<u>Optimization – Programming Assignment</u>

Guidelines:

1. The assignment is due Mon, June 8, 2021 at 23:59.
2. Your classes and functions should be implemented in a single file named *'main.py'*.
3. The assignment will be automatically tested (numerical errors $\leq 10^{-8}$ are acceptable). Make sure classes, functions' names and APIs are **<u>exactly</u>** (case sensitive) as they appear in this document.
4. Solutions are tested for correctness only. That said, writing an efficient and well documented code are good practices that might be beneficial for you.
5. You may add **<u>any additional</u>** class attributes or auxiliary methods for internal usage as long as you maintain the original methods API and functionality.
6. You're allowed to use Python ≥ 3.6 and NumPy only.
7. You're not required to validate inputs.
8. Remove any debug logic in your code prior to submitting your solution.
9. Please send your questions on forum.
10. Assignments will be automatically tested for copying.

Python and NumPy best practices:

1. Keep your code as general as possible. Unless otherwise stated, avoid calling attributes of other classes.
2. NumPy Array dimensions – scalar: (), vector: (n,), matrix: (n,m)
3. All numerical arrays in this assignment should be NumPy arrays. Scalars can be either python or NumPy's *float* or *int*.
4. Setting vector dimensions explicitly to row (1,n) or column (n,1) is typically a bad practice which indicates an incorrect function or an incorrect usage.

General:

1. Write a function *student_id* which returns a tuple of your ID (string) and a string with your **<u>university email</u>** address(@mail.tau.ac.il), example:

```
def student_id():
        return '123456789', r'izhakadiv@mail.tau.ac.il'
```

note the *r* before '<email address>'

Part I – Gradient-based optimization methods (60pts)

In this part you'll implement a Quasi-Newton optimizer and use it to minimize a quadratic function.

1. **Quadratic function (10pts):**
   a. Create a class named *QuadraticFunction* that implements a function of the form

$$f(\underline{x}) = \frac{1}{2}\underline{x}^T Q \underline{x} + \underline{b}^T \underline{x}$$

   where $Q \in \mathbb{R}^{n \times n}$ (not necessarily symmetric), $\underline{b} \in \mathbb{R}^n$ and $n \geq 1$.

   b. Implement the following methods:

__init__(self, Q, b) :

Initializes a quadratic function with NumPy matrix Q and vector b.
Attributes names should be 'Q' and 'b' respectively.

Arguments:

Q : matrix

b : vector

__call__(self, x) :

Evaluates $f(\underline{x})$ at $\underline{x}$

Arguments:

x : vector

Returns:

fx : scalar

grad(self, x) :

Evaluates $g(\underline{x})$, the gradient of $f(\underline{x})$, at $\underline{x}$

Arguments:

x : vector

Returns:

gx : vector

hessian(self, x) :

Evaluates $H(\underline{x})$, the Hessian of $f(\underline{x})$, at $\underline{x}$

Arguments:

x : vector

Returns:
hx : matrix

2. **Newton's method (15pts):**
    a. Create a class *NewtonOptimizer* which implements the Newton's method with constant α.
    b. Implement the following methods:

__init__(self, objective, x_0, alpha, threshold, max_iters) :
Initializes a Newton's method optimizer with an objective function.

Arguments:
objective : callable to an objective function, such as *QuadraticFunction* above

x_0 : vector, initial guess for the minimizer

alpha : (scalar) constant step size

threshold : scalar, stopping criteria $|\underline{x}_{k+1} - \underline{x}_k| <$ threshold, return $\underline{x}_{k+1}$

max_iters : scalar, maximal number of iterations (stopping criteria)

step(self) :
Executes a single step of newton's method.

Return: a tuple (next_x, gx, hx) as follow:
next_x : vector, updated x.

gx : vector, the gradient of $f(\underline{x})$ evaluated at the current $\underline{x}$ (**not** next_x)

hx : matrix, the Hessian of $f(\underline{x})$ evaluated at the current $\underline{x}$ (**not** next_x)

optimize(self) :
Execution of optimization flow

Return:

fmin : scalar, objective function evaluated at x_opt

minimizer : vector, the optimal $\underline{x}$

num_iters : scalar, number of iterations until convergence

Remarks:

- You may call *self.objective()*, *self.objective.grad()* and *self.objective.hessian()*.
- Your implementation of *optimize()* should utilize the optimizer's *step()* method.
- You may use *QuadraticFunction* to test your implementation.

3. **Quasi-Newton BFGS method (40pts):**
   a. Create a class named *BFGSOptimizer* which implements the Quasi-Newton algorithm.
   b. The optimizer uses Backtracking Line Search (Armijo rule) for finding the step size and BFGS algorithm for approximating the inverse Hessian (rank 2 update).
   c. Implement the following methods:

__init__(self, objective, x_0, B_0, alpha_0, beta, sigma, threshold, max_iters) :
Initializes a Newton's method optimizer.

Arguments:

objective : callable to an objective function, such as *QuadraticFunction* above

x_0 : vector, initial guess for the minimizer

B_0 : matrix , initial guess of the inverse Hessian

alpha_0 : scalar, initial step size for Armijo line search

beta : scalar, beta parameter of Armijo line search, a float in range (0,1)

sigma : scalar, sigma parameter of Armijo line search, a float in range (0,1)

threshold : scalar, stopping criteria $|\underline{x}_{k+1} - \underline{x}_k| <$ threshold, return $\underline{x}_{k+1}$

max_iters : scalar, maximal number of iterations (stopping criteria)

update_dir(self):
Computes step direction.

Return:

next_d : vector, the new direction

update_step_size(self):

Compute the new step size using Backtracking Line Search algorithm (Armijo rule). Follow the algorithm described in class (see recording).

Return:

step_size : scalar

update_x(self):

Take a step in the descending direction.

Return:

next_x : vector, updated x

update_inv_hessian(self, prev_x):

Compute the approximator of the inverse Hessian using BFGS algorithm with rank-2 update.

Arguments:

prev_x : vector, previous point x

Return:

next_inv_hessian : matrix, approximator of the inverse Hessian matrix

step(self) :

Executes a single Quasi-Newton step.

Return:

a tuple (next_x, next_d, step_size, next_inv_hessian) as follows:

next_x : vector, updated x

next_d : vector, updated direction

step_size : scalar

next_inv_hessian : matrix, approximator of the inverse Hessian matrix

optimize(self) :
Execution of optimization flow.

Return:
fmin : scalar, objective function evaluated at the minimum

minimizer : vector, the optimal $\underline{x}$

num_iters : scalar, number of iterations until convergence

Remarks:

- You may call *self.objective()* and *self.objective.grad()* only. Assume *self.objective.hessian()* is not necessarily given/known.
- Your implementation of *optimize()* should utilize the optimizer's *step()* method.
- Armijo rule requires to restrict a multi-dimensional function along a line, you may find python's lambda function very useful for that purpose.
- You may use *QuadraticFunction* to test your implementation.

Part II – Total Variation Image Denoising (35pts)

Given an input noisy image X (n×m pixels), we wish to approximate the original image Y. A possible solution is given by minimizing the Total Variation, while keeping Y still close to X. The total variation is given by:

$$\mathcal{L}_{TV}(Y) = \sum_{i,j} \sqrt{\left|y_{i+1,j} - y_{ij}\right|^2 + \left|y_{i,j+1} - y_{ij}\right|^2}$$

We will use a differentiable form instead and add a small number for numerical stability:

$$\mathcal{L}_{TV}(Y) = \sqrt{\left(y_{i+1,j} - y_{ij}\right)^2 + \left(y_{i,j+1} - y_{ij}\right)^2 + \epsilon}$$

In addition, we will use MSE as a measure for closeness:

$$\mathcal{L}_{\text{MSE}}(X,Y) = \frac{1}{n \cdot m} \sum_{i,j} \left(x_{ij} - y_{ij}\right)^2$$

The overall total variation objective is therefore:

$$\begin{aligned}\mathcal{L}(X,Y) &= \mathcal{L}_{\text{MSE}}(X,Y) + \mu \mathcal{L}_{TV}(Y) \\ &= \sum_{i,j} \left\{ \frac{1}{n \cdot m} \left(x_{ij} - y_{ij}\right)^2 + \mu \sqrt{\left(y_{i+1,j} - y_{ij}\right)^2 + \left(y_{i,j+1} - y_{ij}\right)^2 + \epsilon} \right\}\end{aligned}$$

Where μ is a regularization parameter to be set.

Guidelines:

1. Assume that the input is a **grayscale** noisy image represented by (n,m) matrix with its values in range [0,1].
2. Use small images (32×32 or similar) and AWGN noise to test your implementation.
3. Assume Neumann boundary conditions, namely, the gradient is zero where the index is out of range. See examples below:

$$\begin{aligned} y_{i+1,j} - y_{ij} &= 0\ \forall\{(i,j) : i = n-1, 0 \le j \le m-1\} \\ y_{i,j+1} - y_{ij} &= 0\ \forall\{(i,j) : 0 \le i \le n-1, j = m-1\} \\ y_{ij} - y_{i-1,j} &= 0\ \forall\{(i,j) : i = 0, \quad 0 \le j \le m-1\} \\ y_{ij} - y_{i,j-1} &= 0\ \forall\{(i,j) : 0 \le i \le n-1, \quad j = 0\} \end{aligned}$$

4. **Total variation objective (20pts):**
   a. Create a class named *TotalVariationObjective* which implements the total variation objective $\mathcal{L}(X,Y)$ described above.

b. This class is designed such that it is compatible with the BFGS optimizer from part I which assumes that the minimizer is a vector and not a matrix. However, matrix to vector and vector to matrix conversions are simple.
c. Implement the following methods:

__init__(self, src_img, mu, eps):
Initialize a total variation objective.

Arguments:
src_img : (n,m) **matrix**, input noisy image

mu : regularization parameter, determines the weight of total variation term

eps : small number for numerical stability

__call__(self, img):
Evaluate the objective for img.

Arguments:
img : ($n \times m$,) **vector**, denoised image

Return:
total_variation : scalar, objective's value

grad(self, img):
Evaluate the gradient of the objective.

Arguments:
img : ($n \times m$,) **vector**, denoised image

Return:
grad : (nxm,) **vector**, the objective's gradient

2. **Image denoising procedure (15pts):**
   a. Create a function named *denoise_img* which denoises a noisy image by minimizing a total variation objective using a BFGS optimizer.

```
def denoise_img(noisy_img, B_0, alpha_0, beta, sigma, threshold,
max_iters, mu, eps):
    Optimizes a Total Variantion objective using BFGS optimizer to
    denoise a noisy image.

    Arguments:
        noisy_img : (n,m) matrix, input noisy image

        For the rest: see BFGSOptimizer and TotalVariationObjective.

    Return:
        total_variation : loss at minimum

        img : (n,m) matrix, denoised image, values expected range is [0,1]

        num_iters : number of iterations until convergence
```