The Simplex Algorithm

The Simplex algorithm has stood the test of time as one of the most efficient ways to solve an important category of problem: Linear Programming problems. If you haven't come across these yet, this may not be the best activity for you – the algorithm can be confusing, and if you don't already have a reasonable grasp of the meaning behind the operations involved, you probably shouldn't try to code it yet! If, on the other hand, you've learned about linear programming and have a basic understanding of the Simplex algorithm, implementing it in code is an excellent way to cement your understanding and identify any misconceptions or gaps.

Getting the data

First things first: we need some sensible way of converting a linear programming problem from human-readable to machine-readable. We'll simplify the challenge for ourselves somewhat by assuming the inputter has already rewritten the constraint equations and the objective function in the form of equations, incorporating slack, surplus and artificial variables as necessary. To keep things simple, let's assume the data is entered more or less as an in-tact tableau. Let's test it out by making a table of numbers and copy-and-pasting it directly into a text file in repl. So that we have a sensible (but simple) example to start with, let's consider the 2-D linear programming problem below:

Maximise:

$$P = x + 2y$$

subject to the following constraints:

$$x + y \leq 10$$

$$2x - y \leq 8$$

$$x, y \geq 0$$

Rewriting into the appropriate format for a simplex tableau:

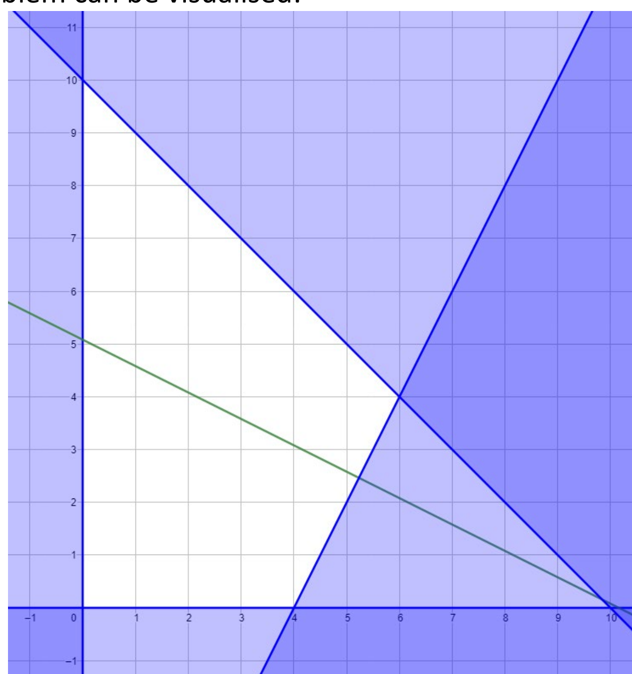
$$x + y + s = 10$$

$$2x - y + r = 8$$

And the objective function:

$$P - x - 2y = 0$$

The linear programming problem can be visualised:

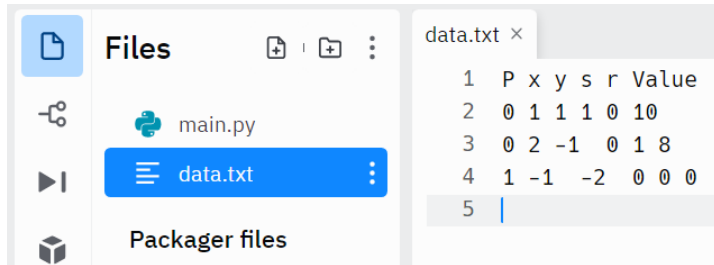


And we can enter the equations into a tableau:

P	x	y	s	r	Value
0	1	1	1	0	10
0	2	-1	0	1	8
1	-1	-2	0	0	0

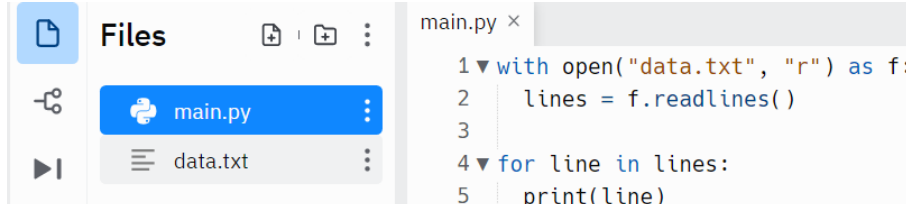
Interpreting the text

Let's try copying and pasting this data straight from the table into a text file in repl:



Click the image of the file with a + on it to add a new file, name it *data.txt* and paste the values directly from a word or excel table.

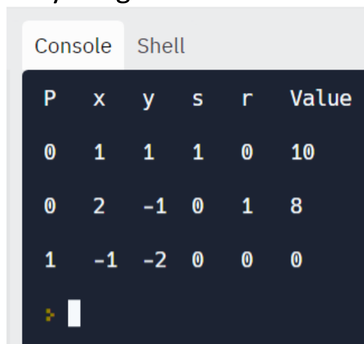
Reading the data



Back in *main.py*, use this code to open the file and read all the lines into a list.

Try printing it to the screen.

Everything looks fine on the surface, but there's some weird behaviour – for a start, it looks… nice.



We wouldn't normally expect that from raw text printed to the screen, but all the columns line up. Curious.

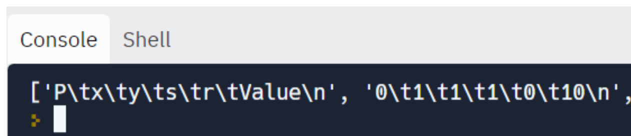
Also, there's a gap between each line and the next.

Let's try printing the list to see the actual text strings…

By asking Python to show us the list of text strings…

`print(lines)`

we see even the hidden characters:



The P, x, y, s, r and Value are all there, but there's a \t between each one, and \n at the end.

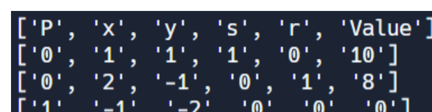
You may recall that \n is the special 'new-line' character which tells the computer to go onto a new line (the equivalent of hitting 'enter'). The \t character is for 'tab'.

Using the delimiter

When text is stored in a *csv* file, the data itself is separated by commas (hence 'comma separated variables'). But the so-called 'delimiter' need not be a comma. Our delimiter happens to be the \t character. If you want to input your data differently (eg typing it in directly to a text file) I suggest you use a comma or space as your delimiter, and tweak the code below accordingly.

```
1  with open("data.txt", "r") as f:
2    lines = f.readlines()
3
4  lines = [line.strip() for line in lines]
5
6  data = []
7  for line in lines:
8    data.append(line.strip().split("\t"))
9
10 for d in data:
11   print(d)
```

The combination of *strip* and *split* allows me to remove the new-line character \n from the end of each line and separate the text string into a list of individual strings, using the tab character \t as the indicator of where one ends and the next begins:



Formatting the data

While the first row is variable names, the rest of the data is numerical, but is currently stored as strings.

```
for i in range(1, len(data)):
    data[i] = [float(text) for text in data[i]]
```

This uses list comprehensions once again, this time to replace all text with floats in every line except the first.

Functional programming

All the steps above can be combined into a single function, to help us clarify the program flow:

```
def get_data(file_name):
    with open(file_name, "r") as f:
        lines = f.readlines()
    lines = [line.strip() for line in lines]
    data = []
    for line in lines:
        data.append(line.strip().split("\t"))
    for i in range(1, len(data)):
        data[i] = [float(text) for text in data[i]]
    return data
```

This is basically the same as the previous code, just squashed up and indented within a *get_data* function. The only changes are that I pass in a file name (so that potentially we could work with multiple different problems all in different documents), and return the *data* list.

Displaying the data

Before diving into running the Simplex algorithm, I should make sure I have a nice clean way to render the current state of the tableau to the screen (or store it in a new text document). In general, I like to deal with the output (whether it be text or graphics) before embarking on the main bulk of the program, because then I can see the effect as I go along, and debug more easily if (or rather, when!) errors occur. We could use the *join* text method to put all the pieces back together in the same format they came out, but tab is an imperfect way to display text on the screen since a longer value can mean columns get all out of alignment. Instead, we'll write our own dedicated *pad* function:

```
def pad(txt):
    txt = str(txt)
    n = 5
    if len(txt) >= 5:
        return txt[:5]
    else:
        return " "*(n - len(txt)) + txt
```

I'm setting the width to 5 initially – should be enough space for most numbers, and I can change it later.
If the text string is too long, I shorten it, and if it's too short, I append an appropriate number of spaces to the beginning.

Note: Python's built-in rjust *and* ljust *functions do a similar job, but don't force the text to be shorter if it spills over. I'd like my table to be properly aligned even if it means truncating some decimals.*

```
def tableau(data):
    n = 5
    text = []
    for line in data:
        strings = [pad(value) for value in line]
        text.append("\t".join(strings))
    print("\n".join(text))

tableau(data)
```

This function displays the current state of the tableau. The list *strings* is formed by padding the entries in each line, and then these are then joined together by the \t delimiter. On its own it can be messy, but if coupled with fixed length elements it works pretty predictably:

```
    P       x       y       s       r   Value
  0.0     1.0     1.0     1.0     0.0    10.0
  0.0     2.0    -1.0     0.0     1.0     8.0
  1.0    -1.0    -2.0     0.0     0.0     0.0
```

Are we done yet?

Another important feature to build in before we get stuck into Simplex proper is a way of checking to see if we've reached an optimal solution. That is, when there are no negative values in the objective row.

```
def optimal(data):
    for value in data[-1]:
        if value < 0:
            return False
    return True
```

This function loops through the values in the final row of the table, and if it finds any negatives it terminates, returning a value of *False*.
If it completes its *for* loop without generating a return value, then (and only then) will it go on to execute the final line, returning *True* because it has failed to find any negatives in the objective row.

Simplex step 1: Identifying the pivot column

The first step of the Simplex algorithm is to look for the **most negative element** in the objective row, since this will tell us the variable that can most profitably be increased.

```
def simplex(data):
  # finding pivot column
  lowest = 0
  pivot_col = -1
  for i in range (len(data[-1])):
    value = data[-1][i]
    if value < lowest:
      lowest = value
      pivot_col = i
  print(f"{pivot_col=}")
  if pivot_col == -1:
    return data
```

I need to loop through every value in the objective row (the final list in *data*), and update my current *lowest* any time I find a more negative value.

By setting the initial value of *lowest* to 0, I only update if I find a negative value.

By setting the initial value of *pivot_col* to -1, I give myself a way to check whether there were any negatives in the row at all (and abort the algorithm early, since the optimal solution is reached).

The *print* statement includes a nifty feature for debugging: I can easily print both the name of the variable and its value:

```
   P      x      y      s      r   Value
 0.0    1.0    1.0    1.0    0.0    10.0
 0.0    2.0   -1.0    0.0    1.0     8.0
 1.0   -1.0   -2.0    0.0    0.0     0.0
pivot_col=2
```

Simplex step 2: Identifying the pivot row

The next step is to determine the row which corresponds to the **least positive theta**, where theta is found by dividing the 'value' (final column element) by the corresponding element in the pivot column.

```
…
    return data
  # finding pivot row
  least_theta = float('inf')
  pivot_row = -1
  for i in range(1, len(data) - 1):
    value = data[i][pivot_col]
    if value != 0:
      theta = data[i][-1] / value
      if theta > 0 and theta < least_theta:
        least_theta = theta
        pivot_row = i
  print(f"{pivot_row=}")
  if pivot_row == -1:
    return data
```

This code follows on directly from the previous block. Recall, if a return statement is executed, the function terminates, so this code only runs if there is a valid pivot column.

Python has a nice way to represent a number larger than any I'll work with: *float('inf')*. This means that *least_theta* will be improved upon by any positive value.

Once again, I use -1 for the default *pivot_row* value, so I can abort the function if no such row exists.

Simplex step 3: Row operations

Having identified the location of the pivot by now (via *pivot_row* and *pivot_col*), I can move on to the next step: dividing all elements of the pivot row by the pivot element and then reducing all other elements in the pivot column to zero via row operations.

```
  # reducing pivot row
  pivot = data[pivot_row][pivot_col]
  data[pivot_row] = [v / pivot for v in data[pivot_row]]
```

This simply identifies the pivot itself (the element in the pivot position), and replaces the pivot row with a scaled version of itself.

```
  # reducing other rows
  for i in range(1, len(data)):
    if i != pivot_row:
      m = data[i][pivot_col]
      data[i] = [data[i][k] - m*data[pivot_row][k] for k in range(len(data[i]))]
  return data
```

This bit is somewhat more confusing: we loop through each row (except the variable names), and compute the multiple required to reduce its pivot column value to zero. Then we use this multiple (*m*) to replace the row with a version created by subtracting that multiple times the corresponding element in the pivot row.

Putting it all together

```
data = get_data("data.txt")
tableau(data)

solved = False
while not solved:
  input("Hit enter to run an iteration of Simplex: ")
  simplex(data)
  tableau(data)
  solved = optimal(data)
print("Solved!")
```

Once all the functions are in place, we can write out the main program. First, get the data and display it in tableau form.

Next, make a loop. This loop won't go indefinitely, even if the problem has no feasible solution, since it waits for user input at each step.

Basic Variables

One thing we can do to improve this immediately is add a Basic Variable column. We determine the basic variable for a given row by determining which column contains a non-zero value (usually 1) for that row and only that row.

```
def get_basic_variables(data):
  bv = []
  for r in range(1, len(data)):
    for c in range(len(data[r])):
      if data[r][c] != 0 and sum([abs(data[i][c]) for i in range(1, len(data)) if i != r]) == 0:
        bv.append(data[0][c])
        break
  return bv
```

First, we create an empty array to hold the basic variables we found. There should be one for every line. Next, we loop through index values for each row and, for a given row, loop through index values for each column. Then...

Yes, that line is a mouthful, and it doesn't get written all at once, so let's break it down:

`if data[r][c] != 0 and` The first condition that must be met is for the value being tested is non-zero.

`sum([abs(data[i][c])` The values in question for the second condition are from all the other rows, but each from the same column (hence using the same column index *c*).

`for i in range(1, len(data)) if i != r])` To select the row for each of these values, we loop through all of the equations (every row but the header), but we add a condition that we don't include the current row.

`sum([abs(data[i][c])` We consider the size of each value, because that way if any of them are non-zero (positive or negative), the sum of their absolute values will be non-zero.

`sum([a ... r]) == 0:` Finally, we require the sum to be zero, implying all values other than the one in the row we are considering must be zero.

Displaying the basic variables

The basic variables will change with every iteration of Simplex, so we should add a call to our new function in *tableau*, putting the appropriate variable name in front of each row.

```
def tableau(data):
  n = 5
  text = []
  bv = ["B.V."] + get_basic_variables(data)
  for line in data:
    strings = [pad(bv.pop(0))] + [pad(value) for value in line]
    text.append("\t".join(strings))
  print("\n".join(text))
```

I've added a line which makes a list whose first entry is the column header, *B.V.*, and the rest are the basic variables from my function. Then whenever a row is converted to text, *pop(0)* is used to add an element from the start of *bv* (and remove it).

```
 B.V.      P      x      y      s      r    Value
    s    0.0    1.0    1.0    1.0    0.0    10.0
    r    0.0    2.0   -1.0    0.0    1.0     8.0
    P    1.0   -1.0   -2.0    0.0    0.0     0.0
Hit enter to run an iteration of Simplex:
 B.V.      P      x      y      s      r    Value
    y    0.0    1.0    1.0    1.0    0.0    10.0
    r    0.0    3.0    0.0    1.0    1.0    18.0
    P    1.0    1.0    0.0    2.0    0.0    20.0
Solved!
```

Our nice simple example is solved after a single iteration, yielding the result $P = 20$ when $y = 10$, $r = 18$ and $x = s = 0$. This is consistent with what we'd expect from the diagram at the start, which is encouraging.

Saving the result

Running the code in the console is all very well, and it's fairly quick to copy in a table of values from word or excel, but it would be nice to be able to copy the result back from the program equally quickly. If we replace our *print* calls with a custom *output* function which both prints and saves to an output file, we can download each iteration of the tableau when the program is done.

```python
def output(text):
  print(text)
  with open("results.txt", "a") as f:
    f.write(f"{text}\n\n")
```

In *tableau*, we swap out the final line to say *output* instead of print, and write this quick function to both print the text given but also append it to the end of a "results.txt" file.

Note that the additional argument "a" stands for append, so that, unlike using "w" for write, the file isn't overwritten each time, but rather new content is added to the end of the current file.

The result: an output in a text file which can be copied and pasted directly into Word or Excel::

B.V.	P	x	y	s	r	Value
s	0.0	1.0	1.0	1.0	0.0	10.0
r	0.0	2.0	-1.0	0.0	1.0	8.0
P	1.0	-1.0	-2.0	0.0	0.0	0.0

B.V.	P	x	y	s	r	Value
y	0.0	1.0	1.0	1.0	0.0	10.0
r	0.0	3.0	0.0	1.0	1.0	18.0
P	1.0	1.0	0.0	2.0	0.0	20.0

B.V.	P	x	y	s	r	Value
s	0	1	1	1	0	10
r	0	2	-1	0	1	8
P	1	-1	-2	0	0	0
B.V.	P	x	y	s	r	Value
y	0	1	1	1	0	10
r	0	3	0	1	1	18
P	1	1	0	2	0	20

One final example

A 4D problem with four constraints taken from the Edexcel Decision 1 book (Exercise 7B Q4):

Maximise $P = 4x_1 - 3x_2 + 2x_3 + 3x_4$

subject to

$$x_1 + 4x_2 + 3x_3 + x_4 + r = 95$$
$$2x_1 + x_2 + 2x_3 + 3x_4 + s = 67$$
$$x_1 + 3x_2 + 2x_3 + 2x_4 + t = 75$$
$$3x_1 + 2x_2 + x_3 + 2x_4 + u = 72$$
$$x_1, x_2, x_3, x_4, r, s, t, u \geqslant 0$$

First, enter the values in a table in Word or Excel:

P	x1	x2	x3	x4	r	s	t	u	Value
0	1	4	3	1	1	0	0	0	95
0	2	1	2	3	0	1	0	0	67
0	1	3	2	2	0	0	1	0	75
0	3	2	1	2	0	0	0	1	72
1	-4	3	-2	-3	0	0	0	0	0

Then copy and paste into the *data.txt* file, and run the program:

B.V.	P	x1	x2	x3	x4	r	s	t	u	Value
r	0.0	1.0	4.0	3.0	1.0	1.0	0.0	0.0	0.0	95.0
s	0.0	2.0	1.0	2.0	3.0	0.0	1.0	0.0	0.0	67.0
t	0.0	1.0	3.0	2.0	2.0	0.0	0.0	1.0	0.0	75.0
u	0.0	3.0	2.0	1.0	2.0	0.0	0.0	0.0	1.0	72.0
P	1.0	-4.0	3.0	-2.0	-3.0	0.0	0.0	0.0	0.0	0.0

The initial tableau is printed at the start.

B.V.	P	x1	x2	x3	x4	r	s	t	u	Value
r	0.0	0.0	3.333	2.666	0.333	1.0	0.0	0.0	-0.33	71.0
s	0.0	0.0	-0.33	1.333	1.666	0.0	1.0	0.0	-0.66	19.0
t	0.0	0.0	2.333	1.666	1.333	0.0	0.0	1.0	-0.33	51.0
x1	0.0	1.0	0.666	0.333	0.666	0.0	0.0	0.0	0.333	24.0
P	1.0	0.0	5.666	-0.66	-0.33	0.0	0.0	0.0	1.333	96.0

After the first iteration, x1 has replaced u as a basic variable, increasing P to a value of 96.

B.V.	P	x1	x2	x3	x4	r	s	t	u	Value
r	0.0	0.0	4.0	0.0	-2.99	1.0	-1.99	0.0	0.999	33.00
x3	0.0	0.0	-0.24	1.0	1.25	0.0	0.749	0.0	-0.49	14.24
t	0.0	0.0	2.75	0.0	-0.75	0.0	-1.24	1.0	0.499	27.25
x1	0.0	1.0	0.749	0.0	0.25	0.0	-0.24	0.0	0.499	19.25
P	1.0	0.0	5.499	0.0	0.5	0.0	0.5	0.0	1.0	105.5

After just one additional iteration, we have an optimal solution of $P = 105.5$ at $x_1 = 19.25, x_3 = 14.24, x_2 = x_4 = 0$.

Note that our formatting reduces accuracy since we truncate numbers. For more precise values, simply change the parameter in the pad *function.*