Lab week 9

This builds directly on week 8.

Building robust functions

Constructing any mathematical algorithm is both an end in itself and a tool for future use. Being able to solve one linear programming problem using the Simplex method is good. Being able to solve any (suitable) linear programming problem using a function implementing the Simplex method allows us to ask more complicated questions. For example, how sensitive is the optimal solution to different constraints? This may be hard to answer analytically, but by calling our Simplex method function with slightly different inputs we can explore this question numerically.

However, this *only* works if the function we are calling is robust. We know that Simplex "fails" on certain inputs (for example, when the problem is unbounded). If we accidentally pass in incorrect input (an unbounded problem, for example), or nonsense (the right inputs, but in the wrong order, for example), then we need our function to catch this error and inform us. If we don't, we might use nonsense output from the Simplex function as "true" results for our more complex questions.

Documentation as contract

```python
import numpy as np
```

```python
def simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs):
    """
    Simplex method

    Parameters
    ----------
    cost_coeffs : vector of float
        Cost coefficients (appear in objective function), length nvars
    rhs_coeffs : vector of float
        Coefficients on RHS of inequalities, length nvars
    lp_coeffs : array of float
        Coefficients on LHS of inequalities, size nvars x nvars

    Returns
    -------
    status : str
        "optimal" or "unbounded" depending on problem.
    z : float
        Minimized objective function.
    x : vector of float
        Optimized coefficients.
    """
    nvars = len(rhs_coeffs)
    tableau = np.zeros((1+nvars, 1+2*nvars))
    tableau[1:, 0] = rhs_coeffs
    tableau[0, 1:nvars+1] = cost_coeffs
    tableau[1:, 1:nvars+1] = lp_coeffs
    tableau[1:, nvars+1:] = np.identity(nvars)
    # Find negative cost coefficients
    negative_cost_idx = np.nonzero(tableau[0, 1:] < 0)[0]
    if len(negative_cost_idx) == 0:
        return "unbounded", np.inf, np.zeros(nvars)
    while len(negative_cost_idx) > 0:
        column = negative_cost_idx[0] + 1
        # Bland's algorithm
        positive_tableau_idx =  np.nonzero(tableau[1:, column] > 0)[0]
        if len(positive_tableau_idx) == 0:
            return "unbounded", np.inf, np.zeros(nvars)
        ratio = tableau[1:, 0] / tableau[1:, column]
        row =
positive_tableau_idx[np.argmin(ratio[positive_tableau_idx])] + 1
        # Pivot
        tableau[row, :] = tableau[row, :] / tableau[row, column]
        for work_row in range(0, nvars+1):
            if row == work_row:
                continue
            scale = tableau[work_row, column] / tableau[row, column]
            tableau[work_row, :] = tableau[work_row, :] - scale *
tableau[row, :]
        negative_cost_idx = np.nonzero(tableau[0, 1:] < 0)[0]
    # Check
    negative_cost_idx = np.nonzero(tableau[0, 1:] < 0)[0]
    z = -tableau[0, 0]
    x =  tableau[1:, 0]
    return "optimal", z, x
```

Above is an implementation of the simplex method. For now, concentrate on the documentation. In particular, note that the inputs are *required* to be arrays, that the arrays are *required* to be floating point (real) numbers, and that the sizes of the arrays are *required* to match.

What happens if we don't match the requirements? Possibly nothing bad:

```
cost_coeffs = np.array([-1, -1])
rhs_coeffs = np.array([4, 3])
lp_coeffs = np.array([[2, -1], [1, 2]])

print(simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs))
```

```
('optimal', -2.6, array([2.2, 0.4]))
```

This looks like we did nothing wrong, but check the types of the numbers we passed in:

```
print(cost_coeffs.dtype)
print(rhs_coeffs.dtype)
print(lp_coeffs.dtype)
```

```
int64
int64
int64
```

We actually passed in integers rather than floats. In many (but not all!) cases this doesn't make a difference, and so failing to perfectly match the requirement is not a problem.

However, in other cases it can really matter:

```
costs_too_long = np.array([1.0, 2.0, 3.0])
print(simplex_method(costs_too_long, rhs_coeffs, lp_coeffs))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

<ipython-input-5-fd1557774531> in <module>
      1 costs_too_long = np.array([1.0, 2.0, 3.0])
----> 2 print(simplex_method(costs_too_long, rhs_coeffs, lp_coeffs))


<ipython-input-2-99868ecd8a46> in simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs)
     24     tableau = np.zeros((1+nvars, 1+2*nvars))
     25     tableau[1:, 0] = rhs_coeffs
---> 26     tableau[0, 1:nvars+1] = cost_coeffs
     27     tableau[1:, 1:nvars+1] = lp_coeffs
     28     tableau[1:, nvars+1:] = np.identity(nvars)


ValueError: could not broadcast input array from shape (3) into shape (2)
```

The sizes were inconsistent, compared to the assumptions that we made within the algorithm. This leads to an error.

The interpretation of this within Python is that *documentation is a contract*. The docstring of a function promises that *if* the inputs match the requirements, *then* the output will match the specifications. If the input doesn't meet requirements, then the function may *try* to produce sensible output, but may produce errors (or worse: it may produce output that looks plausible, but is wrong).

Not all programming languages are as permissive as Python. Some will insist that the input matches very specific requirements and refuse to even try to compute an answer if not.

Python uses what is called "duck-typing": if the input looks like a duck and quacks like a duck, it will be treated as a duck. In the first case above, integers (in this case) behave the same way as floats, and so can be treated as floats.

Trying and failing

The flip side of duck typing and the permissive Python approach is that when errors occur, as in the second case above, they can be difficult to link to the problem that caused them, and the error message can be hard to interpret.

We can improve our function by *adding in our own error messages* to explicitly catch, and explain, the error when calling the function.

Let's extract the key part of the function, using the "incorrect" inputs:

```
cost_coeffs = np.array([1.0, 2.0, 3.0])
rhs_coeffs = np.array([4, 3])
lp_coeffs = np.array([[2, -1], [1, 2]])

nvars = len(rhs_coeffs)
tableau = np.zeros((1+nvars, 1+2*nvars))
tableau[1:, 0] = rhs_coeffs
tableau[0, 1:nvars+1] = cost_coeffs
```

```
-----------------------------------------------------------------------
----

ValueError                                Traceback (most recent call
last)

<ipython-input-6-413a3e940c5b> in <module>
      6 tableau = np.zeros((1+nvars, 1+2*nvars))
      7 tableau[1:, 0] = rhs_coeffs
----> 8 tableau[0, 1:nvars+1] = cost_coeffs


ValueError: could not broadcast input array from shape (3) into shape
(2)
```

We see that we are programmatically setting up the tableau to have a specific size based on the number of equations, which is given by the number of RHS coefficients, which is given by the length of `rhs_coeffs`. However, we have *assumed* that this size, `nvars`, matches the number of variables, and hence the number of cost coefficients. We have documented this in the docstring - it is an underlying assumption in this (limited) implementation - but have not checked that it is true. When it is not true, as with this input, we get the error as the three entries in `cost_coeffs` cannot be stuffed into the two entries in that column of the `tableau`.

This is, indeed, an error in the inputs: they do not match our contract. It is an error in the values of the variable `cost_coeffs`, so calling it a `ValueError` makes sense. However, the message - however correct it may be - does not immediately help us fix the problem. Now that we know what the problem is and where it comes from, we can do better, by *raising the error ourselves*.

Modify the code to *check the inputs match the contract*:

```
nvars = len(rhs_coeffs)
tableau = np.zeros((1+nvars, 1+2*nvars))
tableau[1:, 0] = rhs_coeffs
if len(cost_coeffs) != nvars:
    raise ValueError("The length of cost_coeffs must match the length
of rhs_coeffs.")
tableau[0, 1:nvars+1] = cost_coeffs
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-7-fc3a19c2e27b> in <module>
      3 tableau[1:, 0] = rhs_coeffs
      4 if len(cost_coeffs) != nvars:
----> 5     raise ValueError("The length of cost_coeffs must match the length of rhs_coeffs.")
      6 tableau[0, 1:nvars+1] = cost_coeffs


ValueError: The length of cost_coeffs must match the length of rhs_coeffs.
```

This produces the exact same error as before, but now the error message tells the user calling the function exactly what needs to be fixed.

We can use this to make our original function more robust. We can even modify the code so that we remove the need for a `status` flag. We can raise an exception if the problem is unbounded.

```python
def simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs):
    """
    Simplex method

    Parameters
    ----------
    cost_coeffs : vector of float
        Cost coefficients (appear in objective function), length nvars
    rhs_coeffs : vector of float
        Coefficients on RHS of inequalities, length nvars
    lp_coeffs : array of float
        Coefficients on LHS of inequalities, size nvars x nvars

    Returns
    -------
    z : float
        Minimized objective function.
    x : vector of float
        Optimized coefficients.
    """
    nvars = len(rhs_coeffs)
    tableau = np.zeros((1+nvars, 1+2*nvars))
    tableau[1:, 0] = rhs_coeffs
    if len(cost_coeffs) != nvars:
        raise ValueError("The length of cost_coeffs must match the
length of rhs_coeffs.")
    tableau[0, 1:nvars+1] = cost_coeffs
    if lp_coeffs.shape != (nvars, nvars):
        raise ValueError("The shape of lp_coeffs must be (n, n) to
match the length (n) of rhs_coeffs.")
    tableau[1:, 1:nvars+1] = lp_coeffs
    tableau[1:, nvars+1:] = np.identity(nvars)
    # Find negative cost coefficients
    negative_cost_idx = np.nonzero(tableau[0, 1:] < 0)[0]
    if len(negative_cost_idx) == 0:
        raise ValueError("The problem is unbounded")
    while len(negative_cost_idx) > 0:
        column = negative_cost_idx[0] + 1
        # Bland's algorithm
        positive_tableau_idx =  np.nonzero(tableau[1:, column] > 0)[0]
        if len(positive_tableau_idx) == 0:
            raise ValueError("The problem is unbounded")
        ratio = tableau[1:, 0] / tableau[1:, column]
        row =
positive_tableau_idx[np.argmin(ratio[positive_tableau_idx])] + 1
        # Pivot
        tableau[row, :] = tableau[row, :] / tableau[row, column]
        for work_row in range(0, nvars+1):
            if row == work_row:
                continue
            scale = tableau[work_row, column] / tableau[row, column]
            tableau[work_row, :] = tableau[work_row, :] - scale *
tableau[row, :]
        negative_cost_idx = np.nonzero(tableau[0, 1:] < 0)[0]
    # Check
    negative_cost_idx = np.nonzero(tableau[0, 1:] < 0)[0]
    z = -tableau[0, 0]
    x =  tableau[1:, 0]
    return z, x
```

We can check that this behaves as expected by putting in an unbounded problem:

```
cost_coeffs = np.array([-1, -1])
rhs_coeffs = np.array([-1, -2])
lp_coeffs = np.array([[-1, 1], [-1, -1]])
simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs)
```

```
-----------------------------------------------------------------------
----

ValueError                                Traceback (most recent call
last)

<ipython-input-9-4ff0d23ee693> in <module>
      2 rhs_coeffs = np.array([-1, -2])
      3 lp_coeffs = np.array([[-1, 1], [-1, -1]])
----> 4 simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs)


<ipython-input-8-432b99016f91> in simplex_method(cost_coeffs,
rhs_coeffs, lp_coeffs)
     38         positive_tableau_idx =  np.nonzero(tableau[1:, column]
> 0)[0]
     39         if len(positive_tableau_idx) == 0:
---> 40             raise ValueError("The problem is unbounded")
     41         ratio = tableau[1:, 0] / tableau[1:, column]
     42         row =
positive_tableau_idx[np.argmin(ratio[positive_tableau_idx])] + 1


ValueError: The problem is unbounded
```

We now run into the issue with making better error messages: it blocks exploration. If we wanted to call the simplex method on lots of inputs, and find which was the "best" result, we would be stopped by the error message as soon as any case went wrong. We can see this when trying to varying the first cost coefficient in the unbounded problem:

```
costs = np.linspace(2, 0, 10)
for cost in costs:
    cost_coeffs = np.array([cost, -1])
    print(simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs))
```

```
(1.0, array([-1., -3.]))
(1.0, array([-1., -3.]))
(1.0, array([-1., -3.]))
(1.0, array([-1., -3.]))
(1.0, array([-1., -3.]))



------------------------------------------------------------------------
----

ValueError                                Traceback (most recent call
last)

<ipython-input-10-8bc09ddd2faf> in <module>
      2 for cost in costs:
      3     cost_coeffs = np.array([cost, -1])
----> 4     print(simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs))


<ipython-input-8-432b99016f91> in simplex_method(cost_coeffs,
rhs_coeffs, lp_coeffs)
     38         positive_tableau_idx =  np.nonzero(tableau[1:, column]
> 0)[0]
     39         if len(positive_tableau_idx) == 0:
---> 40             raise ValueError("The problem is unbounded")
     41         ratio = tableau[1:, 0] / tableau[1:, column]
     42         row =
positive_tableau_idx[np.argmin(ratio[positive_tableau_idx])] + 1


ValueError: The problem is unbounded
```

However, we can get around this problem programmatically. The idea is to *try* and do a calculation. If it works, good. If not, we can decide what to do, based on the error. The Python syntax looks very much like `if` / `else` statements:

```python
cost_coeffs = np.array([-1, -1])
try:
    simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs)
except ValueError:
    print("That input does not work")
```

```
That input does not work
```

This means we can complete the loop by ignoring cases where the algorithm is unbounded:

```python
costs = np.linspace(2, 0, 10)
for cost in costs:
    cost_coeffs = np.array([cost, -1])
    try:
        print(cost, simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs))
    except ValueError:
        print("cost", cost, "leads to unbounded problem")
```

```
2.0 (1.0, array([-1., -3.]))
1.7777777777777777 (1.0, array([-1., -3.]))
1.5555555555555556 (1.0, array([-1., -3.]))
1.3333333333333335 (1.0, array([-1., -3.]))
1.1111111111111112 (1.0, array([-1., -3.]))
cost 0.8888888888888888 leads to unbounded problem
cost 0.6666666666666667 leads to unbounded problem
cost 0.44444444444444464 leads to unbounded problem
cost 0.22222222222222232 leads to unbounded problem
cost 0.0 leads to unbounded problem
```

There are many additional error types that can be caught and checked: one that often comes up in mathematical calculations is zero division:

```python
1/0
```

```
---------------------------------------------------------------------------
----

ZeroDivisionError                         Traceback (most recent call
last)

<ipython-input-13-9e1622b385b6> in <module>
----> 1 1/0


ZeroDivisionError: division by zero
```

Exactly the same steps can be used for that case:

```python
xs = [2, 0, -3]
for x in xs:
    try:
        print("10 divide", x, "is", 10/x)
    except ZeroDivisionError:
        print("Cannot divide by zero")
```

```
10 divide 2 is 5.0
Cannot divide by zero
10 divide -3 is -3.3333333333333335
```

Testing

We can make our code more robust but we need to trust that it is correct. To do this we want to test against simple cases where we know the solution and check that it is right.

Note that we do not want to do this once. We have already modified our function a few times. Each time we change something we may be introducing errors. So each time we change something we want to re-run our tests to check we have not introduced a problem when trying to fix another. That means we want to make the tests as easy to run as possible.

Let us first look at a "solved" problem, where the LP coefficients are in the form of the identity matrix, so the optimal values of the coefficients match the RHS coefficients:

```
cost_coeffs = np.array([-1, -1])
rhs_coeffs = np.array([1, 1])
lp_coeffs = np.array([[1, 0], [0, 1]])
print(simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs))
```

```
(-2.0, array([1., 1.]))


<ipython-input-8-432b99016f91>:41: RuntimeWarning: divide by zero
encountered in true_divide
  ratio = tableau[1:, 0] / tableau[1:, column]
```

We knew in advance that we should have $\mathbf{x} = (1, 1)$ and so $z = -2$. So we could check for that:

```
correct_z = -2
correct_xs = np.array([1, 1])
z, xs = simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs)
print("z correct?", z == correct_z)
print("xs correct?", xs == correct_xs)
```

```
z correct? True
xs correct? [ True  True]


<ipython-input-8-432b99016f91>:41: RuntimeWarning: divide by zero
encountered in true_divide
  ratio = tableau[1:, 0] / tableau[1:, column]
```

For the coefficient case, we do not want to look at each coefficient: we want to know that they're all correct. `numpy` allows us to check this:

```
print("xs correct", np.all(xs == correct_xs))
```

```
xs correct True
```

We want to turn this test into a function, so that every time we change our `simplex_method` function we can test it with one line of code. In Python, there are standard conventions for this:

1. The testing function has a name starting `test_`, and typically takes no arguments;
2. The testing function uses `assert` to check that it is giving correct results.

Explicitly, the test above would be:

```python
def test_simple():
    cost_coeffs = np.array([-1, -1])
    rhs_coeffs = np.array([1, 1])
    lp_coeffs = np.array([[1, 0], [0, 1]])
    correct_z = -2
    correct_xs = np.array([1, 1])
    z, xs = simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs)
    assert z == correct_z, "Objective function should be -2"
    assert np.all(xs == correct_xs), "Coefficients should match RHS"
```

```python
test_simple()
```

```
<ipython-input-8-432b99016f91>:41: RuntimeWarning: divide by zero
encountered in true_divide
  ratio = tableau[1:, 0] / tableau[1:, column]
```

This produces no output! This is what we want: if the tests pass everything is fine and we continue. If the tests fail, we want to see that. We can check that by writing a function specifically designed to fail:

```python
def test_simple_fails():
    cost_coeffs = np.array([-1, -1])
    rhs_coeffs = np.array([1, 1])
    lp_coeffs = np.array([[1, 0], [0, 1]])
    correct_z = -1
    correct_xs = np.array([1, 1])
    z, xs = simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs)
    assert z == correct_z, "Objective function should be -2, but we are
comparing to -1"
    assert np.all(xs == correct_xs), "Coefficients should match RHS"

test_simple_fails()
```

```
<ipython-input-8-432b99016f91>:41: RuntimeWarning: divide by zero
encountered in true_divide
  ratio = tableau[1:, 0] / tableau[1:, column]



-----------------------------------------------------------------------
----

AssertionError                            Traceback (most recent call
last)

<ipython-input-20-512710b47034> in <module>
      9     assert np.all(xs == correct_xs), "Coefficients should match
RHS"
     10
---> 11 test_simple_fails()


<ipython-input-20-512710b47034> in test_simple_fails()
      6     correct_xs = np.array([1, 1])
      7     z, xs = simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs)
----> 8     assert z == correct_z, "Objective function should be -2,
but we are comparing to -1"
      9     assert np.all(xs == correct_xs), "Coefficients should match
RHS"
     10


AssertionError: Objective function should be -2, but we are comparing
to -1
```

We can write as many tests as we like, but each should add something new: that way, if a test fails, it tells us something specific has changed in the function we are testing. Here is an example:

```python
cost_coeffs = np.array([-1, -1])
rhs_coeffs = np.array([5, -2])
lp_coeffs = np.array([[7, -5], [5, 3]])

print(simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs))
```

```
(0.6666666666666667, array([ 1.66666667, -0.66666667]))
```

We see that the results are floats, not integers, with long decimal expansions (probably representing e.g. $2/3$ and similar). Testing these with direct equalities is dangerous:

```python
z, xs = simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs)
print(z == 2/3)
print(xs == np.array([5/3, -2/3]))
```

```
False
[False False]
```

Instead, we want to check that the results are *close* to the expected solution.

```
print(np.allclose(z, 2/3))
print(np.allclose(xs, np.array([5/3, -2/3])))
```

```
True
True
```

We can then build this into a test:

```
def test_close():
    cost_coeffs = np.array([-1, -1])
    rhs_coeffs = np.array([5, -2])
    lp_coeffs = np.array([[7, -5], [5, 3]])
    correct_z = 2/3
    correct_xs = np.array([5/3, -2/3])
    z, xs = simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs)
    assert np.allclose(z, correct_z), "Objective function should be 
2/3"
    assert np.allclose(xs, correct_xs), "Coefficients should be 5/3,
-2/3"

test_close()
```

We now have two (useful) tests. Every time we modify `simplex_method` we can now check that we have not introduced a bug by running two lines of code.

Exercise

Move these tests into a file along with your `simplex_method` function: call this `simplex.py`.

Make sure you have `pytest` installed (it should be part of Anaconda: in a terminal type `import pytest` to check).

Then `pytest` can be used to run all the tests. In a terminal, type

```
import pytest
pytest.main(['simplex.py'])
```

The resulting screen output should look something like:

```
============================== test session starts
===============================
...
collected 3 items

simplex.py .F.
[100%]

==================================== FAILURES
====================================
______________________________ test_simple_fails
_______________________________

    def test_simple_fails():
        cost_coeffs = np.array([-1, -1])
        rhs_coeffs = np.array([1, 1])
        lp_coeffs = np.array([[1, 0], [0, 1]])
        correct_z = -1
        correct_xs = np.array([1, 1])
        z, xs = simplex_method(cost_coeffs, rhs_coeffs, lp_coeffs)
>       assert z == correct_z, "Objective function should be -2, but we
are comparing to -1"
E       AssertionError: Objective function should be -2, but we are
comparing to -1

simplex.py:76: AssertionError
=============================== warnings summary
================================
simplex.py::test_simple
simplex.py::test_simple_fails
  /Users/ih3/Desktop/MATH1058/simplex.py:43: RuntimeWarning: divide by
zero encountered in true_divide
    ratio = tableau[1:, 0] / tableau[1:, column]

-- Docs: https://docs.pytest.org/en/stable/warnings.html
=========================== short test summary info
============================
FAILED simplex.py::test_simple_fails - AssertionError: Objective
function sho...
================== 1 failed, 2 passed, 2 warnings in 0.16s
====================
```

This has run all the tests for us and summarized the results. Using a test runner like `pytest` is invaluable when working with large codes, as it means every change can be automatically checked to see it hasn't broken existing code.