

CP468 AI – TERM PROJECT

DECEMBER 01, 2019

MAX NIEBERGALL 160623100

SRIRAM VASUTHEVAN 170408710

JUSTIN HARROTT 161449800

RASHA NASRI 164161160

How to Run

To run this program, you must have python 3.7 or higher installed. <https://www.python.org/downloads/>

Next, install matplotlib using pip. <https://pip.pypa.io/en/stable/quickstart/>

Then, you can run either `dejong_test.py`, `himmelblau_test.py`, or `rosenbrock_test.py` to see the test's results display.

This calls the SGA function in `simple_genetic_algorithm.py` and performs mating, reproduction and mutation operations

on as many generations of a population of specified sizes. The program will plot each generation's fittest member's fitness measure.

This graph can be saved as a .png file.

#Note

It is possible to compare 2 different bit mutation methods by alternating the commented out sections in the `attempt_mutation()` function.

Design Decisions

The only data structure which we used were Python Lists. Lists are dynamic arrays that take $O(n)$ time to initialize, $O(1)$ time to update values and

$O(1)$ amortized time to append to the end of the list (only used for the plot).

As all lists are of fixed size, we could have used Numpy Arrays but we wanted to use vanilla Python as much as possible. The entire script is written in vanilla Python except for plotting.

We used Python modules to make the code easier to follow. Each objective function is in its own module, along with constants that the SGA function uses. `Functions.py` contains several helper functions that are not directly part of the SGA. The SGA function itself calls a different function for each main step: Reproduction, crossover and mutation.

Benchmark Test Results

Comparing 2 methods of character mutation

Method 1

Chooses one bit randomly if random chance falls into the probability that the string should mutate, if `random.random() < probability_of_mutation`.

Method 2

Bit by bit, perform mutation if random chance falls into the probability that the bit should mutate, if `random.random() < probability_of_mutation`.

All graphs are plotted as Fitness Measures (Y-axis [0, 1]) against Generations (X-axis [0, 1000]).

Several tests have shown that method 1 is capable of occasionally outperforming method 2, method 2 is far more consistent and is therefore more reliable.

```

1      #simple_g
2      enetic_algori
3      thm.py
4      from typing import List
5      import random
6      from functions import general_decoder
7      import
8      matplotlib.pyplot
9      t as plt
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35

```

LENG
TH_0
F_DE
CIIMA
L=4

def initialize(pop_size: int, alnum_set: List[str],
var_string_length) -> List[str]: 11 """
Generates pop_size of random strings with characters of their
alphanumeric character set.

Args:
pop_size (int): Number of strings to be generated
alnum_set (List[str]): Valid characters to generate random
string from
var_string_length (int): Number of characters expected to be in
each string

Returns:
(List[str]): List of pop_size random strings made with
characters of their alphanumeric
character set

"""

return ["".join(random.choices(alnum_set, k=var_string_length)) for
_ in
range(pop_size)]

def perform_reproduction(population,
inverse_fitness_func) -> List: 27 """
Takes in a population of strings and probabalistically
candidates for mating

based on the
relative fitness of each string (i.e. the more fit members of
the population will be picked more often
and will therefore make up a bigger portion
of the mating pool). 31

Args:
population (List[str]): List of strings that make up the mating
pool
inverse_fitness_func (<function>): Benchmark function with
known global minima; returned values
closer to zero mean the
given arguments are

```

36                                     closer to optimum
37                                     Args:
38                                     (str): Alphanumeric string
39                                     representing number system
40                                     value(s)
41                                     Returns:
42                                     (float): floating point
43                                     value between 0 to 1;
44                                     results closer to zero
45                                     are closer to
46                                     optimization
47
48     Returns:
49     (List[str]): List of strings that make up the new mating
50     pool generation 43 """
51
52     #compile a list of floating point values, numbers closer to 0 have
53     a higher fitness
54     inverse_fitnesses = [inverse_fitness_func(s) for s in population]
55     min_inv_fit = min(inverse_fitnesses)
56     max_inv_fit =
57     max(inverse_fitnesses) 49
58     #compile a list of inverse fitness values that are normalized
59     to [0,1] with regard to
60     #the range of the population's fitness measures, numbers closer
61     to 0 have a higher fitness
62     inverse_fitnesses = [(X-min_inv_fit)/(max_inv_fit-min_inv_fit+1)
63     for X in
64     inverse_fitnesses] #adding 1 to avoid division by zero
65
66     #invert list of normalized inverse fitness values so that
67     numbers closer to 1 have a higher fitness

```

```

55     fitnesses = [1 / ((s)+1) for s in inverse_fitnesses]
56
57     #sum up all found fitness measures
58     total_fitness = sum(fitnesses)
59
60     #compile a list of each fitness measure's proportion to the sum of all found
    fitness measures
61     #of the mating pool; to be used as a probability that the candidate will mate
62     probabilities_of_reproduction = [fit / total_fitness for fit in fitnesses]
63
64     #create list of randomly chosen strings that are weight biased
65     return random.choices(population=population, weights=probabilities_of_reproduction,
    k=len(population))
66
67
68 def perform_mating(population):
69     """
70     Takes a list of strings, returns a list of strings after potential mating operations.
71
72     Args:
73         population (List[str]): List of strings that make up the mating pool
74     Returns:
75         (List[str]): List of strings that make up the now potentially modified mating
        pool
76     """
77
78     new_pop = []
79
80     while (len(population) > 0):
81         s1 = population.pop(random.randint(0, len(population) - 1))
82         s2 = population.pop(random.randint(0, len(population) - 1))
83
84         s1p, s2p = crossover_pair(s1, s2)
85         new_pop.append(s1p)
86         new_pop.append(s2p)
87
88     return new_pop
89
90
91 def crossover_pair(s1, s2):
92     """
93     Takes two strings for crossover, randomly determines a crossover point
94     between 1 and len(s1)-1 then performs crossover of character data at crossover point.
95     Assumes len(s1) = len(s2)
96
97     Args:
98         s1 (str): string for mating crossover
99         s2 (str): string for mating crossover
100
101     Returns:
102         s1p (str): string from mating crossover
103         s2p (str): string from mating crossover
104     """
105     crossover_point = random.randint(1, len(s1) - 2)
106     s1p = s1[:crossover_point] + s2[crossover_point:]
107     s2p = s2[:crossover_point] + s1[crossover_point:]
108
109     return (s1p, s2p)
110

```

```
111 def perform_mutations(population, probability_of_mutation, alnum_set):
112     """
113     Takes a list of strings, returns a list of strings after potential mutation
114     operation.
115     Args:
116         population (List[str]): List of strings that make up the mating pool
117         probability_of_mutation (float): decimal number between 0 and 1 that represents
```

liklihood

```

118                                     that a character will mutate into another
119                                     character from
119                                     its alphanumeric character set
120     alnum_set (List[str]): Valid alphanumeric characters to genrate number-system
120     value-strings from
121 Returns:
122 (List[str]): List of strings that make up a potentially mutated mating pool
123 """
124
125     return [attempt_mutation(s, probability_of_mutation, alnum_set) for s in population]
126
127
128 def attempt_mutation(s, probability_of_mutation, alnum_set):
129     """
130     Takes a string and probabilisticly performs character mutation
131
132     Args:
133         s (str): string to mutate
134         probability_of_mutation (float): decimal number between 0 and 1 that represents
134         the liklihood
135                                     that a character will mutate into another
136                                     character from
137                                     its alphanumeric character set
137     alnum_set (List[str]): Valid alphanumeric characters to genrate number-system
137     value-strings from
138 Returns:
139     None
140     """
141
142     #bit flipping
143     #on average, method 2 seems to outperform method 1
144
145     # #method 1
146     # #choose one bit randomly if random chance falls into the probability that the
146     string should mutate
147     # if random.random() < probability_of_mutation:
148     #     bit_to_flip = random.randint(0, len(s) - 1)
149     #     s = list(s)
150     #     s[bit_to_flip] = random.choice(alnum_set)
151     #     return "".join(s)
152     # else:
153     #     return s
154
155     #method 2
156     #bit by bit, perform mutation if random chance falls into the probability that the
156     bit should mutate
157     for i in range(len(s)):
158         if random.random() < probability_of_mutation:
159             s = s[:i] + str(alnum_set[random.randint(0, len(alnum_set)-1)]) + s[i +
159             1:]
160     return s
161
162 # program starts here:
163 #
164 -----
165 -----
166
167 def SGA(test_function, pop_size, alnum_set, var_string_length, variable_length,

```



```
166 domain_min, domain_max,  
167 number_of_generations, probability_of_mutation):  
168     """  
169     Simple Genetic Algorithm that finds global minima of test functions through  
170     generational mating,  
    reproduction and bit mutations while printing out each generation's performance  
    results.
```

```

171 Args:
172     test_function (<function>): Benchmark function that has known optimum values
173     (global min)
174     pop_size (int): the number of strings to create for population
175     alnum_set (List[str]): Valid alphanumeric characters to generate number system
176     value-strings from
177     var_string_length (int): character length of string that contains one or more
178     number system value-string variables
179     variable_length (int): character length of one number system value-string
180     variable
181     domain_min (Union[float,int]): min value of operational domain
182     domain_max (Union[float,int]): max value of operational domain
183     number_of_generations (int): number of times to mate / mutate the population in
184     the attempt to hone in
185     on the optimal input values for the given
186     test_function
187     probability_of_mutation (float): decimal number between 0 and 1 that represents
188     the likelihood
189     that a character will mutate into another
190     character from
191     its alphanumeric character set
192 Prints:
193     table: generational performances, avoiding repeat max performance levels
194     between contiguous generations
195 """"
196
197 #initialize a random population of pop_size values to be the starting point for
198 optimization attempt
199 #returns string with (var_string_length * pop_size) number of characters
200 population = initialize(pop_size, alnum_set, var_string_length)
201
202 #small anonymous function used to find fitness measures of a member of a mating
203 pool population,
204 #determined by the given benchmark test function
205 inverse_fitness = lambda string: test_function(*general_decoder(string,
206 variable_length, domain_min, domain_max, len(alnum_set)))
207
208 #print performance of population
209 #header
210 print("\nTested population size: ", pop_size, " Number of generations: ",
211 number_of_generations)
212 pad = str((4 + LENGTH_OF_DECIMAL) * int(var_string_length / variable_length))
213 print(("n{:<16s}{:}<" + pad + "s}\t {:}").format("Generation", "Strongest
214 Candidate", "Fitness"))
215 print("="*80)
216
217 #print off generational performances, avoiding repeat performance levels between
218 contiguous generations
219 last_fit_individual = fittest_individual = [] #coordinate values
220 last_max_fit = 0 #fitness value
221 new_fit = True #is the found fitness value different than the last
222 max_fitness_list = [] #list of all found fitness values to be used for a graph
223 global_max_found = (0, [], 0) #to record the overall best found fitness measure
224 form all generations
225 first_gen_repeat = last_gen_repeat = 0 #to keep track of how many generations have
226 had repeated max_fitness measures
227
228 for i in range(number_of_generations):

```

```
213
214     #create new generation of population
215     population = perform_reproduction(population, inverse_fitness)
216     population = perform_mating(population)
217     population = perform_mutations(population, probability_of_mutation, alnum_set)
218
219     #determine the max fitness measure of this generation
220     max_fitness = max(1/(inverse_fitness(m)+1) for m in population)
```

```

221     max_fitness_list.append(max_fitness)
222
223     #determine the string variable values that have max_fitness of this generation
224     for m in population:
225         if 1/(inverse_fitness(m)+1) == max_fitness:
226             fittest_individual = general_decoder(m, variable_length, domain_min,
227                 domain_max, len(alnum_set))
228
229     #case: if this generation is the last, or it is more fit than its predecessor
230     if (i == number_of_generations - 1) or not (fittest_individual ==
231         last_fit_individual):
232         #case: if is the new fittest member after repeated max performance
233         if (first_gen_repeat != last_gen_repeat) and (last_gen_repeat -
234             first_gen_repeat > 1) :
235             print("\n\tFor generation {} to {}, the max fitness level was
236                 {:. "+str(LENGTH_OF_DECIMAL)+"f}.\n").format(first_gen_repeat,
237                     last_gen_repeat, last_max_fit))
238
239             #print generation's performance results
240             print("{:<16d}[{:<"+ pad + "s}]\t {:>}"").format(i, " ".join(["{:. "+
241                 str(LENGTH_OF_DECIMAL) + "f}", "]).format(x) for x in fittest_individual]),
242                 max_fitness))
243
244             last_fit_individual = fittest_individual
245             last_max_fit = max_fitness
246
247             #record the overall best found fitness measure form all generations
248             if max_fitness > global_max_found[2]:
249                 global_max_found = ("Gen: " + str(i), fittest_individual, max_fitness)
250
251             first_gen_repeat = last_gen_repeat = i
252             new_fit = True
253
254             #case: first repeat of same fittest member between generations
255             elif last_fit_individual == fittest_individual:
256                 new_fit = False
257                 last_gen_repeat = i
258
259     print("=="*80)
260     print("Highest fitness acheived by:\n", global_max_found)
261     print("=="*80)
262     print("")
263     plt.plot(max_fitness_list)
264     plt.show()

```

```

1  #simple_genetic_algorithm.py
2
7  from typing import List
8  import random
9  from functions import general_decoder
10 import matplotlib.pyplot as plt
11
12 LENGTH_OF_DECIMAL=4
13
14 def initialize(pop_size: int, alnum_set: List[str], var_string_length) -> List[str]:
15     """
16     Generates pop_size of random strings with characters of their alphanumeric character
17     set.
18
19     Args:
20         pop_size (int): Number of strings to be generated
21         alnum_set (List[str]): Valid characters to generate random string from
22         var_string_length (int): Number of characters expected to be in each string
23     Returns:
24         (List[str]): List of pop_size random strings made with characters of their
25         alphanumeric character set
26     """
27     return ["".join(random.choices(alnum_set, k=var_string_length)) for _ in
28             range(pop_size)]
29
30 def perform_reproduction(population, inverse_fitness_func) -> List:
31     """
32     Takes in a population of strings and probabilistically candidates for mating based
33     on the relative fitness of each string (i.e. the more fit members of the population will
34     be picked more often and will therefore make up a bigger portion of the mating pool).
35
36     Args:
37         population (List[str]): List of strings that make up the mating pool
38         inverse_fitness_func (<function>): Benchmark function with known global minima;
39         returned values closer to zero mean the given arguments are closer to optimum
40
41     Args:
42         (str): Alphanumeric string representing number system value(s)
43     Returns:
44         (float): floating point value between 0 to 1; results closer to zero are closer to optimization
45     Returns:
46         (List[str]): List of strings that make up the new mating pool generation 43
47     """
48
49     #compile a list of floating point values, numbers closer to 0 have a higher fitness
50     inverse_fitnesses = [inverse_fitness_func(s) for s in population]
51     min_inv_fit = min(inverse_fitnesses)
52     max_inv_fit = max(inverse_fitnesses)
53
54     #compile a list of inverse fitness values that are normalized to [0,1] with regard

```

```
54         #the range of the population's fitness measures, numbers closer to 0 have a higher fitness
55         inverse_fitnesses = [(X-min_inv_fit)/(max_inv_fit-min_inv_fit+1) for X in
inverse_fitnesses]
53         #adding 1 to avoid division by zero
56         #invert list of normalized inverse fitness values so that numbers closer to 1 have
a higher fitness
```

```

57     fitnesses = [1 / ((s)+1) for s in inverse_fitnesses]
58
59     #sum up all found fitness measures
60     total_fitness = sum(fitnesses)
61
62     #compile a list of each fitness measure's proportion to the sum of all found
63     #fitness measures
64     #of the mating pool; to be used as a probability that the candidate will mate
65     probabilities_of_reproduction = [fit / total_fitness for fit in fitnesses]
66
67     #create list of randomly chosen strings that are weight biased
68     return random.choices(population=population, weights=probabilities_of_reproduction,
69                           k=len(population))
70
71 def perform_mating(population):
72     """
73     Takes a list of strings, returns a list of strings after potential mating operations.
74
75     Args:
76         population (List[str]): List of strings that make up the mating pool
77     Returns:
78         (List[str]): List of strings that make up the now potentially modified mating
79         pool
80     """
81
82     new_pop = []
83
84     while (len(population) > 0):
85         s1 = population.pop(random.randint(0, len(population) - 1))
86         s2 = population.pop(random.randint(0, len(population) - 1))
87
88         s1p, s2p = crossover_pair(s1, s2)
89         new_pop.append(s1p)
90         new_pop.append(s2p)
91
92     return new_pop
93
94 def crossover_pair(s1, s2):
95     """
96     Takes two strings for crossover, randomly determines a crossover point
97     between 1 and len(s1)-1 then performs crossover of character data at crossover point.
98     Assumes len(s1) = len(s2)
99
100     Args:
101         s1 (str): string for mating crossover
102         s2 (str): string for mating crossover
103     Returns:
104         s1p (str): string from mating crossover
105         s2p (str): string from mating crossover
106     """
107     crossover_point = random.randint(1, len(s1) - 2)
108     s1p = s1[:crossover_point] + s2[crossover_point:]
109     s2p = s2[:crossover_point] + s1[crossover_point:]
110
111     return (s1p, s2p)

```

```
111 def perform_mutations(population, probability_of_mutation, alnum_set):
112     """
113     Takes a list of strings, returns a list of strings after potential mutation
114     operation.
115     Args:
116         population (List[str]): List of strings that make up the mating pool
117         probability_of_mutation (float): decimal number between 0 and 1 that represents
```


liklihood

```

126                                     that a character will mutate into another
127                                     character from
128                                     its alphanumeric character set
129     alnum_set (List[str]): Valid alphanumeric characters to genrate number-system
130     value-strings from
131 Returns:
132 (List[str]): List of strings that make up a potentially mutated mating pool
133 """
134
135     return [attempt_mutation(s, probability_of_mutation, alnum_set) for s in population]
136
137 def attempt_mutation(s, probability_of_mutation, alnum_set):
138     """
139     Takes a string and probabilisticly performs character mutation
140
141     Args:
142         s (str): string to mutate
143         probability_of_mutation (float): decimal number between 0 and 1 that represents
144         the liklihood
145                                     that a character will mutate into another
146                                     character from
147                                     its alphanumeric character set
148         alnum_set (List[str]): Valid alphanumeric characters to genrate number-system
149         value-strings from
150 Returns:
151     None
152     """
153
154     #bit flipping
155     #on average, method 2 seems to outperform method 1
156
157     # #method 1
158     # #choose one bit randomly if random chance falls into the probability that the
159     # string should mutate
160     # if random.random() < probability_of_mutation:
161     #     bit_to_flip = random.randint(0, len(s) - 1)
162     #     s = list(s)
163     #     s[bit_to_flip] = random.choice(alnum_set)
164     #     return "".join(s)
165     # else:
166     #     return s
167
168     #method 2
169     #bit by bit, perform mutation if random chance falls into the probability that the
170     #bit should mutate
171     for i in range(len(s)):
172         if random.random() < probability_of_mutation:
173             s = s[:i] + str(alnum_set[random.randint(0, len(alnum_set)-1)]) + s[i +
174             1:]
175     return s
176
177 # program starts here:
178 #
179 -----
180 -----
181
182 def SGA(test_function, pop_size, alnum_set, var_string_length, variable_length,

```

```
168 domain_min, domain_max,  
167 number_of_generations, probability_of_mutation):  
168     """  
186     Simple Genetic Algorithm that finds global minima of test functions through  
187     generational mating,  
187     reproduction and bit mutations while printing out each generation's performance  
188     results.
```

```

189 Args:
190     test_function (<function>): Benchmark function that has known optimum values
    (global min)
191     pop_size (int): the number of strings to create for population
192     alnum_set (List[str]): Valid alphanumeric characters to generate number system
    value-strings from
193     var_string_length (int): character length of string that contains one or more
    number system value-string variables
194     variable_length (int): character length of one number system value-string
    variable
195     domain_min (Union[float,int]): min value of operational domain
196     domain_max (Union[float,int]): max value of operational domain
197     number_of_generations (int): number of times to mate / mutate the population in
    the attempt to hone in
198                                     on the optimal input values for the given
    test_function
199     probability_of_mutation (float): decimal number between 0 and 1 that represents
    the likelihood
200                                     that a character will mutate into another
    character from
201                                     its alphanumeric character set
202 Prints:
203     table: generational performances, avoiding repeat max performance levels
    between contiguous generations
186 """"
187
191 #initialize a random population of pop_size values to be the starting point for
    optimization attempt
192 #returns string with (var_string_length * pop_size) number of characters
193 population = initialize(pop_size, alnum_set, var_string_length)
191
202 #small anonymous function used to find fitness measures of a member of a mating
    pool population,
203 #determined by the given benchmark test function
204 inverse_fitness = lambda string: test_function(*general_decoder(string,
    variable_length, domain_min, domain_max, len(alnum_set)))
205
206
207 #print performance of population
208 #header
209 print("\nTested population size: ", pop_size, " Number of generations: ",
    number_of_generations)
210 pad = str((4 + LENGTH_OF_DECIMAL) * int(var_string_length / variable_length))
211 print(("\\n{:<16s}{:<" + pad + "s}\\t {:<}"").format("Generation", "Strongest
    Candidate", "Fitness"))
202 print("="*80)
203
213 #print off generational performances, avoiding repeat performance levels between
    contiguous generations
214 last_fit_individual = fittest_individual = [] #coordinate values
215 last_max_fit = 0 #fitness value
216 new_fit = True #is the found fitness value different than the last
217 max_fitness_list = [] #list of all found fitness values to be used for a graph
218 global_max_found = (0, [], 0) #to record the overall best found fitness measure
    form all generations
219 first_gen_repeat = last_gen_repeat = 0 #to keep track of how many generations have
    had repeated max_fitness measures
220
221 for i in range(number_of_generations):

```

```
213
218     #create new generation of population
219     population = perform_reproduction(population, inverse_fitness)
220     population = perform_mating(population)
221     population = perform_mutations(population, probability_of_mutation, alnum_set)
218
222     #determine the max fitness measure of this generation
223     max_fitness = max(1/(inverse_fitness(m)+1) for m in population)
```

```

224     max_fitness_list.
append(max_fitness) 222
235     #determine the string variable values that have max_fitness of
        this generation
236     for m in population:
237         if 1/(inverse_fitness(m)+1) == max_fitness:
238             fittest_individual = general_decoder(m,
                variable_length, domain_min, domain_max,
                len(alnum_set))
239
240     #case: if this generation is the last, or it is more fit than its
predecessor
241     if (i == number_of_generations - 1) or not (fittest_individual
        ==
last_fit_individual):
242         #case: if is the new fittest member after repeated max
performance
243         if (first_gen_repeat != last_gen_repeat) and
            (last_gen_repeat -
first_gen_repeat > 1) :
244             print("\n\tFor generation {} to {}, the max fitness
                level was
                {:.+str(LENGTH_OF_DECIMAL)+"f}.\n").format(first
_gen_repeat, last_gen_repeat, last_max_fit))
245
246             #print generation's performance results
247             print(("{:<16d}[{:<"+ pad + "s}]\t {:>}").format(i,
                " ".join(["{:-"+str(LENGTH_OF_DECIMAL) +
"f},").format(x) for x in fittest_individual]),
max_fitness))
236
239         last_fit_individual = fittest_individual
240         last_ma
x_fit = max_fitness 239
243         #record the overall best found fitness measure form all
generations
244         if max_fitness > global_max_found[2]:
245             global_max_found = ("Gen: " + str(i),
fittest_individual, max_fitness) 243
251             first_gen_repeat = last_gen_repeat = i
252             new_fit = True
253
254         #case: first repeat of same fittest member between generations
255         elif last_fit_individual == fittest_individual:
256             new_fit = False
257             l
ast_gen_repeat = i
251
252         print("=*80)
253         print("Highest fitness acheived
by:\n", global_max_found) 254 print("=*80)
258         print("")
259         plt.plot(max_fitness_list)
260         plt.show()

```

```

1
2     #d
3     ejong_t
4     est.py
5
6     2
7     from
8     simple_genetic_algorithm import
9     SGA
10    4
11    ALNUM = ["O", "I"]
12    VAR_STRING_LEN = 20
13    NUMBER_OF_VARIABLES = 4
14    VARIABLE_LEN = int(VAR_STRING_LEN / NUMBER_OF_VARIABLES)
15    PROBABILITY_OF_MUTATION = 0.05
16    POP_SIZE = 40
17    NUM_GENERATIONS = 1000
18    DOMAIN_MIN = -5.12
19    DOMA
20    IN_MAX =
21    5.12
22    14
23    def
24    dejong(
25    *xs):
26    """
27    The function is defined on n-dimensional space.
28    The function can be defined on any input domain but it is usually
29    evaluated on x_i element of [-5.12, 5.12] for i = 1, ..., n.
30
31    Takes in n dimensional coordinates and
32    returns output of:
33    f(x,y) = sum[b(x_i + 1 -
34    (x_i)^2)^2 + (a - x_i)^2]]
35    for i = 1, ..., n; and the parameters a and b are constants set to
36    a = 1 and b = 100..
37
38    The function has one global minimum f(x^*) = 0 at x^* =
39    (0, ..., 0).
40    Args:
41    xs (List[num]): n dimensional coordinates for Euclidean (n + 1)-
42    space
43    Returns:
44    (float): f(x,y) = value closer to 0 indicates a coordinate
45    closer to know global minimum
46    """
47
48    return sum(xi ** 2
49    for xi in xs)
50
51    34
52    print("Running Simple Genetic Algorithm on De Jong Sphere benchmark
53    function")
54
55    SGA(dejong, POP_SIZE, ALNUM, VAR_STRING_LEN, VARIABLE_LEN,
56    DOMAIN_MIN, DOMAIN_MAX, NUM_GENERATIONS,
57    PROBABILITY
58    OF_MUTATION)
59
60    38
61    print("\nDe Jong benchmark test
62    complete\n")
63    40

```

```

1  #himmelblau_test.py
2
3  from simple genetic algorithm import SGA
4
5  ALNUM = ["0", "1"]
6  VAR_STRING_LEN = 16
7  NUMBER_OF_VARIABLES = 2
8  VARIABLE_LEN = int(VAR_STRING_LEN / NUMBER_OF_VARIABLES)
9  PROBABILITY_OF_MUTATION = 0.005
10 POP_SIZE = 100
11 NUM_GENERATIONS = 1000
12 DOMAIN_MIN = -6
13 DOMAIN_MAX = 6
14
15 def himmelblau(*xs):
16     """
17     The function is defined on the 2-dimensional space.
18     The function can be defined on any input domain but it is usually evaluated on
19     x_i element of [-6, 6] for i = 1, 2.
20
21     Takes in x, y cartesian values and returns function output of:
22      $f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$ 
23
24     The function has four local minima at:
25      $f(x^*) = 0$  at  $x^* = (3, 2)$ 
26      $f(x^*) = 0$  at  $x^* = (-2.805118, 3.283186)$ 
27      $f(x^*) = 0$  at  $x^* = (-2.779510, -2.283186)$ 
28      $f(x^*) = 0$  at  $x^* = (3.584488, -1.848126)$ 
29
30     Args:
31         xs (List[num]): [x,y] cartesian coordinates
32     Returns:
33         (float): f(x,y) = value closer to 0 indicates a coordinate closer to know
34         global minimum
35     """
36     x = xs[0]
37     y = xs[1]
38     return (x ** 2 + y - 11) ** 2 + (x + y ** 2 - 7) ** 2
39
40 print("Running Simple Genetic Algorithm on Himmelblau benchmark function")
41 SGA(himmelblau, POP_SIZE, ALNUM, VAR_STRING_LEN, VARIABLE_LEN, DOMAIN_MIN, DOMAIN_MAX,
42     NUM_GENERATIONS,
43     PROBABILITY_OF_MUTATION)
44 print("\nHimmelblau benchmark test complete\n")

```

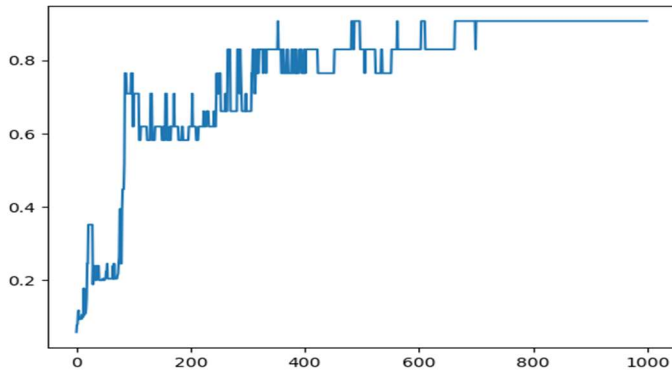
```

1  #rosenbrock_test.py
2  from simple_genetic_algorithm import SGA
3
4  ALNUM = ["0", "1"]
5  VAR_STRING_LEN = 16
6  NUMBER_OF_VARIABLES = 2
7  VARIABLE_LEN = int(VAR_STRING_LEN / NUMBER_OF_VARIABLES)
8  PROBABILITY_OF_MUTATION = 0.01
9  POP_SIZE = 100
10 NUM_GENERATIONS = 1000
11 DOMAIN_MIN = -2
12 DOMAIN_MAX = 2
13
14
15 def rosenbrock(*xs):
16     """
17     The function is defined on n-dimensional space.
18     The function can be defined on any input domain but it is usually evaluated on
19     x_i element of [-5, 10] for i = 1, ..., n.
20
21     Takes in n dimensional coordinates and returns output of:
22     f(x,y) = sum[b(x_i + 1 - (x_i)^2)^2 + (a - x_i)^2]
23     for i = 1, ..., n; and the parameters a and b are constants set to a = 1 and b =
24     100..
25
26     The function has one global minimum f(x^*) = 0 at x^* = (1, ..., 1).
27
28     Args:
29     xs (List[num]): n dimensional coordinates for Euclidean (n + 1)-space
30     Returns:
31     (float): f(x,y) = value closer to 0 indicates a coordinate closer to know
32     global minimum
33     """
34
35     x = xs[0]
36     y = xs[1]
37
38     return (1-x)**2+100*(y-x**2)**2
39
40
41 print("Running Simple Genetic Algorithm on Rosenbrock benchmark function")
42
43 SGA(rosenbrock, POP_SIZE, ALNUM, VAR_STRING_LEN, VARIABLE_LEN, DOMAIN_MIN, DOMAIN_MAX,
44     NUM_GENERATIONS,
45     PROBABILITY_OF_MUTATION)
46
47 print("\nRosenbrock benchmark test complete\n")
48

```


De Jong

Method 1



Running Simple Genetic Algorithm on De Jong Sphere benchmark function

Tested population size: 40 Number of generations: 1000

Generation	Strongest Candidate	Fitness
0	[-1.9200, 1.9200, -1.9200, 2.2400,]	0.05855898060526562
1	[-1.6000, 1.6000, 2.5600, -0.3200,]	0.07827175954915469
2	[-1.9200, 1.9200, -1.9200, 0.6400,]	0.08020017964840241
3	[-0.3200, -1.9200, -1.9200, 0.6400,]	0.11255177381595535
4	[-0.3200, -1.9200, -1.9200, 0.0000,]	0.11799131583915423
5	[-1.6000, -1.6000, -1.9200, 0.9600,]	0.09321401938851603
6	[-1.2800, 1.9200, -1.9200, 0.0000,]	0.09988812529966437
7	[-1.2800, 2.2400, -1.6000, -0.3200,]	0.0969142502713599
9	[-1.2800, 0.6400, 2.2400, -1.6000,]	0.09411233248004816
10	[-0.6400, 0.6400, 2.2400, -1.6000,]	0.10641920653839604
12	[-2.5600, 0.6400, 0.6400, 1.2800,]	0.09988812529966438
13	[-1.2800, 0.9600, -0.6400, 1.2800,]	0.17831669044222537
14	[-1.2800, 0.9600, 0.9600, -2.2400,]	0.10527202290719216
15	[-0.6400, 1.6000, -1.6000, 1.6000,]	0.11001584228128854
17	[-0.6400, 0.9600, 0.9600, -2.2400,]	0.12091313600309538
18	[-1.2800, 0.9600, 0.6400, 1.6000,]	0.15314873805439846
19	[-1.2800, 0.9600, 0.6400, 0.3200,]	0.24557956777996065
21	[-0.6400, 0.9600, 0.6400, 0.3200,]	0.35171637591446264

For generation 21 to 28, the max fitness level was 0.3517.

29 [-0.6400, 0.9600, 0.6400, 1.6000,] 0.18865076969514044

...

552 [-0.3200, 0.0000, 0.3200, 0.0000,] 0.8300132802124832

For generation 552 to 561, the max fitness level was 0.8300.

562 [-0.3200, 0.0000, 0.0000, 0.0000,] 0.9071117561683597

563 [-0.3200, 0.0000, 0.3200, 0.0000,] 0.8300132802124832

For generation 563 to 603, the max fitness level was 0.8300.

604 [-0.3200, 0.0000, 0.0000, 0.0000,] 0.9071117561683597

For generation 604 to 610, the max fitness level was 0.9071.

611 [-0.3200, 0.0000, 0.3200, 0.0000,] 0.8300132802124832

For generation 611 to 662, the max fitness level was 0.8300.

663 [-0.3200, 0.0000, 0.0000, 0.0000,] 0.9071117561683597

For generation 663 to 698, the max fitness level was 0.9071.

699 [-0.3200, 0.0000, 0.3200, 0.0000,] 0.8300132802124832

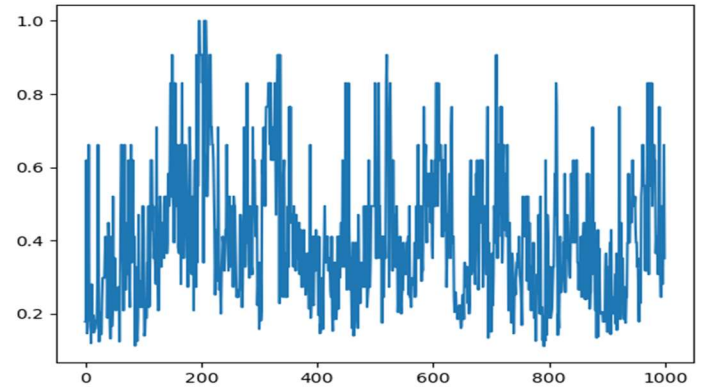
700 [-0.3200, 0.0000, 0.0000, 0.0000,] 0.9071117561683597

For generation 700 to 998, the max fitness level was 0.9071.

999 [-0.3200, 0.0000, 0.0000, 0.0000,] 0.9071117561683597

Highest fitness achieved by:
('Gen: 353', [-0.3200000000000003, 0.0, 0.0, 0.0], 0.9071117561683597)

Method 2



Running Simple Genetic Algorithm on De Jong Sphere benchmark function

Tested population size: 40 Number of generations: 1000

Generation	Strongest Candidate	Fitness
0	[0.6400, -1.2800, 0.0000, 1.6000,]	0.17831669044222542
1	[1.2800, -0.6400, 0.0000, 1.6000,]	0.17831669044222542
2	[0.6400, -0.3200, 0.0000, 0.3200,]	0.6194251734390486
3	[0.6400, -1.2800, 1.9200, 0.3200,]	0.1462672595366253
4	[-0.6400, -0.3200, -0.9600, 1.9200,]	0.16339869281045755
5	[1.2800, 0.6400, 0.0000, 0.9600,]	0.25191455058444173
6	[0.0000, -0.6400, 0.0000, 0.3200,]	0.6613756613756615
7	[0.0000, -0.6400, 1.9200, 0.3200,]	0.19236688211757466
8	[0.0000, -1.9200, -0.6400, -0.6400,]	0.18163324614937523
9	[0.0000, -1.9200, -0.6400, -0.3200,]	0.19236688211757466
10	[0.0000, -2.5600, 0.6400, -0.6400,]	0.11943435887636158
11	[0.6400, 0.6400, -1.2800, 0.3200,]	0.2808988764044944
12	[0.0000, -1.9200, 0.6400, -0.9600,]	0.1661792076575379
13	[0.9600, -1.9200, 0.0000, -0.9600,]	0.15314873805439844
14	[0.0000, -1.9200, 0.0000, -0.6400,]	0.19623233908498198
15	[1.2800, -1.9200, 0.0000, -0.6400,]	0.14849132810643859
16	[0.6400, -1.9200, 0.0000, -0.6400,]	0.18163324614937523
17	[1.2800, -1.2800, 1.2800, -0.6400,]	0.15810776625347833
18	[1.2800, -1.2800, 1.2800, -0.3200,]	0.16617920765753783
19	[0.6400, -1.9200, -0.6400, -0.3200,]	0.17831669044222542
20	[-0.6400, 0.6400, -1.2800, 1.2800,]	0.19623233908498196
21	[-0.6400, -1.2800, 0.0000, -0.6400,]	0.28921795465062466

...

966 [0.0000, -0.6400, 0.3200, -1.2800,] 0.3174200101574403

967 [0.0000, -0.3200, 0.6400, -0.6400,] 0.5203996669442134

968 [1.2800, -0.6400, 0.0000, -0.3200,] 0.31742001015744026

969 [0.0000, -0.3200, 0.0000, -0.6400,] 0.6613756613756615

970 [0.0000, -0.3200, 0.0000, -0.3200,] 0.8300132802124832

971 [0.6400, -0.3200, 0.3200, -0.3200,] 0.5824790307548928

972 [1.2800, -0.6400, 0.3200, -0.3200,] 0.307427447122479

973 [0.0000, 0.0000, 0.3200, -0.3200,] 0.8300132802124832

974 [0.0000, -0.3200, 0.0000, -0.3200,] 0.8300132802124832

975 [0.0000, -0.6400, 0.0000, -0.6400,] 0.5496921723834656

976 [0.0000, -0.3200, 0.6400, -0.3200,] 0.6194251734390486

977 [0.0000, -0.3200, 0.0000, -0.3200,] 0.8300132802124832

978 [0.6400, -0.3200, 0.6400, -0.3200,] 0.4940711462450594

979 [0.0000, -0.3200, 0.3200, -0.6400,] 0.6194251734390486

980 [0.0000, -0.3200, 0.0000, -0.6400,] 0.6613756613756615

981 [0.0000, -0.3200, 0.3200, -0.6400,] 0.6194251734390486

982 [0.0000, -0.9600, 0.6400, -0.6400,] 0.3648569760653825

983 [0.0000, -0.3200, 0.9600, -0.6400,] 0.410913872452334

984 [0.0000, -1.2800, 0.0000, -0.3200,] 0.36485697606538225

985 [0.0000, -0.9600, 0.9600, -0.6400,] 0.3074274471224791

986 [0.6400, 0.6400, 0.0000, -0.6400,] 0.4486719310839916

987 [0.0000, -0.3200, 0.3200, -0.3200,] 0.7649938800489593

988 [1.2800, -0.3200, 0.0000, -0.6400,] 0.31742001015744026

989 [1.2800, -0.3200, 0.6400, -0.9600,] 0.24557956777996065

990 [0.3200, -0.3200, 0.6400, -0.6400,] 0.4940711462450594

991 [0.0000, -0.3200, 0.6400, -0.9600,] 0.410913872452334

992 [1.2800, -0.6400, 0.3200, -0.6400,] 0.28089887640449435

993 [0.0000, -0.6400, 0.0000, -0.9600,] 0.42896362388469467

994 [0.0000, -0.6400, 0.0000, -0.3200,] 0.6613756613756615

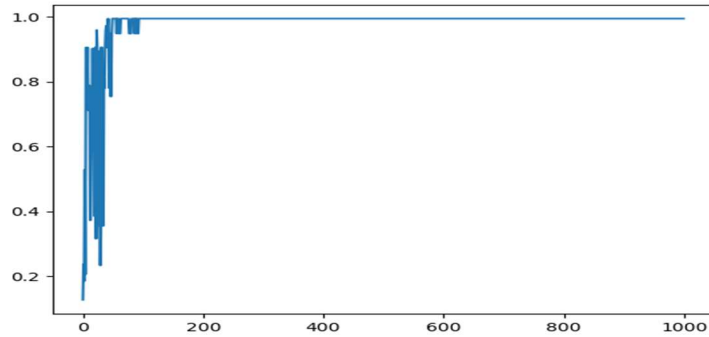
995 [0.6400, -0.3200, 0.6400, -0.9600,] 0.35171637591446264

Highest fitness achieved by:
('Gen: 196', [0.0, 0.0, 0.0, 0.0], 1.0)

De Jong benchmark test complete

Himmelblau

Method 1



Running Simple Genetic Algorithm on Himmelblau benchmark function

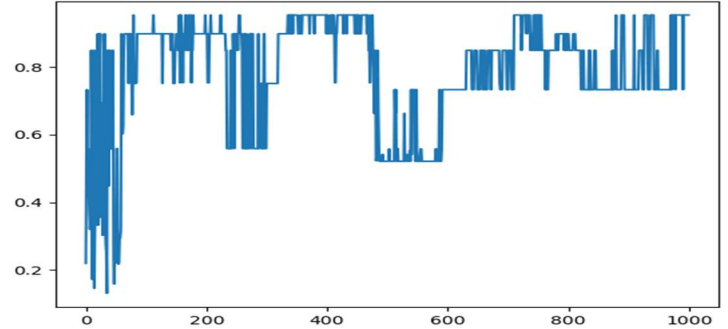
Tested population size: 100 Number of generations: 1000

Generation	Strongest Candidate	Fitness
0	[3.4219, -0.9844,]	0.12685092257472316
1	[3.4219, -2.1562,]	0.23576390181437498
2	[-2.4375, 3.0000,]	0.18418870798658832
3	[3.0469, 1.7344,]	0.5281303795684011
4	[-2.4375, 3.0938,]	0.2049805991176092
5	[-2.7656, 3.0938,]	0.9057085681453921
7	[-2.7656, 3.0469,]	0.7546138259377453
8	[-2.7656, 3.0938,]	0.9057085681453921
9	[-2.9062, 3.0938,]	0.7123313727807917
10	[3.5625, -1.9688,]	0.7879095273411465
12	[3.5625, -2.1562,]	0.3724650854831554
14	[3.5625, -2.0625,]	0.5542531418616059
15	[3.5625, -2.0156,]	0.6684277361309007
16	[3.6094, -1.7812,]	0.9024394166606019
17	[3.5625, -2.0156,]	0.6684277361309007
19	[3.6094, -1.7812,]	0.9024394166606019
20	[3.6094, -2.1562,]	0.38445216363670154
21	[3.6094, -1.9219,]	0.9066419898073936
22	[3.6094, -2.2031,]	0.31530059654322834
23	[3.6094, -1.8281,]	0.9595585892857286
24	[3.6094, -1.7812,]	0.9024394166606019
25	[3.6094, -2.0156,]	0.6887587173430882
26	[3.5625, -1.4531,]	0.35525716342591057

...

57	[-2.8125, 3.1406,]	0.9948443921995289
For generation 57 to 59, the max fitness level was 0.9948.		
60	[-2.7656, 3.1406,]	0.9487880473133589
For generation 60 to 62, the max fitness level was 0.9488.		
63	[-2.8125, 3.1406,]	0.9948443921995289
For generation 63 to 76, the max fitness level was 0.9948.		
77	[-2.7656, 3.1406,]	0.9487880473133589
For generation 77 to 80, the max fitness level was 0.9488.		
81	[-2.8125, 3.1406,]	0.9948443921995289
For generation 81 to 85, the max fitness level was 0.9948.		
86	[-2.7656, 3.1406,]	0.9487880473133589
87	[-2.8125, 3.1406,]	0.9948443921995289
89	[-2.7656, 3.1406,]	0.9487880473133589
90	[-2.8125, 3.1406,]	0.9948443921995289
91	[-2.7656, 3.1406,]	0.9487880473133589
93	[-2.8125, 3.1406,]	0.9948443921995289
For generation 93 to 998, the max fitness level was 0.9948.		
999	[-2.8125, 3.1406,]	0.9948443921995289
Highest fitness achieved by: ('Gen: 42', [-2.8125, 3.140625], 0.9948443921995289)		
Himmelblau benchmark test complete		

Method 2



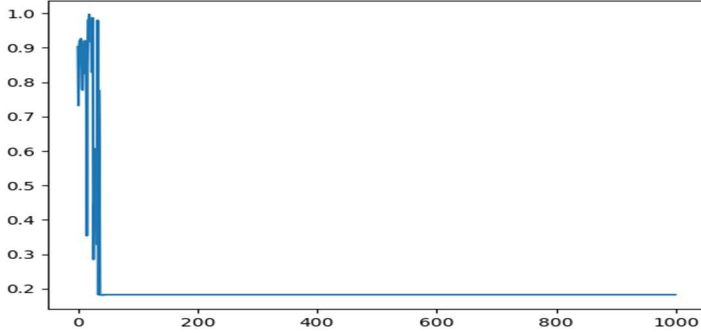
Running Simple Genetic Algorithm on Himmelblau benchmark function

Tested population size: 100 Number of generations: 1000

Generation	Strongest Candidate	Fitness
0	[3.1875, 2.2500,]	0.21972996442665219
1	[-3.6562, -3.1875,]	0.521534039504397
2	[-3.7500, -3.1875,]	0.7333848099282685
3	[-3.7500, -3.0938,]	0.41835122118981877
4	[-3.7500, -3.1406,]	0.5580541659818725
5	[-2.7656, 2.9531,]	0.44446231772630235
6	[3.0000, 2.2969,]	0.3682727405489496
7	[3.1406, 2.2031,]	0.3198419604143811
8	[3.2344, 1.9219,]	0.3426966317380834
9	[-3.7500, -3.3281,]	0.8495425659298472
10	[3.0000, 2.2031,]	0.5649189289312718
11	[-3.0469, 2.8594,]	0.17223202916858715
12	[-2.8125, 2.9531,]	0.452451961679104
13	[-3.7500, -3.3281,]	0.8495425659298472
14	[2.5312, 2.2031,]	0.14580124120040613
15	[3.7031, -1.5469,]	0.3147154801968908
16	[3.6562, -2.1094,]	0.43676270558425573
17	[-3.7500, -3.3281,]	0.8495425659298472
18	[3.4688, -2.1094,]	0.3329600109200428
19	[-3.7500, -3.3281,]	0.8495425659298472
20	[-3.7500, -3.2344,]	0.8985072610951793
21	[3.4688, -2.1094,]	0.3329600109200428
22	[-3.7500, -3.2344,]	0.8985072610951793

...

For generation 936 to 941, the max fitness level was 0.9541.		
942	[-3.7500, -3.1875,]	0.7333848099282685
For generation 942 to 946, the max fitness level was 0.7334.		
947	[-3.7500, -3.3281,]	0.8495425659298472
949	[-3.7500, -3.1875,]	0.7333848099282685
For generation 949 to 969, the max fitness level was 0.7334.		
970	[-3.7500, -3.2812,]	0.9540826957320063
971	[-3.7500, -3.1875,]	0.7333848099282685
972	[-3.7500, -3.2812,]	0.9540826957320063
For generation 972 to 976, the max fitness level was 0.9541.		
977	[-3.7500, -3.1875,]	0.7333848099282685
978	[-3.7500, -3.2812,]	0.9540826957320063
For generation 978 to 989, the max fitness level was 0.9541.		
990	[-3.7500, -3.1875,]	0.7333848099282685
991	[-3.7500, -3.2812,]	0.9540826957320063
For generation 991 to 998, the max fitness level was 0.9541.		
999	[-3.7500, -3.2812,]	0.9540826957320063
Highest fitness achieved by: ('Gen: 79', [-3.75, -3.28125], 0.9540826957320063)		
Himmelblau benchmark test complete		

Rosenbrock**Method 1**

Running Simple Genetic Algorithm on Rosenbrock benchmark function

Tested population size: 100 Number of generations: 1000

Generation	Strongest Candidate	Fitness
0	[1.2031, 1.4219,]	0.9033636766002906
1	[0.9531, 0.9688,]	0.7321505844648348
2	[0.6719, 0.4531,]	0.9025612245710402
3	[1.2500, 1.5781,]	0.9200359389038635
4	[1.2500, 1.5469,]	0.9200359389038635
5	[0.7188, 0.5156,]	0.9266149414114279
6	[0.7500, 0.5781,]	0.9200359389038635
7	[0.7500, 0.5156,]	0.7798933739527799
8	[0.9219, 0.7969,]	0.7771361403442459
9	[0.9844, 0.9375,]	0.9095603828204
10	[0.7969, 0.5938,]	0.8254253515527755
11	[0.7500, 0.5781,]	0.9200359389038635
12	[0.7500, 0.5469,]	0.9200359389038635
14	[0.9375, 0.9219,]	0.8413701021927797
15	[0.7344, 0.6719,]	0.35360758557227123
16	[0.9375, 0.9219,]	0.8413701021927797
17	[0.9531, 0.9219,]	0.9801733757561382
19	[0.9531, 0.9062,]	0.9973271002925899
20	[0.8906, 0.7656,]	0.9190567751421268
21	[0.9531, 0.9219,]	0.9801733757561382
22	[0.8906, 0.7969,]	0.9868706773126782
23	[0.5469, 0.2969,]	0.8293214471223307
24	[0.8750, 0.7969,]	0.8982456140350877
25	[0.8906, 0.7969,]	0.9868706773126782
26	[0.9531, 0.7500,]	0.2846771771491518
27	[0.9531, 0.7969,]	0.44503103266984945
29	[0.9531, 0.7812,]	0.38166311708082545
30	[0.9531, 0.8281,]	0.6070304195138754
32	[0.9531, 0.7656,]	0.328729236951222
33	[0.9531, 0.9219,]	0.9801733757561382
34	[-1.1094, 1.2500,]	0.18226017435828668
35	[0.9219, 0.7969,]	0.7771361403442459
36	[0.9219, 0.7656,]	0.5829041895321386
37	[-1.1094, 1.2500,]	0.18226017435828668
38	[-1.1250, 1.2656,]	0.1813031161473088

For generation 38 to 42, the max fitness level was 0.1813.

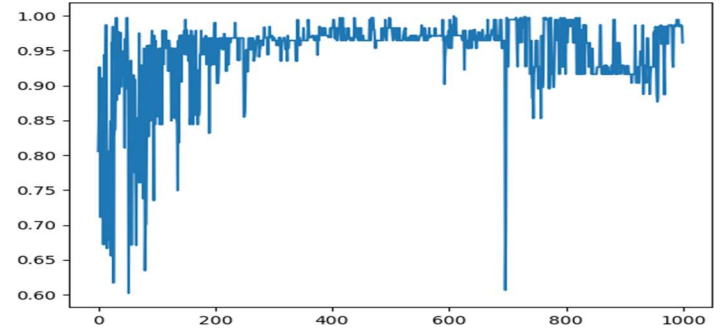
43	[-1.1094, 1.2500,]	0.18226017435828668
44	[-1.1250, 1.2656,]	0.1813031161473088
45	[-1.1094, 1.2500,]	0.18226017435828668

For generation 45 to 998, the max fitness level was 0.1823.

999	[-1.1094, 1.2500,]	0.18226017435828668
-----	---------------------	---------------------

Highest fitness achieved by:
('Gen: 19', [0.953125, 0.90625], 0.9973271002925899)

Rosenbrock benchmark test complete

Method 2

Running Simple Genetic Algorithm on Rosenbrock benchmark function

Tested population size: 100 Number of generations: 1000

Generation	Strongest Candidate	Fitness
0	[0.5781, 0.3594,]	0.8056634380357375
1	[0.7188, 0.5469,]	0.8541535654355582
2	[0.7188, 0.5156,]	0.9266149414114279
3	[0.6094, 0.3906,]	0.8404864689282009
4	[1.1250, 1.2031,]	0.7111111111111111
5	[0.6250, 0.3438,]	0.7351040918880115
6	[1.2969, 1.6719,]	0.9106188394120442
For generation 6 to 8, the max fitness level was 0.9106.		
9	[0.3750, 0.1719,]	0.6719160104986877
10	[1.0938, 1.2188,]	0.9440745049032495
11	[0.8750, 0.7969,]	0.8982456140350877
12	[0.8906, 0.7656,]	0.9190567751421268
13	[0.8906, 0.7969,]	0.9868706773126782
15	[0.4219, 0.2188,]	0.6664623251239198
16	[0.5312, 0.2969,]	0.8056821639430923
For generation 16 to 18, the max fitness level was 0.8057.		
19	[0.3125, 0.0938,]	0.6783422349190577
20	[0.5000, 0.2344,]	0.7846743295019157

...

969	[0.8750, 0.7656,]	0.9846153846153847
971	[0.9375, 0.8594,]	0.9596438821531073
972	[0.8750, 0.7656,]	0.9846153846153847
974	[0.9375, 0.8594,]	0.9596438821531073
975	[0.9062, 0.8281,]	0.9867167533019916

For generation 975 to 979, the max fitness level was 0.9867.

980	[0.8750, 0.7656,]	0.9846153846153847
981	[0.9062, 0.8281,]	0.9867167533019916
982	[0.9062, 0.7969,]	0.9359845468859904
983	[0.7188, 0.5156,]	0.9266149414114279
984	[0.8750, 0.7656,]	0.9846153846153847
986	[0.8906, 0.7969,]	0.9868706773126782
987	[0.8750, 0.7656,]	0.9846153846153847
989	[0.9375, 0.8750,]	0.9945972196928307

For generation 989 to 992, the max fitness level was 0.9946.

993	[0.8750, 0.7656,]	0.9846153846153847
-----	--------------------	--------------------

For generation 993 to 995, the max fitness level was 0.9846.

996	[0.8906, 0.7969,]	0.9868706773126782
998	[0.8750, 0.7656,]	0.9846153846153847
999	[0.8750, 0.7500,]	0.9615023474178404

Highest fitness achieved by:
('Gen: 608', [0.984375, 0.96875], 0.9997499614454374)

Rosenbrock benchmark test complete