Answers: Exercise Sheet No. 4
Nonlinear Equations Pt. II.

Automatic Differentiation in python using `autograd`

No submission for this section. The important idea is to understand why Automatic Differentiation (AD) is useful and how it can be used.

1 Automatic Differentiation in Conjunction with a SciPy Root Solver

You should have a python file something like:

```
import autograd.numpy as np
from autograd import jacobian
import scipy.optimize as sp

def fun(x):
    return np.arctan(x)

jac = jacobian(fun)
sol = sp.root(fun, [3.]. jac=jac, method='lm')
print(sol)
```

LM should converge with a status of 4, i.e. there is an error message, but the algorithm did converge.

2 Flash tank

The code to answer all questions is included below:

```
# -*- coding: utf-8 -*-
"""
Created on Jan 23

@author: E. Turan
"""
import autograd.numpy as np
from autograd import jacobian
import scipy.optimize as sp

# Demonstrate Newton type methods on a flash tank problem
# The problem is solved using the root finding function of scipy.
    optimize

# Antoine parameters
A = np.array([3.97786, 4.00139, 3.93002], dtype=np.float64)
B = np.array([1064.840, 1170.875, 1182.774], dtype=np.float64)
C = np.array([-41.136, -48.833, -52.532], dtype=np.float64)
# feed composition
z = np.array([0.5, 0.3, 0.2], dtype=np.float64)
# feed flow, pressure and temperature
F = 100
p = 5
T = 390
```

```python

# As T and P are specified can calculate K directly
psat = np.float64(10) ** (A - B / (T + C))
K = psat / p

# define the residual function
def fun(inp):
    # the input is a vector of 8 elements, unpack now
    x = inp[0:3]
    y = inp[3:6]
    V = inp[6]
    L = inp[7]
    res_MB = -F * z + V * y + L * x
    res_EQ = -y + K * x
    res_xy = np.array([1 - sum(y), 1 - sum(x)])
    return np.concatenate((res_MB, res_EQ, res_xy))  # return a vector of residuals


ini_guess = np.array([0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 50, 50])  # initial guess

jac = jacobian(fun)  # autograd to calculate the jacobian

sol_flash = sp.root(fun, ini_guess, jac=jac, method="lm")  # solve the problem
print(sol_flash)

# set up problem in a different  formulation


def f_rr(psi):
    return np.sum((z * (K - 1)) / (1 + psi * (K - 1)))


# psi is V/F, and from this everything else can be calculated

# Autograd
jac_ad = jacobian(f_rr)


# solve
sol_psi_hybr = sp.root(f_rr, np.float64(0.5), jac=jac_ad, method="hybr")
sol_psi_lm = sp.root(f_rr, np.float64(0.5), jac=jac_ad, method="lm")


print(sol_psi_hybr)
print(sol_psi_lm)
```

You should find that 'hybr' sometimes has trouble converging, however when started "close ennough" it should converge faster than LM. If you don't specify the derivative function then the method automatically does finite differencing. As this 1) is less accurate and 2) requires repeated evaluations of the function you should find that it requires many more function evaluations (and probably more iterations) to converge to the solution.