Documentation for the Python Genetic Algorithm

This document provides a detailed explanation of the GA2.py, a Python script using the DEAP GA optimization library, to find the best parameters for a physiological simulation to replicate the results from Moshkforoush et al paper.

**Import Statements**

import numpy as np  
from deap import base, creator, tools, algorithms  
from numba import njit  
import pickle  
import os  
from scipy.optimize import curve\_fit

The code starts by importing the required libraries: NumPy for numerically heavy operations, DEAP, the evolutionary computation algorithms library, Numba to gain a performance boost especially in nested for loops, by just-in-time compilation, pickle for check-pointing the simulation so the work does not get lost should the simulation be interrupted, os for smart path manipulation, and scipy.optimize for curve fitting. These libraries form a solid base for the simulation and optimization tasks.

**Constants and Functions**

# Constants and Functions  
MIN\_VALUE = -80  
MAX\_VALUE = 40  
  
@njit  
def safe\_log(x): ...  
  
@njit  
def exponential\_function(x, a): ...  
  
@njit  
def exponential\_decay\_function(x, A, B): ...

To prevent numerical instabilities like division by zero or infinity, a couple of different safeguards have been put in place. Constants, ‘MIN\_VALUE’ and ‘MAX\_VALUE’ define the physical limits of the membrane potential. The ‘safe\_log’ function ensures that the logarithmic calculation happens within safe limits. ‘exponential\_function’ and ‘exponential\_decay\_function’ are the exponential calculations used in modeling the biological processes. Those functions have been compiled with ‘njit’ for performance.

**Simulate Process Function**

@njit(parallel=False)  
def simulate\_process\_modified\_v2(g\_gap\_value, Ibg\_init, Ikir\_coef, cm, dx, K\_o): ...

At the heart of the simulation is the 'simulate\_process\_modified\_v2' function. It uses a set of different inputs to create and modify the behavior of cells in an array over time. Using a discrete time model to do this, while responding to gap junction conductance, background current, and potassium current, kir channel, cell capacitance and so forth…  
  
**Genetic Algorithm Components**

# Genetic Algorithm Components  
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))  
creator.create("Individual", list, fitness=creator.FitnessMin)  
  
# Attribute, individual and population initializers  
toolbox.register("attr\_float", np.random.uniform, 0, 1)  
toolbox.register("individual", tools.initIterate, creator.Individual, ...)  
toolbox.register("population", tools.initRepeat, list, toolbox.individual)  
  
# Genetic operators  
toolbox.register("mate", tools.cxBlend, alpha=0.5)  
toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=1, indpb=0.2)  
toolbox.register("select", tools.selTournament, tournsize=3)  
toolbox.register("evaluate", objective)

The following code is responsible for initializing the genetic algorithm components Using `creator` from DEAP, I define `FitnessMin`, which is a minimization fitness evaluation, and `Individual`, the structure an individual in the population. Functions to initialize the attributes, individuals, and the population are registered in the `toolbox`. Genetic operators including mating by crossover, mutation, selection, and evaluation are also registered. All the above help to evolve the population towards optimal parameter values that minimize a specified fitness function. In this study, this function is the difference between the simulated and expected physiological behavior.

**Detailed explanation of the GA**

**Fitness and Individual Definition:**

creator.create("FitnessMin", base.Fitness, weights=(-1.0,))

A fitness evaluation scheme is defined by this line. FitnessMin means that we want the GA to minimize some quantity - in our case, the difference of the simulated data from expected physiological behavior.

creator.create("Individual", list, fitness=creator.FitnessMin)

An individual is a string of parameters defining how an individual in the population looks like and is linked to the FitnessMin previously defined to evaluate its quality. Start by:

The GA initializes the population with a set of random individuals. Each individual in the population is a set of parameters that will be used in the simulation.

toolbox.register(...)

Included in this code block are several functions necessary for the GA:

Attribute Generator: This function specifies how each attribute (parameter) is generated. In this case, we will generate a floating-point number uniformly distributed between 0 and 1.

Individual and Population Initializers: These functions define how we create one individual and how we create the population from a set of individuals.

**Genetic Operators:**

Crossover (Mating): toolbox.register("mate", tools.cxBlend, alpha=0.5)

registers a blending crossover that creates children by taking the weighted average of the parents. The alpha parameter controls the extent of the blending.

Mutation:

toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=1, indpb=0.2)

sets up Gaussian mutation, which adds a value drawn from a Gaussian distribution to the gene. mu and sigma are the mean and standard deviation of the Gaussian distribution, and indpb is the probability of each attribute being mutated.

Selection:

toolbox.register("select", tools.selTournament, tournsize=3)

The population size is defined when initializing the population. The population size determines the number of individuals in each generation. The probability of choosing an individual pair as parents for recombination equals the fitness of that individual divided by the sum of the fitness of all individuals in the population. Crossover is a method applied to two individuals that produces generalized offspring using point arrangements from both parents. From the start, all variables have only two possible values (0.5 and −0.5).

(pop = toolbox.population(n=60000)).

This is the number of times the GA loop runs, allowing the population to evolve.

(for gen in range(100):).

More generations can lead to a better solution but require more computation time.

Crossover Probability (cxpb):

This setting determines the frequency of crossover, which is how often individuals will be recombined. It is an important balance between exploration (finding new solutions) and exploitation (refining existing solutions) of the search space.

Mutation Probability (mutpb):

Sets the probability of mutation. Mutation provides the opportunity to bring new genetic material into the population, which can help maintain diversity and prevent premature convergence. and prevent premature convergence.

Tournament Size (tournsize):

For tournament selection, what decides how many individuals take part in each tournament? More intense selection can be set by making tournaments bigger and bigger, increasing the pressure on selection, and promoting stronger individuals.

**Statistical Distributions:**

The Gaussian Distribution used in mutation is adding variation based on a normal distribution, which is the most common choice in continuous parameter spaces.

**Objective/Fitness GA function details:**

**Optimization Parameters:**

The parameters to be optimized by the GA Are:

ggap, Ikier\_coef, cm, and K\_o, this code also includes their allowed values range based on literature.

param\_bounds = [

(0.1, 35), # ggap

(0.90, 0.96),# Ikir\_coef

(8, 11), # cm

(1, 8) # K\_o

]

**Simulation of ΔV Ratios:**

The core of the simulation within the objective function calculates the change in membrane potential (ΔV) across cells under certain stimuli.

The ratio of ΔV response to ΔV stimulated is computed. This ratio is crucial as it represents how the cells respond to electrical stimulation, a key aspect of electrophysiology.

The ratios are computed in the following line of the code:

D = np.abs(A[399998, 101:135] - A[99000, 101:135]) / np.abs(A[99000, 101:135])[0]

**Exponential Decay Fit:**

The goal of the objective function is to assume that the ratio if ΔV response to ΔV stimulated will ideally follow the exponential decay pattern. Or in simple words.

The parameters of this function are “optimized” to best fit the simulation data.

Reference Fitness Decay – Experimental Data from Moshksforoush et al (Evyatar et al) paper.

Unlike other simulations, the comparison of our simulation is not among simulations, but comparison is against actual experimental data.

reference\_decay = 1 \* np.exp(-0.003 \* distance\_m).

This decay function represents the expected behavior of cells based on physical experiments where sensors were attached to cells to measure stimulation and response.

The constant -0.003 in the exponent is derived from experimental observations and encapsulates the characteristic decay rate of the response over distance.

**Fitness Calculation - Matching Simulation to Experimentation:**

The goodness of fit or loss function is then calculated as the sum of squared differences between the simulated decay and the experimentally derived reference decay. This step is important to ensure the model behaves physiologically despite possible errors or inter-experimental variability.

**Optimization Objective:**

The overall objective of the objective function is to this calculate function value as low as possible. This makes the genetic algorithm search for the parameter sets which make the simulation look much closer to the observed experimental decay pattern in nature. The lower loss indicated the closer the simulated behavior was to the real behavior in the physical experiments. This would mean that the simulation is reliable and accurate.