GILP

Nov 19, 2020

Contents

GILP (Geometric Interpretation of Linear Programs) is a Python package that utilizes Plotly for visualizing the geometry of linear programs (LPs) and the simplex algorithm. It was developed for the course ENGRI 1101: Engineering Applications of Operations Research at Cornell University. Furthermore, it is part of the forthcoming book by David B. Shmoys, Samuel C. Gutekunst, Frans Schalekamp, and David P. Williamson, entitled Data Science and Decision Making: An Elementary Introduction to Modeling and Optimization.

This site contains multiple tutorials as well as the full GILP *Documentation*. If you are new to linear programming and the simplex algorithm, we provide a breif *Introduction*. It is recommended to start with the *Quickstart Guide* which includes installation instructions and a tutorial. If you are interested in developing on GILP, see *Development*. Lastly, *Examples* contains mutliple example visualizations created using GILP.

CHAPTER 1

Quickstart Guide

This guide is the perfect place to begin exploring GILP. First, we walk through the installation of GILP. Next, we provide a breif introduction to linear programming and the simplex algorithm. Afterwards, we will learn how to visualize the included examples LPs and create LPs of our own! Lastly, we will explore different solver parameters that can be set when running simplex.

1.1 Installation

The quickest way to install the gilp package is with pip. Run the following line in your terminal and you should be good to go!

```
pip install gilp
```

If you are using a Google Colab enviroment, you can use gilp by running the following cell.

```
!pip install gilp
```

1.2 Introduction

Here, we provide a breif introduction to linear programming and the simplex algorithm.

1.2.1 Linear Programming

In a linear program, we have a set of decisons we need to make. We represent each decison as a **decison variable**. For example, say we run a small company that sells 2 types of widgets. We must decide how much of each each widget to produce. Let x_1 and x_2 denote the number of type 1 and type 2 widgets produced respectively.

Next, we have a set of **constraints**. Each constraint can be an inequality ($\leq, \geq$) or an equality ($=$) but **not** a strict inequality ($<, >$). Furthermore, it must consist of a *linear* combination of the decison variables. For example, let's say we have a buget of \$20. Type 1 and type 2 widgets cost \$2 and \$1 to produce respectively. This gives us our first

constraint: $2x_1 + 1x_2 \leq 20$. Furthermore, we can only store 16 widgets at a time so we can not produce more than 16 total. This yeilds $1x_1 + 1x_2 \leq 16$. Lastly, due to enviromental regulations, we can produce at most 7 type 2 widgets. Hence, our final constraint is $1x_1 + 0x_2 \leq 7$.

This leaves the final component of a linear program: the **objective function**. The objective function specifies what we wish to optimize (either minimize or maximize). Like constraints, the objective function must be a *linear* combination of the decison variables. In our example, we wish to maximize our revenue. Type 1 and type 2 widgets sell for \$5 and \$3 respectively. Hence, we wish to maximize $5x_1 + 3x_2$.

Combined, the decison variables, constraints, and objective function fully define a linear program. Often, linear programs are written in standard inequality form. Below is our example in standard inequality form.

max	$5x_1 + 3x_2$
s.t.	$2x_1 + 1x_2 \leq 20$
	$1x_1 + 1x_2 \leq 16$
	$1x_1 + 0x_2 \leq 7$
	$x_1, x_2 \geq 0$

Let us now summarize the three componets of a linear program in a general sense.

- **Decision variables** The decision variables encode each "decision" that must be made and are often denoted $x_1, \ldots, x_n$.
- **Constraints** The set of constraints limit the values of the decision variables. They can be inequalities or equalities ($\leq, \geq, =$) and must consist of a *linear* combination of the decision variables. In standard inequality form, each constraint has the form: $c_1x_1 + \cdots + c_nx_n = b$.
- **Objective Function** The objective function defines what we wish to optimize. It also must be a *linear* combination of the decision variables. In standard inequality form, the objective function has the form: $\max c_1x_1 + \cdots + c_nx_n$.

The decison variables and constraints define the **feasible region** of a linear program. The feasible region is defined as the set of all possible decisions that can feasibly be made i.e. each constraint inequality or equality holds true. In our example, we only have 2 decision variables. Hence, we can graph the feasible region with x_1 on the x-axis and x_2 on the y-axis. The area shaded blue denotes the feasible region. Any point (x_1, x_2) in this region denotes a feasible set of decisions. Each point in this region has some **objective value**. Consider the point where $x_1 = 2$ and $x_2 = 10$. This point has an objective value of $5x_1 + 3x_2 = 5(2) + 3(10) = 40$. You can move the objective slider to see all the points with some objective value. This is called an **isoprofit line**. If you slide the slider to 40, you will see that $(2, 10)$ lies on the red isoprofit line.

We wish to find the point with the maximum objective value. We can solve this graphically. We continue to increase the objective value until the isoprofit line no longer intersects with the feasible region. The point of intersection right before no point on the isoprofit line is feasible is the optimal solution! In our example, we push the objective value to 56 before the isoprofit line no longer intersects the feasible region. The only feasible point with an objective value of 56 is $(4, 12)$. We now know that $x_1 = 4$ and $x_2 = 12$ is an **optimal solution** with an **optimal value** of 56. Hence, we should produce 4 type 1 widgets and 12 type 2 widgets to maximize our revenue!

We now know what a linear program (LP) is and how LPs with 2 decision variables can be solved graphically. In the next section, we will introduce the simplex algorithm which can solve LPs of any size!

1.2.2 The Simplex Algorithm

The simplex algorithm relies on LPs being in **dictionary form**. An LP in dictionary form has the following properties:

- Every constraint is an equality constraint.
- All constants on the RHS are nonnegative.

- All variables are restricted to being nonnegative.
- Each variable appears on the left hand side (LHS) or right hand side (RHS). Not both!
- The objective function is in terms of variables on the RHS.

Let us transform our LP example from standard inequality form to dictionary form. First, we need our constraints to be equalities instead of inequalities. We have a nice trick for doing this! We can introduce another decision variable that represents the difference between the linear combination of variables and the right-hand side (RHS). Hence, the constraint $2x_1 + 1x_2 \leq 20$ becomes $2x_1 + 1x_2 + x_3 = 20$. Note that this new variable x_3 must also be nonnegative. After transforming all of our constraints, we have:

max	$5x_1 + 3x_2$
s.t.	$2x_1 + 1x_2 + x_3 = 20$
	$1x_1 + 1x_2 + x_4 = 16$
	$1x_1 + 0x_2 + x_5 = 7$
	$x_1, x_2, x_3, x_4, x_5 \geq 0$

Recall, we want each variable to appear on only one of the LHS or RHS. We consider the objective function to be on the RHS. Right now, x_1 and x_2 appear on both the LHS and RHS. To fix this, we will move them from the LHS to the RHS in each constraint. Furthermore, we want the constants on the RHS so we will do that now as well. This leaves us with:

max	$5x_1 + 3x_2$
s.t.	$x_3 = 20 - 2x_1 - 1x_2$
	$x_4 = 16 - 1x_1 - 1x_2$
	$x_5 = 7 - 1x_1 - 0x_2$
	$x_1, x_2, x_3, x_4, x_5 \geq 0$

Our LP is now in dictionary form! This is not the only way to write this LP in dictionary form. Each dictionary form for an LP has a unqiue **dictionary**. The dictionary consists of the variables that only appear on the LHS. The corresponding dictionary for the above LP is x_3, x_4, x_5. Furthermore, each dictionary has a corresponding feasible solution. This solution is obtained by setting variables on the RHS to zero. The variables on the LHS (the variables in the dictionary) are then set to the constants on the RHS. The corresponding feasible solution for the dictioary x_3, x_4, x_5 is $x_1 = 0, x_2 = 0, x_3 = 20, x_4 = 16, x_5 = 7$ or just $(0, 0, 20, 16, 7)$.

The driving idea behind the simplex algorithm is that some LPs are easier to solve that others. For example, the objective function $\max 10 - x_1 - 4x_2$ is easily maximized by setting $x_1 = 0$ and $x_2 = 0$. This is because the objective function has only negative coefficients. Simplex algebraically manipulates an LP (without changing the objective function or feasible region) in to an LP of this type.

Let us walk through an iteration of simplex on our example LP. First, we choose a variable that has a positive coefficent in the objective function. Let us choose x_1. We call x_1 our **entering variable**. In the current dictionary, $x_1 = 0$. We want x_1 to enter our dictionary so it can take a positive value and increase the objective function. To do this, we must choose a constraint where we can solve for x_1 to get x_1 on the LHS. Our constraints limit the increase of x_1 so we need to determine the **most limiting constraint**. Consider the constraint $x_3 = 20 - 2x_1 - 1x_2$. Recall, dictionary form enforces all constants on the RHS are nonnegative. Hence, $x_1 \leq 10$ since increasing x_1 by more than 10 would make the constant on the RHS negative. We can do this for every constraint to get bounds on the increase of x_1.

$x_3 = 20 - 2x_1 - 1x_2$	$x_1 \leq 10$
$x_4 = 16 - 1x_1 - 1x_2$	$x_1 \leq 16$
$x_5 = 7 - 1x_1 - 0x_2$	$x_1 \leq 7$

It follows that the most limiting constraint is $x_5 = 7 - 1x_1 - 0x_2$. We now solve for x_1 and get

max	$5x_1 + 3x_2$
s.t.	$x_3 = 20 - 2x_1 - 1x_2$
	$x_4 = 16 - 1x_1 - 1x_2$
	$x_1 = 7 - 0x_2 - 1x_5$
	$x_1, x_2, x_3, x_4, x_5 \geq 0$

Now, we must substitute $7 - 0x_2 - 1x_5$ for x_1 everywhere on the RHS and the objective function so that x_1 only appears on the LHS.

max	$5(7 - 0x_2 - 1x_5) + 3x_2$
s.t.	$x_3 = 20 - 2(7 - 0x_2 - 1x_5) - 1x_2$
	$x_4 = 16 - 1(7 - 0x_2 - 1x_5) - 1x_2$
	$x_1 = 7 - 0x_2 + 1x_5$
	$x_1, x_2, x_3, x_4, x_5 \geq 0$

max	$35 + 3x_2 - 5x_5$
s.t.	$x_3 = 6 - 1x_1 + 2x_5$
	$x_4 = 9 - 1x_1 + 1x_5$
	$x_1 = 7 - 0x_2 + 1x_5$
	$x_1, x_2, x_3, x_4, x_5 \geq 0$

The simplex iteration is now complete! The variable x_1 has entered the dictionary and x_5 has left the dictionary. We call x_5 the **leaving variable**. Our new dictionary is x_1, x_3, x_4 and the corresponding feasible solution is $x_1 = 7, x_2 = 0, x_3 = 6, x_4 = 9, x_5 = 0$ or just $(7, 0, 6, 9, 0)$. Furthermore, our objective value increased from 0 to 35!

We can continue in this fashion until there is no longer a variable with a positive coefficent in the objective function. We then have an optimal solution. Use the iteration slider below to toggle through iterations of simplex on our example. You can see the updating tableau in the top right and the path of simplex on the plot. Furthermore, you can hover over the corner points to see the feasible solution, dictionary, and objective value at that point.

In summary, in every iteration of simplex, we must

1. Choose a variable with a positive coefficient in the objective function.
2. Determine how much this variable can increase by finding the most limiting constraint.
3. Solve for the entering variable in the most limiting constraint and then substitute on the RHS such that the entering variable no longer appears on the RHS. Hence, it has entered the dictionary!

When there are no positive coefficient in the objective function, we are done!

This concludes our breif introduction to linear programming and the simplex algorithm. In the following tutorial, we will learn how one can use GILP to generate linear programming visualizations like the ones seen in this introduction.

This introduction is based on "Handout 8: Linear Programming and the Simplex Method" from Cornell's ENGRI 1101 (Fall 2017).

1.3 Tutorial

First, open up a Jupyter Notebook or Google Colab enviroment. Reminder: if you are using a Google Colab enviroment, you will have to reinstall gilp every time by running the cell `!pip install gilp`.

1.3.1 Example LPs

GILP comes with many LP examples. Before we use them, we must import them.

```
from gilp import examples as ex
```

We can now access the LP examples using `ex.NAME` where `NAME` is the name of the example LP. For example, consider:

$$\begin{array}{ll} \max & 5x_1 + 3x_2 \\ \text{s.t.} & 2x_1 + 1x_2 \leq 20 \\ & 1x_1 + 1x_2 \leq 16 \\ & 1x_1 + 0x_2 \leq 7 \\ & x_1, x_2 \geq 0 \end{array}$$

This example LP is called `ALL_INTEGER_2D_LP`. Let us assign this LP to a variable called `lp`.

```
lp = ex.ALL_INTEGER_2D_LP
```

Now, we can begin visualizing LPs. We import the visualization function below.

```
from gilp.visualize import simplex_visual
```

The function `simplex_visual()` takes an LP and returns a plotly figure. The figure can then be viewed on a Jupyter Notebook inline using

```
simplex_visual(lp).show()
```

If `.show()` is run outside a Jupyter Notebook enviroment, the visualization will open up in the browser. Alternatively, the HTML file can be written and then opened.

```
simplex_visual(lp).write_html('name.html')
```

Here is the resulting visualization from running `simplex_visual(lp).show()`

The resulting visualization has the following components.

- **Plot:** On the left, a plot shows the feasible region of the LP shaded in blue. You can hover over the corner points to see the feasible solution, dictionary, and objective value asscociated with that point.
- **Constraints:** In the middle, there is a list of constraints (not including the nonnegativity constraints). You can click on a constraint to mute it and click again to bring it back.
- **Dictionary Form LP:** The dictionary form for the current iteration of simplex is shown in the top right. If the slider is between iterations, the dictionary form for both the previous and next iteration are shown.
- **Sliders:** The iteration slider allows you to toggle through iterations of simplex. You can see the path of simplex on the plot and the updating corresponding dictionary LPs. The objective slider allows you to see the isoprofit line or plane for various objective values.

1.3.2 Defining LPs

We can also create our own LPs! First, we must import the `LP` class.

```
from gilp.simplex import LP
```

The `LP` class creates linear programs from their standard inequality form. We can represent a standard inequality form LP in terms of three matrices.

max	$c^T x$
s.t.	$Ax \leq b$
	$x \geq 0$

For example, consider the following LP in standard inequality form.

max	$1x_1 + 2x_2$
s.t.	$0x_1 + 1x_2 \leq 4$
	$1x_1 - 1x_2 \leq 2$
	$1x_1 + 0x_2 \leq 3$
	$-2x_1 + 1x_2 \leq 0$
	$x_1, x_2 \geq 0$

In this example, we have $A = \begin{bmatrix} 0 & 1 \\ 1 & -1 \\ 1 & 0 \\ -2 & 1 \end{bmatrix}$, $b = \begin{bmatrix} 4 \\ 2 \\ 3 \\ 0 \end{bmatrix}$, and $c = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$. Note $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

We will use these three matrices to create an instance of `LP`. First, we will import NumPy to create the matrices.

```
import numpy as np
```

Now, using NumPy, we create the matrices and create the `LP` instance.

```
from gilp.simplex import LP

A = np.array([[0, 1],
              [1, -1],
              [1, 0],
              [-2, 1]])
b = np.array([[4],
              [2],
              [3],
              [0]])
c = np.array([[1],
              [2]])
# Alternatively
b = np.array([4,2,3,0])
c = np.array([1,2])

lp = LP(A,b,c)
```

Now, we can visualize it like before!

```
simplex_visual(lp).show()
```

The complete code for defining the LP and visualizing it is given below.

```
1 import numpy as np
2 from gilp.simplex import LP
3 from gilp.visualize import simplex_visual
4
```

(continues on next page)

(continued from previous page)

```
A = np.array([[0, 1],
              [1, -1],
              [1, 0],
              [-2, 1]])
b = np.array([4,2,3,0])
c = np.array([1,2])
lp = LP(A,b,c)

simplex_visual(lp).show()
```

1.3.3 Solver Parameters

The `simplex_visual()` function has some optional solver parameters that can be set. These include an initial solution, iteration limit, and pivot rule. We go over each in more detail using `ex.KLEE_MINTY_3D_LP` as an example. For reference, here is the visualization of the Klee Minty Cube with no solver parameters set.

```
simplex_visual(ex.KLEE_MINTY_3D_LP).show()
```

Setting an Initial Solution

By default, the intial solution is always set at the origin. However, one can choose from any corner point to be the initial solution. For those with previous experience with LPs, the initial solution must be a *basic feasible solution*. An initial solution is set as follows:

```
simplex_visual(lp, initial_solution=x).show()
```

where `x` is a NumPy vector representing the initial solution. Above, you can see the default initial feasible solution is the origin. Let us try setting a different initial solution.

```
x = np.array([[0],[25],[25]])
simplex_visual(ex.KLEE_MINTY_3D_LP, initial_solution=x).show()
```

Iteration Limits

By default, the simplex algorithm will run simplex iterations until an optimal solution is found. Alternatively, an iteration limit can be set:

```
simplex_visual(lp, iteration_limit=1).show()
```

where `l` is an integer iteration limit. Above, you can see it takes 5 simplex iterations to reach the optimal solution. Let's set the iteration limit to be 3.

```
simplex_visual(ex.KLEE_MINTY_3D_LP, iteration_limit=3).show()
```

Setting a Pivot Rule

Be default, the simplex algorithm uses Bland's pivot. In addition to Bland's rule, three other pivot rules are implemented. In an iteration of simplex, the leaving variable is always the minimum (positive) ratio (minimum index to tie break) regardless of the chosen pivot rule. Of the eligible entering variables (those with positive coefficients in the objective function), each pivot rule determines the entering variable as follows:

- **Bland's Rule** (reference as `bland` or `min_index`) Minimum index.
- **Dantzig's Rule** (reference as `dantzig` or `max_reduced_cost`) Most positive reduced cost.
- **Greatest Ascent** (reference as `greatest_ascent`) Most positive (minimum ratio) x (reduced cost).
- **Manual Select** (reference as `manual_select`) Selected by user.

A desired pivot rule is specified as follows.

```
simplex_visual(lp, rule=r).show()
```

where `r` is a string representing the chosen rule. Let us try some other pivot rules on `ex.KLEE_MINTY_3D_LP`!

```
simplex_visual(ex.KLEE_MINTY_3D_LP, rule='dantzig').show()
```

```
simplex_visual(ex.KLEE_MINTY_3D_LP, rule='greatest_ascent').show()
```

```
simplex_visual(ex.KLEE_MINTY_3D_LP, rule='manual_select').show()
```

For this visualization, the chosen entering variables were 2,3, and then 5.

This concludes the quickstart tutorial! See the *Development* section for information on developing for GILP.

CHAPTER 2

Development

This guide is aimed towards those who have prior knowledge of linear programming and the simplex algorithm and would like to add on to the current functionality of GILP. First, we will walkthough an editable installation of gilp. The walkthough will include setting up a Python virtual enviroment for testing and running gilp. After that, we will provide an extensive overview of the gilp package and the modules it contains.

2.1 Installation

To develop and run tests on gilp, first download the source code in the desired directory.

```
git clone https://github.com/henryrobbins/gilp
```

Next, cd into the gilp directory and create a Python virtual enviroment called `env_name`

```
cd gilp
python -m venv env_name
```

Activate the virtual enviroment.

```
source env_name/bin/activate
```

Run the following in the virtual enviroment. The `-e` flag lets you make adjustments to the source code and see changes without re-installing. The `[dev]` installs necessary dependencies for developing and testing.

```
pip install -e .[dev]
```

To run tests and see coverage, run the following in the virtual enviroment.

```
coverage run -m pytest
coverage report --include=gilp/*
```

Next, we will provide an extensive overview of the gilp package and the contained modules.

2.2 Package Overview

This package overview serves as a source of necessary and helpful information for developing gilp. First, we will discuss the structure of the gilp package at a high level.

2.2.1 Package Structure

The gilp package contains 4 modules: `simplex`, `style`, `visualize`, and `examples`. The `simplex` module contains the `LP` class definition as well as an implementation of the revised simplex method. Additionally, it contains some custom exception classes that can be thrown by the `LP` methods and simplex functions. The `style` module mainly serves as a higher level interface with the Plotly Graphing Library. Furthermore, it contains some additional functions for styling text and numbers. The height, width, and background color of the generated visualizations are set with constants in this module. The `visualize` drives most of the gilp package. This module utilizes the `simplex` and `style` modules to generate interactive visualizations. Additionally, it contains a custom exception class and constants which specify properties of the visualization. Lastly, the `examples` module contains 8 example LPs in the form of 8 constants. Now, we will go into each module in more detail.

Simplex Module

The main components of the

Style Module

Visualize Module

Examples Module

CHAPTER 3

Examples

3.1 All Integer 2D LP

A 2D LP where all basic feasible solutions are integral and have integral tableaus.

3.2 Limiting Constraints 2D LP

A 2D LP demonstrating how the most limiting constraint determines the leaving variable.

3.3 Degenerate Fin 2D LP

A 2D LP where the (default) intial feasible solution is degenerate.

3.4 Klee Minty 2D LP

A 2D LP where the 'dantzig' pivot rule results in a simplex path through every bfs. Klee, Victor; Minty, George J. (1972). "How good is the simplex algorithm?"

3.5 All Integer 3D LP

A 3D LP where all basic feasible solutions are integral and have integral tableaus.

3.6 Multiple Optimal Solutions 3D LP

A 3D LP demonstrating the geometry of multiple optimal solutions.

3.7 Square Pyramid 3D LP

A 3D LP which is highly degenerate. It demonstrates that degeneracy can not be solved by removing a seemingly redundant constraint–doing so can alter the feasible region.

3.8 Klee Minty 3D LP

A 3D LP where the 'dantzig' pivot rule results in a simplex path through every bfs. Klee, Victor; Minty, George J. (1972). "How good is the simplex algorithm?"

CHAPTER 4

Documentation

4.1 gilp package

4.1.1 gilp.examples module

gilp.examples.**ALL_INTEGER_2D_LP = <gilp.simplex.LP object>**
A 2D LP where all basic feasible solutions are integral and have integral tableaus.

gilp.examples.**ALL_INTEGER_3D_LP = <gilp.simplex.LP object>**
A 3D LP where all basic feasible solutions are integral and have integral tableaus.

gilp.examples.**DEGENERATE_FIN_2D_LP = <gilp.simplex.LP object>**
A 2D LP where the (default) intial feasible solution is degenerate.

gilp.examples.**KLEE_MINTY_2D_LP = <gilp.simplex.LP object>**
A 2D LP where the 'dantzig' pivot rule results in a simplex path through every bfs. Klee, Victor; Minty, George J. (1972). "How good is the simplex algorithm?"

gilp.examples.**KLEE_MINTY_3D_LP = <gilp.simplex.LP object>**
A 3D LP where the 'dantzig' pivot rule results in a simplex path through every bfs. Klee, Victor; Minty, George J. (1972). "How good is the simplex algorithm?"

gilp.examples.**LIMITING_CONSTRAINT_2D_LP = <gilp.simplex.LP object>**
A 2D LP demonstrating how the most limiting constraint determines the leaving variable.

gilp.examples.**MULTIPLE_OPTIMAL_3D_LP = <gilp.simplex.LP object>**
A 3D LP demonstrating the geometry of multiple optimal solutions.

gilp.examples.**SQUARE_PYRAMID_3D_LP = <gilp.simplex.LP object>**
A 3D LP which is highly degenerate. It demonstrates that degeneracy can not be solved by removing a seemingly redundant constraint–doing so can alter the feasible region.

4.1.2 gilp.simplex module

exception gilp.simplex.**InfeasibleBasicSolution**
Bases: Exception

Raised when a list of indices forms a valid basis but the corresponding basic solution is infeasible.

exception gilp.simplex.**InvalidBasis**
Bases: Exception

Raised when a list of indices does not form a valid basis and prevents further correct execution of the function.

class gilp.simplex.**LP**(*A: numpy.ndarray*, *b: numpy.ndarray*, *c: numpy.ndarray*)
Bases: object

Maintains the coefficents and size of a linear program (LP).

The LP class maintains the coefficents of a linear program in both standard inequality and equality form. A is an m*n matrix describing the linear combination of variables making up the LHS of each constraint. b is a nonnegative vector of length m making up the RHS of each constraint. Lastly, c is a vector of length n describing the objective function to be maximized. Both the n decision variables and m slack variables must be nonnegative. Under these assumptions, the LP must be feasible.

```
inequality          equality
max c^Tx            max c^Tx
s.t Ax <= b         s.t Ax + Is == b
    x >= 0                 x,s >= 0
```

n
number of decision variables (excluding slack variables).

Type int

m
number of constraints (excluding nonnegativity constraints).

Type int

A
An m*n matrix of coefficients.

Type np.ndarray

A_I
An m*(n+m) matrix of coefficients: [A I].

Type np.ndarray

b
A nonnegative vector of coefficients of length m.

Type np.ndarray

c
A vector of coefficients of length n.

Type np.ndarray

c_0
A vector of coefficients of length n+m: [c^T 0^T]^T.

Type np.ndarray

get_basic_feasible_sol(*B: List[int]*) → numpy.ndarray
Return the basic feasible solution corresponding to this basis.

By definition, B is a basis iff A_B is invertible (where A is the matrix of coefficents in standard equality form). The corresponding basic solution x satisfies A_Bx = b. By definition, x is a basic feasible solution iff x satisfies both A_Bx = b and x > 0.

Parameters **B** (*List[int]*) – A list of indices in {0..n+m-1} forming a basis.

Returns Basic feasible solution corresponding to the basis B.

Return type np.ndarray

Raises

- *InvalidBasis* – B
- *InfeasibleBasicSolution* – x_B

get_basic_feasible_solns() → Tuple[List[numpy.ndarray], List[List[int]], List[float]]
Return all basic feasible solutions, their basis, and objective value.

Returns

- List[np.ndarray]: The list of basic feasible solutions for this LP.
- List[List[int]]: The corresponding list of bases.
- List[float]: The corresponding list of objective values.

Return type Tuple

get_equality_form()
Returns n,m,A_I,b,c_0 describing this LP in standard equality form

get_inequality_form()
Returns n,m,A,b,c describing this LP in standard inequality form.

get_tableau(*B: List[int]*) → numpy.ndarray
Return the tableau corresponding to the basis B for this LP.

The returned tableau has the following form:

```
z - (c_N^T - y^TA_N)x_N = y^Tb  where   y^T = c_B^TA_B^(-1)
x_B + A_B^(-1)A_Nx_N = x_B^*    where   x_B^* = A_B^(-1)b
```

Parameters **B** (*List[int]*) – A valid basis for this LP

Returns A numpy array representing the tableau

Return type np.ndarray

Raises *InvalidBasis* – Invalid basis. A_B is not invertible.

exception gilp.simplex.**UnboundedLinearProgram**
Bases: Exception

Raised when an LP is found to be unbounded during an execution of the revised simplex method

gilp.simplex.**invertible**(*A: numpy.ndarray*) → bool
Return true if the matrix A is invertible.

By definition, a matrix A is invertible iff n = m and A has rank n

Parameters **A** (*np.ndarray*) – An m*n matrix

Returns True if the matrix A is invertible. False otherwise.

Return type bool

gilp.simplex.**simplex**(*lp: gilp.simplex.LP*, *pivot_rule: str = 'bland'*, *initial_solution: numpy.ndarray = None*, *iteration_limit: int = None*) → Tuple[List[numpy.ndarray], List[List[int]], float, bool]

Execute the revised simplex method on the given LP.

Execute the revised simplex method on the given LP using the specified pivot rule. If a valid initial basic feasible solution is given, use it as the initial bfs. Otherwise, ignore it. If an iteration limit is given, terminate if the specified limit is reached. Output the current solution and indicate the solution may not be optimal.

PIVOT RULES

Entering variable:

- 'bland' or 'min_index': minimum index
- 'dantzig' or 'max_reduced_cost': most positive reduced cost
- 'greatest_ascent': most positive (minimum ratio) x (reduced cost)
- 'manual_select': user selects among possible entering indices

Leaving variable:

- (All): minimum (positive) ratio (minimum index to tie break)

Parameters

- **lp** (*LP*) – LP on which to run simplex
- **pivot_rule** (*str*) – Pivot rule to be used. 'bland' by default.
- **initial_solution** (*np.ndarray*) – Initial bfs. None by default.
- **iteration_limit** (*int*) – Simplex iteration limit. None by default.

Returns

- List[np.ndarray]: Basic feasible solutions at each simplex iteration.
- List[List[int]]: Corresponding bases at each simplex iteration.
- float: The current objective value.
- bool: True if the current objective value is known to be optimal.

Return type Tuple

Raises

- ValueError – Invalid pivot rule. Select from (list).
- ValueError – Iteration limit must be strictly positive.
- ValueError – initial_solution should have shape (n,1) but was ().

gilp.simplex.**simplex_iteration**(*lp: gilp.simplex.LP, x: numpy.ndarray, B: List[int], pivot_rule: str = 'bland'*) → Tuple[numpy.ndarray, List[int], float, bool]

Execute a single iteration of the revised simplex method.

Let x be the initial basic feasible solution with corresponding basis B. Do one iteration of the revised simplex method using the given pivot rule. Implemented pivot rules include:

Entering variable:

- 'bland' or 'min_index': minimum index
- 'dantzig' or 'max_reduced_cost': most positive reduced cost
- 'greatest_ascent': most positive (minimum ratio) x (reduced cost)
- 'manual_select': user selects among possible entering indices

Leaving variable:

- (All): minimum (positive) ratio (minimum index to tie break)

Parameters

- **lp** (LP) – LP on which the simplex iteration is being done.
- **x** (*np.ndarray*) – Initial basic feasible solution.
- **B** (*List(int)*) – Basis corresponding to basic feasible solution x.
- **pivot_rule** (*str*) – Pivot rule to be used. 'bland' by default.

Returns

- np.ndarray: New basic feasible solution.
- List[int]: Basis corresponding to the new basic feasible solution.
- float: Objective value of the new basic feasible solution.
- bool: An idication of optimality. True if optimal. False otherwise.

Return type Tuple

Raises

- ValueError – Invalid pivot rule. Select from (list).
- ValueError – x should have shape (n+m,1) but was ().

4.1.3 gilp.style module

gilp.style.**BACKGROUND_COLOR = 'white'**
The background color of the figure

gilp.style.**FIG_HEIGHT = 500**
The height of the entire visualization figure.

gilp.style.**FIG_WIDTH = 950**
The width of the entire visualization figure.

gilp.style.**LEGEND_NORMALIZED_X_COORD = 0.39473684210526316**
The normalized x coordinate of the legend (relative to right side).

gilp.style.**LEGEND_WIDTH = 200**
The width of the legend section of the figure.

gilp.style.**TABLEAU_NORMALIZED_X_COORD = 0.6052631578947368**
The normalized x coordinate of the tableau (relative to right side).

gilp.style.**equation**(*fig: plotly.graph_objs._figure.Figure*, *A: numpy.ndarray*, *b: float*, *style: str*, *lb: str = None*) → Union[plotly.graph_objs._scatter.Scatter, plotly.graph_objs._scatter3d.Scatter3d]
Return a styled 2d or 3d trace representing the given equation.

gilp.style.**equation_string**(*A: numpy.ndarray*, *b: float*, *comp: str = ' '*) → str
Return the string representation of an equation.

The equation is assumed to be in standard form: Ax 'comp' b.

gilp.style.**format**(*num: Union[int, float]*, *precision: int = 3*) → str
Return a properly formated string for a number at some precision.

gilp.style.**get_axis_limits**(*fig: plotly.graph_objs._figure.Figure*, *n: int*) → List[float]
Return the axis limits for the given figure.

gilp.style.**intersection**(*A: numpy.ndarray*, *b: float*, *D: numpy.ndarray*, *e: float*) → List[numpy.ndarray]
Return the points where Ax = b intersects Dx <= e.

gilp.style.**label**(*dic: Dict[str, Union[float, list]]*) → str
Return a styled string representation of the given dictionary.

gilp.style.**line**(*x_list: List[numpy.ndarray]*, *style: str*, *lb: str = None*, *i=[0]*) → plotly.graph_objs._scatter.Scatter
Return a 2d line trace in the desired style.

gilp.style.**linear_string**(*A: numpy.ndarray, indices: List[int], constant: float = None*) → str
Return the string representation of a linear combination.

gilp.style.**order**(*x_list: List[numpy.ndarray]*) → List[List[float]]
Return an ordered list of points for drawing a 2d or 3d polygon.

gilp.style.**polygon**(*x_list: List[numpy.ndarray]*, *style: str*, *lb: str = None*) → plotly.graph_objs._scatter.Scatter
Return a styled 2d or 3d polygon trace defined by some points.

gilp.style.**scatter**(*x_list: List[numpy.ndarray]*, *style: str*, *lbs: List[str] = None*) → plotly.graph_objs._scatter.Scatter
Return a styled 2d or 3d scatter trace for given points and labels.

gilp.style.**set_axis_limits**(*fig: plotly.graph_objs._figure.Figure, x_list: List[numpy.ndarray]*)
Set the axes limits of fig such that all points in x are visible.

Given a set of nonnegative 2 or 3 dimensional points, set the axes limits such all points are visible within the plot window.

gilp.style.**table**(*header: List[str], content: List[str], style: str*) → plotly.graph_objs._table.Table
Return a styled table trace with given headers and content.

gilp.style.**vector**(*tail: numpy.ndarray*, *head: numpy.ndarray*) → Union[plotly.graph_objs._scatter.Scatter, plotly.graph_objs._scatter3d.Scatter3d]
Return a styled 2d or 3d vector trace from tail to head.

4.1.4 gilp.visualize module

gilp.visualize.**ISOPROFIT_STEPS = 25**
The number of isoprofit lines or plane to render.

gilp.visualize.**ITERATION_STEPS = 2**
The number of steps each iteration is divided in to.

exception gilp.visualize.**InfiniteFeasibleRegion**
Bases: Exception

Raised when an LP is found to have an infinite feasible region and can not be accurately displayed.

gilp.visualize.**add_isoprofits**(*fig: plotly.graph_objs._figure.Figure*, *lp: gilp.simplex.LP*) → Tuple[List[int], List[float]]

Add the set of isoprofit lines/planes which can be toggled over.

Parameters

- **fig** (*plt.Figure*) – Figure to which isoprofits lines/planes are added
- **lp** (LP) – LP for which the isoprofit lines are being generated

Returns

- List[int]: Indices of all isoprofit lines/planes
- List[float]): The corresponding objective values

Return type Tuple

gilp.visualize.**add_path**(*fig: plotly.graph_objs._figure.Figure, path: List[numpy.ndarray]*) → List[int]

Add vectors for visualizing the simplex path. Return vector indices.

gilp.visualize.**add_tableaus**(*fig: plotly.graph_objs._figure.Figure, lp: gilp.simplex.LP, bases: List[int], tableau_form: str = 'dictionary'*) → List[int]

Add the set of tableaus. Return the indices of each table trace.

gilp.visualize.**get_tableau_strings**(*lp: gilp.simplex.LP, B: List[int], iteration: int, form: str*) → Tuple[List[str], List[str]]

Get the string representation of the tableau for the LP and basis B.

The tableau can be in canonical or dictionary form:

```
Canonical:                                  Dictionary:
-------------------------------------       (i)
| (i) z | x_1 | x_2 | ... | x_n | RHS |
=====================================       max           ... + x_N
|   1   |  -  |  -  | ... |  -  |  -  |     s.t.   x_i = ... + x_N
|   0   |  -  |  -  | ... |  -  |  -  |            x_j = ... + x_N
              ...                                     ...
|   0   |  -  |  -  | ... |  -  |  -  |            x_k = ... + x_N
-------------------------------------
```

gilp.visualize.**isoprofit_slider**(*isoprofit_IDs: List[int], objectives: List[float], fig: plotly.graph_objs._figure.Figure, n: int*) → plotly.graph_objs.layout._slider.Slider

Create a plotly slider to toggle between isoprofit lines / planes.

Parameters

- **isoprofit_IDs** (*List[int]*) – IDs of every isoprofit trace.
- **objectives** (*List[float]*) – Objective values for every isoprofit trace.
- **fig** (*plt.Figure*) – The figure containing the isoprofit traces.
- **n** (*int*) – The dimension of the LP the figure visualizes.

Returns A plotly slider that can be added to a figure.

Return type plt.layout.SLider

gilp.visualize.**iteration_slider**(*path_IDs: List[int], table_IDs: List[int], fig: plotly.graph_objs._figure.Figure, n: int*) → plotly.graph_objs.layout._slider.Slider

Create a plotly slider to toggle between iterations of simplex

Parameters

- **path_IDs** (*List[int]*) – IDs of every simplex path trace.
- **table_IDs** (*List[int]*) – IDs of every table trace.
- **fig** (*plt.Figure*) – The figure containing the traces.
- **n** (*int*) – The dimension of the LP the figure visualizes.

Returns A plotly slider that can be added to a figure.

Return type plt.layout.Slider

gilp.visualize.**lp_visual**(*lp: gilp.simplex.LP*) → plotly.graph_objs._figure.Figure
Render a plotly figure visualizing the geometry of an LP.

gilp.visualize.**plot_lp**(*lp: gilp.simplex.LP*) → plotly.graph_objs._figure.Figure
Return a figure visualizing the feasible region of the given LP.

Assumes the LP has 2 or 3 decision variables. Each axis corresponds to a single decision variable. The visualization plots each basic feasible solution (with their basis and objective value), the feasible region, and each of the constraints.

Parameters **lp** (LP) – An LP to visualize.

Returns A figure containing the visualization.

Return type fig (plt.Figure)

Raises

- *InfiniteFeasibleRegion* – Can not visualize.
- ValueError – Can only visualize 2 or 3 dimensional LPs.

gilp.visualize.**set_up_figure**(*n: int*) → plotly.graph_objs._figure.Figure
Return a figure for an n dimensional LP visualization.

gilp.visualize.**simplex_visual**(*lp: gilp.simplex.LP*, *tableau_form: str = 'dictionary'*, *rule: str = 'bland'*, *initial_solution: numpy.ndarray = None*, *iteration_limit: int = None*) → plotly.graph_objs._figure.Figure
Render a figure showing the geometry of simplex.

Parameters

- **lp** (LP) – LP on which to run simplex
- **tableau_form** (*str*) – Displayed tableau form. Default is 'dictionary'
- **rule** (*str*) – Pivot rule to be used. Default is 'bland'
- **initial_solution** (*np.ndarray*) – An initial solution. Default is None.
- **iteration_limit** (*int*) – A limit on simplex iterations. Default is None.

Returns A plotly figure which shows the geometry of simplex.

Return type plt.Figure

Python Module Index

g

Index

N

O

P

S

T

U

V