# Exercises Lecture 4 (Chapter 6)

Make sure to import Numpy and the `optimize` module from SciPy.

```
import numpy as np
import scipy.optimize as optimize
```

**Question 1**

Execute `fsolve()` with initial guesses in $[-10, -9.5, -9, \dots, 9, 9.5, 10]$ to find roots of the function $f(x) = x \cdot \cos(x)$. You may use a for-loop to iterate over the initial guesses. Store the found roots in a list called `roots`.

The output list `roots` should look like this. Recall that `fsolve()` returns an array with one element (which you can access by indexing it at position 0).

```
[np.float64(-10.995574287564276), np.float64(1.5707963267948954), np.float64(-
7.853981633974491), np.float64(-7.853981633974483), np.float64(-7.853981633974498), np.float64
7.853981633974479), np.float64(-7.853981633974455), np.float64(-7.853981633974483), np.float64
4.712388980384539), np.float64(-4.71238898038469), np.float64(-4.71238898038469), np.float64(-
4.71238898038469), np.float64(-4.71238898038469), np.float64(1.5707963267948966), np.float64(-
1.5707963267948966), np.float64(-1.570796326794901), np.float64(-1.5707963267948974), np.float6
1.5707963267948966), np.float64(-1.5707963267948961), np.float64(0.0), np.float64(0.0), np.flo
1.5707963267948966), np.float64(4.71238898038469), np.float64(4.71238898038469), np.float64(4.7
7.853981633974483), np.float64(10.995574287564276)]
```

**Question 2**

We will write a function that can compute a root of a polynomial function.

a) Write a function that takes as input a number $x$ and an array of coefficients $a = [a_0, \dots, a_{n-1}]$. It should return the function $f(x) = \sum_{i=0}^{n-1} a_i x^i$.

The derivative of $f$ is given by $f'(x) = \sum_{i=1}^{n-1} i \cdot a_i x^{i-1}$.

b) Write a function `f_deriv` that takes as input a number $x$ and an array of coefficients $a = [a_0, \dots, a_{n-1}]$. It should return the derivative $f'(x)$ of $f$ in the point $x$.

c) Write a function `root_polynomial` that takes as input an array of coefficients $a = [a_0, \dots, a_{n-1}]$ and an initial guess $r$. It should output a root of the function $f$ if one is found, and a message saying no root was found otherwise. Use `root_scalar()` with Newton's method and initial guess $r$. The output property `converged` of `root_scalar()` tells you whether or not Newton's method has found a root. Also, you will need to use the `args` keyword argument.

Your functions should give the following output on the input below.

```
# First polynomial
a = np.array([1.5, 0, -2.5, 0.3, 1])
r = 1.5

result = root_polynomial(a,r)
print(result)

# Second polynomial (this function has no root)
a2 = np.array([1.5, 0, -2.5, 0.3, 3])
r2 = 1.5

result2 = root_polynomial(a2,r2)
print(result2)
```

```
The found root is x = -1.5180670079327394
No root was found; it might be that no root exists.
```

**Question 3**

In this problem, we try to solve the 2-by-2 nonlinear system of equations $f(x) = 0$ introduced by Bloggs where

$$f(x) = \begin{bmatrix} f_0(x) \\ f_1(x) \end{bmatrix} = \begin{bmatrix} x_1^2 - x_2 + 1 \\ x_1 - \cos(\frac{\pi}{2}x_2) \end{bmatrix}$$

with Jacobian

$$J(f) = \begin{bmatrix} 2x_1 & -1 \\ 1 & \frac{\pi}{2}\sin(\frac{\pi}{2}x_2) \end{bmatrix}.$$

The problem has solutions $(-1, 2)$, $(0, 1)$, and $(-1/\sqrt{2}, 3/2)$. The goal of this exercise is to show that the `root()` function might, or might not, converge depending on the initial guess that we choose.

a) Implement $f$ and its Jacobian as Python functions called `f` and `jac_f`.

For initial guesses, we will take points on the circle with radius 2 centered around $(0, 0)$. This circle can be parameterized by $[2\cos(\rho), 2\sin(\rho)]$ for $\rho \in [0, 2\pi]$.

b) Write a function `circle()` that takes as input a one-dimensional array $\rho = [\rho_0, \dots, \rho_{k-1}]$ and outputs the points $[2\cos(\rho_i), 2\sin(\rho_i)]$ on the rows of a $k \times 2$ array. Do not use for-loops.

c) For $k = 50$ evenly spaced values of $\rho \in [0, 2\pi]$, use `root()` to find a root of the system $f(x) = 0$ with initial guess $[2\cos(\rho), 2\sin(\rho)]$, Jacobian information, and the `'hybr'` method. For every value of $\rho$ the output should be a short message including 1) the value of $\rho$ and 2) the root that was found, or a message that no root was found. Have a look at the output message of the function `root()` to see how you can determine whether the method found a root or not. For $k = 50$ evenly spaced points in $[0, 2\pi]$ your output should look like this. As you can see, all of the three roots of the system are sometimes found, and also sometimes no root is found.

```
For guess = [2. 0.], found root is [1.5781495e-12 1.0000000e+00]
For guess = [1.98358003 0.25575432], found root is [-2.57655834e-14  1.00000000e+00]
For guess = [1.93458973 0.50730917], found root is [1.23867782e-13 1.00000000e+00]
For guess = [1.85383351 0.75053401], found root is [1.57396466e-13 1.00000000e+00]
For guess = [1.74263741 0.9814351 ], found root is [9.05935849e-13 1.00000000e+00]
For guess = [1.60282724 1.19622106], found root is [1.38409487e-15 1.00000000e+00]
For guess = [1.4366987 1.3913651], found root is [-1.32700583e-12  1.00000000e+00]
For guess = [1.2469796  1.56366296], found root is [-3.80159246e-13  1.00000000e+00]
For guess = [1.03678514 1.71028553], found root is [8.04962149e-11 1.00000000e+00]
For guess = [0.80956669 1.82882525], found root is [1.54091134e-17 1.00000000e+00]
For guess = [0.56905517 1.91733571], found root is [-7.72638622e-15  1.00000000e+00]
For guess = [0.31919979 1.97436357], found root is [-5.90449013e-12  1.00000000e+00]
For guess = [0.06410316 1.99897243], found root is [-0.70710678  1.5       ]
For guess = [-0.19204605  1.99075823], found root is [-0.70710678  1.5       ]
For guess = [-0.44504187  1.94985582], found root is [-1.  2.]
For guess = [-0.69073011  1.87693684], found root is [-1.  2.]
For guess = [-0.92507658  1.77319861], found root is [-1.  2.]
For guess = [-1.14423332  1.64034451], found root is [-0.70710678  1.5       ]
For guess = [-1.34460178  1.48055599], found root is [-0.70710678  1.5       ]
For guess = [-1.52289192  1.29645679], found root is [-0.70710678  1.5       ]
For guess = [-1.67617621  1.0910698 ], found root is [-0.70710678  1.5       ]
For guess = [-1.80193774  0.86776748], found root is [-0.70710678  1.5       ]
For guess = [-1.89811149  0.63021644], found root is [-1.  2.]
For guess = [-1.96311831  0.38231726], found root is [4.60067409e-13 1.00000000e+00]
For guess = [-1.99589079  0.12814044], found root is [4.60088962e-14 1.00000000e+00]
For guess = [-1.99589079 -0.12814044], found root is [-4.39476516e-13  1.00000000e+00]
For guess = [-1.96311831 -0.38231726], found root is [7.42968452e-17 1.00000000e+00]
For guess = [-1.89811149 -0.63021644], found root is [8.20547063e-14 1.00000000e+00]
For guess = [-1.80193774 -0.86776748], found root is [-5.80092403e-14  1.00000000e+00]
For guess = [-1.67617621 -1.0910698 ], found root is [4.97457028e-14 1.00000000e+00]
For guess = [-1.52289192 -1.29645679], found root is [-2.90995034e-14  1.00000000e+00]
For guess = [-1.34460178 -1.48055599], found root is [2.29128908e-16 1.00000000e+00]
For guess = [-1.14423332 -1.64034451], found root is [-3.51617808e-12  1.00000000e+00]
For guess = [-0.92507658 -1.77319861], found root is [2.52638402e-13 1.00000000e+00]
For guess = [-0.69073011 -1.87693684], found root is [1.66343153e-11 1.00000000e+00]
For guess = [-0.44504187 -1.94985582], found root is [-0.70710678  1.5       ]
For guess = [-0.19204605 -1.99075823], found root is [-0.70710678  1.5       ]
For guess = [ 0.06410316 -1.99897243], found root is [-0.70710678  1.5       ]
For guess =  [ 0.31919979 -1.97436357], no root was found
For guess =  [ 0.56905517 -1.91733571], no root was found
For guess =  [ 0.80956669 -1.82882525], no root was found
For guess = [ 1.03678514 -1.71028553], found root is [-0.70710678  1.5       ]
For guess = [ 1.2469796  -1.56366296], found root is [-2.26536179e-16  1.00000000e+00]
For guess = [ 1.4366987 -1.3913651], found root is [9.29740721e-15 1.00000000e+00]
```

```
For guess = [ 1.60282724 -1.19622106], found root is [9.29961312e-13 1.00000000e+00]
For guess =  [ 1.74263741 -0.9814351 ], no root was found
For guess = [ 1.85383351 -0.75053401], found root is [-7.74845403e-14  1.00000000e+00]
For guess = [ 1.93458973 -0.50730917], found root is [1.66798808e-12 1.00000000e+00]
For guess = [ 1.98358003 -0.25575432], found root is [-9.27679759e-14  1.00000000e+00]
For guess = [ 2.0000000e+00 -4.8985872e-16], found root is [1.57814971e-12 1.00000000e+00]
```

**Question 4**

Let $A \in \mathbb{R}^{n \times n}$ be a two-dimensional array, $b \in \mathbb{R}^n$ a one-dimensional array and $c \in \mathbb{R}$ a scalar. Consider the function

$$f(x) = f(x_0, \dots, x_{n-1}) = x^T A x + b^T x + c = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_{ij} x_i x_j + \sum_{i=0}^{n-1} b_i x_i + c$$

that takes as input a one-dimensional array $x$. The gradient of this function is given by $\nabla f(x) = Ax + b$.

Write a function `quadratic_minimizer` that takes as input $A, b$ and $c$. It should compute a (local) minimizer of the function $f$ using `minimize()` with gradient information, the `'CG'` method, and initial guess $x = [1, 1, \dots, 1]$.

It should give the following output on the given input.

```python
# Example usage
A = np.array([[3, 0, 0], [0, 4, 0], [0, 0, 5]])
b = np.array([-1, -4, 1])
c = 6

result = quadratic_minimizer(A, b, c)
print("Optimal solution:", result.x)
print("Function value at optimal point:", result.fun)
```

```
Optimal solution: [ 0.33333333  1.         -0.2       ]
Function value at optimal point: 6.0
```

Because the matrix $A$ is symmetric and has positive eigenvalues, the function $f$ is convex, meaning that an optimal solution can be found at the point where the gradient is the all-zeros vector, that is, $x^* = -A^{-1}b$. This is indeed the solution that we found.

```python
eig_A, _ = np.linalg.eig(A)

print(f"The eigenvalues of A are {eig_A}")

# Solution to gradient = 0
x = -1*(np.linalg.inv(A) @ b)

print("Point where gradient is zero:", x)
print("Function value at point where gradient is zero:", f(x,A,b,c))
```

```
The eigenvalues of A are [3. 4. 5.]
Point where gradient is zero: [ 0.33333333  1.          -0.2        ]
Function value at point where gradient is zero: 6.0
```

**Question 5**

Write a function `capacitated_norm()` taking as input vectors $a, b, w \in \mathbb{R}^n_{\geq 0}$ and a capacity value $W$, that returns the solution to the problem

$$\begin{aligned} \max \quad & \sqrt{\sum_{i=0}^{n-1} x_i^2} \\ \text{s.t.} \quad & \sum_{i=0}^{n-1} w_i x_i \leq W \\ & a_i \leq x_i \leq b_i \text{ for } i = 0, \dots, n-1 \end{aligned}$$

using `minimize()` with the `'trust-constr'` method. Take $a$ as the initial guess. You may assume that $\sum_i w_i a_i \leq W$, so that this problem always has a solution. Hint: You may want to have a look at the `constraints` keyword argument in the documentation of `minimize()` to see how you can input additional arguments to a constraint.

```python
n = 7
a = np.array([0,0,1,4,1,0,7])
b = np.array([1,4,3,6,8,1,9])
w = np.array([0,6,5,4,6,5,3])
W = 2*np.sum(w*a)

x = capacitated_norm(a,b,w,W)
x = np.around(x, decimals=2) # Round solution to two decimals

print(x)
```

```
[1.   0.   1.   6.   6.67 0.   9.  ]
```