Manual for LP_Solvers: Instructions, Fundamentals, and Guidelines for Primal and Dual Simplex Methods

Written By:
Corey R. Randall
Colorado School of Mines
MEGN 686 Course Project

5/2/2019

Contents

1 Introduction

This document serves as a user manual for the accompanying files found in the LP_Solvers repository. The current version of the software package is v1.2.*, but this document will be updated when necessary as additional options are implemented.

LP_Solvers stands for Linear Program (LP) Solvers. These types of programs have been developed and used to optimize problems in a diverse set of fields for many years. Professional solvers such as CPLEX [1] and Gurobi [2] have been developed to efficiently solve realistic large-scale LPs. The code developed here brings similar algorithms found in these solvers to the open-source Python programming language. All of this work has been done as part of a graduate-level course project at Colorado School of Mines during the spring 2019 semester.

In addition to providing instructions on how to use this software package, this document will also briefly cover background information on LPs, Simplex Method algorithms, and recommendations on how to tune the algorithms using the LP_User_Inputs.py file.

2 LP Fundamentals

2.1 Notation and Problem Structure

In their most basic form, linear programs involve an objective function and a set of constraints which each consist of linear terms with continuous variables. A common way to express these problems is in standard form as seen below.

$$\begin{array}{ll} \text{min} & cx \\ \text{subject to} & Ax = b \\ & x \geq 0 \end{array}$$

In this form, c represents a row vector of the objective coefficients, A represents the constraint coefficients in matrix form where each row corresponds to a different constraint, b represents a column vector of the right-hand-side (RHS) values of the constraints, and x represents a column vector of the variables in the system. Although an equality is used in standard form, it is not necessary when developing a set of constraints to describe a problem. Any inequality expression can be changed to an equality by adding slack variables or subtracting excess variables. Additionally, although a minimization is shown here, a system may equivalently be expressed as a maximization problem by using min (cx) = -1$*$max (-cx). For the remainder of this document, problems will primarily be discussed in terms of a minimization so that the algorithms can be generalized.

2.2 Duality Theory

Duality theory states that each LP has an alternate form which provides the same optimal value [3]. This alternate form is referred to as the dual problem. The relationship below shows our original problem, referred to as the primal problem ($\mathcal{P}$), in standard form next to its dual problem ($\mathcal{D}$).

$$\begin{array}{llll} (\mathcal{P}) & \text{min} & cx & \\ & \text{subject to} & Ax = b & \\ & & x \geq 0 & \end{array} \quad \Bigg| \quad \begin{array}{llll} (\mathcal{D}) & \text{max} & yb & \\ & \text{subject to} & yA \leq c & \\ & & y \text{ unrestricted} & \end{array}$$

In the dual problem, y is used as the vector of variables since it does not provide the same dimension or values as x from the primal problem. These variables are also comprised of a row vector instead of a column

vector, which is why the multiplication order has been changed. Aside from these two problems using the same values from $A,\ b,$ and c to result in the same optimal objective value, they are related in a very specific way. Throughout each iteration of the Simplex method, the algorithm calculates values referred to as dual or shadow prices. At optimality, the dual prices of $(\mathcal{P})$ are equal to the decision variable values from $(\mathcal{D})$. Likewise, the dual prices of $(\mathcal{D})$ are equal to the decision variables of $(\mathcal{P})$ when at optimality as well. This relationship is known as strong duality and it means that if a feasible finite solution exists for either one of these problems, then both problems have a feasible solution that can be simultaneously obtained by solving either problem, i.e., without the need to explicitly solve both.

When either of these problems are not at optimality, weak duality theory states that the objective function value of $(\mathcal{D})$ is greater than that of $(\mathcal{P})$ for primal problems that are minimizations [3]. This relationship provides an explanation for how the Primal and Dual Simplex Methods operate which are provided in the following section.

3 Simplex Methods

The Simplex Method algorithms were some of the first to be developed for solving large LPs [4]. Although they have been shown to have exponential complexity [5], their practical performance has shown them to be fairly efficient for solving real world problems. Although, either of these algorithms could operate on either the primal or dual problem, they will be discussed in this section as operating on $(\mathcal{P})$.

3.1 Basic Feasible Solutions

All LPs are convex which means that their optima lie at extreme points in the system, i.e., the intersection of constraints [6]. At these intersections, a number of the variables are equal to zero. In fact, extreme points can in general be found by taking a system that has m constraints and n variables such that $m < n$ in the following way: place a LP in standard form, set $n - m$ variables equal to zero, take the resulting submatrix of A (call it A_B), invert this matrix to get A_B^{-1}, and left multiply this inverted matrix by b to get $x_B = A_B^{-1}b$. Here, we use the subscript B to denote what is referred to as the basis. The basis is made up of the remaining m variables that were not set equal to zero in the previous steps. These variables are called basic variables, while the $n - m$ variables that were originally set equal to zero are called nonbasic. When all m variables in the basis are nonnegative, they form a primal feasible solution since they do not violate any of the constraints used in standard form. Alternatively, if there are any values in the basis that are negative then primal feasibility is violated.

Although some basic solutions can violate primal feasibility, they may still be dual feasible if $c - yA \geq 0$ so that all of the constraints of $(\mathcal{D})$ hold true. Without converting to the dual problem, this criteria can be shown by calculating the dual prices of $(\mathcal{P})$ at the current iteration via $y = c_B A_B^{-1}$ where c_B is the subset of objective function coefficients that correspond to the variables in the current basis. Regardless of the feasibility of the starting basis, there exists a Simplex algorithm that will iterate to find an optimal solution if one exists. The Simplex method that begins with a point that is primal feasible is referred to as the Primal Simplex Method. On the other hand, the Simplex method that begins with a point that is dual feasible is referred to as the Dual Simplex Method. Each of these Simplex algorithms ensure that their respective feasibility is maintained throughout each iteration. It should be noted that finding an initial feasible solution for either algorithm is not a trivial matter, but will not be discussed here. In fact, there are many basic solutions that can be generated that are neither primal nor dual feasible. The following subsections provide more detailed information about the steps that are involved within each iteration for each algorithm. These steps will only cover a Phase II of the algorithms. Phase I will be assumed to have already been completed such that an initial basic feasible solution has been found and can be used as a starting point for the remaining iterations.

3.2 Primal Simplex

The Primal Simplex Method maintains primal feasibility, i.e., $Ax = b$ and $x \geq 0$ are not violated. From an initial starting point that comes from a Primal Phase I (not discussed here), adjacent extreme points are examined to see if an improving direction exists. If such a direction does not exist then the algorithm terminates; otherwise, a variable is pivoted into the basis to replace another. The variable that gets removed from the basis is chosen using a minimum ratio test so that primal feasibility is not lost. In a concise manner, this algorithm can be comprised of four steps:

Step 0. Start with a basic primal feasible solution with basis B from a Primal Phase I.

Step 1. Calculate the current basic variable values as $\bar{b} = A_B^{-1}b$, dual prices as $\bar{y} = c_B A_B^{-1}$, and reduced costs as $\bar{c} = c - \bar{y}A$. If all of the reduced costs are nonnegative, then the current solution is optimal. Otherwise, continue to **Step 2**.

Step 2. Select an incoming variable from the reduced costs that were negative. The indices of the positions of the negative reduced costs each correspond to a nonbasic variable that could improve the objective by pivoting it into the basis. It is common to select the index of the most negative entry, but not necessary. Let this index be referred to as t.

Step 3. Perform a minimum ratio test using the formula below where $\bar{A}_{\cdot t} = A_B^{-1}A_{\cdot t}$. This will provide a variable index to leave the basis while ensuring that primal feasibility is not lost. Let the index of this leaving variable be referred to as r.

$$\frac{\bar{b}_r}{\bar{A}_{rt}} = \min_{i \ni \bar{A}_{it} > 0} \left\{ \frac{\bar{b}_i}{\bar{A}_{it}} \right\}$$

Step 4. Update the basis by replacing the r^{th} index of B with the variable in the t^{th} position of the original problem. This step also requires updating c_B, A_B, and A_B^{-1}. For large problems, inverting matrices can be computationally expensive, so many algorithms use a permutation matrix P to get $A_{B_{\text{new}}}^{-1} = PA_{B_{\text{old}}}^{-1}$ where P is defined below. Once these updates have been made, return to **Step 1**.

$$P = \begin{pmatrix} 1 & 0 & \dots & -\bar{A}_{1t}/\bar{A}_{rt} & 0 & 0 & 0 & 0 \\ 0 & 1 & \dots & -\bar{A}_{2t}/\bar{A}_{rt} & 0 & 0 & 0 & 0 \\ 0 & 0 & \dots & -\bar{A}_{3t}/\bar{A}_{rt} & 0 & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1/\bar{A}_{rt} & 0 & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & -\bar{A}_{m-1,t}/\bar{A}_{rt} & 0 & 0 & 1 & 0 \\ 0 & 0 & \dots & -\bar{A}_{mt}/\bar{A}_{rt} & 0 & 0 & 0 & 1 \end{pmatrix}$$

This permutation matrix is made up of the identity matrix aside from the r^{th} column. This column is replaced with $-\bar{A}_{\cdot t}/\bar{A}_{rt}$ for every row except for the r^{th} row. In the r^{th} row and r^{th} column, the value of $1/\bar{A}_{rt}$ replaces the 1 from the identity matrix. Ultimately, this matrix serves as a way to divide out the r^{th} column of A_B^{-1} and multiply by the t^{th}. Multiplying these two matrices together is much less computationally expensive than inverting the new A_B from scratch.

3.3 Dual Simplex

As opposed to the Primal Simplex Method, the Dual Simplex Method does not maintain primal feasibility. Instead, the Dual Simplex Method preserves dual feasibility, i.e., $yA \leq c$ at each iteration. Although dual feasibility is maintained, there is never a need to convert the problem to its dual form. From weak duality, it is known that ($\mathcal{P}$) and ($\mathcal{D}$) cannot both be feasible when not at optimality. This means that while dual feasibility is maintained, primal feasibility is violated until optimality is reached. From an initial starting point that comes from a Dual Phase I (not discussed here), values that violate $x \geq 0$ are removed from the basis and replaced with another. This effectively moves the solution from one extreme point to another that is adjacent and in a direction that reduces the infeasibility of ($\mathcal{P}$). A new variable is selected to enter the basis based on a minimum ratio test that ensures dual feasibility is not lost. This algorithm can also be summarized in four steps:

Step 0. Starting with a basic dual feasible solution with basis B from a Dual Phase I.

Step 1. Calculate the current basic variable values as $\bar{b} = A_B^{-1}b$, dual prices as $\bar{y} = c_B A_B^{-1}$, and reduced costs as $\bar{c} = c - \bar{y}A$. If all of the $\bar{b}$ values are nonnegative, then the current solution is optimal. Otherwise, continue to **Step 2**.

Step 2. Select an outgoing variable from the $\bar{b}$ values that were negative. The indices of the positions of the negative $\bar{b}$ values each correspond to a basic variable that contributes toward primal infeasibility. Removing any of these variables from the basis could reduce primal infeasibility and therefore progress toward optimality, i.e., where ($\mathcal{P}$) and ($\mathcal{D}$) are both feasible. It is common to select the index of the most negative entry, but not necessary. Let this index be referred to as r.

Step 3. Perform a minimum ratio test using the formula below where $\bar{A}_{r\cdot} = (A_B^{-1})_{r\cdot}A$. This will provide a variable to enter the basis while ensuring that dual feasibility is not lost. Let the index of this incoming variable be referred to as t.

$$\frac{-\bar{c}_t}{\bar{A}_{rt}} = \min_{j \ni \bar{A}_{rj} < 0} \left\{ \frac{-\bar{c}_j}{\bar{A}_{rj}} \right\}$$

Step 4. Update the basis by replacing the r^{th} index of B with the variable in the t^{th} position of the original problem. This step also requires updating c_B, A_B, and A_B^{-1}. The permutation matrix from the Primal Simplex Method can again be used here to improve computational efficiency, but $\bar{A}_{\cdot t}$ must be calculated. In **Step 3** of the Primal Simplex Method, this column vector had already been found for use in the minimum ratio test, but there is no part of the Dual Simplex Method where it is previously calculated. Once the basis and its corresponding vectors and matrices have been updated, return to **Step 1**.

3.4 Phase I

Although obtaining an initial feasible solution for the Simplex algorithms was not discussed here, the software package does include a routine for this initialization. This means that it is not necessary for the user to spend time attempting to find this point on their own. It should be noted that the current implementation of Phase I for Primal Simplex is robust and should work for all types of problems. The Phase I of Dual Simplex is still in development and may throw exceptions to recommend the user switch algorithms. Currently, it is recommended to only use the Dual Simplex algorithm if the problem is a minimization with all positive objective coefficients and inequality constraints. This form is equivalent to a maximization problem with all negative objective function coefficients and inequality constraints. For problems that have a small number of equality constraints, this initialization may still provide a starting point in a reasonable amount of time; however, if it does not, then the appropriate settings option should be changed to use the Primal Simplex Method instead.

An update for Dual Simplex Phase I is on its way and this document will be updated accordingly once it has been tested and proven to be robust and consistent.

4 Inputting a Problem

4.1 Requirements

This code was written using Python3 so the user should set up a suitable environment that can interpret the appropriate syntax. There are a minimal number of external modules that were utilized in writing this software package, but the user should ensure that their environment has access to: `numpy`, `scipy`, `math`, `os`, `sys`, `time`, and `warnings`.

4.2 Define an Objective

The objective inputs are separated into two components: the sense of the objective and its coefficients. Within the code, the variable name "objective" has already been purposed for recording the sense of the objective. This variable takes on a string input as 'maximize' or 'minimize' depending on the formulation of the input problem. Although the algorithms coded here operate on the minimization problem, a conversion routine in the code performs the necessary operations to switch back and forth between objective senses so that the user does not need to reformulate.

In addition to the sense of the objective, its coefficients for the decision variables must also be supplied. The variable "c_coeff" has been reserved for these values. Using a list, values for the objective coefficients should be input in a known order. If some decision variables have an objective coefficient of zero, they should still be input into the list as placeholders. Although it is necessary to input zeros for certain decision variables in this list, it is not necessary to input zeros for the slack and excess variables. When the conversion routine changes the problem into its standard form, this vector is automatically extended by the appropriate amount of zeros to account for the added slack and excess variables.

4.3 Construct Constraints

The constraint set is passed to all routines in the software package together as a dictionary. Because of this, the dictionary should be initialized as `constraint = {}`. Following this line, the constraints can each be input with an increasing integer index, starting at "1", using a list. This list should contain elements for the name of the constraint, coefficients, in/equality, and RHS value. The name and in/equality are represented as strings, the coefficients are a list in the same variable order as the objective coefficients, and the RHS value is a float. A generalized example is shown below.

```
constraint[1] = ['name_1', [coeff1, coeff2, ...], '>=' or '<=' or '=', RHS]
```

Note that for the in/equality position in the constraint, only one of the options shown above should be chosen. Additionally, although it is common in coding languages to have "==" represent equals to in logic statements, the user only needs to input '=' for an equality constraint. It is currently a requirement that all formulations done using this input file include nonnegavitiy for all variables. Due to this, it is not necessary to explicitly write out any constraints to enforce nonnegativity. Any lower bounds for variables that are not zero still need to be specified as a constraint. For users wishing to solve systems with unconstrained variables, they must formulate the problem in terms of $x = x' - x''$ if they would like to allow a variable x to result in a value less than zero. The algorithm will only solve for the values of x' and x'', but the user can easily post process these values to get the value of their original variable as long as they track their indexing.

4.4 Problem Scripts

Most LPs that describe meaningful real-world problems are made up of thousands of variables and constraints. Some problems have even been developed that consist of millions of variables and constraints. For much larger and more complex problems, the user may wish to use an external script that can be comprised of sets, parameters, variables, an objective function, and constraints. Two examples have been provided in the repository of scripts that incorporate all of these pieces for larger problems. The files are titled pv_batt_sizing.py and pv_batt_sizing_sub.py. Both of these files refer to the same problem of sizing a battery and photovoltaic (PV) system for a building on the Colorado School of Mines campus. The model was originally developed by Mohammad Fathollahzadeh and Karl Heine. Their work was presented at the 2019 ASHRAE Winter Conference in Atlanta, Georgia. With their permission, the model was formatted to work with the algorithms presented here in order to test more complex problems and test different efficiency techniques that were implemented into the code. In these scripts, the most important thing to pay attention to is the indexing of the variables. If this is done in a smart way, then inputting the rest of the problem is simple. Vectors of zeros can be initialized using a module such as numpy so that the user does not have to explicitly type out the zero placeholders, but instead only needs to fill in the nonzero values. Constraints that hold true for multiple indices of a set can also be simplified by using for loops or nested for loops.

Large sets of data are often preprocessed and passed in as coefficients to LPs. These data sets can be processed and stored externally to these scripts as well. In the repository, the pv_batt_sizing_data.csv file contains daily load and generation values for an entire year that correspond to the previously mentioned model script files. To see how a data file can be read in and then have its values assigned as coefficients, please reference one of these examples.

4.5 Example Input

Consider the following problem, not in standard form:

$$\begin{aligned} \text{min} \quad & x_1 - 2x_2 \\ \text{subject to} \quad & x_1 + x_2 - x_3 \leq 2 \\ & x_1 - x_2 - x_4 \geq -1 \\ & x_2 + x_5 = 3 \\ & x_i \geq 0 \ \ \forall \, i \end{aligned}$$

This problem can be formatted in the LP_User_Inputs.py file as:

```
# Objective Function:
objective = 'minimize'
c_coeff = [1, -2, 0, 0, 0]

# Constraints:
constraint = {}
constraint[1] = ['name_1', [1, 1, -1, 0, 0], '<=', 2]
constraint[2] = ['name_2', [1, -1, 0, 1, 0], '>=', -1]
constraint[3] = ['name_3', [0, 1, 0, 0, 1], '=', 3]
```

Note that this input code follows all of the guidelines listed in the previous subsections: inputting the zero values as placeholders in the objective function and constraint coefficients, starting the constraint indices with "1", not explicitly stating nonnegavitiy, etc. This example was chosen due to its simplicity and small size. It also provides constraints that are of each type of inequality in addition to an equality constraint. Overall, this input has been written to allow users to input problems in a simple and familiar way when considering LPs. No additional time should be taken to remove negative values from the RHS or to convert inequalities to

equalities since all of these conversions are already handled in a routine included with the software package. Additional small examples can be found in the Sample_Problems.py file included in the repository.

4.6 Settings and Options

This code has already been developed to offer the user different options that affect performance or provide additional information. Within the LP_User_Inputs.py file, these settings and options can easily be changed. The following list provides a description of what each of these settings and options actually do in the code as well as what inputs they can be changed to.

- `sensitivity` = a toggle that takes in a string and allows the user to perform a RHS sensitivity analysis once optimality has been reached (options = 'on' or 'off').
- `pricing` = an option that controls the pricing scheme used to select incoming and outgoing variables in **Step 2** of the Primal and Dual Simplex Methods respectively. All pricing schemes used are simple; however, they can dramatically affect iteration counts. Currently only two string inputs for this control exist (options = 'bland' or 'most'). The 'bland' option will use Bland's rule [**?**] and select either the first or last indexed variable that provides a negative reduced cost or variable value. It is recommended to use the 'most' option as often as possible which instead selects the most negative reduced cost or variable value instead.
- `incoming` = an option that controls Bland's rule if it was selected as the pricing scheme. It takes in a string value that allows the variable from **Step 2** to be chosen based on the first or last index of the negative reduced cost or basic variable values (options = 'first' or 'last').
- `problem` = an option that takes in a string value which allows the algorithm to explicitly solve ($\mathcal{P}$) supplied by the user or to instead convert to ($\mathcal{D}$) to explicitly solve the dual without a need for the user to reformulate (options = 'primal' or 'dual'). Note: even when the dual is explicitly solved, the output organizes the variables to be associated with the primal problem. To see the dual variable values, the user should print the duals output from the solution.
- `method` = an option that takes in a string value which allows the user to control the algorithm used. As previously discussed, the current methods that are available are for the Primal and Dual Simplex algorithms (options = 'primal' or 'dual').
- `verbose` = a toggle that takes in a string which allows the user to print out the iteration number and objective function value at each iteration (options = 'on' or 'off').
- `tolerance` = an absolute tolerance which treats any input data or calculated values within the absolute value of this tolerance as zero. This variable takes in a float as its input that be expressed as a decimal or in scientific notation. It is recommended to set this tolerance no larger than 1e-1, but 1e-6 is input as the default.
- `decimals` = an option that allows the solution input printed in the terminal to be cleaned up. This variable takes in an integer value which allows the decision variables and objective function values printed in the terminal to be rounded to the specified number of decimals. The higher precision outputs are still stored and are not affected by this input.
- `var_names` = a list of string variable names that the user would like to see printed in the terminal after the solution is found. For small problems or LPs in which the variable names were not specified as shown in the script examples, this option should be set to "None" (without the quotes). All of the variable values are stored in the final solution dictionary and can still be viewed after the solution is obtained even without providing this list.

5 Accessing Outputs

LP_Solvers is made up of multiple routines that each contribute to the overall software. After each routine is used in the background of solving a problem, the outputs are all saved in separate dictionaries that the user may choose to access based on their needs. The following sections provide the names of these dictionaries as well as a list of their contents.

5.1 Terminal Outputs

After the LP_User_Inputs.py file has been filled out, the user can run the file in the terminal or in a console associated with a graphic user interface (GUI) such as Spyder. The terminal or console will automatically output certain information the user would likely want to know including: a statement when Phase I terminates, iteration count for Phase I, time for Phase I, termination criteria for Phase I (infeasible, unbounded, etc.), iteration count for Phase II, time for Phase II, termination criteria for Phase II (convergence with optimal objective, infeasible, unbounded, etc.), and variable values.

5.2 User-Specified Settings

After the user inputs or imports a problem, the original problem and all of the chosen settings are saved in a dictionary named `user_inputs`. The keys for this dictionary can be checked at any time by typing `user_inputs.keys()` into the terminal or console. The following list provides the keys within the dictionary and their significance. In order to print the values associated with a specific key, the user should type `user_inputs['key']` into the terminal or console for the key element that they are interest in.

- `'objective'` = the sense of the objective function set by the user.
- `'c_coeff'` = the provided objective function coefficients.
- `'sensitivity'` = the toggle input for the `sensitivity` option.
- `'constraint'` = the created constraint dictionary set within the LP_User_Inputs.py file or imported from an external script.
- `'tolerance'` = the absolute tolerance value specified by the user.
- `'pointer'` = a dictionary of pointers used by the developer for generic indexing.
- `'pricing'` = the chosen pricing scheme set as the `pricing` variable in the inputs file.
- `'incoming'` = the string value input for the `incoming` variable to control Bland's rule.
- `'problem'` = which problem the user selected to solve, i.e., the primal or dual.
- `'method'` = choice of algorithm that the user specified to use, i.e., Primal or Dual Simplex.
- 'verbose' = the decision to print or not print the iteration number and objective function values while running Phase II.
- `'dec'` = the requested number of decimals to print in the terminal upon obtaining the optimal solution.

5.3 Problem in Standard Form

As previously stated, there is no need for users to formulate their problem in standard form. A routine converts all problems to a minimization with all equality constraints and stores this form in a dictionary named `conversion`. If the user would like to access the matrices or dimensions of their input problem in standard form, they can check the keys of this dictionary at any time by typing `conversion.keys()` into the terminal or console. If they would like to see the value associated with a specific key they can type `conversion['key']` to print it in the terminal. A list of keys and their significance for this dictionary are provided below.

- `'A'` = the A matrix of the user-specified LP including columns for slack and excess variables. Additionally, any rows whose RHS was negative are alternated by multiplying by -1.
- `'b'` = the b vector of RHS values from the input LP converted so that all values are nonnegative.
- `'c_coeff'` = the objective coefficients for the original LP extended to include zeros for any slack and excess variables added when converting the problem into standard form.
- `'n'` = the number of decision variables associated with the LP being solved.
- `'m'` = the number of constraints associated with the LP begin solved.
- `'n_slack'` = the number of slack and excess variables added to the LP in order to change inequality constraints to equalities.
- `'precision'` = the number of decimals associated with the precision determined from the `tolerance` input.
- `'eq_ind'` = the indices of slack or excess variables associated with equality constraints when using the Dual Simplex Method.

5.4 Phase I Solution

Once an initial feasible solution has been found, it is stored in a dictionary named `initial_solution`. This is important for users who would like to add new variables or constraints to their system. If columns are being added, it is likely that primal feasibility could be maintained and using the same initial feasible solution could provide a warm start and save computational time. Similarly, if rows are added, it is likely that dual feasibility could be maintained and using the same starting point with the Dual Simplex Method provides the ability to skip repeating Phase I. As with the other dictionaries, to access the keys the user can type `initial_solution.keys()` into the terminal. A list of these keys and their values are shown below.

- `'Basis'` = the indices of the basic variables used in the initial feasible solution.
- `'variables'` = values for all of the decision variables, both basic and nonbasic, associated with the initial feasible solution.
- `'slacks'` = values for all of the slack and excess variables, both basic and nonbasic, associated with the initial feasible solution.
- `'feasibility'` = whether ($\mathcal{P}$) was found to be feasible or not after Phase I. This key only exists when using the Primal Simplex Method since Phase I of the Dual Simplex Method does not determine whether or not ($\mathcal{P}$) is feasible.
- `'bounded'` = whether ($\mathcal{P}$) was found to be bounded or not after Phase I. This key only exists when using the Dual Simplex Method since Phase I of the Primal Simplex Method does not determine whether or not ($\mathcal{P}$) is bounded.

- `'count'` = the number of iterations that were taken in Phase I to obtain an initial feasible solution.
- `'time'` = the amount of computational time that was taken in Phase I to obtain an initial feasible solution.

5.5 Phase II Solution

Since the terminal output once optimality is reached only prints limited information, the full solution is saved in a dictionary named `solution`. The keys for this dictionary can be accessed by typing `solution.keys()` into the terminal. Values for these keys can be accessed by typing `solution['key']` into the terminal for the desired key. A list of the stored keys and their significance for this dictionary are listed below.

- `'feasibility'` = whether or not the primal problem was found to be feasible at the end of Phase II.
- `'bounded'` = whether or not the primal problem was found to be bounded at the end of Phase II.
- `'count'` = the number of iterations that were taken in Phase II to obtain the optimal solution.
- `'time'` = the amount of computational time that was taken in Phase II to obtain the optimal solution.
- `'Objective'` = the optimal objective function value, if one exists.
- `'Basis'` = the indices of the basis that provides an optimal solution, if one exists.
- `'variables'` = the values of the decision variables, both basic and nonbasic, that were found at the end of Phase II.
- `'slacks'` = the values of the slack and excess variables, both basic and nonbasic, that were found at the end of Phase II.
- `'duals'` = the duals for each constraint at optimality. These values are also equal to the decision variables of the dual problem at optimality.

Note that it the problem is either infeasible or unbounded, the values for `'Objective'`, `'Basis'`, `'variables'`, `'slacks'`, and `'duals'` are not recorded. Also, all outputs are in terms of the primal problem, even if the user explicitly solve the dual problem. The user can access the dual variable values with the `'duals'` key. Additionally, the objective function value should be the same if a finite solution existed.

5.6 RHS Sensitivity Analysis

If the user chose to run a sensitivity analysis on the RHS values of each of the constraints, they can access the output from the `rhs_sensitivity_report` dictionary. The dictionary is indexed to correspond to the user-provided constraint set. This means that in order to see the report for constraint i, the user should type `rhs_sensitivity_report[i]` into the terminal. The output for this line of code will return the constraint name, RHS value absolute range for that constraint, and the dual for the specified constraint. An example would look like:

```
rhs_sensitivity_report[1] = ('name_1_range', [2.0, 'inf'], 'dual:'  1.667)
```

This output can be interpreted as: the RHS value of constraint 1 can be changed to any value between 2.0 and positive infinity without affecting the current basis. Additionally, the dual on constraint 1 is equal to 1.667 (rounded) which means that each unit increase in the RHS value of constraint 1 will increase the objective function value by 1.667 units.

6 Algorithm Tuning

Depending on the structure of the problem being solved, tuning the algorithms presented in this software package can dramatically improve performance. The following subsections discuss some of the inefficiencies or ill conditioning that may exist in formulations and how the settings/options can be changed to impact the performance of the algorithms.

6.1 Numerical Stability

Since the computation performed in these algorithms is done using floating point numbers, there is a finite amount of precision that can be maintained. If the user supplies a formulation with ill conditioned data or selects a tolerance that is inappropriate to distinguish between their data and round-off error, an incorrect solution may be returned. The example below comes from [7] and helps to demonstrate this idea.

$$\begin{aligned} \text{min} \quad & x_1 + x_2 \\ \text{subject to} \quad & -x_1 + 24x_2 <= 21 \\ & x_1 <= 3 \\ & x_2 >= 1.00000008 \end{aligned}$$

In this formulation, constraint 3 causes this problem to be infeasible if in fact the extra 0.00000008 of the RHS is specified as meaningful by the user-input tolerance. Setting the tolerance to 1e-7 will result in a correct solution of infeasible being returned as shown in Output Log #1. If instead, the tolerance is set to 1e-6 or more, then the computation does not see the 0.00000008 as significant and an optimal solution is found and returned as shown in Output Log #2 below.

Output Log #1:

```
STOP (P1): infeasible - artificials in Basis
iterations = 2 , time = 0.02 s
```

Output Log #2:

```
End Phase 1... begin Phase 2.
iterations = 2 , time = 0.01 s

STOP (P2): c_bar values were all >= 0.
Solution:
   Optimal Objective Value: 4.0
   iterations = 0 , time = 0.0 s
   x = [3. 1.]
```

If the values for x_1 and x_2 are assigned as 3 and 1 respectively as shown from the output of Output Log #2, then it can be seen that the solution violates constraint 3. This illustrates the necessity for the user to preprocess the data they are using for their formulation to have an appropriate precision and to pick a tolerance that allows their data and calculations to be distinguished from round-off error. Another way to solve this issue for problems with data that have values much smaller than zero is to scale the problem. For example, the constraint $0.000005x_1 + 0.000003x_2 = 0.000002$ can be multiplied by 1e6 to result in an equivalent expression as $5x_1 + 3x_2 = 2$. Scaling a problem ahead of time can assist with picking a more appropriate tolerance and avoiding numerical stability issues that occur due to the finite precision used here.

6.2 Degeneracy

Certain problems may be formulated that have objective function coefficients or RHS values that are similar for multiple variables or constraints. When this occurs, the Simplex methods can often pivot variables into the basis without improving the objective function value. This phenomena is referred to as degeneracy and can cause the Simplex algorithms to take excessive iterations in order to find an optimal solution. Output Log #3 shows an example of this as pv_batt_sizing.py was solved with 'most' as the pricing scheme, 'dual' as the problem type, and 'primal' as the algorithm method. Using these particular settings, nearly 35,000 iterations are needed which result in over 10 minutes being needed to provide a solution.

Output Log #3:

```
STOP (P2): c_bar values were all >= 0.
Solution:
   Optimal Objective Value: 31387.552
   iterations =  34817  , time =  636.2 s
   x = [ 290.129 1474.208    0.    ...    0.       0.       0.   ]
```

There are multiple ways that degeneracy can be avoided if the user is aware that it may exist. One of the least effective ways is to switch the pricing scheme to Bland's rule in order to avoid cycling. Output Log #4 shows the result of doing just that. It can be seen that performance was dramatically improved by avoiding degeneracy with this different pricing scheme since the number of iterations was reduced to just under 4,100 and the time was reduced to less that 90 seconds.

Output Log #4:

```
STOP (P2): c_bar values were all >= 0.
Solution:
   Optimal Objective Value: 31387.552
   iterations =  4091  , time =  86.66 s
   x = [ 290.129 1474.208    0.    ...    0.       0.       0.   ]
```

Additional ways to avoid degeneracy are to attempt to use a different algorithm or to switch to explicitly solving the alternate problem, i.e., solving ($\mathcal{P}$) instead of ($\mathcal{D}$) if it is known that ($\mathcal{D}$) is likely to be degenerate. Output Log #5 shows that when the problem is switched to 'primal' instead of 'dual', an even larger improvement is made to the computational efficiency.

Output Log #5:

```
End Phase 1... begin Phase 2.
iterations =  1095  , time =  12.73 s

STOP (P2): c_bar values were all >= 0.
Solution:
   Optimal Objective Value: 31387.552
   iterations =  532  , time =  13.97 s
   x = [ 290.129 1474.208    0.    ...    0.       0.       0.   ]
```

Even when the Phase I iteration count and time is combined with the Phase II time, switching to solving ($\mathcal{P}$) over ($\mathcal{D}$) shows that the total number of iterations was just over 1,600 and solved in about 26 seconds. Giving the user the ability to easily switch between pricing schemes and problem structures can dramatically improve algorithm performance in many ways without the need to reformulate. It should be noted that the Phase I iteration counts and time for Output Logs #3 and #4 were neglected here because they required zero

iterations and were able to initialize in less than a second due to their problem structure. The Dual Simplex algorithm's performance on reducing degeneracy for this problem was not able to be tested due to its limited Phase I capabilities.

6.3 Excessive Iterations

Although Bland's rule was shown in the previous section to assist with degeneracy, it is often much less efficient compared to the alternate 'most' pricing scheme. There are also more complex pricing schemes that may be added later that even further reduce the number of iterations; however, these pricing schemes take more computational time per iteration so there is a balance to consider when using them. To illustrate some of the causes of excessive iterations, the pv_batt_sizing.py and pv_batt_sizing_sub.py files were used to generate the following output logs. The difference between these two script files is that the original pv_batt_sizing.py file incorporated multiple constraints that were equality expressions that only consisted of variables, e.g., $x_1 = x_2 + x_3 - x_4$. These expressions were used to reformulate the problem via substitution so that the problem size could be reduced by the number of constraints and variables used in this way. The more concise formulation was created and named pv_batt_sizing_sub.py to specify that these substitutions were made, but that the formulation remained equivalent. Before the substitutions were made, the problem size in standard form involved 2,555 constraints and 3,287 variables. Output Log #6 shows the statistics from running this original problem with the 'most' pricing scheme used on the primal problem with the Primal Simplex Method.

Output Log #6:

```
End Phase 1... begin Phase 2.
iterations = 1095 , time = 10.21 s

STOP (P2): c_bar values were all >= 0.
Solution:
   Optimal Objective Value: 31387.552
   iterations = 532 , time = 12.02 s
   x = [ 290.129 1474.208    0.    ...    0.       0.       0.   ]
```

Once the substitutions were made to the original problem, only 1,095 constraints and 1,827 variables remained. These substitutions had a large impact on the number of iterations in Phase I as well as on the time per iteration. Output Log #7 shows that reducing the problem size removed 730 iterations from Phase I of the algorithm when the same input settings were used. Although the output shows that the same number of Phase II iterations were taken, the overall time that was used in Phase II was reduced to less than half of that seen from the original formulation. It should be noted that the variable value in the third position of Output Log #7 is different from that of Output Log #6 because many variable were removed and this index no longer represented the same variable. The creators of this formulation were primarily interested in the variables that were placed in the first and second index of the variable vector. These variables were intentionally left in the formulation to show that they, as well as the objective function value, remained the same after these substitutions.

These two formulations show that the time that users take to formulate the problem can impact overall solution time and impact having excessive iterations. Most state-of-the-art solver use a presolver that reduces problem sizes for the user by checking for these types of substitutions or redundant constraints. Currently, there is not a presolver routine incorporated into this code, so it is the users responsibility to formulate their problems in a concise way if they are looking to gain efficiency in the amount of time or number of iterations taken while utilizing these algorithms.

Output Log #7:

```
End Phase 1... begin Phase 2.
iterations = 365 , time = 2.87 s

STOP (P2): c_bar values were all >= 0.
Solution:
   Optimal Objective Value: 31387.552
   iterations = 532 , time = 4.85 s
   x = [ 290.129 1474.208 4289.826 ...    0.       0.       0.   ]
```

Since both Output Logs #6 and #7 used the 'most' pricing scheme, Output Log #8 shows the pv_batt_sizing_sub.py script being run with Bland's rule to show that this less efficient pricing scheme can also cause excessive iterations. Although this problem was already made more efficient with its reformulation, a 12% increase in Phase II iterations came from changing the pricing scheme to a less effective method. These additional iterations caused nearly a 30% increase in computation time for Phase II.

Output Log #8:

```
End Phase 1... begin Phase 2.
iterations = 365 , time = 2.79 s

STOP (P2): c_bar values were all >= 0.
Solution:
   Optimal Objective Value: 31387.552
   iterations = 598 , time = 6.23 s
   x = [ 290.129 1474.208 4289.826 ...    0.       0.       0.   ]
```

One last method that can be used to reduce excessive iterations in these algorithms is to be familiar with LP structures and be aware when certain algorithms can exploited. In the PV and battery sizing problem that has been used throughout these output logs, the objective function consists of a minimization with all positive coefficients. This particular structure provides an initial dual feasible solution of all zeros without the need to take any Phase I iterations. Output Log #9 shows that when the Dual Simplex Method is used on the pv_batt_sizing_sub.py script, that the problem solves much faster than any of the other previous output logs. This is because as previously mentioned, the structure allows the initialization to be done in zero iterations and only requires 0.12 seconds. Phase II of this output shows that the number of iterations taken was slightly more than using the Primal Simplex Method; however, the total number of iterations between Output Logs #7 and #9 are much different with a total of 345 iterations being saved from using the Dual Simplex Method.

Ouput Log #9:

```
End Phase 1... begin Phase 2.
iterations = 0 , time = 0.12 s

STOP (P2): b_bar values were all >= 0.
Solution:
   Optimal Objective Value: 31387.552
   iterations = 552 , time = 2.77 s
   x = [ 290.129 1474.208 4289.826 ...    0.       0.       0.   ]
```

7 IMPORTANT Instructions

This code was organized in such a way that the user can use it without the need to edit any file other than LP_User_Inputs.py. In order to clean up the installation and reduce any accidental changes to other files, the user must store all of the files other than LP_User_Inputs.py in a folder titled "Simplex_Files" (without the quotes). The LP_User_Inputs.py file should be saved in the same directory as this folder, but not within it. If the user would like to import a script to the LP_User_Inputs.py file, they can save it in the same directory and use the line of code below replacing pv_batt_sizing_sub with the name of their python script file. The script file must at a minimum import: `objective, c_coeff,` and `constraint`. If variables were assigned names in a `v` dictionary as in these examples, that may also be imported so that the `var_names` option can be used to display specific variables in the terminal once the algorithm has terminated.

```
from pv_batt_sizing_sub import v, objective, c_coeff, constraint
```

If multiple script files are being created, it may be beneficial to create another folder to store all of them in. If this folder is in the same location as LP_User_Inputs.py then the scripts can still be imported without having to move them in an out of the folder by temporarily changing directories within python. For example, a folder titled "Problem_Scripts" (without the quotes) that contains the pv_batt_sizing_sub.py file could be accessed by using the code below. The variable `cwd` stands for current working directory and is already created at the top of the user inputs file. It is important to change into this directory to pull the appropriate script, but also to change back to the current directory after this import for the remainder of the code to function.

```
os.chdir(cwd + '/Problem_Scripts')
from pv_batt_sizing_sub import v, objective, c_coeff, constraint
os.chdir(cwd)
```

8 Conclusions

While more powerful LP optimization tools have already been developed, bringing similar algorithms to open source software languages provides more access to scientists. The package developed here has implemented sparse matrices, simple pricing schemes, two Simplex algorithms, duality theory, and controllable tolerance conditions to provide a robust starting point for scientific and engineering communities to utilize an LP solver without the need for expensive licenses.

In addition to providing the code and user instructions, this document also outlined the essentials of LPs for users that are new to the subject. The author of this code and manual is open to additional development upon request of specific features. Contact information to request these features or to ask for additional support on getting an LP to run with the software is posted below.

Name: Corey R. Randall
e-mail: coreyrandall@mines.edu

References

[1] IBM, ILOG CPLEX. Incline Village, NV, 2012.

[2] Gurobi, 2012. Gurobi Optimizer. Houston, TX.

[3] R. J. Vanderbei, *Linear Programming Foundations and Extensions*. Boston, MA: Springer US, 2001.

[4] G. B. Dantzig, *Linear Programming and Extensions: by George B. Dantzig*. Princeton, NJ: Princeton University Press, 1963.

[5] D. Goldfarb, "On the Complexity of the Simplex Method. In: Gomez S., Hennart JP. (eds) Advances in Optimization and Numerical Analysis." Mathematics and Its Applications, vol 275. Springer, Dordrecht, 1994.

[6] H. D. Sherali, J. J. Jarvis, and M. S. Bazaraa, *Linear Programming And Network Flows*. John Wiley Sons Incorporated, 2013.

[7] E. Klotz and A. M. Newman, "Practical guidelines for solving difficult linear programs," *Surveys in Operations Research and Management Science*, vol. 18, no. 1-2, pp. 1–17, 2013.