ECI 289I: Applied Evolutionary Computing

(1) (Evolution Strategies)

(a) In class we wrote the Evolution Strategy (ES) algorithm, which mutates a population of solutions using Gaussian noise with standard deviation σ. The code for this is in `L5-ES-mcommal.py`.

In this problem, implement the "1/5th rule": at the end of each generation, if more than 1/5th of children perform better than their parents, increase σ; if less than 1/5th perform better, decrease σ. You can choose how much to change σ, for example whether you add or multiply, it's up to you.

Use this new self-adaptive method on the "peaks" function like we did in class, with $\mu = 5, \lambda = 30$. Include a plot of the objective function value vs. NFE for a 10 random seeds. Also include a plot of σ vs. NFE to show the self-adaptive standard deviation value over time.

(b) Modify this algorithm to implement a $(\mu + \lambda)$ Evolution Strategy, in which parent solutions compete to survive in the next generation. Compare the convergence speed to your algorithm from Part (a). Is there a benefit to this "elitist" strategy?

(2) (Differential Evolution)

(a) Read the original DE paper from Storn and Price (1997) to get a sense for how the method works and how they set up their algorithm comparison.

In their results, "Function 2" in Table 1 is the Rosenbrock function with 2 decision variables. Using the authors' same parameter settings ($PopSize = 10, F = 0.9, CR = 0.9$), see if you can replicate their result in Table 1 (654 NFE to converge within a tolerance of 10^{-6}). Perform a few random trials and discuss.

You're welcome to use the code from class (`L6-DE-rosenbrock.py`), or any third-party library like SciPy (`scipy.optimize.differential_evolution`) or R's `DEOptim`. You may need to change a few things to report the NFE to convergence, rather than running for a fixed NFE as we have been doing in class.

(b) Expand your test from Part (a) to try multiple values of *F* and *PopSize* (leave *CR* = 0.9). Create a plot like below showing the NFE-to-converge for each combination of these parameters. What can we learn about the sensitivity of the algorithm to its parameters?