assignment_3

May 10, 2022

1 Assignment 3

Assignment 3: Evolutionary Algorithms

Goal: Get familiar with evolutionary algorithms by implementing it and applying it to a given non-differentiable optimization task.

In this assignment, we are going to learn about evolutionary algorithms (EAs). The goal is to implement components of an evolutionary algorithm: a recombination operator, a mutation operator, and selection mechanisms, and analyze their behavior. This assignment is open to any choice of the aforementioned components as long as they are well motivated.

We are interested in optimizing a given **black-box** function that could be queried (i.e., it returns a value of the objective for given input values), but the gradient wrt the input cannot be calculated. The input to the system is a vector:

$$\mathbf{x} = [\alpha_0, n, \beta, \alpha]^\top \in [-2, 10] \times [0, 10] \times [-5, 20] \times [500, 2500].$$

The optimized function is based on the gene repressilator model. For details, please see Section 4.2 in HERE.

1.1 1. Understanding the problem

The considered problem is about finding parameter values of a model widely used in biology, namely, the gene repressilator model. This model represents a simple network in which a gene (mRNA) is produced by a protein, and then this gene is used to produce another protein. Altogether, there are 3 genes and 3 proteins that are connected as follows: $m_1 \to p_1$, $p_1 \to m_2$, $m_2 \to p_2$, $p_2 \to m_3$, $m_3 \to p_3$, $p_3 \to m_1$.

Please run the code below and spend a while on analyzing the signals in the model. Think of the osscilatory character of the signals.

If any code line is unclear to you, please read on that in numpy or matplotlib docs.

[1]:
```
import pickle
import copy
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

EPS = 1.e-7
```

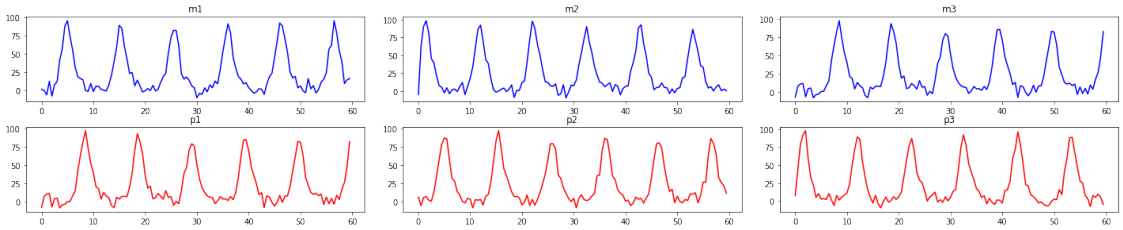
[2]:
```
# PLEASE DO NOT REMOVE!
# This is the main class for the gene repressilator model.
# There is no need to understand how it works! You can treat it as a black-box.
# It is important to realize that we can always ask this object to give us
# an evaluation of given parameter values.
class Repressilator(object):
    def __init__(self, y_real, params):
        super().__init__()
        self.y_real = y_real.copy()
        self.params = params.copy()

  # The definition of the gene repressilator model as a system of ODEs.
    def repressilator_model(self, t, y):
        m1, m2, m3, p1, p2, p3 = y[0], y[1], y[2], y[3], y[4], y[5]

        alpha0 = self.params['alpha0']
        n = self.params['n']
        beta = self.params['beta']
        alpha = self.params['alpha']

        dm1_dt = -m1 + alpha / (1. + max(p3, 0)**n) + alpha0
        dp1_dt = -beta * (p1 - m1)
        dm2_dt = -m2 + alpha / (1. + max(p1, 0)**n) + alpha0
        dp2_dt = -beta * (p2 - m2)
        dm3_dt = -m3 + alpha / (1. + max(p2, 0)**n) + alpha0
        dp3_dt = -beta * (p3 - m3)

        return dm1_dt, dm2_dt, dm3_dt, dp1_dt, dp2_dt, dp3_dt

  # A numerial solver for the model (here we use Runge-Kutta 4.5)
    def solve_repressilator(self):
        # we need to use lambda function if we want to pass some parameters
        solution = solve_ivp(lambda t, y: self.repressilator_model(t, y),
                             t_span=(self.params['t0'], self.params['t1']),
                             y0=self.params['y0'],
                             method='RK45', t_eval=self.params['t_points'])
        y_points = np.asarray(solution.y)
        return self.params['t_points'], y_points

  # An auxiliary function: setting parameters.
    def set_params(self, x):
        self.params['alpha0'] = x[0]
        self.params['n'] = x[1]
        self.params['beta'] = x[2]
        self.params['alpha'] = x[3]

  # Calculating the objective function.
```

```
 # Here, we use the Euclidean distance between the real data and the synthetic␣
↪data.
    @staticmethod
    def loss(y_real, y_model):
        # we assume only m's are observed!
        y_r = y_real[0:3]
        y_m = y_model[0:3]
        if y_r.shape[1] == y_m.shape[1]:
            return np.mean(np.sqrt(np.sum((y_r - y_m)**2, 0)))
        else:
            return np.infty

    def objective(self, x):
        if len(x.shape) > 1:
            objective_values = []
            for i in range(x.shape[0]):
                xi = x[i]
                self.set_params(xi)

                _, y_model = self.solve_repressilator()
                objective_values.append(self.loss(self.y_real, y_model))

            objective_values = np.asarray(objective_values)
        else:
            self.set_params(x)

            _, y_model = self.solve_repressilator()
            objective_values = self.loss(self.y_real, y_model)

        return objective_values
```

[3]:
```
# PLEASE DO NOT REMOVE!
# Initialize the problem.
# Here we set the real parameters and generate "real" data. To make the problem
# more realistic, we add a small Gaussian noise.
params = {}
params['alpha0'] = 1.1
params['n'] = 2.9
params['beta'] = 5.5
params['alpha'] = 500
params['t0'] = 0.
params['t1'] = 60.5
params['t_points'] = np.arange(0, 60, 0.5)
params['x0'] = np.asarray([[5.64167522, 2.07180539, 3.56690274, 7.0015145 ]])
params['y0'] = np.asarray([0.0, 0.0, 0.0, 2.0, 1.0, 3.0])

# Generate "real" data
```

```
r = Repressilator([0], params)
_, y_real = r.solve_repressilator()
del(r) # we remove the object, just in case
y_real = y_real + np.random.randn(*y_real.shape) * 5. # add noise!
```

[4]:
```
# PLEASE DO NOT REMOVE
# Here, we plot all signals in the model.
# Note that later on, we assume that only gene expression (i.e., m1, m2, m3)
# are observed (blue plots) and proteins (i.e. p1, p2, p3) are unobserved (in␣
 ↪red).
t = params['t_points']

fig_data, axs = plt.subplots(2,3,figsize=(20, 4))
fig_data.tight_layout()

for i in range(2):
    for j in range(3):
        if i == 0:
            title = 'm'
            color = 'b'
        else:
            title = 'p'
            color = 'r'

        axs[i,j].plot(t, y_real[2*i+j], color)
        axs[i,j].set_title(title + str(j+1))
```



1.2 2. Evolutionary Algorithms

In this assignment, you are asked to implement an evolutionary algorithm (EA). Please take a look at the class below and fill in the missing parts. Please keep in mind that you are free to choose any operations as lons as they are appropriate for evolutionary algorithms.

NOTE: Please pay attention to the inputs and outputs of each function.

Question 1 (0-0.5pt): Do you use a mutation operator? If yes, please explain in detail your choice. If not, please explain why not.

Answer: I do use a mutation operator. The mutation operator I have chosen adds samples from a

Gaussian distribution with mean around **0** and a diagonal covariance matrix which is proportional to the amplitude of each parameter. This way the mutations to every paramater of each child are indiviualized as opposed to, e.g., generating a mutation for n that is the same as a mutation for α. This makes sense considering how different the scales are. The covariance matrix is also multiplied by the standard deviation hyperparameter which adds the ability to regulate how much the points are mutated.

Question 2 (0-0.5pt): Do you use a cross-over operator? If yes, please explain in detail your choice. If not, please explain why not.

Answer: I do use a cross-over operator. The way I have it implemented is that a fraction of the parents, best ones, is selected and then at random, features from those parents are selected and again at random inserted in into the worst parents. Finally, I clone the individuals based on how many parents there are so that there would always be more children than the population size and since each child gets different mutations later, every individual, even if it's a copy, is potentially valuable. The reason for cloning the parents to create more children is that so there would alwasy be more children than the population size and in theory the next generation would not have to include any parents.

Question 3 (0-0.5pt): What kind of selection mechanism do you use? Please explain in detail and motivation your choice.

Answer: For survival selection, I use the round-robin tournament method where for each individual (combination of children and parents), I check its evaluation against a fraction of the population and count the number of "wins" where the evaluation is higher. Then the individuals with the best winning scores are selected. This makes up 80% of the new population, the other 20% are selected at random to some diversity and help find a better optimum.

For parent selection I simply use a random uniform selection as the parents are already the best individuals of their generation. Selecting the best ones might lead to greater dominance and the convergance may be inhibited and stopped at a bad local optimum.

Question 4 (0-0.5pt): Do you use any other operation? If yes, please explain and motivate your choice.

Answer: I have implemented a dynamic standard deviation system where the standard deviation would decrease linearly for 80% of the generations through a specified range and then stay at the minimum for the remaining 20% of the epochs. This is very similar to the temperature in simulated annealing and is done for the same purpose. The points mutated early are more spread out and help find better optima and then at the end the lower standard deviation makes sure that more points are mutated closer so as to get to the very best point in the (local) optimum.

Question 5 (0-0.5-1pt): Please provide a pseudo-code for your evolutionary algorithm. Please try to be as formal as possible!

Answer:

```
define step(points_old, values_old):
    parents, values = select_parents(points_old, values_old)
    children = recombination(parents, values)
    children = mutate(children)
    children_values = evaluate(children)
    points_new, values_new = survivor_selection(children, children_values, points_old, values_
```

```
    update_std()

    return points_new, values_new


define select_parents(parents, values):
    return specified number of parents and corresponding values selected at random

define recombination(parents, values):
    sort the parents based on values and split them into best and worst
    select features from best parents at random
    loop through worst parents and insert features at random
    concatenate best and worst parents and clone them so that length(parents) > population_size
    return the new points

define mutate(points):
    scale = upper_bounds - lower_bounds
    create covariance matrix E from scale
    mutations = sample multivariate normal distribution with E for every child
    return points + mutations

define survivor_selection(children, children_values, points_old, values_old):
    points = concatenate children and points_old
    values = concatenate children_values and values_old

    for every value in values
        select 0.1*population_size random values and and compare them with value
        append the number of times value is larger

    sort points and values based on how many wins each point has
    return the best population_size points and values

define update_std():
    if dynamic_std is true
        std = next value from std list
```

[5]:
```
#=========
# GRADING:
# 0
# 0.5 pt if code works but some functions are incorrect and it is badly explained
# 1.0 pt if code works but some functions are incorrect
# 2.0 pt if code works but it does not correspond to the description above and it is badly explained
# 2.5 pt if code works and it is well explained, but it does not correspond to the description above
```

```
# 3.0 pt if code works and it is aligned with the description about, but it is␣
 ↪badly explained
# 3.5 pt if code works and it is aligned with the description about, and it is␣
 ↪well explained
# 4.0 pt if code works, it is as it was described, it is well explained, and␣
 ↪the proposed operations are beyond the lecture!
#=========
# Implement the Evolutionary Algorithm (EA) algorithm.
# It is equivalent to implementing the step function.
class EA(object):
    def __init__(self, repressilator, pop_size, std=20, parents=None,␣
 ↪bounds_min=[], bounds_max=[]):
        self.repressilator = repressilator
        self.pop_size = pop_size
        self.std = std
        self.parents = parents if parents is not None else pop_size
        self.bounds_min = np.asarray(bounds_min)
        self.bounds_max = np.asarray(bounds_max)
        self.stds = []
        self.t = None

    # Turn on diminishing standard deviation which linearly goes down and stays␣
 ↪at the minimum for the final 20% of generations
    def set_dynamic_std(self, std_high, std_low, epochs):
        self.stds = np.concatenate([np.linspace(std_high, std_low, int(epochs*0.
 ↪8)),
                                    int(np.ceil(epochs*0.2))*[std_low]])
        self.t = 0

    # Updating standard deviation when using dynamic standard deviation
    def __update_std(self):
        if self.t is not None:
            self.std = self.stds[self.t]
            self.t += 1

    # Select and return a random number of parents equal to parents␣
 ↪hyperparameter
    def parent_selection(self, x_old, f_old):
        ind = np.random.choice(self.pop_size, self.parents, replace=False)
        return x_old[ind], f_old[ind]

    # Takes features from best parents and inserts them into the worst ones
    def recombination(self, x_parents, f_parents):
        best_parents, worst_parents = self.
 ↪__get_best_and_worst_parents(x_parents, f_parents)
        crossover_features = self.__get_crossover_features(best_parents)
```

```
        worst_parents = self.__insert_features(worst_parents,␣
↪crossover_features)
        x_parents = np.concatenate([best_parents, worst_parents])
        x_parents = np.tile(x_parents, (int(np.ceil(self.pop_size/self.
↪parents))+1, 1))

        return np.clip(x_parents, self.bounds_min, self.bounds_max)

    # Splits parents into 10% best and 90% worst
    def __get_best_and_worst_parents(self, x, f):
        x = x[np.argsort(f)]
        threshold = int(np.ceil(self.parents/10))
        return x[:threshold], x[threshold:]

    # Extracts random features from the best parents
    def __get_crossover_features(self, best):
        crossover_features = np.zeros(best.size)
        ind = np.random.choice(best.size, self.parents//10)
        crossover_features[ind] = best.reshape(-1)[ind]

        return crossover_features.reshape(best.shape)

    # Inserts the random features to the worst parents
    def __insert_features(self, parents, features):
        for i, parent in enumerate(parents):
            feature = features[np.random.randint(features.shape[0])]
            parent[feature != 0] = 0
            parents[i] = parent + feature

        return parents

    # Add some noise from a multivariate normal distribution to every child
    def mutation(self, x_children):
        scale = self.bounds_max - self.bounds_min
        scale /= np.max(scale) # Normalize the scale
        mutation = np.random.multivariate_normal(
            np.zeros(4), np.diag(scale)*self.std,
            size=x_children.shape[0])

        return np.clip(x_children + mutation, self.bounds_min, self.bounds_max)

    # Select 80% of survivors based on the round robin tournament selection␣
↪method
    # The other 20% are selected randomly to help escape local optima
    def survivor_selection(self, x_old, x_children, f_old, f_children):
        x = np.concatenate([x_old, x_children])
        f = np.concatenate([f_old, f_children])
```

```
        scores = self.__winning_scores(f)

        ind = np.argsort(scores)
        x = x[ind]
        f = f[ind]

        threshold = int(np.ceil(self.pop_size*0.8))
        ind = np.random.choice(range(threshold, x.shape[0]), int(0.2*self.
↪pop_size), replace=False)
        return np.concatenate([x[:threshold], x[ind]]), np.concatenate([f[:
↪threshold], f[ind]])

    # For every individual, finds the number of wins against random opponents
    def __winning_scores(self, f):
        scores = []
        for f_i in f:
            ind = np.random.choice(x.shape[0], int(np.ceil(self.pop_size/10)),␣
↪replace=False)
            scores.append(sum([f_i >= i for i in f[ind]]))

        return np.asarray(scores)

    # Evaluation step: DO NOT REMOVE!
    def evaluate(self, x):
        return self.repressilator.objective(x)

    def step(self, x_old, f_old):
        x_parents, f_parents = self.parent_selection(x_old, f_old)
        x_children = self.recombination(x_parents, f_parents)
        x_children = self.mutation(x_children)
        f_children = self.evaluate(x_children)
        x, f = self.survivor_selection(x_old, x_children, f_old, f_children)
        self.__update_std()

        return x, f
```

NOTE 1

Since this assignment allows you implementing your own operations, this is difficult to prepare a code for that. Therefore, please use the code below to find the best set of your hyperparameters in a separate file, and then present your analysis for the best values of the hyperparameters here.

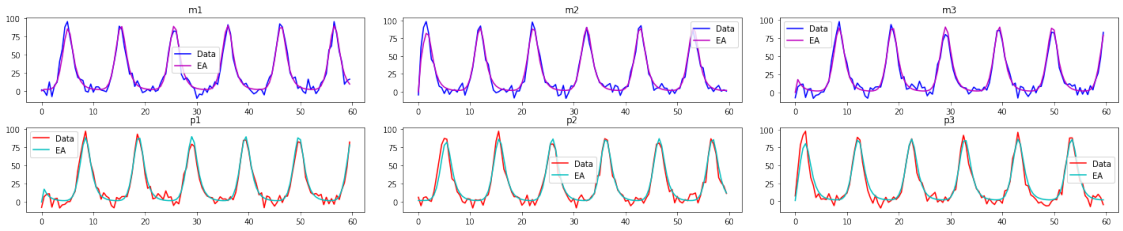
NOTE 2

Additionally, please do try various population sizes (25, 50, 100, ...). You will be asked about it later.

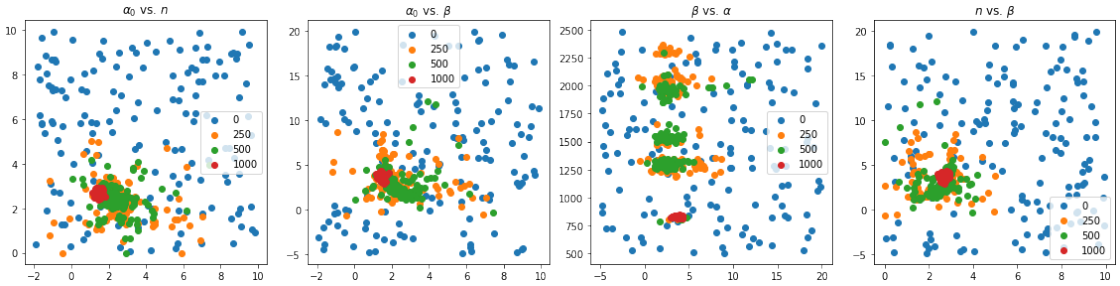
[6]:
```
# PLEASE DO NOT REMOVE!
num_generations = 1000 # if necessary, please increase the number of generations
pop_size = 150
bounds_min = [-2., 0., -5., 500.]
bounds_max = [10., 10., 20., 2500.]
#-------
# PLEASE FILL IN!
# Your hyperparams go here.
std_high = 500
std_low = 4
parents = max(1, pop_size//4) # divisor must be greater or equal to population␣
 ↪size
#-------

# Initialize the repressilator
repressilator = Repressilator(y_real, params)

#-------
# PLEASE FILL IN!
# Your object goes here:
ea = EA(repressilator, pop_size, parents=parents,
        bounds_min=bounds_min, bounds_max=bounds_max)
ea.set_dynamic_std(std_high, std_low, num_generations)
#-------

# Init the population
x = np.random.uniform(low=bounds_min, high=bounds_max, size=(pop_size, 4))
f = ea.evaluate(x)

# We want to gather populations and values of the best candidates to further
# analyze the algorithm.
populations = []
populations.append(x)
f_best = [f.min()]

# Run the EA.
for i in range(num_generations):
    if i % int(num_generations * 0.1) == 0:
        print('Generation: {}, best fitness: {:.2f}'.format(i, f.min()))
    x, f = ea.step(x, f)
    populations.append(x)
    if f.min() < f_best[-1]:
        f_best.append(f.min())
    else:
        f_best.append(f_best[-1])
print('FINISHED!')
```

```
Generation: 0, best fitness: 43.78
Generation: 100, best fitness: 15.42
Generation: 200, best fitness: 13.40
Generation: 300, best fitness: 11.81
Generation: 400, best fitness: 11.36
Generation: 500, best fitness: 11.36
Generation: 600, best fitness: 10.68
Generation: 700, best fitness: 10.68
Generation: 800, best fitness: 10.13
Generation: 900, best fitness: 9.10
FINISHED!
```

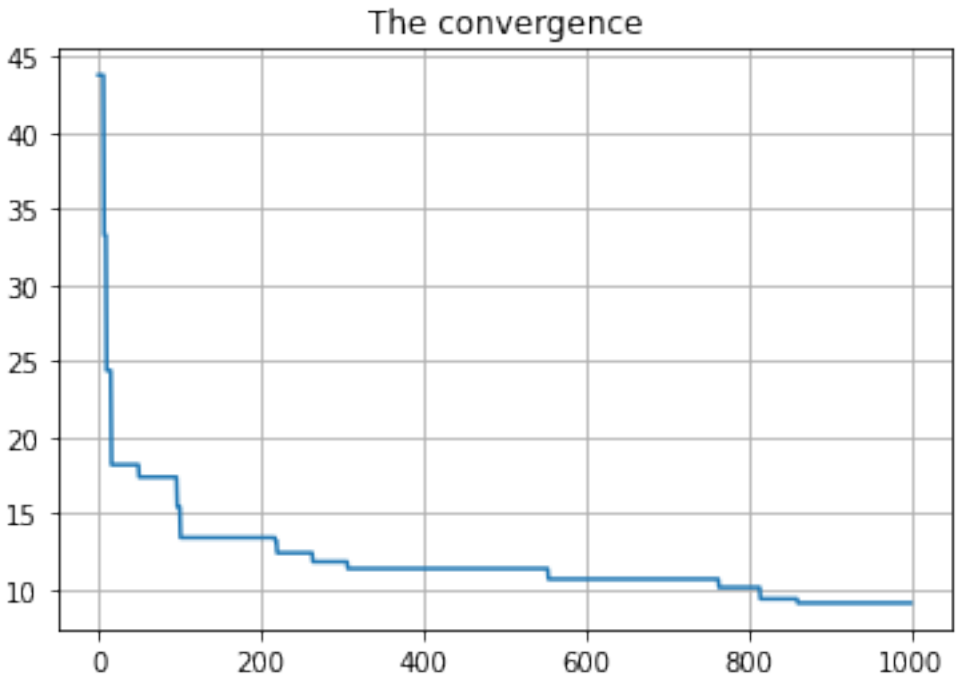
[7]:
```python
# PLEASE DO NOT REMOVE
# Generate signals for the best performing parameter values
repressilator.set_params(x[f.argmin()],)
t, y_best = repressilator.solve_repressilator()

# Create plots
fig_ea, axs = plt.subplots(2,3,figsize=(20, 4))
fig_ea.tight_layout()

for i in range(2):
    for j in range(3):
        if i == 0:
            title = 'm'
            color = 'b'
            color_m = 'm'
        else:
            title = 'p'
            color = 'r'
            color_m = 'c'

        axs[i,j].plot(t, y_real[2*i+j], color, label='Data')
        axs[i,j].plot(t, y_best[2*i+j], color_m, label='EA')
        axs[i,j].set_title(title + str(j+1))
        axs[i,j].legend()
```



[8]:
```
# PLEASE DO NOT REMOVE
# Generate scatterplots for the populations
gens = [0, num_generations//4, num_generations//2, num_generations]
fig_ea_pops, axs = plt.subplots(1,4,figsize=(16, 4))
fig_ea_pops.tight_layout()

for i in range(len(gens)):
    pop_i = populations[gens[i]]
    axs[0].scatter(pop_i[:,0], pop_i[:,1], label=str(gens[i]))
    axs[1].scatter(pop_i[:,0], pop_i[:,2], label=str(gens[i]))
    axs[2].scatter(pop_i[:,2], pop_i[:,3], label=str(gens[i]))
    axs[3].scatter(pop_i[:,1], pop_i[:,2], label=str(gens[i]))
axs[0].legend(), axs[1].legend(), axs[2].legend(), axs[3].legend()
axs[0].set_title(r'$\alpha_0$ vs. $n$')
axs[1].set_title(r'$\alpha_0$ vs. $\beta$')
axs[2].set_title(r'$\beta$ vs. $\alpha$')
axs[3].set_title(r'$n$ vs. $\beta$')

plt.show()
```



[9]:
```
# PLEASE DO NOT REMOVE
# The convergence plot: the best value of the fitness function for each␣
 ↪generation
plt.plot(range(len(f_best)), f_best)
plt.title('The convergence')
plt.grid()
```



1.3 4. Final remarks: Analysis

Eventually, please answer the following last questions that will allow you to conclude the assignment.

Question 6 (0-0.5pt): After analyzing the results, do you think your algorithm discovered the real values of the parameters? Why? Please indicate which plots indicate that? If not, what is a potential explanation?

Answer: As the real graph has noise added to it, it's hard to say if the real values have been discovered. The peaks and troughs overlap very well and when I tested what the graphs look like without noise, I found that the values my algorithm discovered were very close and that there was an almost exact overlap. The reason why it may not find the very right values is because of how many local optima there appear to be and the algorithm may easily get stuck. Given enough time though, the random nature of mutations should guarantee that the global optimum is found.

Question 7 (0-0.5pt): How the population influences the performance of evolutionary algorithms? What population sizes did you use?

Answer: I used population sizes of 25, 50, 75, 100, 150, 200, and 250. For each size, I ran the algorithm 5 times with other hyperparameters frozen and for a few population sizes I also tested different environments. The larger values found a good solution fast and then got stuck in local minima while for the lower values, it usually took a bit longer to reach a good solution but after 50 generations, almost always some local minimum was found no matter the population size. When running for a long time (No. of generations ≥ 1000) or using the cooled standard deviation mechanism, the larger populations sizes appear likelier to wriggle out of current local optimum and find marginally better points.

Question 8 (0-0.5pt): What are the advantages of your approach?

Answer: It uses a plethora of evolutionary mechanisms which allow for greater tinkering and adjusting to help with the search for the global or at least an adequate local optimum.

Question 9 (0-0.5pt): What are the drawbacks of your approach?

Answer: Because of the large number of mechanisms at work it is substantially slower and the search for the right hyperparameter and configurations is very tedious and time-consuming.

Question 10 (0-0.5pt): How could you improve convergence speed of your algorithm? Please provide very specific answer, ideally supported with literature and mathematical formulas.

Answer: Lowering the standard deviation, selecting survivors just based on evaluation, and increasing the population size are all tested concepts which have proven to converge faster. Despite that, I choose to do the opposite of optimizing for fast convergence because the faster the algorithm converges, the worse is the optimum it converges to. To improve convergence speed without impairing the ability to find a good solution a deeper understanding of the problem must be acquired. For example if you know that the parameter n is especially sensetive to small changes, perhaps a greater emphasis can be put on exploring larger and smaller mutations of n.

Question 11 (0-0.5pt): How does an EA compares with the Metropolis-Hastings algorithm? What are the similarities? What are the differences?

Answer: Both algorithms generate new points which are evaluated and either accpeted or rejected. But that's where the similarities stop. MH is used for sampling, not optimization and vice versa for EA. EAs are also far more modular and there are a greater number of components that can be used such as mutation and cross-over.