Open in Colab

Numerical Optimisation. Project 1

Team Information

Group 1
Participants information in alphabetical order

#	Name	Lastname	Matr Number
1	David	Kürnsteiner	11820336
2	Christian	Peinthor	11815592
3	Elias	Ramoser	11918558
4	Georg	Storz	11918811

Implementation

All points x are represented as numpy arrays. Function f returns a scalar with grad_f and hessian_f returning numpy arrays.

Imports

Describe how to install additional packages, if you have some, here

In [1]:
```
import numpy as np
import scipy
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns
from numpy.linalg import *
from sklearn.datasets import make_spd_matrix
```

Stopping criteria

Function returns True if the gradient of f at xk relative to x0 is smaller than parameter tol.

Additionally there is an upper bound for iterations to stop non converging algorithms.

In [2]:
```
def stop_crit(grad_f, xk, x0, i, tol=1e-8, max_iter=5000):
  if i > max_iter:
    return True
  elif norm(grad_f(xk)) <= tol * norm(grad_f(x0)):
    return True
  return False
```

Varibales scaling

Place your reasoning here, how your algorithm behave with respect to this problem. You can also try rescaling your problems This is additional task, which can earn you several points.

In [ ]:
```

```

Stabilising algorithm

Place your reasoning here, how your algorithm behave with respect to this problem. You can also try rescaling your problems This is additional task, which can earn you several points.

In [3]:
```
#your function for stabilising goes here
```

Fighting floating-point numbers and roundoff error

Place your reasoning, how your algorithm behave with respect to this problem. You can also try rescaling your problems This is additional task, which can earn you several points.

Inverting matrices

linear_solve() provides a way to solve linear systems of equations using a LU-factorization of A and subsequent forward and backward substitution as described in the book. This solver proves to be quite unstable though in practical applications. We therefore use the numpy implementation of solve().

In [4]:
```
def forward_substitution(L, b):

    n = L.shape[0]
```

```
    y = np.zeros_like(b, dtype=np.double);

    y[0] = b[0] / L[0, 0]

    for i in range(1, n):
        y[i] = (b[i] - np.dot(L[i,:i], y[:i])) / L[i,i]

    return y

def back_substitution(U, y):

    n = U.shape[0]

    x = np.zeros_like(y, dtype=np.double);

    x[-1] = y[-1] / U[-1, -1]

    for i in range(n-2, -1, -1):
        x[i] = (y[i] - np.dot(U[i,i:], x[i:])) / U[i,i]

    return x

def linear_solve(A, b):

    P, L, U = scipy.linalg.lu(A)

    y = forward_substitution(L, P @ b)

    return back_substitution(U, y)
```

Gradients calculation

Following functions are wrapper functions that provide approximations of the gradient and hessian of f using the forward-difference approach as described in the book.

In [5]:

```
def e_i(size, index):
  arr = np.zeros(size)
  arr[index] = 1.0
  return arr

def approx_grad(f, e=1.1*10**-8):
  def grad_f(x):
    if x.size == 1:
      return (f(x + e) - f(x)) / e
    return np.array([(f(x + e * e_i(x.size, i)) - f(x)) / e for i in range(x.size)])
  return grad_f

def approx_hessian(f, e=1.1*10**-8):
  def hessian_f(x):
    if x.size == 1:
      return (f(x + 2*e) - 2*f(x + e) + f(x)) / e**2
    return np.array([[(f(x + e * e_i(x.size, i) + e * e_i(x.size, j)) - f(
                      x + e * e_i(x.size, i)) - f(x + e * e_i(x.size, j)) + f(
                      x)) / e**2 for j in range(x.size)] for i in range(x.size)])
  return hessian_f
```

Additional objects you implemented

The class Problem() provides an object to generate and set up quadratic and non some non quadratic test problems for the algorithms with.

In [6]:

```
class Problem():

  def __init__(self):

    self.f = None
    self.grad_f = None
    self.hessian_f = None
    self.min_x = None

  def quadratic(self, n_dim, rseed):

    rng = np.random.RandomState(rseed)
    A = make_spd_matrix(n_dim, random_state=rseed)
    x = rng.randint(-10, 10, n_dim)
    b = A @ x

    def f(x):
      return 0.5 * x.T @ A @ x - b @ x

    def grad_f(x):
      return A @ x - b

    def hessian_f(x):
      return A

    self.f = f
    self.grad_f = grad_f
    self.hessian_f = hessian_f
    self.min_x = x
    self.A = A
    self.b = b

  def rosenbrock(self):

    def f(x):
      return 100*(x[1] - x[0]**2)**2 + (1 - x[0])**2

    def grad_f(x):
      return np.array([-400*x[0]*(x[1] - x[0]**2) - 2*(1 - x[0]),
          200*(x[1] - x[0]**2)])

    def hessian_f(x):
      return np.array([[-400*(x[1] - 3*x[0]**2) + 2, -400*x[0]],
          [-400*x[0], 200]])

    self.f = f
    self.grad_f = grad_f
    self.hessian_f = hessian_f
    self.min_x = np.array([1,1])

  def himmelblau(self):

    def f(x):
      return (x[0] ** 2 + x[1] - 11) ** 2 + (x[0] + x[1] ** 2 - 7) ** 2

    def grad_f(x):
```

```
        return np.array([4*x[0]*(x[0]**2 + x[1] - 11)+2*(x[0] + x[1]**2 - 7),
                         4*x[1]*(x[1]**2 + x[0] - 7)+2*(x[1] + x[0]**2 - 11)])

      def hessian_f(x):
        return np.array([[12*x[0]**2 + 4*x[1] - 42, 4*(x[1] + x[0])],
                         [4*(x[1] + x[0]), 12*x[1]**2 + 4*x[0] - 26]])

      self.f = f
      self.grad_f = grad_f
      self.hessian_f = hessian_f
      self.min_x = np.array([[3,2], [-2.805118, 3.131312], [-3.779310, -3.283186], [3.

    def poly_1(self):

      def f(x):
        return ((x - 7)**2 * (x - 3)**2) / 4

      def grad_f(x):
        return (x - 7) * (x - 5) * (x - 3)

      def hessian_f(x):
        return 3 * x**2 - 30 * x + 71

      self.f = f
      self.grad_f = grad_f
      self.hessian_f = hessian_f
      self.min_x = np.array([[3],[5],[7]])

    def poly_2(self):

      def f(x):
        return (x**2 * (x**2 - 16*x + 40)) / 4

      def grad_f(x):
        return x * (x - 2) * (x - 10)

      def hessian_f(x):
        return 3 * x**2 - 24 * x + 20

      self.f = f
      self.grad_f = grad_f
      self.hessian_f = hessian_f
      self.min_x = np.array([[0],[2],[10]])

    def poly_3(self):

      def f(x):
        return (x * (3 * x**3 - 64 * x**2 + 414 * x - 648)) / 12

      def grad_f(x):
        return (x - 1) * (x - 6) * (x - 9)

      def hessian_f(x):
        return 3 * x**2 - 32 * x + 69

      self.f = f
      self.grad_f = grad_f
      self.hessian_f = hessian_f
      self.min_x = np.array([[1],[6],[9]])
```

Optimising algorithm itself

alpha_wolfe() returns a step length satisfiying the weak wolfe conditions using a bisection approach as described in [1]. newton_method() implements the line search Newton method solving the equation H @ pk = -grad instead of computing the inverse of H. This is done using np.solve() for stability reasons.

In [7]:
```
def alpha_wolfe(f, grad_f, xk, pk, c1=1e-4, c2=0.9):

  alpha = 0
  beta = np.Inf
  t = 1

  while True:

    if f(xk + t * pk) > (f(xk) + c1 * t * (pk @ grad_f(xk))):
      beta = t
      t = 0.5 * (alpha + beta)
    elif (-pk @ grad_f(xk + t * pk)) > (-c2 * pk @ grad_f(xk)):
      alpha = t
      t = (2 * alpha if beta == np.Inf else 0.5 * (alpha + beta))
    else:
      return t

def newton_method(x0, f, grad_f=None, hessian_f=None):

  conv_tol = 1e-8
  if grad_f == None:
    grad_f = approx_grad(f)
    hessian_f = approx_hessian(f)
    conv_tol = 1e-6
  i = 0
  x = x0

  while not stop_crit(grad_f, x, x0, i, tol=conv_tol):

    #pk = np.array([- (1 / hessian_f(x)) @ grad_f(x)]) if x.size == 1 else - inv(hes
    pk = np.array([- (1 / hessian_f(x)) @ grad_f(x)]) if x.size == 1 else solve(hess
    x = x + alpha_wolfe(f, grad_f, x, pk) * pk
    i += 1

  print(f"\nsearch terminated at iteration {i}, result: {x}")
  return x
```

Testing on 5-10 variables, Quadratic objective

Implement functions to optimise over

Place for additional comments and argumentation

In [8]:
```
rseed = [1,4,6,7,8]
quadratic_probs = []
```

```python
for i in range(5):
  prob = Problem()
  prob.quadratic(10, rseed[i])
  quadratic_probs.append(prob)
```

Run 5 tests

Note: *After every test print out the resulsts.*
For your convinience we implemented a function which will do it for you. Function can be used in case after running optimisation you return $x_{optimal}$, and if you have implemented your gradient approximation. Feel free to bring your adjustments.
Additionaly print how many iterations your algotithm needed. You might also provide charts of your taste (if you want).

Place for your additional comments and argumentation

In [9]:

```python
def final_printout(x_0,x_optimal,x_appr,f,grad,args,tolerance):
    """
    Parameters
    -------------------------------------------------------------------------------
    x_0: numpy 1D array, corresponds to initial point
    x_optimal: numpy 1D array, corresponds to optimal point, which you know, or have
    x_appr: numpy 1D array, corresponds to approximated point, which your algorithm
    -------------------------------------------------------------------------------
    f: function which takes 2 inputs: x (initial, optimal, or approximated)
                                      **args
       Function f returns a scalar output.
    -------------------------------------------------------------------------------
    grad: function which takes 3 inputs: x (initial, optimal, or approximated),
                                         function f,
                                         args (which are submitted, because you migh
                                              to call f(x,**args) inside your gradie
          Function grad approximates gradient at given point and returns a 1d np arr
    -------------------------------------------------------------------------------
    args: dictionary, additional (except of x) arguments to function f
    tolerance: float number, absolute tolerance, precision to which, you compare opt
    """

    print(f'Initial x is :\t\t{x_0}')
    print(f'Optimal x is :\t\t{x_optimal}')
    print(f'Approximated x is :\t{x_appr}')
    print(f'Is close verificaion: \t{np.isclose(x_appr,x_optimal,atol=tolerance)}\n'
    f_opt = f(x_optimal,**args)
    f_appr = f(x_appr,**args)
    print(f'Function value in optimal point:\t{f_opt}')
    print(f'Function value in approximated point:   {f_appr}')
    print(f'Is close verificaion:\t{np.isclose(f_opt,f_appr,atol=tolerance)}\n')
    print(f'Gradient approximation in optimal point is:\n{grad(f,x_optimal,args)}\n'
    grad_appr = grad(f,x_appr,args)
    print(f'Gradient approximation in approximated point is:\n{grad_appr}\n')
    print(f'Is close verificaion:\n{np.isclose(grad_appr,np.zeros(grad_appr.shape),a
```

In [10]:

```python
for i, prob in enumerate(quadratic_probs):
  print(f"Problem {i+1}: ")
```

```python
    print("approximated gradient: ")
    newton_method(np.zeros(10), prob.f)
    print("\n exact gradient: ")
    newton_method(np.zeros(10), prob.f, prob.grad_f, prob.hessian_f)
    print(f"\n actual minimum: {prob.min_x}\n")
```

```
Problem 1:
approximated gradient:

search terminated at iteration 1, result: [ -5.00000258   0.99999819   2.00000026  -
2.00000564  -0.99999859
   0.99999838  -4.99999869   5.00000137 -10.00000407   5.99999942]

 exact gradient:

search terminated at iteration 1, result: [ -5.   1.   2.  -2.  -1.   1.  -5.   5. -
10.   6.]

 actual minimum: [ -5   1   2  -2  -1   1  -5   5 -10   6]

Problem 2:
approximated gradient:

search terminated at iteration 1, result: [ 4.00002782 -5.00000623 -8.99998079 -2.00
000111 -1.99999684  7.99999184
 -0.99999169 -2.9999854   3.00000612 -1.99998371]

 exact gradient:

search terminated at iteration 1, result: [ 4. -5. -9. -2. -2.  8. -1. -3.  3. -2.]

 actual minimum: [ 4 -5 -9 -2 -2  8 -1 -3  3 -2]

Problem 3:
approximated gradient:

search terminated at iteration 1, result: [ 1.23802768e-05 -1.00001235e+00 -6.999996
10e+00 -9.61930436e-06
  3.00000796e+00  4.99999656e+00  4.32208402e-06  5.99999836e+00
 -8.99998924e+00  9.99981305e-01]

 exact gradient:

search terminated at iteration 1, result: [ 7.98428146e-15 -1.00000000e+00 -7.000000
00e+00 -6.22645543e-15
  3.00000000e+00  5.00000000e+00  1.87717696e-15  6.00000000e+00
 -9.00000000e+00  1.00000000e+00]

 actual minimum: [ 0 -1 -7  0  3  5  0  6 -9  1]

Problem 4:
approximated gradient:

search terminated at iteration 1, result: [ 5.00000314e+00 -5.99999693e+00 -7.000004
53e+00  9.00000132e+00
 -2.99999596e+00  4.00000797e+00 -2.00000696e+00  4.00000806e+00
  4.78438651e-06 -1.99999405e+00]

 exact gradient:

search terminated at iteration 1, result: [ 5.00000000e+00 -6.00000000e+00 -7.000000
00e+00  9.00000000e+00
 -3.00000000e+00  4.00000000e+00 -2.00000000e+00  4.00000000e+00
  1.07364444e-14 -2.00000000e+00]

 actual minimum: [ 5 -6 -7  9 -3  4 -2  4  0 -2]

Problem 5:
approximated gradient:
```

```
search terminated at iteration 1, result: [-7.00000826  6.99999811 -1.00000677 -5.00
00075  -2.00000849  9.00000521
 -1.99999628  5.99999763  2.99998715  6.99999355]

 exact gradient:

search terminated at iteration 1, result: [-7.  7. -1. -5. -2.  9. -2.  6.  3.  7.]

 actual minimum: [-7  7 -1 -5 -2  9 -2  6  3  7]
```

This method is the best for solving quadratic problems, as it is able to calculate the exact solutions with the gradient and hessian provided in the first iteration. For the approximated gradient and hessian it also provides good results after the first iteration

Testing on functions of 1-2 variables, Non-quadratic objective

---

Implement functions to optimise over

Place for additional comments and argumentation

In [ ]:

```

```

Run 5 tests

Place for your additional comments and argumentation

In [11]:

```
prob = Problem()
prob.rosenbrock()
print(f"Problem rosenbrock: \n")
print("approximate gradient/hessian: ")
newton_method(np.array([1.2,1.2]), prob.f)
print("\nexact gradient: ")
newton_method(np.array([1.2,1.2]), prob.f, prob.grad_f, prob.hessian_f)
print(f"\nactual minimum: {prob.min_x}")

prob = Problem()
prob.himmelblau()
print(f"\nProblem himmelblau: \n")
print("approximate gradient/hessian: ")
newton_method(np.array([2.5,1.5]), prob.f)
print("\nexact gradient: ")
newton_method(np.array([2.5,1.75]), prob.f, prob.grad_f, prob.hessian_f)
print(f"\nactual minimum: {prob.min_x}")

prob = Problem()
prob.poly_1()
```

```python
print(f"\nProblem poly_1: \n")
print("approximate gradient/hessian: ")
newton_method(np.array([1.2]), prob.f)
print("\nexact gradient: ")
newton_method(np.array([1.2]), prob.f, prob.grad_f, prob.hessian_f)
print(f"\nactual minimum: {prob.min_x}")

prob = Problem()
prob.poly_2()
print(f"\nProblem poly_2: \n")
print("approximate gradient/hessian: ")
newton_method(np.array([9.5]), prob.f)
print("\nexact gradient: ")
newton_method(np.array([1.2]), prob.f, prob.grad_f, prob.hessian_f)
print(f"\nactual minimum: {prob.min_x}")

prob = Problem()
prob.poly_3()
print(f"\nProblem poly_3: \n")
print("approximate gradient/hessian: ")
newton_method(np.array([8.5]), prob.f)
print("\nexact gradient: ")
newton_method(np.array([8.5]), prob.f, prob.grad_f, prob.hessian_f)
print(f"\nactual minimum: {prob.min_x}")
```

```
Problem rosenbrock:

approximate gradient/hessian:

search terminated at iteration 7, result: [0.99999669 0.99999338]

exact gradient:

search terminated at iteration 8, result: [1. 1.]

actual minimum: [1 1]

Problem himmelblau:

approximate gradient/hessian:

search terminated at iteration 5001, result: [2.5 1.5]

exact gradient:

search terminated at iteration 5, result: [3. 2.]

actual minimum: [[ 3.         2.       ]
 [-2.805118   3.131312]
 [-3.77931   -3.283186]
 [ 3.584428 -1.848126]]

Problem poly_1:

approximate gradient/hessian:

search terminated at iteration 6, result: [2.99999982]

exact gradient:

search terminated at iteration 6, result: [3.]

actual minimum: [[3]
 [5]
 [7]]

Problem poly_2:
```

```
approximate gradient/hessian:

search terminated at iteration 5001, result: [9.57583333]

exact gradient:

search terminated at iteration 6, result: [10.]

actual minimum: [[ 0]
 [ 2]
 [10]]

Problem poly_3:

approximate gradient/hessian:

search terminated at iteration 5001, result: [8.69881056]

exact gradient:

search terminated at iteration 4, result: [9.]

actual minimum: [[1]
 [6]
 [9]]
```

The quality of the outcome depends on the function that should be solved. The rosenbrock function for example can be solved very fast, while the himmelblau function can just be solved if the exact gradient and hessian are provided

Template for teachers' tests

Set up a template, how one can run your code

Template should include sceletons for:

- *custom function to optimise over*
- *values initialisation to submit into otimising algorithm*
- *optimiser function call*
- *report print out call*

Provide descriptions and comments.

In [12]:

```
algorithm_to_test = newton_method

# Here you can set your individual starting point
x_0 = np.array([0.15, -0.4])

# Here you can enter your individual function
def f(x):
    return 6*x[0]**2 + 3*x[1]**2

# Here you can enter the exact gradient of your function
```

```python
# This function will just be used in the second test
def grad_f(x):
    return np.array([12*x[0], 6*x[1]])


# Here you can enter the exact hessian of your function
# This function will just be used in the second test
def hessian_f(x):
    return np.array([[12, 0],
                     [0, 6]])


# Test run:

print("Test with approximate gradient:")
algorithm_to_test(x_0, f)

print('\n'*2) # Print some lines between the tests

print("Test with exact gradient:")
algorithm_to_test(x_0, f, grad_f, hessian_f)

print()
```

```
Test with approximate gradient:

search terminated at iteration 3, result: [-5.50000006e-09 -5.50003364e-09]



Test with exact gradient:

search terminated at iteration 1, result: [0.00000000e+00 5.55111512e-17]
```

References

[1] https://sites.math.washington.edu/~burke/crs/408/notes/nlp/line.pdf