45010: Optimization I **2019W**

Project 5: Genetic algorithm with QWOP

Project member(s): *Date:*

Note: *Here is a guideline for your project.*

- *The project is designed to help understand the topic, so please do not copy and paste.*
- *You are welcome to work in groups (no more than 3 in one group), but you should write it up for your submission/exercises on your own.*
- *The standard submission of project contains the following documents:*
  - *The code files*
  - *A* `README` *file*
  - *A brief writeup document answering the questions*

In your `README` *file, there are several things you need to declare:*

- *What language are you using to write the program;*
- *How to run the code(s);*
- *Contributor(s).*

You can add additional information such as possible issues, future plan in the `README` *file as well. For the writeup document, you will be answering the questions in the project.*

If you are only submitting the `ipynb` file, you can include everything in it.

5.1 Project descriptions

If you haven't heard about QWOP, here is what you should do first, open a browser on your computer, type in this link, http://www.foddy.net/Athletics.html and give it try.

This game was created in 2010. The game is to let your avatar run a 100-meter event at the Olympic Games by controlling his/her thigh and knee angles (calves).

This small game has 4 controls, namely Q,W,O,P, you can press them to control the athlete to move his thighs and calves. It is not trivial to make it work for starters. Although it is an old flash game, it reveals a fact that controlling a robot is only more difficult. Here we only have 4 controls for the game while a real life bipedal robot could have hundreds or thousands.

Playing the flash game is boring, let us write a program for it. The QWOP physics engine has been implemented in both Python and MATLAB. The function takes a list of 40 floating point numbers as input (corresponds to 20 instructions for the angle of the thighs and 20 for the angle of the knees). The output of the function is a single number, representing the final x-position of the *head* of the avatar. The more

efficiently you get the avatar to run, the further it will go and the larger the output number. The goal of this problem is for you to find an input that makes the avatar run as far as possible, which means finding a vector of length 40 that maximize the evaluation of QWOP code.

This project has been also used in some school's artificial intelligence classes or machine learning classes, e.g. Stanford CS229, University Washington CS168. So do not copy the codes from others.

5.2 Before starting the project

First, download the code (either MATLAB or Python), try to call the `sim(plan)` in Python or `qwop(plan)` in MATLAB console, here `plan` is a Python list of 40 floating numbers or a MATLAB vector of 40 floating numbers in the range of $[-1, 1]$. Both codes come with the visualization of the simulated game, you can suppress the animation by putting an additional argument of "0" in the MATLAB function, i.e. `qwop(paln, 0)`.

After you have made the code run, you should be able to print out the output from the function and the generated video (if you do not suppress).

If you cannot make this happen, you can ask help from others or email me for instructions.

5.3 Optimization

After you have tried out the game and the code, you should be able to seriously think about how to solve this problem. If you consider the distance traveled as an objective function f, you are actually doing the *minimization* for $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_{40})$.

$$f(\mathbf{x}) = -\texttt{QWOP}(\mathbf{x}) \tag{5.1}$$

subject to the constraints $-1 \leq \mathbf{x}_i \leq 1$ for each entry. The objective function can be run many times rapidly, but we are not supplied with the gradient function (actually we do not know if there is any gradient), and one important thing is: this objective function does not look convex. So it means the function might have a lot of local minimums. However a bruteforce search in 40 dimension space is of course impossible. One need to find another way out.

The "Differential Evolution" (DE) – Wikipedia link is a popular genetic algorithm and can be used in this case. One important feature of DE is: it does not use the gradient information of the function being optimized, not even requiring the function to be differentiable (the classical methods like line search and trust region are requiring this).

The algorithm's idea is keeping a population of solutions (genes) and create new solutions by combining existing ones according to very simple rules. Then keep whichever solutions that has the best score or fitness on the optimization problem. In this way, the optimization problem is more like treated as a blackbox that only provides the information of the quality of the solution.

We introduce the basic DE in the following. There are some parameters supplied by users:

1. Population size m, default 20.
2. crossover probability $p_{cr} \in [0, 1]$, default 0.7.
3. max iteration number `maxIter`, default 1000.
4. mutation weight μ, default 0.8.

1. Initially, spawn a number of candidate solutions $\{X_1, X_2, \ldots, X_m\}$ randomly in the search space $[-1, 1]^n$, we call them `population`. For each solution in the population, compute the value of objective function $\{f(X_1), \ldots, f(X_m)\}$. Set the score $F_{best} = \min_k f(X_k)$. Set iteration number $k = 0$.
2. When $k <$ `maxIter`, we repeat the following:
   (a) For each candidate X_l in the population, we do:
      i. Pick 3 other candidates X_a, X_b, X_c which are distinct from each other and different from X_l.
      ii. Pick a random index $R \in \{1, 2, \ldots, n\}$, where n is the length of X_l.
      iii. Compute the candidate's potential new mutant $Y = [y_1, \ldots, y_n]$ as follows:
         A. For each $i \in \{1, 2, \ldots, n\}$, pick a uniformly distributed random number $r_i \sim U(0, 1)$.
         B. If $r_i < p_{cr}$, then set
         $$y_i = X_a + \mu * (X_b - X_c)$$
         then clip back into search space
         $$y_i = \begin{cases} 1, & y_i > 1 \\ -1, & y_i < -1 \\ y_i, & \text{otherwise} \end{cases}$$
         otherwise set $y_i = X_{l,i}$, which is the i-th entry of X_l.
      iv. If $f(Y) < f(X_l)$ then replace X_l by its mutant Y in the population, which is $X_l \leftarrow Y$.
   (b) $k \leftarrow k + 1$, keep track of the best score $F_{best} = \min_k f(X_k)$.
3. Pick the best of the population and return.

5.4 Tasks

5.4.1 Implement the DE algorithm

Your algorithm will have signature as: `de(objFunc, mPopulation, crossover, mutant, maxIter)` The meaning of each variable is self-explanatory.

5.4.2 Run your DE against the QWOP

Try to find a good solution for the game QWOP. You can also integrate your method with other technique that your have learned in this class as well. But you should not copy others' code and result. When you submit the results, please make sure you submit your best 40 floating point vector along with the source code.

Caution: You should never change the QWOP's physics engine. Since your submission will be tested on my local environment.

5.5 Scoring

1. Your score for the implementation is 10pts.
2. Your score for the QWOP is the integer part of $\frac{d^2}{8}$ pts, d is your best result from the game. If this score is larger than 10, then it is your bonus points.

5.6 Starter code kit

No starter code is provided.

5.7 Submission

If you prefer to use the GitHub or Bitbucket, etc. to store your code, then it is OK to simply submit the repository's link on a text file (if your repository is public), otherwise, please submit all files in a zip file through Canvas.