Developing Optimisers in Python
Coursework Feedback Report

Dr Alma Rahat, and Ms Sophie Sadler

December 22, 2020

Academic year: 2020-21
Module code: CSC372/M72
Module title: Optimisation
Student ID: 956213
Marks obtained: 15/20

Summary.

Excellent work. Please read the comments below to see if or how you would be able to further improve your submission.

Specific issues and comments.

Problem Implementation. Function g2 not quite correct

Implementation of RS. You should consider returning the best solutions as well as its function value.

Proposing and Implementing a Stochastic Optimiser. For computing delta, you need to consider the f_center rather than f_best_penalty. Your implementation can be more flexible: pass an array of constraint functions, and an initial ti. Otherwise a very good implementation. Again consider returning the best solution.

Use of Constraint Handling Techniques. Excellent choices.

Performance Comparison. Good attempt but needed to use a statistical test (the Mann-Whitney U test). Your counters are not quite correct: you should reset them before running the next iteration.

Documentation and Code Quality. Solid documentation, but can add more comments in the code.

General remarks.

Here are some general comments and remarks.

- The problem at hand is a constrained optimisation problem: it has one objective function and four constraint functions.
- All three decision variables are continuous. Thus the decision space is continuous. The last variable – the number of active coils – may seem like a discrete variable. Here, think about what happens if you cut a spring in half when there are odd number of coils: you will get a fractional number of coils. So, it is possible that this variable is continuous as it is based on a continuous string or thread.
- For Random Search, it is best to use a Uniform distribution for generating candidate solutions. You can generate them at first and then evaluate one at a time. However, since the budget on function evaluations was fixed, with rejection sampling, you could generate a number of feasible solutions equal to the budget first, and evaluate them afterwards. This way you can make sure that you are hitting the target number of solutions. Note that this may take a while as you are potentially rejecting a large number of samples.
- For Simulated Annealing, one should use a Gaussian distribution to alter a central solution and generate a new candidate solution. With bound constraints, one should use a truncated distribution per dimension. Pay close attention when computing the acceptance probabilities, and see that you are following the given formula properly.
- For Genetic Algorithms, a continuous representation should be used; this is different from the binary representation used in the lab, and we covered this in the lecture. With this, a linear crossover and Gaussian mutation strategies would be most appropriate. Also, note that you do not need to reevaluate the solutions from the current population that you have already evaluated. In other words, in the population, if you have a set of solutions that were already evaluated, it is just a bit wasteful to re-evaluate them.

- For constraint handling, it is best to use rejection sampling or death penalty with RS, and static penalty (or a sophisticated penalty system) with SA (or GA). This is because RS is a naive algorithm, and it does not care what the evaluated function values are. On the other hand, SA (or GA) are smarter and will exploit the information regarding the evaluated solutions.

- In performance comparison, if your experiments were unpaired, which is the case for almost all of you, you should use Mann-Whitney-U test. Another point to remember is if you have multiple comparable datasets, you should consider adjusting α to correct for potential errors.

- Some of you noted that SA performs worse than RS. These are stochastic methods, and we cannot guarantee that one would be better than another in every context: this is known as the no-free-lunch theorem. However, often, this is because RS may have had an unfair advantage with rejection sampling: all of the 3000 function evaluations for the objective function were feasible solutions. Even then, it was possible to perform hyper-parameter optimisation to find better hyper-parameters for SA that could derive better performance. In any case, if the same budge on function evaluations were applicable on the constraint functions too, it is highly likely that SA would outperform RS. The sole reason of this part of the exercise was to make you realise that the precise set up for experiments matter, and you should pay close attention when designing experiments to ensure that you are being fair to the competing algorithms.

- The same no-free-lunch theorem applies to SA and GA too. One may be better for some problems, and vice versa. They are both applicable to discrete, continuous and mixed (i.e. some variables are discrete and some are continuous) problem domains. The success is more dependent on the function landscape and the respective hyper-parameters.