a C++ class possessing an attribute that represents the pivot selection rule to be used at each iteration. Users specify the pivot rule of their choice by setting this attribute appropriately, the value of the attribute being another C++ class that abstracts the pivot rule itself. New pivot rules may be implemented by subclassing the latter class and overriding certain of its methods. The definition of such a pivot rule in Python can be significantly shorter in terms of number of lines of code and easier in terms of development effort. For example, Dantzig's pivot rule implementation in CLP takes 58 lines of code while a straightforward Python

implementation takes only 18 lines. A C++ implementation of the positive edge pivot rule takes 106 lines while we can obtain the same functionality in Python in 38 more readable lines. Conciseness can be crucial in more complex pivot rules. We estimate that the steepest edge method, whose implementation takes about 3800 lines of code in CLP, could be written in less than 500 lines in Python. This makes a Python implementation remarkably easier to develop and debug. Furthermore, since low-level programming details such as memory management are no longer a concern, the programmer can focus almost exclusively on the logic of the pivot rule.

3 The positive edge rule

Positive edge [30] is a rule designed to handle degenerate LPs efficiently by allowing us to identify degenerate pivots. Let $Q := A_{\mathcal{B}}^{-1}$. Define $\mathcal{Z} = \{i = 1, \dots, m \mid \bar{b}_i = 0\}$ and $\mathcal{P} = \{i = 1, \dots, m \mid \bar{b}_i > 0\}$. Accordingly, partition Q and A row-wise as

$$Q = \begin{bmatrix} Q^{\mathcal{P}} \\ Q^{\mathcal{Z}} \end{bmatrix} \quad \text{and} \quad A = \begin{bmatrix} A^{\mathcal{P}} \\ A^{\mathcal{Z}} \end{bmatrix}.$$

Denoting $\bar{A} := A_{\mathcal{B}}^{-1} A$, we have

$$\begin{bmatrix} \bar{A}^{\mathcal{P}} \\ \bar{A}^{\mathcal{Z}} \end{bmatrix} := \begin{bmatrix} Q^{\mathcal{P}} A \\ Q^{\mathcal{Z}} A \end{bmatrix} = \begin{bmatrix} Q^{\mathcal{P}} A_{\mathcal{B}} & Q^{\mathcal{P}} A_{\mathcal{N}} \\ Q^{\mathcal{Z}} A_{\mathcal{B}} & Q^{\mathcal{Z}} A_{\mathcal{N}} \end{bmatrix} = \begin{bmatrix} I & 0 & Q^{\mathcal{P}} A_{\mathcal{N}} \\ 0 & I & Q^{\mathcal{Z}} A_{\mathcal{N}} \end{bmatrix}, \tag{1}$$

where the last equality uses the identity $Q A_{\mathcal{B}} = I$.

A variable x_j, $j \in \mathcal{B} \cup \mathcal{N}$ is said to be *compatible* if and only if

$$Q^{\mathcal{Z}} A_j = \bar{A}_j^{\mathcal{Z}} = 0.$$

Using the usual identification of $\mathcal{B}$ with $\{1, \dots, m\}$, we observe from this definition and (1) that for $j \in \mathcal{B}$, x_j is compatible if $j \in \mathcal{P}$ and is incompatible if $j \in \mathcal{Z}$. But we are particularly interested in the nonbasic compatible variables because selecting one of them as the entering variable ensures a nondegenerate pivot. From the definition of compatibility we deduce that for a given compatible entering variable x_j $(j \in \mathcal{N})$ and a leaving variable x_i $(i \in \mathcal{B})$, the improvement in the objective value is strictly positive, and a non-degenerate pivot is performed. But calculating $\bar{A}_j^{\mathcal{Z}}$ for all variables requires the matrix–matrix product $Q^{\mathcal{Z}} A$. For positive edge to be efficient we must lower the complexity of this identification.

For an arbitrary vector $v \in \mathbb{R}_+^{|\mathcal{Z}|}$, define $w = (Q^{\mathcal{Z}})^T v$. If the variable x_j is compatible, we have

$$w^T A_j = v^T Q^{\mathcal{Z}} A_j = 0. \tag{2}$$

Conversely, if $w^T A_j = 0$, can we affirm that x_j is compatible, i.e., $Q^{\mathcal{Z}} A_j = 0$? There are two possibilities: either $Q^{\mathcal{Z}} A_j = 0$, in which case x_j is compatible, or

$Q^{\mathcal{Z}} A_j \perp v$. For random v, the probability of the latter happening in $\mathbb{R}^{|\mathcal{Z}|}$ is zero. However, in IEEE double precision arithmetic, this probability can be shown to be 2^{-62} [30]. Therefore, the probability that the statement

$$w^T A_j = 0 \Longrightarrow x_j \text{ is compatible}$$

be erroneous is 2^{-62}, and this would result in a single degenerate pivot. Since this method does not involve the calculation of updated columns $\bar{A}_j$ its complexity reduces to $O(mn)$—the complexity of the dense matrix-vector product $w^T A$.

The details of the positive edge method are given in Algorithm 3.1. It involves a parameter $0 < \psi < 1$ that specifies the preferability of compatible variables. A value $\psi = 0.4$ means that we prefer a compatible variable over an incompatible one even if the reduced cost of the former is 0.4 that of the latter.

Algorithm 3.1 The Positive Edge Rule

Step 0. If w is not initialized or updating w is required, set $\mathcal{P} := \{i \in \mathcal{B} \mid \bar{b}_i > \epsilon\}$ where $\epsilon > 0$ is a prescribed tolerance. Let $v \in \mathbb{R}^m$ be a random vector and fix $v_i = 0$ for all $i \in \mathcal{P}$. Compute $w := A_{\mathcal{B}}^{-T} v$.

Step 1. Let $r_{\min} = r_{\text{comp}} = 0$. For each $j \in \mathcal{N}$ such that $r_j < r_{\text{comp}}$, do:

1. if $|w^T A_j| < \epsilon$ (x_j is likely compatible) then set $r_{\text{comp}} = r_j$
2. set $r_{\min} = \min(r_{\min}, r_j)$.

Step 2. If $r_{\text{comp}} < \psi\, r_{\min}$, choose the variable corresponding to r_{comp} to enter the basis. Otherwise choose the variable corresponding to $r_{\min}$ and demand an update of w at the next iteration.

In Step 1 of Algorithm 3.1, r_j denotes the j-th component of the vector of reduced costs, r_{comp} is the best reduced cost over compatible variables, and $r_{\min}$ is the best overall reduced cost found so far. For more information on the design of the positive edge criterion, we refer the reader to [30].

4 Software design and numerical tests

Commercial implementations of simplex typically allow users to choose a pivot rule among a set of predefined rules, but not to plug in customized or experimental pivot rules. It therefore appears that open-source solvers are the best option if one wishes to experiment with new pivot rules. One of the leading open-source LP solvers, CLP, is part of the COIN-OR project [23] and has several advantages that make it our solver of choice for the development of CyLP. Firstly, CLP has an object-oriented structure that makes it convenient to extend or modify. Secondly, CLP is written in C++, a language for which compilers are freely available on most platforms. Finally, the large COIN-OR user base gives confidence that CLP has been exercised and debugged to satisfaction.

In CLP it is possible to define customized pivot rules but a solid understanding of its internal structure and of C++ are required. To simplify the exposition we show a partial UML class diagram [22] of CLP in Fig. 1. A class `ClpSimplex` implements

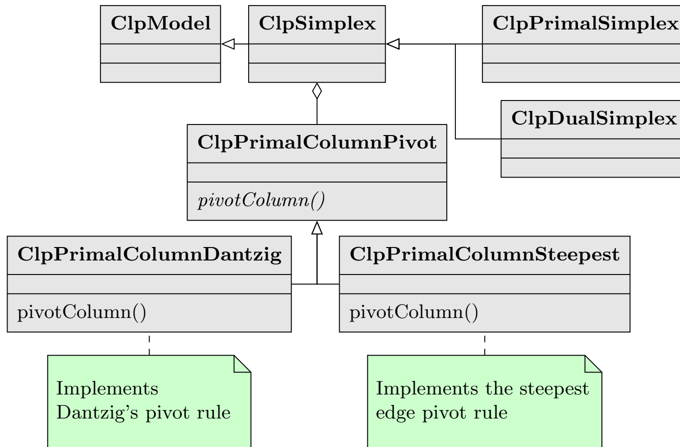

Fig. 1 Partial UML class diagram for CLP

the simplex method. It has an attribute of type `ClpPrimalColumnPivot`—the base class common to all pivot rules. Every pivot rule must derive from the latter and implement a method called `pivotColumn()` that returns an integer—the index of the entering variable. Figure 1 also shows two pivot rules already implemented in CLP.

One of the goals of CyLP is to offer users practical and efficient means to implement `pivotColumn()` directly in Python or Cython. Pivot rules need full access to different aspects of a problem, which demands that CyLP wrap a number of components of CLP. In addition, implementing pivot rules often requires defining and maintaining new data structures, vectors and matrices. The standard Python modules Numpy and Scipy [32] facilitate such tasks and make them reasonably efficient. CyLP must therefore be able to interact with those standard packages.

CyLP consists of three layers. The first layer is the auxiliary C++ layer whose role is to enable or facilitate the communication between C++ and Cython. This layer is often required because of technicalities.

The second and most important layer is the Cython layer, whose role is to ensure seamless communication between Python and CLP. A collection of Cython files interfaces CLP either directly or indirectly via the auxiliary layer. In the Cython layer, special attention is paid to handling matrices and vectors efficiently while passing them back and forth between C++ and Python.

The third and final layer is the Python layer. This is where we define the callback functions that implement custom pivot rules. These functions will be called from the Cython and/or C++ layers. In a typical use case, we define a pivot rule in Python and pass it over for CLP to use while iterating. The CyLP layers are illustrated in Fig. 2. More implementation details are provided in [36].

To test the software, we first examine the performance hit caused by implementing a pivot rule in Python or Cython as opposed to C++. We choose Dantzig's pivot selection rule because it is simple enough to ensure a fair comparison across different

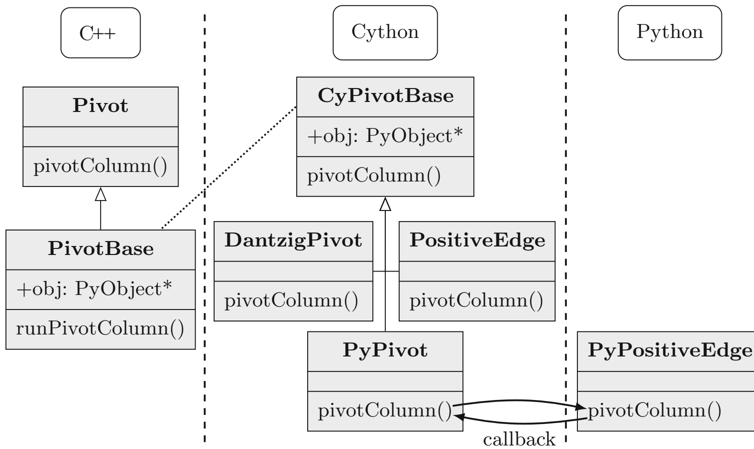

Fig. 2 Schema of the three layers of CyLP

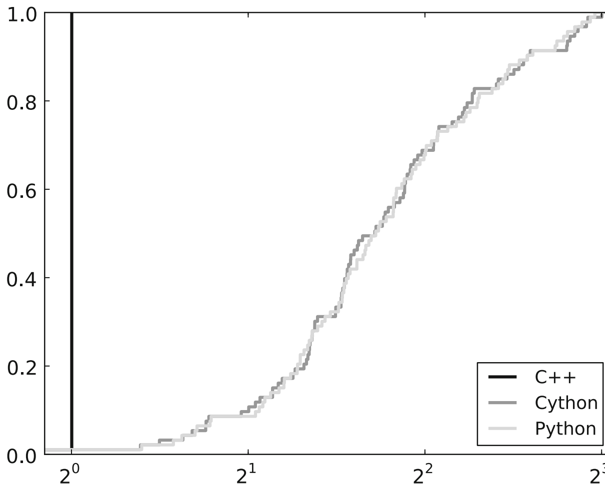

Fig. 3 Performance profile for the execution time of the primal simplex algorithm using the C++, Cython and Python implementations of Dantzig's pivot rule

implementations. In a second stage, we demonstrate how CyLP can be used to examine the effectiveness of the positive edge pivot rule.

We choose the Netlib LP test set [25] for the first part, which contains 93 LPs of diverse dimensions and sparsity. All our experiments are conducted on computers with Intel Xeon 2.4 GHz CPUs and 49 GB of total shared memory. Let t^d_{cpp}, t^d_{cy} and t^d_{py} denote the execution times of the primal simplex method using C++, Cython and Python versions of Dantzig's pivot rule, respectively.

Figure 3 show the performance profile [12] of the execution times. The figure illustrates that, as expected, the C++ implementation is always faster but that the Cython and Python versions are essentially equivalent to one another. It also demonstrates that

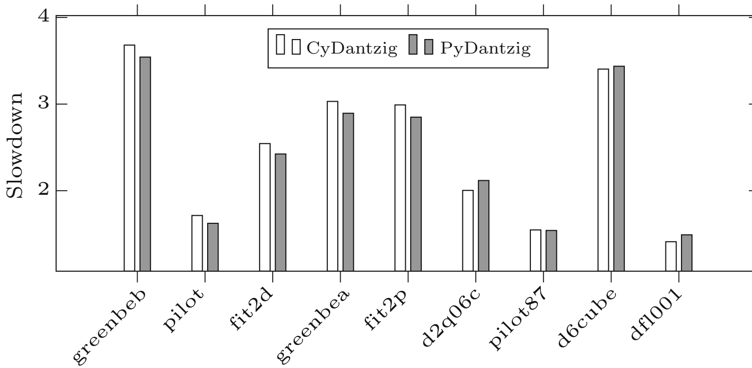

Fig. 4 Performance hit caused by Python and Cython compared to C++

for 50 % of the instances, the Cython and Python versions are less than 3 times slower than the C++ version.

We next select those Netlib instances that take more than 5 s to solve using the C++ implementation of Dantzig's rule and measure the performance hit caused by using Cython and Python by computing the *slowdown* factors t^d_{cy}/t^d_{cpp} and t^d_{py}/t^d_{cpp}. The results are given in the form of a bar chart in Fig. 4. Problems are sorted by C++ execution time from `greenbeb`, taking 5 s, to `dfl001`, taking 9729 s. The average slowdown is 2.3 and in the most difficult instance, `dfl001`, is about 1.4. Our observation is that as problems become moderately large, the performance gap shrinks to a point where it no longer has a significant impact.

For the second part of the numerical tests we choose from among the `pds` instances from Mittlemann's benchmark [4], which are large, sparse and highly degenerate—good target problems for the positive edge method.

Let t^p_{cpp} and t^p_{py} be the execution times of the positive edge method using C++ and Python implementations, respectively. We compute t^d_{cpp}/t^p_{cpp} and t^d_{py}/t^p_{py} which indicate the speedups gained by using the positive edge rule relative to Dantzig's rule in C++ and Python. Results of these tests appear in Table 1. In this table, n and m denote the number of variables and constraints of the instance, respectively, i_d and i_p are the numbers of iterations necessary to solve the instance using Dantzig's pivot rule and the positive edge rule, respectively. Python reports higher speedups than C++ but the iteration reductions are identical. The reason is that positive edge is adding an almost equal overhead to each iteration in C++ and Python. For example in `pds-06`, the average C++ iteration time is 0.0002 s for Dantzig and 0.0004 s for positive edge. As for Python, the average iteration time is 0.0019 s for Dantzig and 0.0017 s for positive edge. This means that positive edge is adding an extra 0.0002 s on average to each Dantzig iteration which is relatively more costly for C++. This also shows that a careful implementation of the positive edge rule in Python runs at almost the same speed as in C++, each iteration on average taking 3×10^{-6} s longer.

Table 1 Speedup of the positive edge method relative to Dantzig's rule

Instance	n	m	i_d	i_p	C++ speedup	Python speedup
pds-02	7535	2953	897	997	0.3	0.8
pds-06	28,655	9881	12,842	3734	1.0	3.5
pds-10	48,763	16,558	35,070	7039	2.5	5.3
pds-20	105,728	33,874	321,465	31,382	3.9	10.1
pds-30	154,998	49,944	727,418	71,453	4.2	9.5
pds-40	212,859	66,844	1,444,478	164,289	5.2	7.8

Table 1 compares the performance of the positive edge pivot rule in Python and in C++. As expected, in both languages we observe the superiority of the positive edge method over Dantzig's pivot rule for larger `pds` instances, both in terms of run time and in terms of number of pivots. We emphasize that what is important here is being able to make similar performance analysis in both languages, rather than validating the efficiency of the positive edge method per se.

5 Using CyLP

We solve problems in CyLP either by reading the problem from an MPS or LP file or by using CyLP's modeling facilities. Suppose that we have a LP defined in an MPS file `problem.mps`. To solve this problem in Python using the positive edge pivot method we use the excerpt given in Listing 1.1.

```
1 s = CyClpSimplex()
2 s.readMps("problem.mps")
3 s.setPivotMethod(PositiveEdgePivot(s))
4 s.primal()              # Executes primal simplex.
```

Listing 1.1 Using a Custom Pivot Rule

In Listing 1.1, we first create an instance `s` of `CyClpSimplex`—a class that interfaces CLP's `ClpSimplex`. After reading the problem from `lp.mps`, we create an instance of `PositiveEdge` and register it with `s`. Then we solve the model using CLP's primal simplex method.

As an alternative to reading from a file, CyLP provides intuitive modeling facilities to express linear programming problems. Consider the following LP

$$\begin{aligned} \text{minimize} \quad & c^T x + [2,2]^T y \\ \text{subject to} \quad & Ax \le a \\ & [2,2]^T \le Bx + Dy \le b, \end{aligned} \tag{3}$$

where $c \in \mathbb{R}^3$, $x \in \mathbb{R}^3$, $y \in \mathbb{R}^2$, $A \in \mathbb{R}^{2\times3}$, $a \in \mathbb{R}^2$, $B \in \mathbb{R}^{2\times3}$, $D \in \mathbb{R}^{2\times2}$, and $b \in \mathbb{R}^2$.

Listing 1.2 demonstrates how to define the objective function and the constraints of (3) in CyLP.

```
1 s = CyClpSimplex()
2 # Adding Constraints
3 s += A * x <= a
4 s += 2 <= B * x + D * y <= b
5 # Adding the objective function
6 s += c * x + 2 * y.sum()
```

Listing 1.2 Modeling LP (3) using CyLP

6 Mixed integer programming with CyLP

Much in the same way as CyLP enables to customize the pivot selection rule in CLP, it also enables to customize the solution process of MIPs, be it by defining an instance-specific branch-and-cut strategy or by defining a special cut generator. This is made possible by interfacing COIN-OR's CBC library. Specifically, custom cuts may be input by the user by way of COIN-OR's CGL Cut-Generation Library [6]. CGL supplies generators for a collection of well-known cuts, such as Gomory, clique and knapsack cover cuts. CyLP facilitates access to these cut generators. The interface offers a certain level of flexibility which, when combined with user-designed pivot rules, may produce powerful variants of the solution process.

In this section, we illustrate how to add integrality restriction to variables, as well as how to solve a MIP and tune the solution process. Finally, we explain how CyLP can be used to write Python callbacks to customize the branch-and-cut process.

To mark variables as integers we use the `setInteger()` method. In the LP of Listing 1.2, for example, x_1 and x_2 can be marked as integers by adding `s.setInteger(x[1:3])`. Once the MIP is modeled, we solve it using an instance of the `CyCbcModel` class.

Considering the case of reading the problem from file, Listing 1.3 illustrates how to solve a MIP using two cut generators.

To access the LP's variables in line 8, we need the problem's `CyLPModel`—an object containing the data structures describing the LP. To obtain it, we read the LP using `extractCyLPModel()` instead of `readMps()`. We mark a subset of these variables as integers in line 9. Calling `getCbcModel()` solves the initial relaxation using CLP and returns an instance of the `CyCbcModel` class, which is an interface to CBC's `CbcModel` class. The latter implements a branch-and-cut procedure. We then add two cut generators—one generating Gomory cuts of at most 100 variables and the other generating knapsack cover cuts. Afterwards, we solve the problem and print out the solution.

```
s = CyClpSimplex()

# Read problem from file. To have access to the CyLPModel of
# the problem we use extractCyLPModel method instead of readMps.
model = s.extractCyLPModel('lp.mps')

# Mark variables x5 to x9 as integers
x = model.getVarByName('x')
s.setInteger(x[5:10])

# Solve initial relaxation and obtain a CyCbcModel object.
cbcModel = s.getCbcModel()

# Create a Gomory cut generator and a Knapsack cover cut
generator.
gomory   = CyCglGomory(limit=100)
knapsack = CyCglKnapsackCover(maxInKnapsack=50)

# Add cut generators to CyCbcModel.
cbcModel.addCutGenerator(gomory,   name="Gomory")
cbcModel.addCutGenerator(knapsack, name="Knapsack")

cbcModel.branchAndBound()  # Solve.
print cbcModel.primalVariableSolution
```

Listing 1.3 Mixed Integer Programming with CyLP

6.1 Customizing CBC's branch-and-cut

CBC provides the capability to customize its branch-and-cut node selection process by writing C++ callbacks. CyLP enables us to use Python instead. Suppose that we wish to implement a simplistic approach of traversing the branch-and-cut tree. The strategy is to look for nodes with the least number of unsatisfied integrality constraints. In case of a tie, we first select the deepest node (i.e., a *depth-first* strategy). But whenever an integer solution is found we break the tie by choosing the highest node (*i.e., a breadth-first* strategy). Listing 1.4 illustrates how to implement this strategy by defining a class that inherits from the relevant base class `NodeCompareBase`.

Our subclass, named `SimpleNodeCompare`, must implement `compare()` to determine the preference in node selection, `newSolution()`, which will be run by CBC after a solution is found to perform a possible change of strategy, and `every1000Nodes()`, which is similar to `newSolution()` but is called after CBC has visited 1000 nodes. To use `SimpleNodeCompare` in Listing 1.3, we set the node comparison method by registering an instance `snc` of `SimpleNodeCompare` with the `CbcModel` object using `cbcModel.setNodeCompare(snc)`.

6.2 Introducing custom cuts to CBC

Another highly used way of customizing a MIP solver is to define effective cuts that are specialized for a type of problem. While CBC can be equipped with many ready-to-use well-known cuts through CGL, it provides a framework to define custom cuts in C++. CyLP presents the same feature in Python. At the core of a CyLP cut is a function

```
1  class SimpleNodeCompare(NodeCompareBase):
2      def __init__(self):
3          self.method = 'depth'    # Default strategy.
4
5      def compare(self, x, y):
6          "Return True if node y is better than node x."
7          if x.nViolated != y.nViolated:
8              return (x.nViolated > y.nViolated)
9          if x.depth == y.depth:
10             return x.breakTie(y)    # Break ties consistently.
11         if self.method == 'depth':
12             return (x.depth < y.depth)
13         return (x.depth > y.depth)
14
15     def newSolution(self, model, objContinuous,
16  nInfeasContinuous):
17         "Cbc calls this after a solution is found in a node."
18         self.method = 'breadth'
19
20     def every1000Nodes(self, model, nNodes):
21         "Cbc calls this every 1000 nodes for possible change of
22         strategy."
23         return False         # Do not demand a tree re-sort.
```

Listing 1.4 A simple node comparison implementation in Python

that returns a list of constraints. Suppose that we have two cuts to add to a problem, $a_1^T x \leq b_1$ and $a_2^T x \leq b_2$. Listing 1.5 demonstrates the main part of the implementation.

```
1 class MyCutGenerator:
2     def generateCuts(self, solverInterface, cglTreeInfo):
3
4         # Compute a_1, a_2, b_1, b_2 using solverInterface
5
6         cuts = []
7         cuts.append(a_1 * x <= b_1)
8         cuts.append(a_2 * x <= b_2)
9         return cuts
```

Listing 1.5 A Python custom cut in CyLP

To add this cut to the example of Listing 1.3, we add the following code before calling `branchAndBound()`.

```
    mycut = MyCutGenerator(m)
    cbcModel.addPythonCutGenerator(mycut, name='mycut')
```

7 Discussion and future work

CyLP, available from http://mpy.github.io/CyLP, provides a high-level scripting framework to define and customize aspects of the solution process of LPs and MIPs using COIN-OR's CLP and CBC. It uses callback methods to let users define new primal and dual pivot rules in Python and have CLP use them during the simplex iterations. We demonstrate this feature by implementing the positive edge pivot rule in C++ and Python. We believe that the ease of programming and flexibility offered

by implementing pivot rules in Python outweigh the slowdown caused by using a high-level interpreted programming language.

Besides the pivot selection rules discussed throughout this paper, we also implemented the Last In First Out and the Most Often Selected rules described by [35], each in less than 30 lines of code. However, those rules also demand to restrict the leaving variable selection, which is not currently possible in CyLP. The reason is that in CLP, entering variable selection is designed to be customized by users and is defined in separate classes whereas a leaving variables rule is built into CLP's `ClpSimplexPrimal` class. Nevertheless, future improvements to CyLP will remove this limitation.

In follow-up research, we consider the application of the positive edge rule to quadratic and nonlinear programming. CyLP and its modeling facilities will let us construct and solve subproblems easily. We hope other users find CyLP equally valuable in their research.

Cython is a powerful intermediate language to enable interaction between low-level high-performance libraries and Python. We expect that other types of optimization solvers would benefit from similar scripting capabilities. In nonconvex optimization, the flexibility and power of solvers such as IPOPT [37] would, in our opinion, be greatly enhanced were users able to plug in their own linear system solver or barrier parameter update using Python.

References

1. Aides, A.: Cython wrapper for IPOPT. http://code.google.com/p/cyipopt. Accessed 2 Nov 2011 (Online)
2. Berkelaar, M.: lpsolve, A Mixed Integer Linear Programming Software. http://lpsolve.sourceforge.net. Accessed 2 Nov 2011 (Online)
3. Bland, R.G.: New finite pivoting rules for the Simplex method. Math. Oper. Res. **2**(2), 103–107 (1977). (ISSN 0364765X)
4. Carolan, W., Hill, J., Kennington, J., Niemi, S., Wichmann, S.: Empirical evaluation of the KORBX algorithms for military airlift applications. Oper. Res. **38**, 240–248 (1990)
5. CBC: COIN-OR Branch-and-Cut. http://projects.coin-or.org/Cbc. Accessed 2 Nov 2011 (Online)
6. CGL: COIN-OR Cut Generation Library. http://projects.coin-or.org/Cgl. Accessed 2 Nov 2011 (Online)
7. Cipra, B.A.: The best of the 20th century: editors name top 10 algorithms. SIAM News **33**(4), 1–2 (2000)
8. CLP: COIN-OR Linear Programming. https://projects.COIN-OR.org/Clp. Accessed 2 Nov 2011 (Online)
9. CPLEX: http://www.cplex.com. Accessed 2 Nov 2011 (Online)
10. Cython: http://www.cython.org. Accessed 1 Sept 2013 (Online)
11. Dantzig, G.B.: Linear Programming and Extensions. Princeton University Press, NJ (1963)
12. Dolan, E., Moré, J.: Benchmarking optimization software with performance profiles. Math. Program. B **91**, 201–213 (2002)
13. Dongarra, J., Sullivan, F.: Guest editors' introduction: the top 10 algorithms. Comput. Sci. Eng. **2**, 22–23 (2000). doi:10.1109/MCISE.2000.8146052. (ISSN 1521-9615)
14. Forrest, J.J., Goldfarb, D.: Steepest-edge simplex algorithms for linear programming. Math. Program. **57**, 341–374 (1992). doi:10.1007/BF01581089. (ISSN 0025-5610)
15. Goldfarb, D., Reid, J.K.: A practicable steepest-edge simplex algorithm. Math. Program. **12**, 361–371 (1977). doi:10.1007/BF01593804. (ISSN 0025-5610)
16. Greenberg, H.J.: An analysis of degeneracy. Naval Res. Log. Q. **33**, 635–655 (1986)
17. Gurobi: http://www.gurobi.com. Accessed 2 Nov 2011 (Online)

18. Harris, P.M.J.: Pivot selection methods of the Devex LP code. In: Cottle, R.W., et al., (eds.) Computational Practice in Mathematical Programming, Mathematical Programming Studies, vol. 4, pp. 30–57. Springer, Berlin (1975). doi:10.1007/BFb0120710 (ISBN 978-3-642-00766-8)
19. Hoffman, K.L., Padberg, M.: Solving airline crew scheduling problems by branch-and-cut. Manag. Sci. **39**, 675–682 (1993). doi:10.1287/mnsc.39.6.657
20. Koepke, H.: Cython wrapper for CPLEX. http://www.stat.washington.edu/~hoytak/code/pycpx/index.html. Accessed 2 Nov 2011 (Online)
21. Koepke, H.: Cython wrapper for lpsolve. http://www.stat.washington.edu/~hoytak/code/pylpsolve/index.html. Accessed 2 Nov 2011 (Online)
22. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edn. Prentice Hall, Upper Saddle River (2001)
23. Lougee-Heimer, R.: The common optimization interface for operations research. IBM J. Res. Dev. **47**(1), 57–66 (2003). doi:10.1147/rd.471.0057. http://www.COIN-OR.org
24. Makhorin, A.: GLPK, GNU Linear Programming Kit. http://www.gnu.org/s/glpk. Accessed 2 Nov 2011 (Online)
25. Netlib: http://www.netlib.org/lp/data. Accessed 2 Nov 2011 (Online)
26. Numpy: http://www.numpy.org. Accessed 1 Sept 2013 (Online)
27. Padberg, M., Rinaldi, G.: A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. SIAM Rev. **33**, 60–100 (1991). doi:10.1137/1033004. (ISSN 0036-1445)
28. PuLP: An LP modeler written in Python. http://code.google.com/p/pulp-or. Accessed 2 Nov 2011
29. Python. http://www.python.org. Accessed 1 Sept 2013 (Online)
30. Raymond, V.,Soumis, F., Metrane, A., Desrosiers, J.: Positive edge: a pricing criterion for the identification of non-degenerate simplex pivots. In: Cahier du GERAD G-2010-61. GERAD, Montreal (2010)
31. Raymond, V., Soumis, F., Orban, D.: A new version of the improved primal simplex for degenerate linear programs. Comput. OR **37**(1), 91–98 (2010). doi:10.1016/j.cor.2009.03.020
32. Scipy: http://www.scipy.org. Accessed 1 Sept 2013 (Online)
33. Silva, P.J.S.: Pycoin, interface to some COIN packages (2005). http://www.ime.usp.br/~pjssilva/software.html. Accessed 2 Nov 2011 (Online)
34. SoPlex: An open-source LP solver. http://soplex.zib.de. Accessed 1 Sept 2013 (Online)
35. Terlaky, T., Zhang, S.: Pivot rules for linear programming: a survey on recent theoretical developments. Ann. Oper. Res. **46–47**, 203–233 (1993). doi:10.1007/BF02096264. (ISSN 0254-5330)
36. Towhidi, M., Orban, D.: Customizing the solution process of coin-or's linear solvers with python. In: Cahier du GERAD G-2012-07. GERAD, Montreal (2012)
37. Wächter, A., Biegler, L.T.: On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. Math. Programm. **106**, 25–57 (2006). doi:10.1007/s10107-004-0559-y. https://projects.COIN-OR.org/Ipopt