Optimization – Programming Assignment

Guidelines:

1. The assignment is due Sunday, June 7, 2020 at 23:59.
2. Submission is on Moodle, submit a single file named *'main.py'*.
3. You're allowed to use Python ≥ 3.6 and NumPy only.
4. The assignment will be automatically tested, make sure classes and functions names are **exactly** (case sensitive) as they appear in this document.
5. You're not required to verify proper inputs (NumPy arrays).
6. The only scoring parameter is correctness, implementations will not be scored by efficiency. That said, writing efficient code is a good practice and it might be beneficial for you.
7. You may add **any additional** class attributes or auxiliary methods for internal usage.
8. You may add documentation (docstrings and comments) to your code for clarity (in case things go wrong) but it is not mandatory.
9. Assignments will be automatically tested for copying

General:

1. Your classes and functions should be implemented in a single file named *'main.py'*.
2. Write a function *student_id* which returns a tuple of your ID (integer) and a string with your **university email** 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 (55pts)

In this part you'll implement two gradient based optimizers and test them with a quadratic function.

1. **Quadratic function (15pts):**
    a. Create a class *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), $\underline{b} \in \mathbb{R}^n$
    b. Implement the following methods:

```
__init__(self, Q, b) :
        Initializes a quadratic function with NumPy matrix Q and vector b
__call__(self, x) :
```

Returns the function value, evaluated at point x, $f(\underline{x})$

grad(self, x) :

Returns the function's gradient, evaluated at point x, $g(\underline{x})$

hessian(self, x) :

Returns the function's Hessian, evaluated at point x, $h(\underline{x})$

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, func ,alpha, init) :

Initializes a Newton's method optimizer with function.

<u>Inputs:</u>

func :

An handle to initialized objective function, such as *QuadraticFunction* above

alpha :

a constant step size (scalar)

init :

initial point, an N dimensional NumPy vector

step(self) :

Executes a single step of newton's method.

Returns a tuple (next_x, gx, hx) as follow:

next_x :

The new $\underline{x}$, an N-dimensional NumPy vector.

gx :

The gradient of $f(\underline{x})$ evaluated at the current $\underline{x}$ (the one used for the step calculation), an N-dimensional NumPy vector.

hx :

The Hessian of $f(\underline{x})$ evaluated current $\underline{x}$ (the one used for the step calculation), an $N \times N$ NumPy matrix.

optimize(self, threshold, max_iters) :

Execution of optimization flow

<u>Inputs:</u>

threshold :

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

max_iters :

maximal number of iterations (stopping criteria)

Return:

fmin :

Objective function evaluated at the minimum

minimizer :

The optimal $\underline{x}$

num_iters :

Number of iterations

Remarks:

- Use *self.func.grad()* and *self.func.hessian()* internally if needed. Don't forget to update the current value of $\underline{x}$ for the next iteration (call to *step()* method).
- Your implementation *optimize()* should utilize the optimizer's *step()* method.

3. **Conjugate gradient (25pts)**:
   a. Create a class *ConjGradOptimizer which* implements the Conjugate Gradient method. Assume that *func* is a quadratic function with symmetric Q matrix.
   b. Implement the following methods:

__init__(self, func , init) :

Initializes a Conjugate Gradient method optimizer with function.

Inputs:

func :

An handle to initialized objective function, such as *QuadraticFunction* above

init :

initial point, an N dimensional NumPy vector

Use these inputs to initialize the rest of the required attributes.

update_grad(self) :

Calculates and returns $\left(\underline{g}_k, \underline{g}_{k-1}\right)$ the new and previous gradients correspondingly.

update_dir(self, prev_grad) :

Calculates and returns the new direction $\underline{d}_k$

Inputs:

prev_grad : an (N-1)-dimensional NumPy vector, previous gradient computation $\underline{g}_{k-1}$

update_alpha(self) :

Calculates and returns the new step size α_k

update_x(self) :

Calculates and returns $\underline{x}_k$

step(self) :

Executes a single step of newton's method.

Returns a tuple $(\underline{x}_k, \underline{g}_k, \underline{d}_k, \alpha_k)$ where $\underline{x}_k, \underline{g}_k, \underline{d}_k$ are N-dimensional NumPy vectors and α_k. is a scalar.

optimize(self) :

Returns:

fmin :

Objective function evaluated at the minimum

minimizer :

The optimal $\underline{x}$

num_iters :

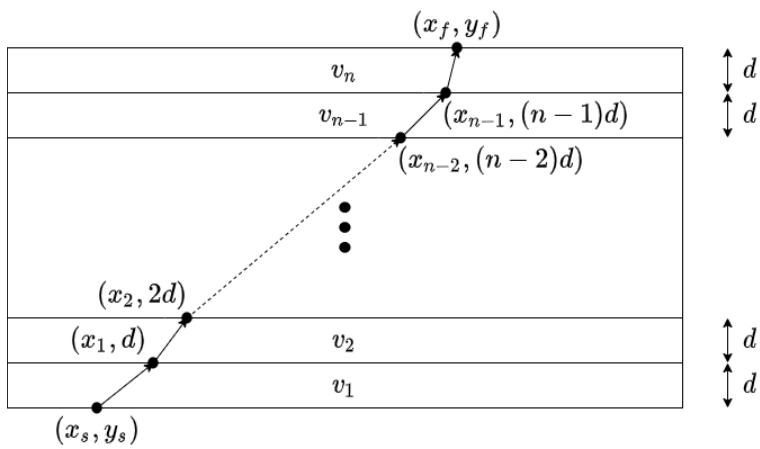
Number of iterations

Remarks:

- Use *self.func.grad()* and *self.func.hessian()* internally if needed. Don't forget to update the current value of $\underline{x}$ for the next iteration (call to *step()* method).
- Your implementation *optimize()* should utilize the optimizer's *step()* method.
- What's the stopping criteria in this case?

Part II – Test Case (45pts + 5pts bonus)

Consider the following optimization problem:



The objective is to find a fast route to deliver a package from (x_s, y_s) to (x_f, y_f) in minimal *time*, namely, find $\underline{x} = (x_1, x_2, \dots, x_{n-1})$ such that the total time to travel from the starting point to the end point is minimal (not necessarily global minimum), given that the velocity in each region $(i-1)d \le y < i \cdot d$ is v_i $\forall i \in \{1, \dots, n\}$.

1. Create a class ***FastRoute*** class that implements the objective function. The class must include the following methods **(35pts)**:

__init__(self, start_x, start_y, finish_x, finish_y, velocities) :

Inputs:

start_x :

x coordinate of starting point

start_y :

y coordinate of starting point

finish_x :

x coordinate of finish point

finish_y :

y coordinate of finish point

velocities :

*A*n N-dimensional numpy vector,
the i-th element is the velocity at the i-th layer
assume all velocities are greater than zero

__call__(self, x) :

Calculates the time for the route given by a vector of x's

<u>Input:</u>

x :

An (N-1)-dimensional numpy vector with x coordinates of cross points

<u>Return:</u>

total_time :

*T*he total travel time of the route

grad(self, x) :

Returns the function's gradient, evaluated at point <u>*x*</u>

<u>Inputs:</u>

x :

An (N-1)-dimensional numpy vector with x coordinates of cross points

<u>Return:</u>

grad :

An (N-1)-dimensional numpy vector, the gradient of the objective w.r.t to (x[1],...,x[n-1])

hessian(self, x) :

Evaluates the hessian of the objective function at x=(x[1],...,x[n-1])

<u>Input:</u>

x :

An (N-1)-dimensional NumPy vector with x coordinates of crossing points

<u>Return:</u>

hessian :

*A*n $(N-1) \times (N-1)$ NumPy matrix, the hessian of the objective evaluated at (x[1],...,x[n-1])

2. Create a function *find_fast_route* that optimizes *FastRoute* objective using Newton's method optimizer to find a fast route between the starting point and finish point. **(10pts)**

<u>Inputs:</u>
objective :
    An initialized *FastRoute* object with preset start and finish points, velocities and initialization vector.
init :
    An (N-1)-dimensional NumPy vector with x coordinates of initial cross points
alpha :
    Step size for the *NewtonOptimizer*
threshold :
    Stopping criteria $|\underline{x}_{k+1} - \underline{x}_k| <$ threshold
max_iters :
    Maximal number of iterations (stopping criteria)
<u>Return:</u>
route_time :
    The objective evaluated for the best route
route :
    An (N-1)-dimensional NumPy vector with x coordinates of the best route
num_iters :
    Number of optimizer steps until convergence

3. **Bonus (5pts):** add a function *find_alpha* that, given the starting and finish points and the number of velocity layers, finds a step size for the Newton Method. There's no analytic solution for that questions, but heuristics based on the scales may help. Explain your selection *very shortly* in a line or two in the docstring. The function's API should be as follow:

<u>Inputs:</u>
start_x :
    x coordinate of starting point
start_y :
    y coordinate of starting point
finish_x :
    x coordinate of finish point
finish_y :
    y coordinate of finish point
num_layers :

The number of velocity layers in the problem

<u>Return:</u>

alpha :

A heuristic-based step-size for the Newton's Method