Lab week 8

We have seen control flow, loops, and `numpy` arrays. Now we are going to use them on the simplex method.

Script

Linear programming is a core part of operational research. We'll look at solving the following linear program

$$\begin{array}{lrll} \max & x_1 & +x_2 & \\ & 2x_1 & -x_2 & \leq 4 \\ & x_1 & +2x_2 & \leq 3 \end{array}$$

where $x_1, x_2 \geq 0$. We'll code up the simplex method.

Let us bring the problem to standard form by introducing slack variables $s_1, s_2 \geq 0$ and transforming the problem into a minimization problem:

$$\begin{array}{lrrrrl} \min & -x_1 & -x_2 & & & \\ & 2x_1 & -x_2 & +s_1 & & = 4 \\ & x_1 & +2x_2 & & +s_2 & = 3 \end{array}$$

The corresponding tableau reads:

$$\left(\begin{array}{c|cccc} 0 & -1 & -1 & 0 & 0 \\ \hline 4 & 2 & -1 & 1 & 0 \\ 3 & 1 & 2 & 0 & 1 \end{array}\right)$$

Recall that we count rows and columns from zero here.

The tableau contains the reduced cost coefficients (row 0), the right-hand side (RHS) coefficients (column 0, ignoring its first component), which should be kept nonnegative at all times, and the coefficients of the various constraints. To be basic feasible, a collection of its columns should form an identity matrix (which is the case here: consider columns $3, 4$).

From an algebraic perspective (slightly more abstract than the one adopted in the lectures), the simplex method performs a sequence of row operations (pivoting operations) to remove all negative numbers from the top row (row 0), at each stage choosing a pivot which guarantees that the nonnegativity of the RHS coefficients (column 0, ignoring is first entry, i.e., the entry in position $(0, 0)$) is preserved.

First, we want to store the tableau. We'll do this using `numpy.`

```
import numpy
```

```
tableau = numpy.array([ [0, -1, -1, 0, 0],
                        [4,  2,  -1, 1, 0],
                        [3,  1,  2, 0, 1] ], dtype=numpy.float64)
print(tableau)
```

```
[[ 0. -1. -1.  0.  0.]
 [ 4.  2. -1.  1.  0.]
 [ 3.  1.  2.  0.  1.]]
```

We've explicitly said that the tableau entries are floating point numbers here, as we know we'll end up with non-integer results.

Note that the vector of right-hand sides has been stored in column `0`, from row `1` to row `2`. The entry in position `(0,0)` corresponds to the opposite of the objective function value.

First, we inspect the row of reduced costs (row `0`).

```
print(tableau[0, :])
```

```
[ 0. -1. -1.  0.  0.]
```

We see negative entries in columns `1` and `2`. Applying Bland's rule, we pivot on the left-most column: column `1`. As, carrying out the min ratio test, we observe `4/2 < 3/1`, we select row `1`, pivoting in position `(1,1)`.

First, we divide row 1 through by 2 (the entry in the pivot position):

```
tableau[1, :] = tableau[1, :] / tableau[1,1]
print(tableau)
```

```
[[ 0.  -1.  -1.   0.   0. ]
 [ 2.   1.  -0.5  0.5  0. ]
 [ 3.   1.   2.   0.   1. ]]
```

Next, we subtract the new row `1` scaled by `-1` (the entry in position `(0,1)`) from row `0`:

```
tableau[0, :] = tableau[0, :] - tableau[0, 1] * tableau[1, :]
print(tableau)
```

```
[[ 2.   0.  -1.5  0.5  0. ]
 [ 2.   1.  -0.5  0.5  0. ]
 [ 3.   1.   2.   0.   1. ]]
```

Next, we subtract the new row `1` scaled by `1` (the entry in position `(2,1)`) from row `2`:

```
tableau[2, :] = tableau[2, :] - tableau[2, 1] * tableau[1, :]
print(tableau)
```

```
[[ 2.   0.  -1.5  0.5  0. ]
 [ 2.   1.  -0.5  0.5  0. ]
 [ 1.   0.   2.5 -0.5  1. ]]
```

After this, a column of the identity matrix shows up in colum `1`. Row `0` (the reduced costs row) still contains a negative value in column `2`. As the column contains a single positive coefficient, the minumum ratio test is not necessary. Pivoting takes place in position `(2,2)`. First, we divide row `2` by `2.5`:

```
tableau[2, :] = tableau[2, :] / tableau[2, 2]
print(tableau)
```

```
[[ 2.   0.  -1.5  0.5  0. ]
 [ 2.   1.  -0.5  0.5  0. ]
 [ 0.4  0.   1.  -0.2  0.4]]
```

Then, we subtract the new row `2` from rows `0` and `1`, scaled by, respectively, `-1.5` and `-0.5`. We obtain:

```
tableau[0, :] = tableau[0, :] - tableau[0, 2] * tableau[2, :]
tableau[1, :] = tableau[1, :] - tableau[1, 2] * tableau[2, :]
print(tableau)
```

```
[[ 2.6  0.   0.   0.2  0.6]
 [ 2.2  1.   0.   0.4  0.2]
 [ 0.4  0.   1.  -0.2  0.4]]
```

Since the reduce cost vector (row `0`) is now componentwise nonnegative, we have found an optimal solution. Note that, as the two columns of the identity matrix are in columns `1` (corresponding to x_1) and `2` (corresponding to x_2), we deduce $x_B = (x_1, x_2)$. We can read the value of both variables and the corresponding objective function value as follows (remember to flip the sign of the element in position `(0,0)` as the value in `(0,0)` is the opposite of the objective function value):

```
print("z =", -tableau[0, 0])
print("x_1 =", tableau[1, 0])
print("x_2 =", tableau[2, 0])
```

```
z = -2.6
x_1 = 2.2
x_2 = 0.4
```

Working with parts of the tableau

To turn the steps above into a *general* function we need to find the candidate column on which we can pivot, and then the best row (out of the set of candidate rows) on which to actually pivot. This means getting a list, or array, containing the row and/or column indexes that we want to use.

With `numpy` directly we can check whether something is true or false using standard logical comparison operations. For example, let us rebuild our `tableau` and check for negative cost coefficients:

```
tableau = numpy.array([ [0, -1, -1, 0, 0],
                        [4,  2,  -1, 1, 0],
                        [3,  1,  2, 0, 1] ], dtype=numpy.float64)
print(tableau[0, 1:] < 0)
```

```
[ True  True False False]
```

We excluded the first column, as that is the objective function, not a cost coefficient.

To get the indexes, we can then use the `nonzero` function (as `False` is equivalent to `0`, this tells us the indexes where the comparison is `True`):

```
negative_cost_idxs = numpy.nonzero(tableau[0, 1:] < 0)[0]
print(negative_cost_idxs)
```

```
[0 1]
```

`nonzero` returns a tuple, giving us the index in each dimension of the array. As we're dealing with a 1d array we only care about the first dimension, hence the `[0]` at the end.

Note that we could (should!) check that any costs are negative. If not, then we have reached optimality. We can do this by using `len` to check the number of appropriate columns.

This tells us that the first and second cost coefficients are negative, and therefore are candidate columns. We can therefore choose the first of these as the column on which to pivot:

```
column = negative_cost_idxs[0] + 1
```

We had to add `1` in order to get the index into the `tableau`, as we had ignored the first column (as it corresponds to the objective function).

We can perform similar operations to find the row on which to pivot. First, store the row indexes where the entry is positive (as this is a requirement for Bland's algorithm):

```
positive_tableau_idxs = numpy.nonzero(tableau[1:, column] > 0)[0]
print(positive_tableau_idxs)
```

```
[0 1]
```

We can now use `argmin` to find the index with the minimum ratio, in order to do the minimum ratio test:

```
ratio = tableau[1:, 0] / tableau[1:, column]  # Compute all ratios
row_argmin = numpy.argmin(ratio[positive_tableau_idxs])  # Only check
positive entries
row = positive_tableau_idxs[row_argmin] + 1 # Correct for ignoring
first row
print(row)
```

```
1
```

Exercise

Turn the operations we carried out above into a function `simplex_method` which takes a basic feasible tableau as input and returns 1) the status (as a string) of the problem (either optimal or unbounded) and, for bounded problems, 2) the optimal solution value and 3) an optimal solution. It's up to you to decide what to return for unbounded problems. The method should apply Bland's rule when choosing what the pivoting pair of row and column.

Remember that, given a `numpy` array `tableau`, `tableau.shape` is a list of the sizes of each dimension.

Exercise

Check the correctness of your function by running on the problems seen in the lectures and problem classes/sheets.