Casio Decision Mathematics Suite

# Analysis

## Problem Identification

Myself and my friends in our Further Mathematics class have encountered a problem with the Decision Mathematics modules. It is difficult to check if the answers we have given to questions are correct as the questions are complicated and the algorithms in use have multiple steps involved. It is easy to make a mistake while going through the steps of these algorithms, and it causes some confusion as to what the correct answer is. This is where it would be useful to have applications available to us in the lesson that can solve these problems for us. Most of us in the class have available to us a Casio FX-CG50 graphical calculator, which can have graphical programs written for it using the SDK.

The project aim is to create a suite of applications to solve all the problems available in the Edexcel Further Mathematics Decision module. These applications can be installed onto the Casio calculators and used as any other program on the calculator would be.

## Stakeholders

The stakeholders in this project would be myself and my classmates. Each of us would be capable of checking, testing and evaluating the individual applications and I would be able to frequently get new feedback from them.

Every one of us has a use for it as an academic tool – it allows us to write our own questions for revision and also mark them ourselves.

## Research

### Simplex

It is possible to solve simplex problems on the Casio FX-CG50 manually, using a matrix and matrix row operations. However, this is not an automated solution as wanted, and so is not exactly suited to the stakeholder’s requirements.

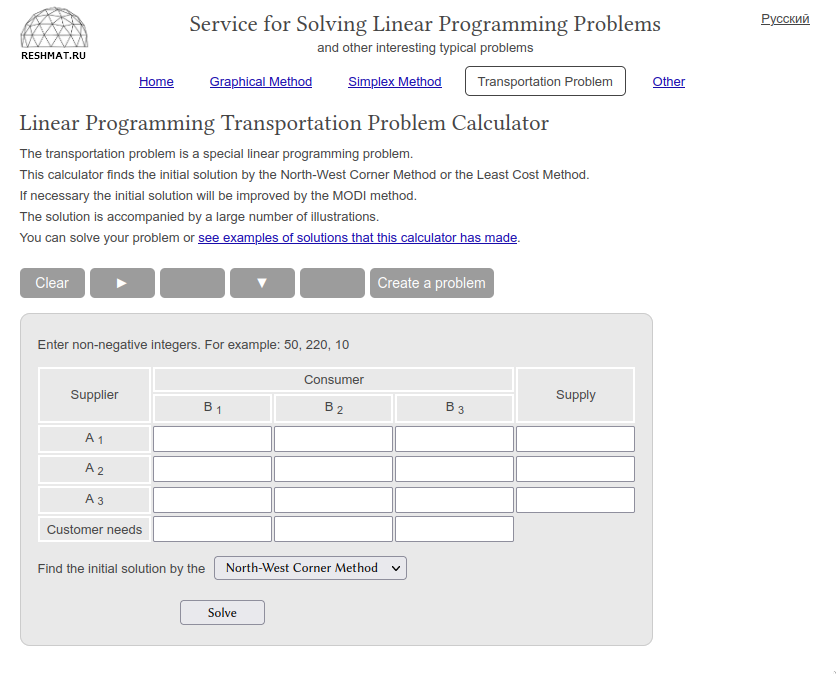
However, it would be useful to be able to enter a simplex tableau manually rather than have it automatically generated at all times, so this is definitely something I can take away as an idea.

### Transportation Problems

Transportation problems are a special case of linear programming problems, which makes them solvable by the simplex method, but it is generally more computationally efficient to solve them by different means.

I have been unable to find an implementation of the Stepping Stone algorithm (the method used to solve these problems) for Casio calculators. Additionally, it is not possible to use any of the features already available on the calculator to solve them manually.

Thusly the only available examples are online solvers such as this one:

(taken from <http://reshmat.ru/transportation_problem_lp.html>)

I will use a similar design for the program on the calculators – this is how the tables are laid out in the questions too, so it will be very easy to transfer the information to the calculator from the questions.

[remove]

### Allocation Problems

Allocations problems are problems where there are an equal number of workers and tasks to be completed. Every worker has a number associated with every task (representing time to complete, cost, etc.) and the goal is to find a solution that uses the most optimal selection of workers for each task.

An example of this could be selection for athletes in a multi-stage swimming relay. Each athlete will swim different stokes and will take different times to complete laps in particular strokes. The table representing this problem could look as such:

| Swimmer / Stroke | Breaststroke | Front Crawl | Backstroke | Butterfly |
| --- | --- | --- | --- | --- |
| Mark | 21 | 35 | 32 | 30 |
| Nicky | 34 | 20 | 36 | 18 |
| Nigel | 25 | 38 | 15 | 40 |
| Susie | 39 | 18 | 29 | 27 |

I have also been unable to find any implementation to solve allocation problems for the Casio calculators. Like Transportation Problems, it is possible to solve allocation problems as a linear programming problem, although the set-up for the tableau is prohibitively repetitive, and it is more efficient to solve the problems by the algorithms designed exactly for them.

Because of this there is very little information I can gather about existing solutions, and so I will need to develop my own implementation without any guidance. I will likely use a similar set-up as for the Transportation Problem solver.

[/remove]

### Research Conclusions

Throughout the whole of my research I did not find any examples of a whole suite of applications like the one I am suggesting. I will be taking inspiration from the tools I found. However, the table layouts used in the tools are already what is standard and what is taught in the book, so not much new information can be garnered from these tools.

## Essential Features

The desired solution will be a suite of applications, including an application to solve Transportation Problems, Allocation Problems and Linear Programming problems. The users will be able to input the problem information, and the solution will be automatically generated, with the steps to attain such a solution shown if required.

### Talking with Stakeholders

I sent my stakeholders, Toby, Jess and Guy the following message:

“The core idea of the project is to provide us with applications to use on our calculators to solve a variety fo the Decision problems, i.e. Transportation Problems, Linear Programming and Allocation Problems. For these problems you’d be able to enter all the information in matrix forma like it’s given to you, and it would generate a solution for you, optionally showing the steps taken to attain the solution.”

Toby told me he was happy with the idea currently, and wanted to ensure that:

* There was “**outputs at all stages and highlighted changes in/ a different colour**”,
* And a **“usable ui”**

I asked for more detail about what a “usable ui” is:

It must be “obvious as to where numbers need to go, how to put them there and stuff”

Jess said:

“could you make it so that you could enter the same problem and see all the methods of getting the optimal solution? Also could you make it so that you could start at any stage of the problem? ”

## Limitations

The limitations of my solution will stem largely from the hardware they are running on. The Casio FX-CG50 has only 61 kB of RAM available, and so the calculator would be unable to solve large problems, such as those that would be encountered in industry. This, however, will not come up to be an issue, as the solution developed for the calculators would not be used in an industry circumstance. The second limitation would be the way that data is entered into the solvers. Because the screen of the calculator is relatively small, it can only display a small amount of information on it at any one time. This small screen is less suited to entering larger amounts of data for larger problems. Thankfully, as this solution is only intended for use by students studying Further Mathematics Decision, the problems entered into the calculator would be restricted in their size.

## Requirements

### Hardware Requirements

A Casio FX-CG50 calculator, or similar calculator from the same line, will be required to run the applications, as well as the appropriate cables to install the software on the calculator. The cables usually come provided with the calculator. Additionally, a computer will be required to download the applications to get them onto the calculator, or use a calculator that already has the software installed and a link cable. It is preferable to install them from a computer as it is much faster than using the link cable between two calculators.

### Software Requirements

There are no additional software requirements, the calculators come with everything required to run the applications. It is recommeded, for testing or evaluation of the software, that anyone not in possesion of a Casio Graphics Calculator installs the Casio Calculator emulator: <https://education.casio.co.uk/download-emulator-cg>

## Success Criteria

|  |  |
| --- | --- |
| Criteria | Evidence |
| Application to solve LP problems | Picture of application installed on a calculator, pictures showing the application working |
| Application to solve Transportation Problems | Picture of application installed on a caluclator, pictures showing the application working |
| Changes highlighted in solution steps | Picture of step before and step after, showing the changes that have been highlighted |
| In all applications, able to start at any part of the solution process | Pictures showing each available part in the process to start at, for each application |

# Design

## Common Matrix Design

Many of the applications will share a common matrix input screen, and so this can be developed as a modular solution and applied to each of the applications through importing an external utilities file.

## Separated Applications

It was considered to produce a single application that performs every different task required by the stakeholders, but I have decided against it, favouring a separated application suite. This will allow users to install whichever application(s) they require, rather than needing to fill the calculator’s limited storage space with an application they will not utilize fully.

## Overall Structure

To start with, only three applications will be developed; this is for multiple reasons. The primary reason being time constraint. I do not wish to be over-ambitious with the number of applications that can be developed in the time given, so I am restricting myself to three, before I can expand to possibly others. The second reason is for future planning: by developing only these three applications to start with, I can focus on the tools required to produce the programs effectively. This way I will be better able to consider the best way to develop more applications in the future, as well as will have more of my own tools/workflows available to me, having developed a few already.

The programs will be:

* Simplex Solver
* Transportation Problem Solver

These applications will share a common matrix design, which will accelerate application development to begin with.

### Simplex Solver

Upon opening the application, the user will immediately be asked how many variables the problem they are solving uses. After inputing their answer (which will likely be capped, an exact cap can be worked out later in development once the space complexity of the solution is attained). Then the user will then be asked if they are maximizing or minimizing the objective function. Once they have answered the previous, will be whether the problem uses >= constraints. If the problem does not, then the user will be asked to fill in their tableau at a matrix input screen. Else, the user will be asked if they would like to use Two Stage Simplex or Big M Method to solve the problem. The difference between these two methods will be detailed later in the section.

The Simplex algorithm is as follows (from Edexcel AS and A level Further Mathematics – Decision Mathematics 1):

1) Look along the objective row for the most negative entry: this indicates the **pivot column**.

2) Calculate the θ values, for each of the constraint rows, where θ = (the term in the value column) ÷ (the term in the pivot column)

3) Select the row with the smallest, positive θ value to become the pivot row.

4) The element in the pivow row and pivot column is the pivot.

5) Divide the row found in step 3 by the pivot, and change the basic variable at the start of the row to the variable at the top of the pivot column. This is now the pivot row.

6) Use the pivot row to eliminate the pivot’s variable from the other rows. This means that the pivot column now contains one 1 and zeros.

7) Repeat steps 3 to 8 until there are no more negative numbers in the objective row.

8) The tableau is now optimal, and the non-zero values can be read off using the basic variable column and value column

If the user wanted to minimize the objective function, rather than maximize, there is no difference in the algorithm itself. Instead, define a new objective function that is the negative of the original objective function. After the new objective function has been maximized, write the solution as the negative of this value, which will minimize the original objective function.

Two Stage Simplex method

The Two Stage Simplex method is one method used to solve linear programming problems containing constraints in the form a1x1 + a2x2 + … + anxn ≥ K

It is detailed as followed:

1) Use slack, surplus, and artificial variables, as necessary, to write all the contraints as equations.

2) Define a new objective function to minimize the sum of all the artificial variables.

3) Use the simplex methodto solve this problem

4) If the minimum sum of the artificial values is 0 then the solution found is a basic feasible solution of the original problem, which is then the starting point for the second stage. Use the simplex method again to solve this problem.

5) If the minimum sum of the artificial variables is not 0 then the original problem has no feasible solution.

The Big-M method

The Big-M method uses a conceptual number – M, which is arbitrarily large, to modifiy the objective function to also contain the artificial variables. By subtracting M x (sum of artificial variables) from the objective function, it becomes necessary for the algorithm to elminiate all artificial variables to optimize the tableau. The algorithm itself is the same as the simplex method, only using M for some values.

Once the user has decided what method they want to use to solve the problem, they will be taken to a matrix input screen, where they can fill in their tableau. Once the user has filled the tableau, they can press a function button on the calculator to run the algorithm. There will be two options, one to run through step-by-step, showing changes each time, and another to run all the way through and only show the end result.

### Transportation Problem Solver

When the user opens the application they will first be asked the number of supply and demand points in their solution. After filling both answers, the user may continue to fill in the cost matrix. They will be taken to a matrix input screen with dimensions corresponding to how many supply and demand points there are.

Once the user has input all cell costs, they can select one of two buttons – one to solve, showing each step, and another to run all the way through and only show the solution.

Once a button is selected, it must first be checked that the solution is **balanced**. This means that both the sums of supply and demand are equal. If they are not then a **dummy point** will need to be added to the matrix. If the demand is less than the supply, a **dummy demand point** will be added, equal to the difference between the supply and demand, making supply and demand now equal. Conversely, if supply is less than demand, then a **dummy supply point** is added, again with value equal to the difference. The cost of every cell for a demand point is 0.

Once any disagreements in supply and demand have been rectified, the full algorithm can run its course. In fact, there are multiple different algorithms/methods used in the process of solving a transportation problem, and each of them is named below, in order.

* North-West Corner method
* Finding shadow costs
* Finding improvement indices
* Stepping-Stone method

North-West Corner method

The North-West Corner method is used to find an initial solution to the problem. It generates a transport matrix that satisfies the conditions, but does not generate an optimal solution. It goes as follows:

1. Begin with the top left-hand corner. Allocate the maximum available quantity to meet the demand at this destination (whilst not exceeding the stock at this source).
2. As each stock is emptied, move one square down and allocate as many units as possilbe from the next source until the demand of the destination is met. As each demand is met, move the square to the right and again allocate as many units as possible.
3. When all the stock is assigned, and all the demands are met, stop.

Finding shadow costs

Shadow costs describe the “cost” of transporting a unit to/from a supply/demand point, agnostic of destination/source. They are required to find **improvement indices**. They are calculated as follows:

1. Start with the north-west corner, and set the cost linked with its source to zero.
2. Move along the row to any other non-empty squares. Set the cost linked with these destinations equal to the total transportation cost for that route (since the source cost for the first row is 0)
3. When all possible destination costs for that row have been established, go to the start of the next row.
4. Move along this row to any non-empty squares and use the destination costs found earlier, to establish the source cost for the row. Once that has been done, find any further unknown destination costs.
5. Repeat steps 3 and 4 until all source and destination costs have been found.

Finding improvement indices

Improvement indices show what cells not currently being used to transport stock are the cheapest. By identifying which cells are least costly, they can be introduced into the solution, eliminating those which have higher cost. Finding improvement indices is much simpler, and does not require an algorithm as such, rather a per-cell calculation:

Improvement IndexPQ = CostPQ – Source CostP – Destination CostQ

The cell with the most negative improvement index becomes the **entering cell** in the stepping-stone method. **If all improvement indices are zero or positive, the solution is optimal.**

Stepping-Stone method

The stepping-stone method is the “optimize away” step. It introduces the entering-cell into the solution, and in doing so reduces the use of other, more expensive, cells. It is done like so:

1. Create the cycle of adjustments. The two basic rules are:
   1. Within any row and any column there can only be one increasing cell and one decreasing cell.
   2. Apart from the entering cell, adjustments are only made to non-empty cells.
2. Once the cycle of adjustments has been found, you transfer the maximum number of units throuhg this cycle. This will be equal to the smallest number in the decreasing cells.
3. You then adjust the solution to incorporate this improvement.

Doing an iteration of the stepping stone method does not guarantee an optimal solution, only an improvement (unless the solution is already optimal). The whole process may need to be repeated many times to generate an optimal solution.

### [Remove this?]

### Allocation Problems

When the user first opens the application, they will immediately be asked to provide the dimensions of the problem: how many workers are available, and how many tasks need to be completed. If the two are not equal – as they must be to attain a solution (allocation problems only allow one worker to one task, as they are all completed simultaneously), then a dummy row or collumn will be added automatically. The user will then immediately be taken to a matrix input screen, with any dummy rows or collumns already filled with zeros, grayed out and unable to take input. Once the user has filled all cells they may continue to solve the problem. Again, there will be two options, one to solve showing steps and changes, and another to go straight through and

## [/Remove this?]

## Required Data Structures and Classes

It will be necessary to develop a table/matrix class that can perform row and column operations, as well as hold both tuples and numbers in its cells. I plan to represent numbers such as (1 + M) or (3 – 5M) as tuples like so: (1, M()) and (3, M(-5)).

It will also be necessary to create an M data type, which will need a value attribute, for representations greater than one M. In all cases, when compared with a normal number, M will be larger. If it is to be compared with another M, it will be necessary to look at the values of both to determine which is larger.

It will also be useful to have a generic matrix/table display class that can be used to display an n×m matrix. As many of the applications will use this kind of input, this is necessary to avoid constantly rewriting the same thing. The display class will take a matrix class as an input, and use it as the data to be displayed.

## Usability Features

All applications will have the capability to take the user a step back by pressing the Exit key on the calculator. This will prevent the user having their time wasted if they accidentally enter wrong data, as they will not have to go through the whole process if they enter wrong data accidentally, and can instead just go back.

## Test Data

It will be very important to consider how much memory and CPU the applications will use. For this reason I will be collecting detailed profiling information for every application. When it comes to testing the individual applications, I will be using valid and extreme valid, as the way I plan on writing the applications they will not accept invalid data. This will also need testing in some way, so I will attempt to enter invalid data at every stage. For valid testing I will be using questions from the textbooks provided, as these come with solutions, and for extreme valid I will be creating problems that have no solution.

# Development of the Coded Solution

## Stage 1: Library Development

The first part of development should be to write the classes and data types needed for the rest of development, and to review them before writing them into the full solution. This way it can be ensured that they are as efficient as possible. I will begin by writing the matrix class. I am going to prototype everything in Python first, as development in Python is quicker to iterate. Once I consider the prototype working, I will then translate all the code I have to C.

First, ensuring that any data passed into the class is validated. Running this code, which passes too much data to the class, correctly raises an AttributeError:

**class** **Matrix**:

**def** \_\_init\_\_(self, dims: (int, int), data: [float]):

**if** len(data)/dims[0] != dims[1]:

**raise** **AttributeError**(

f"Dimensions **{**dims**}** do not match data len (**{**len(data)**}**)")

self.dims = dims

self.data = data

A = Matrix((2, 2), [1, 2, 3, 4, 5])

Traceback (most recent call last):

File "/home/leet/Documents/GitHub/NEA-Project/src/python-prototyping/lib.py", line 10, **in** <module>

A = Matrix((2, 2), [1, 0, 1, 0, 1])

File "/home/leet/Documents/GitHub/NEA-Project/src/python-prototyping/lib.py", line 4, **in** \_\_init\_\_

**raise** **AttributeError**(

**AttributeError**: Dimensions (2, 2) do **not** match data len (5)

Next, I implement a \_\_getitem\_\_() function so that our matrices can be indexed. Indexing a matrix will return a row list, which can be indexed by default.

*# …*

*# ...*

**def** \_\_getitem\_\_(self, index: int):

start\_index = self.dims[1]\*index

end\_index = self.dims[1]\*index+self.dims[0]

**return** self.data[start\_index:end\_index]

*# This represents a matrix like such:*

*# | 1 3 |*

*# | 2 4 |*

A = Matrix((2, 2), [1, 2, 3, 4])

*# Should print [1, 2]*

print(A[0], A[1])

This code outputs:

[1, 2] [3, 4]

This is correct, and testing it with a larger, non-square matrix (dimensions 5x3):

A = Matrix((5, 3), [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15])

*# Should print [1, 2, 3] [13, 14, 15]*

print(A[0], A[4])

Outputs:

[1, 2, 3, 4, 5] [13, 14, 15]

This is wrong, and I believe that this is due to the way I find the end index. Changing the code for finding the end index to:

end\_index = self.dims[1]\*(index + 1)

works exactly as intended, and the code now prints what it should:

[1, 2, 3] [13, 14, 15]

Next, I would like to be able to add or subtract numbers from rows and columns. There is no need to implement two functions for adding and subtracting, as I can simply add a negative number. This will reduce code size, which is important in these applications. In order to be able to change the

values in rows and columns, I need a \_\_setitem\_\_ function.

*# Sets the column at `index` to `new\_col`*

**def** \_\_setitem\_\_(self, index: int, new\_col: [float]):

*# The value by which all changes are shifted*

shift = index\*self.dims[1]

**for** col\_index, item **in** enumerate(new\_col):

self.data[shift+col\_index] = new\_col[col\_index]

Now, trying to change the first column in a 2x2 matrix:

*# This represents a matrix like such:*

*# | 1 3 |*

*# | 2 4 |*

A = Matrix((2, 2), [1, 2, 3, 4])

A[0] = [5, 6]

*# Should print [5, 6] [3, 4]*

print(A[0], A[1])

Prints:

[5, 6] [3, 4]

I forgot to implement any kind of validation, so have now implemented it. The \_\_setitem\_\_ function now looks like this:

*# Sets the column at `index` to `new\_col`*

**def** \_\_setitem\_\_(self, index: int, new\_col: [float]):

*# If new\_col is wrong length, raise*

**if** len(new\_col) != self.dims[1]:

**raise** **ValueError**(

f"Error assigning new value to matrix column. New value has length**\**

**{**len(new\_col)**}** but matrix has height **{**self.dims[1]**}**")

*# The value by which all changes are shifted*

shift = index\*self.dims[1]

*# Iterate through the new column, and change*

*# appropriate cells to new values*

**for** col\_index, item **in** enumerate(new\_col):

self.data[shift+col\_index] = new\_col[col\_index]

If we try to set an inappropriate length column to the matrix:

A = Matrix((2, 2), [1, 2, 3, 4])

A[0] = [5, 6, 7]

The program correctly errors:

Traceback (most recent call last):

File "/home/leet/Documents/GitHub/NEA-Project/src/python-prototyping/lib.py", line 41, **in** <module>

A[0] = [5, 6, 7]

File "/home/leet/Documents/GitHub/NEA-Project/src/python-prototyping/lib.py", line 21, **in** \_\_setitem\_\_

**raise** **ValueError**(

**ValueError**: Error assigning new value to matrix column. New value has length 3 but matrix has height 2

I also need to be able to change individual cells this way, and reflecting on this, I have decided to change both get and set functions accordingly. The new functions look like this:

*# Returns the value at index `index`*

**def** \_\_getitem\_\_(self, index: (int, int)):

item\_index = self.dims[1]\*index[0] + index[1]

**return** self.data[item\_index]

*# Sets the value at `index` to `val`*

**def** \_\_setitem\_\_(self, index: (int, int), val: float):

*# If val is wrong type, error*

**if** **not** isinstance(val, numbers.Number):

**raise** **ValueError**(

f"Error assigning new value to matrix cell. New value has type**\**

**{**type(val)**}**, which is not a number type.")

item\_index = self.dims[1]\*index[0] + index[1]

self.data[item\_index] = val

I have needed to import the “numbers” module, which is part of the Python standard library. This is needed to validate that the new value is a number type. The following code:

Outputs, correctly:

2

7

Now I can continue to write my rowadd and coladd functions.

*# Add `num` to every cell in row `row\_index`*

def rowadd(self, row\_index: int, num: float):

**for** i **in** **range**(self.dims[0]):

self.**data**[self.dims[1]\*i + row\_index] += num

def coladd(self, col\_index: int, num: float):

**for** j **in** **range**(self.dims[1]):

self.**data**[self.dims[1]\*col\_index + j] += num

Testing with the following code:

A = Matrix((2, 2), [1, 2, 3, 4])

*# Should print [1, 3]*

print([A[0, 0], A[1, 0]])

*# Changes 1 to 11 and 3 to 13*

A.rowadd(0, 10)

*# Should print [11, 13]*

print([A[0, 0], A[1, 0]])

*# Should print [13, 4]*

print([A[1, 0], A[1, 1]])

*# Changes 13 to 23 and 4 to 14*

A.coladd(1, 10)

*# Should print [23, 14]*

print([A[1, 0], A[1, 1]])

Outputs, correctly:

[1, 3]

[11, 13]

[13, 4]

[23, 14]

Now that I can add to columns and rows, I would also like to be able to multiply across them. Implementing very similar functions is very easy.

*# Multiply every cell in row `row\_index` by `num`*

**def** rowmul(self, row\_index: int, num: float):

**for** i **in** range(self.dims[0]):

self.data[self.dims[1]\*i + row\_index] \*= num

*# Multiply every cell in column `col\_index` by `num`*

**def** colmul(self, col\_index: int, num: float):

**for** j **in** range(self.dims[1]):

self.data[self.dims[1]\*col\_index + j] \*= num

Tested with the following code:

A = Matrix((2, 2), [1, 2, 3, 4])

*# Should print [1, 3]*

print([A[0, 0], A[1, 0]])

*# Changes 1 to 10 and 3 to 30*

A.rowmul(0, 10)

*# Should print [10, 30]*

print([A[0, 0], A[1, 0]])

*# Should print [30, 4]*

print([A[1, 0], A[1, 1]])

*# Changes 30 to 15 and 4 to 2*

A.colmul(1, 0.5)

*# Should print [15, 2]*

print([A[1, 0], A[1, 1]])

Outputs, correctly:

[1, 3]

[10, 30]

[30, 4]

[15.0, 2.0]

At this point I consider the Matrix data type fully developed as I need it currently. I may find myself adding to it later in development, but this is all the functionality I need for the Simplex Solver. The final code for the entire class is such:

**class** **Matrix**:

**def** \_\_init\_\_(self, dims: (int, int), data: [float]):

*# If the amount of data provided does*

*# not fit within the dimensions, error*

**if** len(data)/dims[0] != dims[1]:

**raise** **ValueError**(

f"Dimensions **{**dims**}** do not match data len (**{**len(data)**}**)")

self.dims = dims

self.data = data

*# Returns the value at index `index`*

**def** \_\_getitem\_\_(self, index: (int, int)):

item\_index = self.dims[1]\*index[0] + index[1]

**return** self.data[item\_index]

*# Sets the value at `index` to `val`*

**def** \_\_setitem\_\_(self, index: (int, int), val: float):

*# If val is wrong type, error*

**if** **not** isinstance(val, numbers.Number):

**raise** **ValueError**(

f"Error assigning new value to matrix cell. New value has type**\**

**{**type(val)**}**, which is not a number type.")

item\_index = self.dims[1]\*index[0] + index[1]

self.data[item\_index] = val

*# Add `num` to every cell in row `row\_index`*

**def** rowadd(self, row\_index: int, num: float):

**for** i **in** range(self.dims[0]):

self.data[self.dims[1]\*i + row\_index] += num

*# Add `num` to every cell in column `col\_index`*

**def** coladd(self, col\_index: int, num: float):

**for** j **in** range(self.dims[1]):

self.data[self.dims[1]\*col\_index + j] += num

*# Multiply every cell in row `row\_index` by `num`*

**def** rowmul(self, row\_index: int, num: float):

**for** i **in** range(self.dims[0]):

self.data[self.dims[1]\*i + row\_index] \*= num

*# Multiply every cell in column `col\_index` by `num`*

**def** colmul(self, col\_index: int, num: float):

**for** j **in** range(self.dims[1]):

self.data[self.dims[1]\*col\_index + j] \*= num

The next class I need is a way to represent Big M. I will write my own M class which will have a regular number value and an M value.

**class** **M**(numbers.Number):

*# M class is a number with two values.*

*# One regular number value, and another Big M value.*

**def** \_\_init\_\_(self, num\_val, m\_val):

self.num\_val = num\_val

self.m\_val = m\_val

**def** \_\_repr\_\_(self):

**if** self.m\_val < 0:

**return** f"(**{**self.num\_val**}** - **{**abs(self.m\_val)**}**M)"

**return** f"(**{**self.num\_val**}** + **{**self.m\_val**}**M)"

I have defined the core of the class. I can now create an instance of M, which I can also print using its \_\_repr\_\_.

num = M(-1, 3)

*# Should print (-1 + 3M)*

print(num)

Outputs:

(-1 + 3M)

Now that I can instantiate and print an M class, I want to be able to perform comparisons with it.

**def** \_\_gt\_\_(self, other):

**if** isinstance(other, M):

**return** self.m\_val > other.m\_val

**return** **True**

**def** \_\_lt\_\_(self, other):

**if** isinstance(other, M):

**return** self.m\_val < other.m\_val

**return** **False**

**def** \_\_gte\_\_(self, other):

**if** isinstance(other, M):

**return** self.m\_val >= other.m\_val

**return** **True**

**def** \_\_lte\_\_(self, other):

**if** isinstance(other, M):

**return** self.m\_val <= other.m\_val

**return** **False**

**def** \_\_eq\_\_(self, other):

**if** isinstance(other, M):

**return** self.m\_val == other.m\_val **and** self.num\_val == other.num\_val

**return** **False**

This is the code I had originally written to perform comparisons with, although after some consideration I have realized it is faulty. When performing comparisons, for all of them except \_\_eq\_\_, I have neglected to consider the non M part. This leads to incorrect results:

num1 = M(10, 4)

num2 = M(0, 4)

*# Should print True*

print(num1 > num2)

Outputs:

**False**

I have corrected the code for comparisons, and the new code looks like this:

**def** \_\_gt\_\_(self, other):

**if** isinstance(other, M):

**return** (self.m\_val > other.m\_val) **or** (self.m\_val == other.m\_val **and**

self.num\_val > other.num\_val)

**return** **True**

**def** \_\_lt\_\_(self, other):

**if** isinstance(other, M):

**return** (self.m\_val > other.m\_val) **or** (self.m\_val == other.m\_val **and**

self.num\_val > other.num\_val)

**return** **False**

**def** \_\_gte\_\_(self, other):

**if** isinstance(other, M):

**return** (self.m\_val >= other.m\_val) **or** (self.m\_val == other.m\_val **and**

self.num\_val >= other.num\_val)

**return** **True**

**def** \_\_lte\_\_(self, other):

**if** isinstance(other, M):

**return** (self.m\_val <= other.m\_val) **or** (self.m\_val == other.m\_val **and**

self.num\_val <= other.num\_val)

**return** **False**

**def** \_\_eq\_\_(self, other):

**if** isinstance(other, M):

**return** self.m\_val == other.m\_val **and** self.num\_val == other.num\_val

**return** **False**

Using the same test code for comparisons, the output is now, correctly:

**True**

Now that I have working code for comparisons, I would like to be able to perform arithmetic operations on M types. This will make it easier to add, subtract multiply etc.

**def** \_\_add\_\_(self, other):

**if** isinstance(other, M):

self.m\_val += other.m\_val

self.num\_val += other.num\_val

**return** self

self.num\_val += other

**return** self

**def** \_\_sub\_\_(self, other):

**if** isinstance(other, M):

self.m\_val -= other.m\_val

self.num\_val -= other.num\_val

**return** self

self.num\_val -= other

**return** self

**def** \_\_mul\_\_(self, other):

**if** isinstance(other, M):

self.m\_val \*= other.m\_val

self.num\_val \*= other.num\_val

**return** self

self.num\_val \*= other

**return** self

**def** \_\_div\_\_(self, other):

**if** isinstance(other, M):

self.m\_val \*= other.m\_val

self.num\_val \*= other.num\_val

**return** self

self.num\_val \*= other

**return** self

Now that I’ve written the arithmetic code, testing it:

num1 = M(10, 4)

num2 = M(0, 4)

*# Should print (10, 8M)*

print(num1 + num2)

Output:

(10 + 8M)

### Review

Reviewing the Matrix class, I have noticed that the way the index is calculated is in fact wrong. I also wanted to be able to use negative indices to access the end of rows and columns. The new modified code below generates the correct index from the row and column numbers:

index\_0, index\_1 = index

**if** index\_0 < 0:

index\_0 = self.dims[0]+index\_0

**if** index\_1 < 0:

index\_1 = self.dims[1]+index\_1

item\_index = index\_0 + self.dims[0]\*index\_1

I have also implemented a \_\_repr\_\_ function so that I can print out a whole matrix, which makes developing and debugging easier. In addition I have modified all matrix functions to use the matrix indexing rather than directly modifying self.data.

**def** \_\_repr\_\_(self):

output = ""

**for** j **in** range(self.dims[1]):

**for** i **in** range(self.dims[0]):

output += str(self[i, j])

output += ' '

output += "**\n**"

**return** output

Another change I would like to make is to the matrix class. I would like to be able to select individual rows and columns from the matrix and to be able to add rows and columns together. I have implemented this below.

**def** row(self, row\_index):

**return** MatrixRow([self[i, row\_index] **for** i **in** range(self.dims[0])])

**def** column(self, col\_index):

**return** MatrixCol([self[col\_index, j] **for** j **in** range(self.dims[1])])

*# Add `num` to every cell in row `row\_index`*

**def** rowadd(self, row\_index: int, other: float):

**for** i **in** range(self.dims[0]):

**if** isinstance(other, Number):

self[i, row\_index] += other

**elif** isinstance(other, list):

self[i, row\_index] += other[i]

**else**:

**raise** **ValueError**(f"Type of other is not number or list: **{**type(other)**}**")

*# Add `num` to every cell in column `col\_index`*

**def** coladd(self, col\_index: int, other: float):

**for** j **in** range(self.dims[1]):

**if** isinstance(other, Number):

self[col\_index, j] += other

**elif** isinsance(other, list):

self[col\_index, j] += other[j]

**else**:

**raise** **ValueError**(f"Type of other is not number or list: **{**type(other)**}**")

I have used custom MatrixRow and MatrixCol types to achieve this. They are defined as such.

**class** **MatrixRow**(list):

**def** \_\_getitem\_\_(self, key):

**return** super(MatrixRow, self).\_\_getitem\_\_(key)

**def** \_\_setitem\_\_(self, key):

**return** super(MatrixRow, self).\_\_setitem\_\_(key)

**def** \_\_mul\_\_(self, other):

**for** i **in** range(len(self)):

self[i] \*= other

**class** **MatrixCol**(list):

**def** \_\_getitem\_\_(self, key):

**return** super(MatrixCol, self).\_\_getitem\_\_(key)

**def** \_\_setitem\_\_(self, key):

**return** super(MatrixCol, self).\_\_setitem\_\_(key)

**def** \_\_mul\_\_(self, other):

**for** i **in** range(len(self)):

self[i] \*= other

## Stage 2: Simplex Solver

In a new file, I have begun writing the code for the simplex solver. The first step is to find the pivot column.

**def** pivot\_column(matrix):

objective\_row = [matrix[i, matrix.dims[1]-1] **for** i **in** range(matrix.dims[0])]

**return** objective\_row.index(min(objective\_row))

Testing this with the following code:

*# x y z r s t Value*

data = [-0.25, 0, -0.5, 1, 0, -0.75, 8, *# r*

2.5, 0, 2, 0, 1, -0.5, 92, *# s*

0.75, 1, 0.5, 0, 0, 0.25, 24, *# y*

3, 0, -6, 0, 0, 5, 480] *# P*

mat = Matrix((len(column\_names), len(row\_names)), data)

*# Should print 2*

print(pivot\_column(mat))

Outputs:

2

Next, I need to find the pivot row and thereby the pivot cell.

**def** pivot\_row(matrix, column):

thetas = []

indices = []

**for** j **in** range(matrix.dims[1]-1):

theta = matrix[-1, j] / matrix[column, j]

**if** theta >= 0:

thetas.append(theta)

indices.append(j)

index = indices[0]

theta = thetas[0]

**for** ind, thet **in** zip(indices, thetas):

**if** thet < theta:

theta = thet

index = ind

**return** index

Testing with this code:

*# x y z r s t Value Theta*

data = [-0.25, 0, -0.5, 1, 0, -0.75, 8, *# r -16*

2.5, 0, 2, 0, 1, -0.5, 92, *# s46*

0.75, 1, 0.5, 0, 0, 0.25, 24, *# y 48*

3, 0, -6, 0, 0, 5, 480] *# P*

mat = Matrix((len(column\_names), len(row\_names)), data)

column = pivot\_column(mat)

row = pivot\_row(mat, column)

pivot\_cell = (column, row)

*# Should print (2, 1)*

print(pivot\_cell)

Outputs:

(2, 1)

Next is step 5 and 6 of the algorithm. I need to divide the pivot row by the value in the pivot cell, and then subtract multiples of the pivot row from the other rows, such that the pivot column only contains one 1.

**def** simplex(matrix):

column = pivot\_column(matrix)

row = pivot\_row(matrix, column)

matrix.rowmul(row, 1/matrix[column, row])

**return** matrix

Testing this with the same data matrix:

mat = Matrix((len(column\_names), len(row\_names)), data)

*# Should show that (2, 1) has become 1*

print(simplex(mat))

Outputs:

-0.25 0 -0.5 1 0 -0.75 8

1.25 0.0 1.0 0.0 0.5 -0.25 46.0

0.75 1 0.5 0 0 0.25 24

3 0 -6 0 0 5 480

This is correct, but the output is extremely difficult to read without significant effort, so I have decided that I am going to rewrite the \_\_repr\_\_ for the matrix class to pad between each value, so that all values in the same column are lined up. The function now looks like this:

**def** \_\_repr\_\_(self):

output = ""

length = [0 **for** i **in** range(self.dims[0])]

**for** i **in** range(self.dims[0]):

**for** j **in** range(self.dims[1]):

**if** len(str(self[i, j])) > length[i]:

length[i] = len(str(self[i, j]))

**for** j **in** range(self.dims[1]):

**for** i **in** range(self.dims[0]):

output += f" **{**' '\*(length[i]-len(str(self[i, j])))**}{**self[i, j]**}** "

output += "**\n**"

**return** output

And the output is much cleaner:

-0.25 0 -0.5 1 0 -0.75 8  
 1.25 0.0 1.0 0.0 0.5 -0.25 46.0  
 0.75 1 0.5 0 0 0.25 24  
 3 0 -6 0 0 5 480

The next step is to reduce the rest of the columns, such that the pivot column contains a single 1, in the pivot cell.

**def** simplex(matrix):

column = pivot\_column(matrix)

row = pivot\_row(matrix, column)

matrix.rowmul(row, 1/matrix[column, row])

**for** i **in** range(matrix.dims[1]):

**if** i == row:

**continue**

row\_modifier = matrix.row(row)\*(-matrix[column, i])

matrix.rowadd(i, row\_modifier)

**return** matrix

Testing, now with a smaller matrix from the textbook:

*# x y r s Value*

data = [ 5, 7, 1, 0, 70, *# r*

10, 3, 0, 1, 60, *# s*

-3, -2, 0, 0, 0] *# P*

mat = Matrix((len(column\_names), len(row\_names)), data)

print(mat)

*# Should show that the 0th column has one 1 and the*

*# rest are zeros*

print(simplex(mat))

Outputs:

5 7 1 0 70

10 3 0 1 60

-3 -2 0 0 0

0.0 5.5 1.0 -0.5 40.0

1.0 0.30000000000000004 0.0 0.1 6.0

0.0 -1.0999999999999999 0.0 0.30000000000000004 18.0

While this output is correct: the zeroth column is how it should be, I have encountered an issue with floating point arithmetic. Floating point arithmetic is prone to inaccuracies, for example -1.0999999999999999 is meant to represent 1.1. This issue can be solved with part of Python's standard library, the "fractions" module. The fractions module allows you to represent any rational number as a fraction without losing precision. In order to implement the fractions module I will need to make some changes to my library.

I have modified the \_\_init\_\_ function for the Matrix class, it now converts all values to fractions during initialisation.

self.data = [Fraction(num) **for** num **in** data]

This now works without problem, and I have regained the accuracy I require.

5 7 1 0 70

10 3 0 1 60

-3 -2 0 0 0

0 11/2 1 -1/2 40

1 3/10 0 1/10 6

0 -11/10 0 3/10 18

Now that I can perform one iteration on the matrix, I want to be able to check if the solution is optimal. This is done by seeing if the most negative entry in the objective row is negative. If it is not, the solution is optimal.

**def** isoptimal(matrix, objective\_row=-1):

**return** min(matrix.row(objective\_row)) >= 0

The reasoning behind the objective\_row parameter is that the objective row changes during the course of the Two Stage Simplex algorithm. I modified the simplex function so that it will run until the solution is optimal.

**def** simplex(matrix):

**while** **not** isoptimal(matrix):

column = pivot\_column(matrix)

row = pivot\_row(matrix, column)

matrix.rowmul(row, 1/matrix[column, row])

**for** i **in** range(matrix.dims[1]):

**if** i == row:

**continue**

row\_modifier = matrix.row(row)\*(-matrix[column, i])

matrix.rowadd(i, row\_modifier)

**return** matrix

Testing, the function now returns the optimal solution that is written in the textbook.

# x y r s Value

data = [ 5, 7, 1, 0, 70, # r

10, 3, 0, 1, 60, # s

-3, -2, 0, 0, 0] # P

mat = Matrix((len(column\_names), len(row\_names)), data)

print(simplex(mat))

Outputs:

0 1 2/11 -1/11 80/11

1 0 -3/55 7/55 42/11

0 0 1/5 1/5 26

It is important that for the solution to be useful, we can see the row names and column names of the matrix. I am going to implement a DisplayMatrix class for this. This class will use the same \_\_repr\_\_, \_\_getitem\_\_ and \_\_setitem\_\_ as the Matrix class, but will not have any of the other functionality available. The important distinction is that the DisplayMatrix class takes a string as its values.

**class** **DisplayMatrix**():

**def** \_\_init\_\_(self, matrix, col\_names, row\_names):

self.dims = (matrix.dims[0]+1, matrix.dims[1]+1)

self.data = [" "]

self.data.extend(col\_names)

**for** i **in** range(len(row\_names)):

row = matrix.row(i)

row.insert(0, row\_names[i])

self.data.extend(row)

**for** i **in** range(len(self.data)):

self.data[i] = str(self.data[i])

*# Returns the value at index `index`*

**def** \_\_getitem\_\_(self, index: (int, int)):

index\_0, index\_1 = index

**if** index\_0 < 0:

index\_0 = self.dims[0]+index\_0

**if** index\_1 < 0:

index\_1 = self.dims[1]+index\_1

item\_index = index\_0 + self.dims[0]\*index\_1

**if** item\_index > self.dims[0]\*self.dims[1]:

**raise** **IndexError**(f"Index [**{**index[0]**}**, **{**index[1]**}**] out of range")

**return** self.data[item\_index]

*# Sets the value at `index` to `val`*

**def** \_\_setitem\_\_(self, index: (int, int), val: str):

index\_0, index\_1 = index

**if** index\_0 < 0:

index\_0 = self.dims[0]+index\_0

**if** index\_1 < 0:

index\_1 = self.dims[1]+index\_1

item\_index = index\_0 + self.dims[0]\*index\_1

**if** item\_index > self.dims[0]\*self.dims[1]:

**raise** **IndexError**(f"Index [**{**index[0]**}**, **{**index[1]**}**] out of range")

self.data[item\_index] = val

**def** \_\_repr\_\_(self):

output = ""

length = [0 **for** i **in** range(self.dims[0])]

**for** i **in** range(self.dims[0]):

**for** j **in** range(self.dims[1]):

**if** len(str(self[i, j])) > length[i]:

length[i] = len(self[i, j])

**for** j **in** range(self.dims[1]):

**for** i **in** range(self.dims[0]):

output += f" **{**' '\*(length[i]-len(self[i, j]))**}{**self[i, j]**}** "

output += "**\n**"

**return** output

Next, I have modified the simplex function so that it performs the necessary changes to the row names.

*# ...*

matrix.rowmul(row, 1/matrix[column, row])

row\_names[row] = column\_names[column]

*# ...*

Now I can output the matrix, including the row and column names.

column\_names = ["x", "y", "r", "s", "Value"]

row\_names = ["r", "s", "P"]

# x y r s Value

data = [ 5, 7, 1, 0, 70, # r

10, 3, 0, 1, 60, # s

-3, -2, 0, 0, 0] # P

mat = Matrix((len(column\_names), len(row\_names)), data)

mat, column\_names, row\_names = simplex(mat, column\_names, row\_names)

print(DisplayMatrix(mat, column\_names, row\_names))

Outputs:

x y r s Value

y 0 1 2/11 -1/11 80/11

x 1 0 -3/55 7/55 42/11

P 0 0 1/5 1/5 26

This now is much more useful. Now I can read the solution and say that the optimal solution, for 26 profit, requires 80/11 y and 42/11 x.

I want to implement Toby's requests now, specifically to be able to show the changes of each iteration.

I have updated the display matrix \_\_init\_\_ so that the pivot cell is indicated by being surrounded with \*. In the final application this will be highlighting, but a more rudimentary approach here is better to just get it implemented.

**class** **DisplayMatrix**():

**def** \_\_init\_\_(self, matrix, col\_names, row\_names, changes=(-1, -1)):

*# ...*

*# ...*

**if** sum(changes) >= 0:

self[changes[0]+1, changes[1]+1] = f"\***{**self[changes[0]+1, changes[1]+1]**}**\*"

I've also updated the simplex function. It is now a generator, so it must be called as such.

**def** simplex(matrix, column\_names, row\_names, show\_changes=**False**):

**while** **not** isoptimal(matrix):

column = pivot\_column(matrix)

row = pivot\_row(matrix, column)

matrix.rowmul(row, 1/matrix[column, row])

row\_names[row] = column\_names[column]

**for** i **in** range(matrix.dims[1]):

**if** i == row:

**continue**

row\_modifier = matrix.row(row)\*(-matrix[column, i])

matrix.rowadd(i, row\_modifier)

**if** show\_changes:

**yield** matrix, row\_names, (column, row)

Running with slightly different code:

mat = Matrix((len(column\_names), len(row\_names)), data)

**for** mat, row\_names, pivot\_cell **in** simplex(mat,

column\_names,

row\_names,

show\_changes=**True**):

print(DisplayMatrix(mat,

column\_names,

row\_names,

changes=pivot\_cell)

)

Outputs:

x y r s Value

r 0 11/2 1 -1/2 40

x \*1\* 3/10 0 1/10 6

P 0 -11/10 0 3/10 18

x y r s Value

y 0 \*1\* 2/11 -1/11 80/11

x 1 0 -3/55 7/55 42/11

P 0 0 1/5 1/5 26

The next step is Two Stage Simplex. This is actually a fairly simplex addition. I do need to make some changes to the simplex function and the way I find the pivot column first, however. I need to ensure that the pivot column is not chosen from either of the objective rows in the Two Stage case, as the matrix for two stage contains two objective rows.

**def** pivot\_row(matrix, column, two\_stage=**False**):

thetas = []

indices = []

rows = [j **for** j **in** range(matrix.dims[1] - 1)]

**if** two\_stage:

rows = [j **for** j **in** range(matrix.dims[1] - 2)]

**for** j **in** rows:

**if** matrix[column, j] == 0:

theta = inf

**else**:

theta = matrix[-1, j] / matrix[column, j]

**if** theta >= 0:

thetas.append(theta)

indices.append(j)

index = indices[0]

theta = thetas[0]

**for** ind, thet **in** zip(indices, thetas):

**if** thet < theta:

theta = thet

index = ind

**return** index

I have added an optional `two\_stage` flag, which if enabled changes the rows checked to no longer include the extra objective row. Now I just need to add this flag to the simplex function, as other than that the algorithm is not affected.

**def** simplex(matrix,

column\_names,

row\_names,

objective\_row=-1,

two\_stage=**False**,

show\_changes=**False**):

**while** **not** isoptimal(matrix, objective\_row=objective\_row):

column = pivot\_column(matrix, objective\_row=objective\_row)

row = pivot\_row(matrix, column, two\_stage=two\_stage)

matrix.rowmul(row, 1 / matrix[column, row])

row\_names[row] = column\_names[column]

**for** i **in** range(matrix.dims[1]):

**if** i == row:

**continue**

row\_modifier = matrix.row(row) \* (-matrix[column, i])

matrix.rowadd(i, row\_modifier)

**if** show\_changes:

**yield** matrix, row\_names, (column, row)

**if** **not** show\_changes:

**yield** matrix, row\_names, (-1, -1)

Now that this is done, I can test it:

column\_names = ["x", "y", "z", "s1", "s2", "s3", "a1", "a2", "Value"]

row\_names = ["s1", "s2", "a1", "P", "I"]

*# x y z s1 s2 s3 a1 a2 Value*

data = [1, 1, 2, 1, 0, 0, 0, 0, 10, *# s1*

-3, 1, 0, -1, 0, 1, 0, 5, *# a1*

1, 0, 0, 0, -1, 0, 1, 8, *# a2*

-3, 2, -1, 0, 0, 0, 0, 0, 0, *# P*

-3, 2, -1, 0, 1, 1, 0, 0, -13] *# I*

mat = Matrix((9, 5), data)

**for** matrix, row\_names, pivot\_cell **in** two\_stage\_simplex(mat,

column\_names,

row\_names,

show\_changes=**True**):

print(DisplayMatrix(matrix, column\_names, row\_names))

Outputs:

x y z s1 s2 s3 a1 a2 Value

s1 0 5/2 3/2 1 1/2 0 -1/2 0 15/2

x 1 -3/2 1/2 0 -1/2 0 1/2 0 5/2

a1 0 5/2 -1/2 0 1/2 -1 -1/2 1 11/2

P 0 -5/2 1/2 0 -3/2 0 3/2 0 15/2

I 0 -5/2 1/2 0 -1/2 1 3/2 0 -11/2

x y z s1 s2 s3 a1 a2 Value

s1 0 0 2 1 0 1 0 -1 2

x 1 0 1/5 0 -1/5 -3/5 1/5 3/5 29/5

y 0 1 -1/5 0 1/5 -2/5 -1/5 2/5 11/5

P 0 0 0 0 -1 -1 1 1 13

I 0 0 0 0 0 0 1 1 0

x y z s1 s2 s3 a1 a2 Value

s1 0 0 2 1 0 1 0 -1 2

x 1 1 0 0 0 -1 0 1 8

s2 0 5 -1 0 1 -2 -1 2 11

P 0 5 -1 0 0 -3 0 3 24

x y z s1 s2 s3 a1 a2 Value

s3 0 0 2 1 0 1 0 -1 2

x 1 1 2 1 0 0 0 0 10

s2 0 5 3 2 1 0 -1 0 15

P 0 5 5 3 0 0 0 0 30

This is the correct output and it is seen that the program correctly removed the ‘I’ row.

The final addition is the Big M Simplex algorithm. Ideally, there should be no need to modify the simplex algorithm itself, and I should be able to make the algorithm work with the M type naturally.

I have stumbled across several problems in developing the Big M solution, mostly to do with the M type itself. I found that the arithmetic functions I had written were incorrect, and that my equality function was also invalid. The corrected functions are below.

**def** \_\_eq\_\_(self, other):

**if** isinstance(other, M):

**return** self.m\_val == other.m\_val **and** self.num\_val == other.num\_val

**return** self.m\_val == 0 **and** self.num\_val == other

**def** \_\_add\_\_(self, other):

new = M(0, 0)

**if** isinstance(other, M):

new.m\_val = self.m\_val + other.m\_val

new.num\_val = self.num\_val + other.num\_val

**return** new

new.num\_val = self.num\_val + other

new.m\_val = self.m\_val

**return** new

**def** \_\_sub\_\_(self, other):

new = M(0, 0)

**if** isinstance(other, M):

new.m\_val = self.m\_val - other.m\_val

new.num\_val = self.num\_val - other.num\_val

**return** new

new.num\_val = self.num\_val - other

new.m\_val = self.m\_val

**return** new

**def** \_\_mul\_\_(self, other):

new = M(0, 0)

**if** isinstance(other, M):

new.m\_val = self.m\_val \* other.m\_val

new.num\_val = self.num\_val \* other.num\_val

**return** new

new.num\_val = self.num\_val \* other

new.m\_val = self.m\_val \*other

**return** new

**def** \_\_div\_\_(self, other):

new = M(0, 0)

**if** isinstance(other, M):

new.m\_val = self.m\_val / other.m\_val

new.num\_val = self.num\_val / other.num\_val

**return** new

new.num\_val = self.num\_val / other

new.m\_val = self.m\_val / other

**return** new

**def** \_\_neg\_\_(self):

**return** M(-self.num\_val, -self.m\_val)

I was mistakenly modifying the object itself rather than returning the modified object, which led to a lot of confusing problems where the pivot cell would change values when it should not. Having corrected these errors, I wrote some tests to ensure that all variations of the algorithm work without issue.

**import** **test\_simplex**, **test\_two\_stage**, **test\_big\_m**

**from** **simplex** **import** simplex, two\_stage\_simplex

solvers = [(test\_simplex, "Simplex", simplex),

(test\_two\_stage, "Two-Stage Simplex", two\_stage\_simplex),

(test\_big\_m, "Big-M method", simplex)]

**def** test\_one(start\_matrix, end\_matrix, solver):

columns = [i **for** i **in** range(start\_matrix.dims[0])]

rows = [i **for** i **in** range(start\_matrix.dims[1])]

**for** calculated\_end\_matrix, \_, \_ **in** solver(start\_matrix, columns, rows):

**for** calculated\_mat\_item, end\_mat\_item **in** zip(calculated\_end\_matrix.data,

end\_matrix.data):

**try**:

**assert** calculated\_mat\_item == end\_mat\_item

**except**:

print(calculated\_mat\_item)

print(calculated\_end\_matrix)

print(end\_matrix)

**raise**

print("Success!")

**def** test\_all(start\_matrices, end\_matrices, test, solver):

print(f"Testing **{**test**}**...")

**for** start, end **in** zip(start\_matrices, end\_matrices):

test\_one(start, end, solver)

**return** **True**

**for** solver **in** solvers:

test\_all(solver[0].start\_matrices, solver[0].end\_matrices, solver[1], solver[2])

The test\_simplex, test\_two\_stage and test\_big\_m files simply contain the individual data for each test, and are not important code. All tests pass:

Testing Simplex...

Success!

Success!

Testing Two-Stage Simplex...

Success!

Testing Big-M method...

Success!

Now that the prototype code is fully functional, I can start to develop the final coded solution in C.