GERAD

GROUPE D'ÉTUDES ET DE RECHERCHE
EN ANALYSE DES DÉCISIONS

Les Cahiers du GERAD

G-2012-07
March 2012

CITATION ORIGINALE / ORIGINAL CITATION

Towhidi, M., Orban, D., Customizing the solution process of COIN-OR's linear solvers with Python, Mathematical Programming Computation, 8(4), 377-391, 2016.
Doi: 10.1007/s12532-015-0094-2

GERAD HEC Montréal
3000, ch. de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

Les Cahiers du GERAD ISSN: 0711–2440

Customizing the Solution Process of COIN-OR's Linear Solvers with Python

M. Towhidi
D. Orban

G–2012–07

March 2012

Les textes publiés dans la série des rapports de recherche HEC n'engagent que la responsabilité de leurs auteurs. La publication de ces rapports de recherche bénéficie d'une subvention du Fonds québécois de la recherche sur la nature et les technologies.

Customizing the Solution Process of COIN-OR's Linear Solvers with Python

Mehdi Towhidi
Dominique Orban

GERAD & Department of Mathematics and Industrial Engineering
École Polytechnique de Montréal
Montréal (Québec) Canada, H3C 3A7

mehdi.towhidi@gerad.ca
dominique.orban@gerad.ca

March 2012

Les Cahiers du GERAD
G–2012–07

Copyright © 2012 GERAD

Abstract

Implementations of the Simplex method differ only in very specific aspects such as the pivot rule. Similarly, most relaxation methods for mixed-integer programming differ only in the type of cuts and the exploration of the search tree. Implementing instances of those frameworks would therefore be more efficient if linear and mixed-integer programming solvers let users customize such aspects easily. We provide a scripting mechanism to easily implement and experiment with pivot rules for the Simplex method by building upon COIN-OR's open-source linear programming package CLP. Our mechanism enables users to implement pivot rules in the Python scripting language without explicitly interacting with the underlying C++ layers of CLP. In the same manner, it allows users to customize the solution process while solving mixed-integer linear programs using the CBC and CGL COIN-OR packages. The Cython programming language ensures communication between Python and COIN-OR libraries and activates user-defined customizations as callbacks. For illustration, we provide an implementation of a well-known pivot rule as well as the positive edge rule—a new 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.

Key Words: Linear Programming, Mixed-Integer Linear Programming, Python, Cython, COIN-OR, CLP, Simplex Pivot.

Acknowledgments: Research partially supported by NSERC Discovery Grant 299010-04.

1 Introduction

The Simplex algorithm created by Dantzig is considered by many to be among the top ten algorithms of the twentieth century in terms of its scientific and practical impact (Cipra, 2000; Dongarra and Sullivan, 2000). Simplex 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 causing cycling to significantly impacting the performance. In order to ensure convergence theoretically one needs to modify an essential component of the algorithm—the *pivot selection* (e.g. Bland (1977)). 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 (IPS) of Raymond et al. (2010b) and the positive edge pivot rule of Raymond et al. (2010a) are recent efforts in that direction.

Commercial implementations of Simplex typically do not allow users to plug in customized pivot rules. Raymond et al. (2010a) work around this limitation by solving auxiliary dynamically-generated LPs at each step—a substantial overhead. A promising alternative is to modify an open-source Simplex implementation, such as Makhorin's GLPK or COIN-OR's CLP, typically written in a low-level programming language such as C or C++ . Delving into large-scale open projects in such 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.

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[1]. CyLP builds upon CLP and provides flexible and easy to use mechanisms to substitute CLP's built-in pivot rules with a user-defined pivot rule written in Python. Aside from LP, effective pivot rules are also crucial in mixed-integer linear programming (MIP). CyLP provides facilities to customize the solution of MIPs using Python, by allowing users to inject cuts of their own design. In particular, we interface COIN-OR's CBC[2] which provides tools to solve MIPs using branch-and-cut (Hoffman and Padberg, 1993; Padberg and Rinaldi, 1991). 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 pivot rule is part of the nerve center of any implementation of Simplex since it must be executed at each iteration. One worry is thus that implementing it in an interpreted language seriously affects performance. In order to limit the performance hit as much as possible, our choice is to write the communication layer between Python and CLP in the Cython[3] 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 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 `github.com/mpy/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 Section 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. It also covers how CyLP is used to solve MIPs, how it can be used to examine different solution strategies scripted in Python, and summarizes its modeling capabilities. Section 5 documents numerical experience. We conclude and look ahead in Section 6.

[1] `www.python.org`
[2] `projects.coin-or.org/Cbc`
[3] `www.cython.org`

Related Research

Two of the major commercial LP solvers, CPLEX[4] and Gurobi[5], 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 is a Python modeler for LP and provides interfaces to existing open-source and commercial LP solvers such as GLPK, CLP and CPLEX. PyCPX (Koepke, a) is a Cython interface to CPLEX that leverages the power of Numpy[6]—a library defining the standard array type in Python—and provides more convenient modeling facilities than the default CPLEX Python API. Pylpsolve (Koepke, b) is a similar interface to lpsolve (Berkelaar) and Pycoin (Silva, 2005) 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 (Aides) is a wrapper for the interior-point optimizer IPOPT (Wächter and Biegler, 2006).

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.

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, the i-th row of M by M^i and the i-th element of M_j by m_{ij}. 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 Euclidian norm.

2 Implementing Simplex Pivot Rules

In this section, we give a high-level, and by no means complete, account of the Simplex method and an overview of the CLP implementation.

2.1 The Simplex Method

The linear programming problem in standard form is

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

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. It divides variables into two categories: basic and non-basic. Let $\mathcal{B}$ be the index set of basic variables, also called the *basis*, and $\mathcal{N}$ be that of the non-basic variables. At every Simplex iteration, $|\mathcal{B}| = m$ and $|\mathcal{N}| = n - m$. Non-basic variables are fixed to zero because if (LP) has a solution, then there exists a solution with at most m nonzero elements (Dantzig, 1998). Using those index sets, we partition A, x, and c as

$$A = \begin{bmatrix} A_{\mathcal{B}} & A_{\mathcal{N}} \end{bmatrix}, \quad x = \begin{bmatrix} x_{\mathcal{B}} \\ x_{\mathcal{N}} \end{bmatrix} \quad \text{and} \quad c = \begin{bmatrix} c_{\mathcal{B}} \\ c_{\mathcal{N}} \end{bmatrix}.$$

For simplicity of exposition, we assume here that the m-by-m submatrix $A_{\mathcal{B}}$ is nonsingular at every iteration. This assumption is not required in practice as all that is required is the solution of a linear system with coefficient matrix $A_{\mathcal{B}}$.

[4] www.cplex.com
[5] www.gurobi.com
[6] www.numpy.org

Upon re-writing the equality constraint in (LP) as $A_{\mathcal{B}}x_{\mathcal{B}} + A_{\mathcal{N}}x_{\mathcal{N}} = b$, we extract $x_{\mathcal{B}} = A_{\mathcal{B}}^{-1}(b - A_{\mathcal{N}}x_{\mathcal{N}})$. In the same manner and using this expression for $x_{\mathcal{B}}$, the objective function becomes $c^T x = c_{\mathcal{B}}^T A_{\mathcal{B}}^{-1} b + (c_{\mathcal{N}} - A_{\mathcal{N}}^T A_{\mathcal{B}}^{-T} c_{\mathcal{B}})^T x_{\mathcal{N}}$. Each iteration of Simplex ensures $x_{\mathcal{B}} \geq 0$. Because Simplex fixes $x_{\mathcal{N}} = 0$, these expressions further simplify to $x_{\mathcal{B}} = A_{\mathcal{B}}^{-1} b$ and $c^T x = c_{\mathcal{B}}^T A_{\mathcal{B}}^{-1} b$. If an improvement is possible, it has to involve increasing a non-basic variable. The last expression of $c^T x$ shows that unit changes in $x_{\mathcal{N}}$ change the objective value at the rate $r = c_{\mathcal{N}} - A_{\mathcal{N}}^T A_{\mathcal{B}}^{-T} c_{\mathcal{B}}$. Therefore, if $r \geq 0$, $(x_{\mathcal{B}}, 0)$ is an optimal solution. The vector r is called the *reduced cost* vector. At each Simplex iteration we look for variables with a negative reduced cost and move one of them—the *entering variable*—into the basis. Since we maintain m basic variables at all times, one variable has to leave the basis—the *leaving variable*. This process of swapping two variables in and out of the basis is called *pivoting*. If we define $\bar{A}_{\mathcal{N}} := A_{\mathcal{B}}^{-1} A_{\mathcal{N}}$ and $\bar{b} := A_{\mathcal{B}}^{-1} b \geq 0$, the choice of the leaving variable is guided by the requirement that the next $x_{\mathcal{B}} = \bar{b} - \bar{A}_{\mathcal{N}} x_{\mathcal{N}} \geq 0$. For a given entering variable x_j ($j \in \mathcal{N}$), we select the leaving variable x_k ($k \in \mathcal{B}$) as that allowing the largest increase possible in the value of x_j, i.e., $k \in \operatorname{argmin}_{i \in \mathcal{B}} \{\bar{b}_i / \bar{a}_{ij} \mid \bar{a}_{ij} > 0\}$. From the definition of reduced cost, when x_j enters the basis the objective function improvement is equal to $r_j \bar{b}_k / \bar{a}_{kj}$ where k is the index of the leaving variable. The Simplex method is outlined in Algorithm 2.1. There are many pivot rules for choosing the entering and the leaving variables (Terlaky and Zhang, 1993). Moreover, pivot rules can have a significant impact on the practical performance of the Simplex method.

Algorithm 2.1 Generic outline of the Simplex method

Step 0. Determine initial $\mathcal{B}$ and $\mathcal{N}$ by identifying a feasible starting point $x = (x_{\mathcal{B}}, 0)$ with $x_{\mathcal{B}} \in \mathbb{R}^m$.

Step 1. Compute the reduced cost vector $r = c_{\mathcal{N}} - A_{\mathcal{N}}^T A_{\mathcal{B}}^{-T} c_{\mathcal{B}}$. If $r \geq 0$, the solution is $(x_{\mathcal{B}}, 0)$. Otherwise choose j such that $r_j < 0$ and set x_j as the *entering variable*.

Step 2. If $\bar{A}_j \leq 0$ then the problem is unbounded. Otherwise, set x_k as the *leaving variable* where $k \in \operatorname{argmin}_{i \in \mathcal{B}} \left\{ \bar{b}_i / \bar{a}_{ij} \mid \bar{a}_{ij} > 0 \right\}$.

Step 3. Pivot: set $\mathcal{N} \leftarrow \mathcal{N} \setminus \{j\} \cup \{k\}$ and $\mathcal{B} \leftarrow \mathcal{B} \setminus \{k\} \cup \{j\}$. Return to Step 1.

LP solvers, commercial or free, typically implement several predefined pivot rules. Most of these rules fall into one of the following two categories. The first category is that of rules similar to the original pivot rule presented by Dantzig (1998), where at each iteration a non-basic variable with the smallest negative reduced cost is chosen to enter the basis. Variations on this method choose a variable with a negative reduced cost that is not necessarily the minimum—an approach called *partial pricing*. The second category contains rules based on the steepest-edge method (Goldfarb and Reid, 1977; Harris, 1975; Forrest and Goldfarb, 1992), which work with normalized reduced costs.

It is possible to show that using Dantzig's rule, the iterate moves from the current vertex to an adjacent vertex along an edge d of the feasible polyhedron such that the directional derivative $c^T d$ is as negative as possible. By contrast, a steepest edge rule selects an edge d such that the directional derivative along the normalized edge direction $c^T d / \|d\|$ is as negative as possible. Steepest edge rules are known to outperform Dantzig's original pivot selection by large margins (Wolfe and Cutler, 1963) but require a significantly higher programming effort.

2.2 Implementing New Pivot Rules

In Step 2 of Algorithm 2.1, the definition of $x_{\mathcal{B}}$ implies that $\bar{b} \geq 0$. Therefore, if there exists a $k_0 \in \mathcal{B}$ such that $\bar{b}_{k_0} = 0$ then all choices of k are such that $\bar{b}_k / \bar{a}_{ij} = 0$, and from §2.1 we know that performing the pivot will cause no improvements in the objective function. This type of pivot is called a degenerate pivot and an LP for which such pivots occur is said to be degenerate (Greenberg, 1986). Simplex may be very slow on degenerate LPs or even fail to converge because of cycling. Simple modifications to pivot selection can help avoid cycling, e.g., the method proposed by Bland (1977). Degeneracy occurs frequently in real-world LPs including but not limited to large-scale set partitioning problems. Raymond et al. (2010a) report manpower planning problems with a degeneracy level of 80%, i.e. at each iteration of Simplex we have $\bar{b}_i = 0$ for typically 80% of $i \in \mathcal{B}$. Likewise the patient distribution system (pds) instances of Carolan et al. (1990) 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. (2010a) 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. In the remainder of this section, we explain how new rules may be implemented in CLP and motivate the need for the higher-level mechanism provided by CyLP.

CLP implements both Dantzig's pivot selection and two variants of steepest edge methods with optional partial pricing. In CLP, an 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 19 lines—see Listing 1. 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

The positive edge rule is a recent method to handle degenerate LPs efficiently. To explain this method we consider (LP) and the notation from §2.1.

We stated in §2.2 that if there is a row i for which $\bar{b}_i = 0$ and $\bar{a}_{ij} > 0$ for some j then we are facing degeneracy. In large-scale LPs it is possible to spend the majority of the solution time performing degenerate pivots.

The positive edge criterion of Raymond et al. (2010a) allows us to identify degenerate pivots. Let $Q := A_{\mathcal{B}}^{-1}$ to simplify notation. Define $\mathcal{Z} = \{i = 1, \ldots, m \mid \bar{b}_i = 0\}$ and $\mathcal{P} = \{i = 1, \ldots, m \mid \bar{b}_i > 0\}$. Note that if $\mathcal{Z} = \emptyset$, a nondegenerate Simplex iteration is guaranteed to exist. Accordingly, we partition Q and A row-wise as

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

where $Q^{\mathcal{P}} \in \mathbb{R}^{|\mathcal{P}| \times m}$, $Q^{\mathcal{Z}} \in \mathbb{R}^{|\mathcal{Z}| \times m}$, $A^{\mathcal{P}} \in \mathbb{R}^{|\mathcal{P}| \times n}$, and $A^{\mathcal{Z}} \in \mathbb{R}^{|\mathcal{Z}| \times n}$. Using a notation similar to that of §2.1, 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 $QA_{\mathcal{B}} = I$.

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 $\{1, \ldots, m\}$, 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 chosen by the ratio test (§2.1) the improvement in the objective value, $r_j \bar{b}_i / \bar{a}_{ij}$, 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 happenning in $\mathbb{R}^{|\mathcal{Z}|}$ is zero. However, in IEEE double precision arithmetic, this probability is proven to be 2^{-62} (Raymond et al., 2010a). Therefore, the probability that the statement

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

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. In order to obtain a Simplex algorithm equipped with the positive edge rule, Algorithm 3.1 should replace Step 1 of Algorithm 2.1.

The positive edge method has a parameter that specifies the preferability of compatible variables. We denote this parameter by $0 < \psi < 1$. 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 $j \in \mathcal{N}$, do:

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

Step 2. If $r_{\text{comp}} < \psi\, r_{\min}$, choose the compatible 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.

At Step 1 of Algorithm 3.1, r_j denotes the j-th component of the vector of reduced costs defined in §2.1, 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 (Raymond et al., 2010a).

4 Implementation Details and Examples

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 is to experiment with new pivot rules. One of the leading open-source LP solvers, CLP, is part of the COIN-OR project (Lougee-Heimer, 2003) and has several advantages that make it our solver of choice for the development of CyLP. Firstly, CLP has an object-oriented structure which 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 implements the state of the art, and has been exercised and debugged to satisfaction.

In CLP it is possible to define customized pivot rules but a good understanding of its internal structure and of C++ are required. To simplify the exposition we show a partial UML class diagram (Larman, 2001) of CLP in Figure 1. A class `ClpSimplex` implements 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.

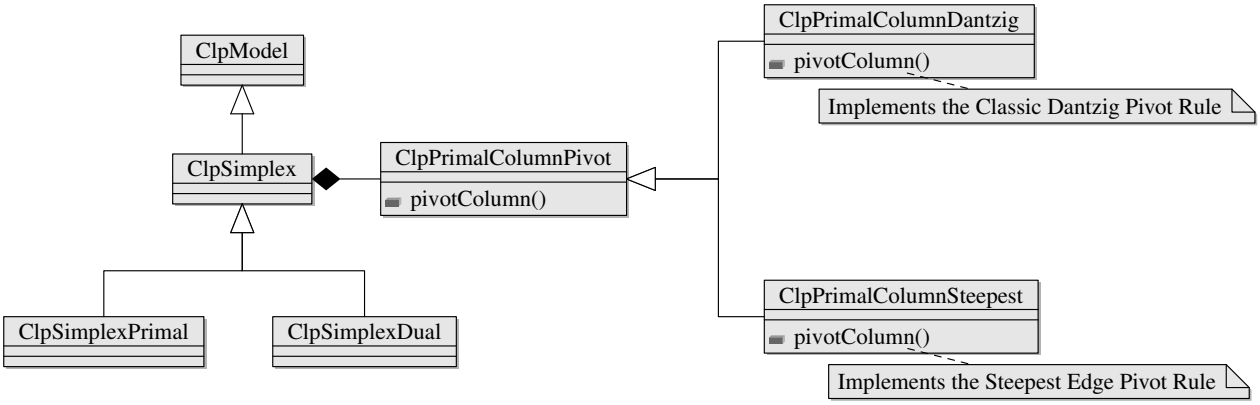

Figure 1: Partial UML Class Diagram for CLP

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[7] 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 such as the fact that call-by-reference arguments are currently not supported in Cython. Thus, for Cython to communicate with C++ code, it may be necessary to wrap certain functions and methods so as to modify their apparent signature or return value. Consider for example a function that takes an argument by reference, `f(A &a)` where `A` is a C++ class. One way to work with `f()` in Cython is to interface instead with a C++ wrapper of `f()`: `void f_wrap(A *a) {f(*a);}`. Another example is that of a function that returns an array generated in C++ , say, in the form of a `double*`. Although we can use a `double*` array directly in Cython, we may prefer working with a Numpy array not only because it provides the same constant access time to array elements, but also because it is endowed with a variety of array operations we may need. However, if we require to use the array in Python, the conversion to a Python-understood type like a Numpy array is inevitable. A role of the auxiliary C++ layer is to expose the return array to Cython and Python by wrapping it into a Numpy array data structure. This is achieved by returning the array to Cython as a `PyObject*`, i.e., a pointer to a generic Python object, and subsequently casting this pointer as a pointer to a Numpy array inside Cython. Though it is possible to return a `double*` directly and initialize the Numpy array at the Cython level, we decided against this method for performance reasons.

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 efficienly 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 Figure 2. The rest of this section is devoted to explaining each of these layers in more detail. To simplify the presentation, we abbreviate `ClpPrimalColumnPivot` to just `Pivot`.

[7] `www.scipy.org`

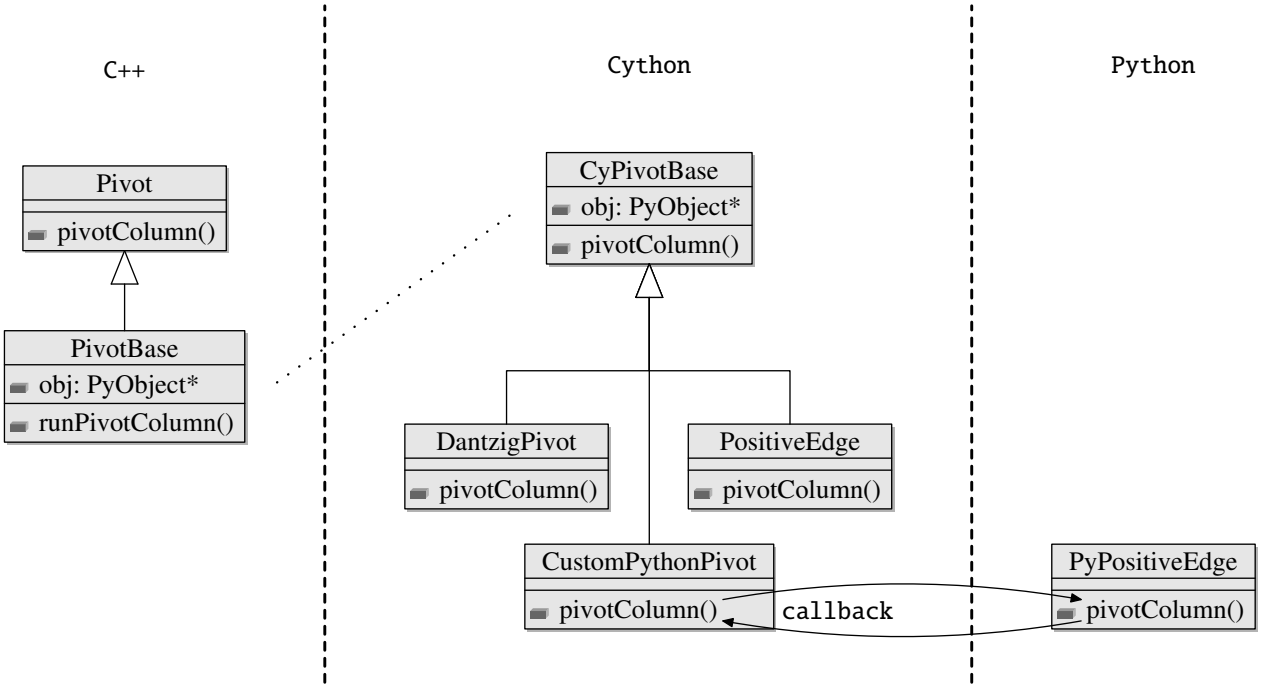

Figure 2: Schema of the three layers of CyLP

In §4.1 we go into some of the details of our implementation of CyLP, the reason being that interfacing a C++ library is not obvious. Moreover, those details may be relevant in other contexts. A reader who wishes to skip over those details may safely go directly to §4.2.

4.1 Implementation Details

As a superset of the Python language, Cython is itself object oriented. If Cython supported inheritance from C++ classes, we could create a Cython subclass of `Pivot` and override `pivotColumn()`. Unfortunately it is currently not possible (at least in an automated way) to inherit from a C++ class inside Cython and we have to find a workaround[8]. What we want is to have a class equivalent to `Pivot` in Cython, say `CyPivot`, which has the same functionality, i.e., users should be able to define Cython subclasses of `CyPivot` and implement pivot rules by overriding the `pivotColumn()` method. The key to achieving this is understanding the programming concept of *name binding*, which is roughly the process of associating names with objects. The difficulty at the CyLP level can be understood by first considering the following simplification of the definitions in CLP.

```
1  class Pivot{                        // Base class for the user to subclass.
2    public:
3      int pivotColumn() {return -1;} // Generic method for the user to override.
4  };
5
6  class Simplex{
7      public:
8          Pivot *pivotMethod;         // User must bind to a pivot method.
9  };
10
11 class MyFirstPivot: public Pivot{  // User-defined pivot rule.
12   public:
13     int pivotColumn() {return 0;}
14 };
15
16 class MyOtherPivot: public Pivot{  // User-defined pivot rule.
17   public:
18     int pivotColumn() {return 1;}
19 };
20
```

[8]Based on the idea discussed in `tinyurl.com/6sqvd3l`

```
21 int main(void) {
22     Simplex S;
23     S.pivotMethod = new MyFirstPivot();
24     printf("%d", S.pivotMethod->pivotColumn());
25     S.pivotMethod = new MyOtherPivot();
26     printf("%d", S.pivotMethod->pivotColumn());
27 }
```

The main program above outputs `-1` twice because both times, the `pivotMethod` attribute of `S` is bound to an object of type `Pivot` at compile time, not to objects of type `MyFirstPivot` or `MyOtherPivot`. This behavior is called *static binding*. If we change the signature of the `pivotColumn()` method of the `Pivot` class and declare it `virtual`, binding will be delayed until runtime when the actual type of `pivotMethod` is known, causing the main program to output `0` and `1`. This kind of binding is called *dynamic binding*. This allows CLP to run a `pivotColumn()` of a user-defined pivot rule class.

However, defining a pivot rule class in Cython breaks the inheritance chain—CLP will not be able to recognize our class as a `Pivot` type. As a result, we must bind the C++ `pivotMethod()` to the Cython implementation inside Cython where its type is known.

For instance suppose we define a class `CyDantzigRule` and implement the classic Dantzig pivot rule in its `pivotColumn()` method. To be able to pass a `CyDantzigRule` object to C++ —where `CyDantzigRule` is not recognized as a type—we make use of generic (void) pointers. Suppose we have a void pointer `ptr`, pointing to a `CyDantzigRule` instance. Because casting the void pointer is necessary, Cython requires prior knowledge of the user-defined class, here `CyDantzigRule`, to execute the corresponding `pivotColumn()` method, as illustrated in the following listing:

```
(<CyDantzigRule>(ptr)).pivotColumn()
```

To overcome this limitation we define a Cython class `CyPivot` which shall be the parent of every pivot rule implemented in Cython. Now we are certain that the void pointer can safely be up-cast to a `CyPivot` pointer. The Cython function defined in the following listing

```
cdef int RunPivotColumn(void *ptr):
    return (<CyPivot>(ptr)).pivotColumn()
```

performs this task and runs the `pivotColumn()` method. This is where the call to `pivotColumn()` is bound (dynamically) to `CyDantzigRule.pivotColumn()`.

Taking an additional step ahead, implementing a pivot rule directly in Python would be much more convenient. To this end, we define the `CustomPythonPivot` subclass of `CyPivot` as we would have done had we wanted to define another pivot rule in Cython. At variance with `CyPivot`, instances of `CustomPythonPivot` have a `pivotMethod` attribute that will be bound to a user-defined Python object implementing `pivotColumn()`. This binding occurs when the user registers their pivot rule for use in the Simplex implementation. This registration stage is illustrated in §4.4. For example suppose a user defines the `PyDantzigRule` class and implements a Python version of Dantzig's pivot rule in the `pivotColumn()` method. Upon registering this pivot rule, `CustomPythonPivot.pivotColumn()` is set to call `pivotMethod.pivotColumn()` (instead of implementing a pivot rule itself). In brief, we provide to `CustomPythonPivot` a reference to an actual pivot implementation—a *callback*.

4.2 Implementation of a Classic Pivot Rule in Cython and Python

Listing 1 gives the definition of the classic Dantzig pivot rule in Python. We use the Numpy package to gain performance. We define a class deriving from the `PyPivot` class. In a method that is (and must be) called `pivotColumn()` we first fetch the reduced costs. Then, using the Numpy `where()` function, we gather indices of the variables that are unbounded or those which are at their bounds and whose reduced costs are large enough and have a favorable sign. Among these variables, we choose the one with the maximum absolute value.

```python
import numpy as np

class PyDantzig(PyPivot):
    def pivotColumn(self):
        s = self.clpModel
        rc = s.reducedCosts
        tol = s.dualTolerance()

        indicesToConsider = np.where(s.varNotFlagged & s.varNotFixed &
                                     s.varNotBasic &
                                     (((rc > tol) & s.varIsAtUpperBound) |
                                     ((rc < -tol) & s.varIsAtLowerBound) |
                                     s.varIsFree))[0]

        abs_rc = abs(rc[indicesToConsider])

        if len(indicesToConsider) > 0:
            return  indicesToConsider[np.argmax(abs_rc)]
        return -1
```

Listing 1: Dantzig's Pivot Rule in Python

Implementing Dantzig's pivot rule in Cython is essentially similar to the Python implementation in Listing 1. The major difference is that in Cython the class is defined as `cdef class CyDantzig(CyPivot)`.

4.3 Implementation of the Positive Edge Rule

In this section we demonstrate how to implement the positive edge rule of §3.

The core of the positive edge pivot rule implementation is essentially similar to that of Dantzig's rule, illustrated in Listing 1. The difference is that it should incorporate Algorithm 3.1 into its pivot selection process.

We define the Python class `PositiveEdge`, a subclass of `PyPivot`, possessing an `updateW()` method used to perform the optional update of `w` specified in Step 0 of Algorithm 3.1, as shown in Listing 2. First this method populates `z` with indices of the constraints for which the right-hand-side is smaller in absolute value than `self.EPSILON`—a predefined threshold. Then it sets corresponding elements in `self.w` to a random number. Finally the call to `vectorTimesB_1()` multiplies `self.w` by A_B^{-1} in place.

```python
    def updateW(self):
        z = np.where(np.abs(self.rhs) <= self.EPSILON)[0]
        self.w[z] = np.random.random(len(z))
        s = self.clpModel
        s.vectorTimesB_1(self.w)
```

Listing 2: Updating w

To implement the `pivotColumn()` method of the postive edge rule we use Dantzig's rule implementation in Listing 1 as a starting point.

Each variable compatibility check requires a dot product, i.e. $A_j^T w$ for $j \in \mathcal{N}$. Instead, we choose to check the compatibility of all the non-basic variables at once by performing a vector by matrix multiplication, i.e. $A_{\mathcal{N}}^T w$, which is more efficient. To this end, we use a wrapper of `ClpModel`'s `transposeTimesSubset()` using the call

```
s.transposeTimesSubset(idx, w, Aw)
```

which multiplies w by the rows of $A_{\mathcal{N}}^T$ specified in the list `idx`. The result is stored in `Aw`, which is a Numpy array. To get the indices of compatible variables we can use Numpy's function `where()` once again:

```
compVars = idx[np.where(abs(Aw[idx]) < self.EPSILON)[0]]
```

We next identify a compatible variable with maximum reduced cost:

```
compRc = abs(rc[compVars])          #Reduced costs of the compatible variables
maxCompIdx = compVars[np.argmax(compRc)]
```

We compare `rc[maxCompIdx]` with the reduced cost of the variable chosen by Dantzig's method considering the preferability of compatible variables, by the predefined parameter ψ, and decide either to choose a compatible or an incompatible variable. If an incompatible variable is chosen, we call `updateW()` to reconstruct w before moving on to the next iteration.

As a result, we are able to implement the positive edge method in Python in 38 lines while the same implementation in C++ takes 106 lines.

4.4 A Complete Example Usage of CyLP

Suppose that we have a LP defined in an MPS file `lp.mps`. To solve this problem in Python using the positive edge pivot method we use:

```
s = CyClpSimplex()
s.readMps("lp.mps")
s.preSolve(tol=1.0e-8)  # Optional presolve step.
pivot = PositiveEdgePivot(s)
s.setPivotMethod(pivot)
s.primal()              # Executes primal Simplex.
```

Listing 3: Using Custom Pivot Rules

where we first create an instance of `CyClpSimplex`—a class which 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 the CLP's primal Simplex method.

4.5 Modeling Facilities

As an alternative to reading from a file, CyLP provides intuitive modeling facilities to express linear programming problems. Listing 4 shows how to model (3), solve it using primal Simplex, add a new constraint and solve again.

$$
\begin{array}{lrcrcrcrcrcr}
\text{minimize} & & & x_0 & - & 2x_1 & + & 3x_2 & + & 2y_0 & + & 2y_1 & & \\
\text{subject to} & & & x_0 & + & 2x_1 & & & & & & & \leq & 5 \\
 & & & x_0 & & & + & x_2 & & & & & \leq & 2.5 \\
 & 2 & \leq & x_0 & & & & & + & y_0 & + & 2y_1 & \leq & 4.2 \\
 & 2 & \leq & & & & & x_2 & & & + & y_1 & \leq & 3 \\
 & & & & & & & & & & & (y_0, y_1) & \geq & 0 \\
 & & & & & & & & & & & 1.1 \leq x_1 & \leq & 2 \\
 & & & & & & & & & & & 1.1 \leq x_2 & \leq & 3.5.
\end{array}
\tag{3}
$$

```
import numpy as np
from CyClpSimplex import CyClpSimplex
from CyLP.python.modeling.CyLPModel import CyLPModel, CyLPArray

model = CyLPModel()  # Initialize model.
x = model.addVariable('x', 3)
y = model.addVariable('y', 2)

# Define coefficient matrices and bound vectors.
A = np.matrix([[1., 2., 0],[1., 0, 1.]])
B = np.matrix([[1., 0, 0], [0, 0, 1.]])
D = np.matrix([[1., 2.],[0, 1]])
a = CyLPArray([5, 2.5])
b = CyLPArray([4.2, 3])
u = CyLPArray([2., 3.5])

# Add constraints and bounds to model.
model.addConstraint(A * x <= a)
model.addConstraint(2 <= B * x + D * y <= b)
model.addConstraint(y >= 0)
model.addConstraint(1.1 <= x[1:3] <= u)

# Define the objective function
c = CyLPArray([1., -2., 3.])
```

```
25 model.objective = c * x + 2 * y
26
27 s = CyClpSimplex(model) # Create CyClpSimplex object using model.
28 s.primal()              # Solve. Solution: [0.2  2.   1.1  0.   0.9]
29
30 s.addConstraint(x[2] + y[1] >= 2.1) # Add a cut.
31 s.primal()              # Warm start. Solution: [ 0.  2.  1.1  0.   1.]
```

Listing 4: Creating and solving an LP using CyLP modeling facility

`CyLPModel` objects encapsulate an LP's variable and constraint information. The `addVariable()` method returns a `CyLPVar` object which we use later to add constraints and bounds.

Lines 10–15 define coefficient matrices and vectors. Vectors are defined using `CyLPArray` objects instead of the more familiar Numpy arrays, whereas matrices may be defined using Numpy's `matrix` objects. The reason lies in the way Numpy array operators take precedence. If we compare a `CyLPVar` object, `x` with a Numpy array `b` using the expression `b >= x` the `>=` operator of `b` is called and returns a Numpy array with the same dimension as `b` and all its elements set to `False`, which is of no significance since `b` is trying to compare itself with another object that it does not know of. We would expect, instead, the `<=` operator of `x` to execute, and to save `b` as the lower bound on `x`. This information could then be used later to construct a `CyClpSimplex` object. To this end, we use `CyLPArray` objects which are Numpy arrays in most respects except that they concedes performing an operation if the other operand is a `CyLPVar` object.

Constraints and variable bounds are declared using the `addConstraint()` method as in lines 18–21. The objective function is defined by setting `CyLPModel`'s `objective` attribute. Once the LP is defined, the `writeMps()` method allows us to write the problem to file in `mps` format. This makes the choice of modeling with CyLP independent of the choice of solver. In CyLP, LPs may be solved using primal or dual Simplex. The modeling tool also makes it easy to generate a problem dynamically, as in cut-generation or branch-and-cut techniques in integer programming, by allowing users to add constraints at any time and re-solve the problem with a *warm start*, i.e., using the solution of the last execution as the initial point for the new problem. In the next section we demonstrate how CyLP can be used to solve MIPs and how users can customize the branch-and-cut process using Python callbacks.

4.6 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 using a branch-and-cut strategy. 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[9]. 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. Although the interface is still at a preliminary stage, it already 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, 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.

Suppose that in the LP of Listing 4, x_0, x_1 and x_2 should be integer. This may be specified using the intuitive syntax `s.setInteger(x[1:3])`. Once the MIP has been modeled, it may be solved using an instance of the `CyCbcModel` class. Listing 5 illustrates how to solve a MIP using two cut generators. For problems that were not modeled using `CyLPModel`, specific variables may be marked as integer using `copyInIntegerInformation()` as in line 5. In lines 4–5, we read an LP from an `mps` file and mark all variables as integers. 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, that 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.

[9] `projects.coin-or.org/Cgl`

```
clpModel = CyClpSimplex()

# Read problem from file and mark all variables as integers.
clpModel.readMps('lp.mps')
clpModel.copyInIntegerInformation(np.array(clpModel.nCols * [True], np.uint8))

# Solve initial relaxation and obtain a CyCbcModel object.
cbcModel = clpModel.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 5: Integer Programming with CyLP

```
class SimpleNodeCompare(NodeCompareBase):
    def __init__(self):
        self.method = 'depth'   # Default strategy.

    def compare(self, x, y):
        "Return True if node y is better than node x."
        if x.nViolated != y.nViolated:
            return (x.nViolated > y.nViolated)
        if x.depth == y.depth:
            return x.breakTie(y)   # Break ties consistently.
        if self.method == 'depth':
            return (x.depth < y.depth)
        return (x.depth > y.depth)

    def newSolution(self, model, objContinuous, nInfeasContinuous):
        "Cbc calls this after a solution is found in a node."
        self.method = 'breadth'

    def every1000Nodes(self, model, nNodes):
        "Cbc calls this every 1000 nodes for possible change of strategy."
        return False        # Do not demand a tree re-sort.
```

Listing 6: A simple node comparison impelementation in Python

CBC provides the capability to customize its branch-and-cut node selection process by writing C++ callbacks. For this purpose, CyLP enables us to, instead, use Python. 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, at first we 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 6 illustrates how to implement this strategy by defining a class that inherits from the relevant base class `NodeCompareBase`.

Our subclass 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 5, we set the node comparison method by registering an instance `snc` of `SimpleNodeCompare` with the `CbcModel` object using `cbcModel.setNodeCompare(snc)`.

5 Numerical Experiments

In this section, 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 implementations. Later, we demonstrate how CyLP can be used to examine the effectiveness of the positive edge pivot rule.

We choose the Netlib LP benchmark[10] for the first part, which contains 93 LPs of diverse dimensions and sparsity. All our experiments are conducted on computers with Intel Xeon 2.4GHz CPUs and 49GB of total shared memory. Let t^d_{cpp}, t^d_{cy} and t^d_{py} denote the execution times of Algorithm 2.1 using C++ , Cython and Python versions of Dantzig's pivot rule, respectively.

Figure 3 show the performance profile (Dolan and Moré, 2002) 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 for 50% of the instances, the Cython and Python versions are less than 3 times slower than the C++ version.

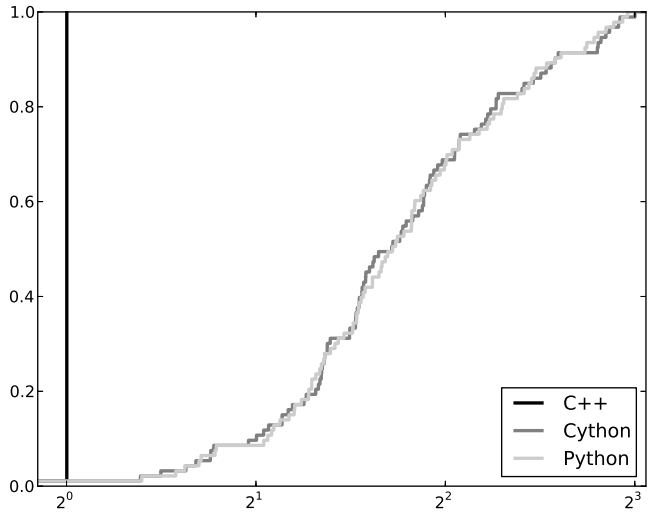

Figure 3: Performance profile for the execution time of the primal Simplex Algorithm using the C++ , Cython and Python implementations of Dantzig's pivot rule

We next select those Netlib instances that take more than 5 seconds 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 Figure 4. Problems are sorted by C++ execution time from `greenbeb`, taking 5 seconds, to `dfl001`, taking 9729 seconds. 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.

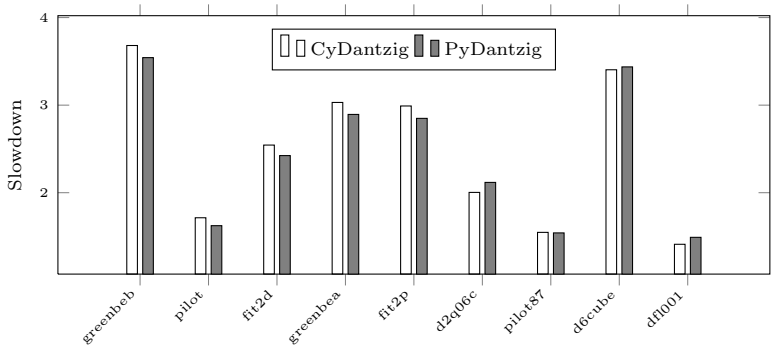

Figure 4: Performance hit caused by Python and Cython compared to C++

[10] `www.netlib.org/lp/data`

For the second part of the numerical tests we choose from among the `pds` instances from Mittlemann's benchmark (Carolan et al., 1990), 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 seconds for Dantzig and 0.0004 seconds for positive edge. As for Python, the average iteration time is 0.0019 seconds for Dantzig and 0.0017 seconds for positive edge. This means that positive edge is adding an extra 0.0002 seconds 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 taking 3×10^{-6} seconds longer.

Table 1 demonstrates the superiority of the positive edge method over Dantzig's pivot rule for larger `pds` instances. In fact, increasing speedups for both implementations show that the effectiveness of the positive edge rule increases as the problem size grows, both in terms of run time and in terms of number of pivots.

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	553	583	0.32	0.78
pds-06	28655	9881	7759	2816	1.03	2.85
pds-10	48763	16558	37939	6890	2.50	6.12
pds-20	105728	33874	293668	31584	3.92	9.78
pds-30	154998	49944	597447	65657	4.20	9.75
pds-40	212859	66844	1504587	139376	5.20	10.85

6 Discussion and Future Work

CyLP, available from `github.com/mpy/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 pivot rules in Python and have CLP use them during primal Simplex. We demonstrated this feature by implementing the positive edge pivot rule in C++ and Python. We feel 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. Moreover, this slowdown becomes minor as problem size grows.

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 Terlaky and Zhang (1993), 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.

Currently, custom pivot rules may only be passed to the primal Simplex solver. In the future, we wish to provide facilities to implement dual pivot rules in Python. As an improvement to the integer programming facilities—where we are already capable of defining the branch-and-cut node comparison rule in Python—we will consider adding the capability to script cut generators in the same manner.

In follow-up research, we consider Wolfe's pivot rule to solve the KKT system of a convex quadratic program (Wolfe, 1959). The KKT system of a QP is a set of linear equations if we set aside the complementarity conditions. Wolfe proposes to solve an LP to find a feasible point for this system by using a specific pivot

strategy to take care of complementarity. Our goal in doing so will be to investigate the application of the positive edge rule and of the constraint aggregation techniques of Raymond et al. (2010b) to convex quadratic programming. The interface to CLP described in the present paper will let us implement Wolfe's rule and construct modified linear programs 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 (Wächter and Biegler, 2006) 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

A. Aides. Cython wrapper for IPOPT. `http://code.google.com/p/cyipopt`. [Online; accessed 2-November-2011].

M. Berkelaar. lpsolve, A Mixed Integer Linear Programming Software. `http://lpsolve.sourceforge.net`. [Online; accessed 2-November-2011].

R. G. Bland. New finite pivoting rules for the Simplex method. *Mathematics of Operations Research*, 2(2):103–107, 1977. ISSN 0364765X.

W. Carolan, J. Hill, J. Kennington, S. Niemi, and S. Wichmann. Empirical Evaluation of the KORBX Algorithms for Military Airlift Applications. *Operations Research*, 38:240–248, 1990.

B. A. Cipra. The best of the 20th century: editors name top 10 algorithms. *SIAM News*, 33(4):1–2, 2000.

CLP. COIN-OR Linear Programming. `https://projects.COIN-OR.org/Clp`. [Online; accessed 2-November-2011].

G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1998. ISBN 0691059136.

E. Dolan and J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming B*, 91:201–213, 2002.

J. Dongarra and F. Sullivan. Guest editors' introduction: The top 10 algorithms. *Computing in Science and Engineering*, 2:22–23, 2000. ISSN 1521-9615. DOI: 10.1109/MCISE.2000.814652.

J. J. Forrest and D. Goldfarb. Steepest-edge Simplex algorithms for linear programming. *Mathematical Programming*, 57:341–374, 1992. ISSN 0025-5610. DOI: 10.1007/BF01581089.

D. Goldfarb and J. K. Reid. A practicable steepest-edge Simplex algorithm. *Mathematical Programming*, 12:361–371, 1977. ISSN 0025-5610. DOI: 10.1007/BF01593804.

H. J. Greenberg. An analysis of degeneracy. *Naval Research Logistics Quarterly*, 33:635–655, 1986.

P. M. J. Harris. Pivot selection methods of the Devex LP code. In R. W. Cottle, L. C. W. Dixon, B. Korte, M. J. Todd, E. L. Allgower, W. H. Cunningham, J. E. Dennis, B. C. Eaves, R. Fletcher, D. Goldfarb, J.-B. Hiriart-Urruty, M. Iri, R. G. Jeroslow, D. S. Johnson, C. Lemarechal, L. Lovasz, L. McLinden, M. J. D. Powell, W. R. Pulleyblank, A. H. G. Rinnooy Kan, K. Ritter, R. W. H. Sargent, D. F. Shanno, L. E. Trotter, H. Tuy, R. J. B. Wets, E. M. L. Beale, G. B. Dantzig, L. V. Kantorovich, T. C. Koopmans, A. W. Tucker, P. Wolfe, M. L. Balinski, and Eli Hellerman, editors, *Computational Practice in Mathematical Programming*, volume 4 of *Mathematical Programming Studies*, pages 30–57. Springer Berlin Heidelberg, 1975. ISBN 978-3-642-00766-8. DOI: 10.1007/BFb0120710.

K. L. Hoffman and M. Padberg. Solving airline crew scheduling problems by branch-and-cut. *Management Science*, 39:675–682, June 1993. DOI: 10.1287/mnsc.39.6.657.

H. Koepke. Cython wrapper for CPLEX. `http://www.stat.washington.edu/~hoytak/code/pycpx/index.html`, a. [Online; accessed 2-November-2011].

H. Koepke. Cython wrapper for lpsolve. `http://www.stat.washington.edu/~hoytak/code/pylpsolve/index.html`, b. [Online; accessed 2-November-2011].

C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd Edition*. Prentice Hall, 2001.

R. Lougee-Heimer. The common optimization interface for operations research. *IBM Journal of Research and Development*, 47(1):57–66, 2003. DOI: 10.1147/rd.471.0057. URL `www.COIN-OR.org`.

A. Makhorin. GLPK, GNU Linear Programming Kit. `http://www.gnu.org/s/glpk`. [Online; accessed 2-November-2011].

M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Rev.*, 33:60–100, February 1991. ISSN 0036-1445. DOI: 10.1137/1033004.

PuLP. An LP modeler written in Python. `http://code.google.com/p/pulp-or`. [Online; accessed 2-November-2011].

V. Raymond, F. Soumis, A. Metrane, and J. Desrosiers. Positive edge: A pricing criterion for the identification of non-degenerate Simplex pivots. Cahier du GERAD G-2010-61, GERAD, Montréal, Québec, Canada, 2010a.

V. Raymond, F. Soumis, and D. Orban. A new version of the improved primal Simplex for degenerate linear programs. *Computers & OR*, 37(1):91–98, 2010b. DOI: 10.1016/j.cor.2009.03.020.

P. J. S. Silva. Pycoin, interface to some COIN packages. `http://www.ime.usp.br/~pjssilva/software.html`, 2005. [Online; accessed 2-November-2011].

T. Terlaky and S. Zhang. Pivot rules for linear programming: A survey on recent theoretical developments. *Annals of Operations Research*, 46-47:203–233, 1993. ISSN 0254-5330. DOI: 10.1007/BF02096264.

A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106:25–57, 2006. DOI: 10.1007/s10107-004-0559-y. URL `https://projects.COIN-OR.org/Ipopt`.

P. Wolfe. The Simplex method for quadratic programming. *Econometrica*, 27(3):382–398, 1959.

P. Wolfe and L. Cutler. Experiments in linear programming. In Graves and Wolfe, editors, *Recent Advances in Mathematical Programming*. McGraw-Hill, New York, 1963.