Project 3: Nelder-Mead and Genetic Algorithms

OVERVIEW

In this project, you will investigate the effectiveness of two gradient-free optimization techniques in solving a geometric optimization problem: maximizing the ratio of area to squared perimeter for an arbitrary \( n \)-sided polygon.

BACKGROUND

Since the optimization objective has a well-known solution, we can compare both algorithms to the best possible solution. Furthermore, since the objective allows us to select different values of \( n \), we can investigate how performance scales with problem size.

We will represent our \( k \)-sided polygon by storing the \( x \)- and \( y \)-coordinates of its points in a vector of size \( n = 2k \):

\[
\mathbf{x} = (x_1, x_2, \ldots, x_k, y_1, y_2, \ldots y_k)
\]

In MATLAB, we can generate a random initial polygon by using the built-in \texttt{rand} function:

\[
\texttt{xinit = rand(2*n, 1)}
\]

Your goal is to generate a number of random initial \( \mathbf{x} \) vectors for a variety of values of \( k \), and test the performance of both algorithms on each vector.

TASKS

Solve the problem using Nelder-Mead. Use the Nelder-Mead implementation you wrote for Homework 5 to optimize the objective that we wrote in class. You should be able to verify that the output converges on an optimal or near-optimal solution for small values of \( k \) (3 or 4 for example).

Now modify your Nelder-Mead implementation to terminate when total number of objective function evaluations exceeds a user-defined maximum. That is, the function should be defined as
function x = neldermead(f, xinit, h, maxfunc)

where maxfunc indicates the maximum number of function evaluations.

Implement a Genetic Algorithm using your notes from class. You will need to implement the functions select, which selects a parent from the population, crossover, which produces an offspring given two parents, and mutate, which randomly mutates an individual by making small perturbations.

Note that for the select function, we would like to have a higher probability of selecting individuals with a lower objective function score. One way to accomplish this task is to set the probability of selecting an individual \( x_j \) with probability proportional to

\[
p_j = e^{-\lambda f(x_j)}
\]

where \( \lambda \geq 0 \) is a scaling factor that determines the correlation between objective function score and probability. When \( \lambda = 0 \), all individuals have an equal chance of being selected, and as \( \lambda \to \infty \), only the very best individuals would end up getting selected.

To implement your mutate function, you should consider not only adding small random perturbations to each \( x \) or \( y \)-coordinate with a low probability, but also occasionally performing operations like swapping the order of two points within the genome, which may substantially impact fitness as well.

Finally you will need to decide on a couple of additional parameters. I would suggest a population size of 20 individuals, and retaining the top 2 individuals from generation to generation.

Like the Nelder-Mead algorithm above, you should configure your genetic algorithm to terminate when a predefined number of objective function evaluations has been performed (for instance, number of generations times number of individuals).

To initialize your population, you should use a number of copies of the initial \( x_{\text{init}} \) vector, each one slightly perturbed using the mutate function.

During development, you will want to test your genetic algorithm on small values of \( n \) to make sure that it produces reasonable results, before moving on to the next section.

Compare the two approaches for \( n = 5, 10, 20, \) and 100. For each value of \( n \), test each algorithm 10 times on the same initial randomly generated vector. For both algorithms, limit the total number of objective function evaluations to 10,000.
For each algorithm and value of $n$, document the average amount of time required to perform the optimization. Produce a bar graph summarizing the average runtimes.

Also, for each algorithm and value of $n$, produce a graph where the $x$-axis represents the iteration number (from 1 to 10,000) and the $y$-axis represents the average minimum objective function value at that iteration across all 10 runs of the algorithm.

**WHAT TO TURN IN**

You should submit all of your source code along with a PDF writeup containing your plots and answers to these questions (a few sentences on each should suffice).

- Which algorithm achieves the best performance? Explain the performance results.
- Which algorithm takes longer? Explain why.
- How does each algorithm’s output compare to the best possible objective function value? (To answer this, you will need to compute the area and perimeter of a regular $k$-sided polygon.)

Also, include representative plots of the best and worst polygons (remember to plot using `axis equal` to show the aspect ratio correctly).