intro:

the focus of this assignment will be solving two minimisation optimisation problems using an evolutionary algorithm. The first problem function is

$$f(x) = (x_1 - 1)^2 + \sum_{i=2}^{d} i\left(2x_i^2 - x_{\{i-1\}}\right)^2$$

where -10 ≤ x ≤ 10, d=20. The second problem function is

$$f(x) = 418.9829d - \sum_{i=1}^{d} x_i \sin\left(\sqrt{|x_i|}\right) + 31$$

where -500 ≤ x ≤ 500, d=20 and 31 refers to my student number (22010031). Evolutionary algorithms (Eiben and Smith, 2003) are particularly good for tasks like this because they gradually improve solutions across generations by mimicking natural selection. They are effective at exploring large, complex search spaces without needing derivative information. They iteratively improve solutions through selection, mutation and crossover, being particularly effective for optimisation problems like the two this report will be exploring. In order to implement and analyse an evolutionary algorithm to solve the two given problems, exploring how changes in parameters like mutation rate, mutation step size, population size and number of generations affect the performance, and comparing these results with other optimisation methods will be vital.

The algorithm starts by initialising a population of potential solutions within the parameter ranges of each function. The fitness of each individual is then evaluated, followed by applying mutation and crossover to generate new individuals. Selection and replacement are used to ensure that the best solutions are kept, leading to gradual improvement over generations.

The experimentation part of this report will focus on analysing how different algorithm parameters affect the performance of the algorithm for both problems, specifically evaluating aspects such as convergence speed, stability, and solution quality. The main parameters are the mutation rate, the mutation step size, population size and number of generations. These parameters control the behaviour of the algorithm and should have significant effects on the speed of the minimisation and the quality of the solutions. In the case of these two problems, the initialisation is relatively similar, with them both being initialised with 50 individuals, shown by the P value, and each individual being made up of 20 values in the ranges of [-10,10] and [-500,500] in the first and second problem respectively, shown by the N value. In both cases, the algorithm runs for 100 generations. the fitness of each individual in the population is calculated by plugging the individual into the test function, and in both cases, a lower value indicates a better solution. Each generation undergoes a cycle of mutation and crossover. For mutation, each gene has a probability defined by the mutation rate, shown by MUTRATE, with small random changes applied to it within the defined mutation step size, shown by

MUTSTEP. Crossover happens between pairs of individuals in the population, where a random crossover point is chosen and their genes are swapped, creating two offspring. These offspring are then used in the next generation, with their fitness calculated like the individuals before them. Initially, both problems were tested with a MUTRATE of 1/N and a MUTSTEP of 0.1. These values were chosen to provide a balanced starting point, making it possible to observe the initial behaviour of the algorithm and establish a benchmark for further experimentation. The graphs below show the evolution of both problems fitness over 100 generations, with each having 6 graphs shown together to show how the algorithm initially works. The best fitness, along with average fitness, gradually improves in both cases, minimising towards 0. This suggests that the population is converging towards an optimal solution, but does not reach it efficiently with the baseline parameters. With further experimentation, parameters like MUTSTEP, MUTRATE, P, and the number of generations will be modified to find more optimal solutions. These changes are expected to affect how well the algorithm explores and refines solutions, helping us determine the best settings for these problems.

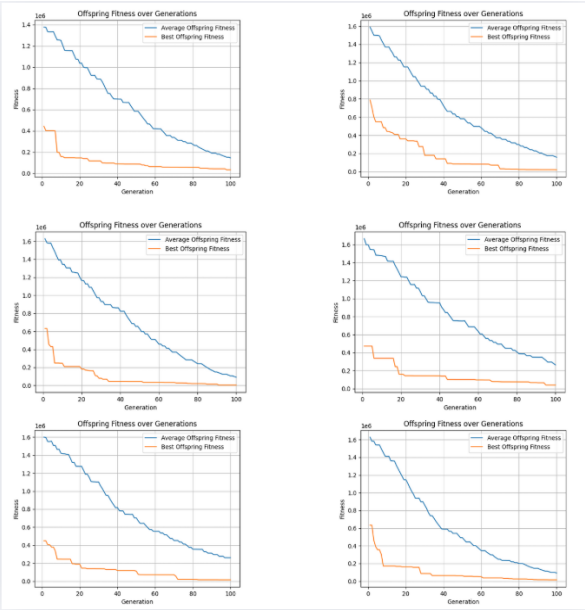

Figure 1: baseline graphs of function 1

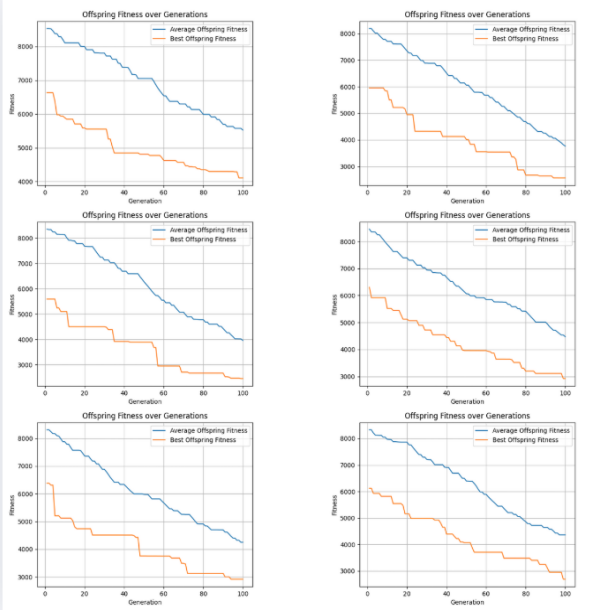

Figure 2: baseline graphs of function 2

The experiments that were conducted on the functions were done so to determine the best parameter values that would lead to best convergence and lowest fitness value. Each parameter is tested systematically while keeping all other parameters at their baseline value as shown above. This way of testing allowed a clearer understanding of how the individual parameters affected the performance of the evolutionary algorithm, which, along with finding the optimal solution, is the goal of this report. In the experiments, the mutation rate, or MUTRATE, was tested with 3 different values, a ‘low’ value of 0.01, a ‘medium’ value of 0.1, and a ‘high’ value of 0.5. this was in order to

assess how the mutation rate affected the algorithm exploring the problem space. A low mutation rate was expected to limit exploration, potentially causing the algorithm to get stuck in local optima. A high mutation rate was expected to be mostly unstable and possibly jump in and out of local optima's, but be more diverse as well. Mutation step size, or MUTSTEP, was tested with the same low, medium, and high values of 0.01, 0.1, and 0.5 respectively. In theory, a smaller step size meant more fine tuning of solutions, but possibly not being able to explore as well as a larger step size, whereas the high value might mean overshooting the target. In the case of the population size, values of 20, 50 and 100 were tested to explore their effect on diversity of the individuals and the speed of convergence. A smaller population was expected to have less diversity, but converge to 0 faster, with a faster program. A larger population was expected to maintain diversity for longer but take longer for the program to execute, which would be an issue in larger problems. Lastly, the number of generations was tested to see when the point of diminishing returns is in relation to computational power. Values of 50, 100 and 200 were tested, with the higher value of iterations expected to give the algorithm more opportunities to improve, but at the risk of converging early and wasting resources. Each experiment was repeated 5 times in order to account for the inherent randomness of evolutionary algorithms. When one parameter was being tested, the others were set to their baseline values of P = 50, MUTSTEP = 0.1, MUTRATE = 1/N, and Generations = 100.

MUTRATE	Avg best fit	Avg avg fit	Avg best fit (g1)	Avg best fit (g50)
.01	21264.8563	156632.8046	552056.5853	77986.40599
.1	26346.92031	143824.4822	526122.4506	82531.9001
.5	20544.96131	134284.2079	424135.6688	73328.6592

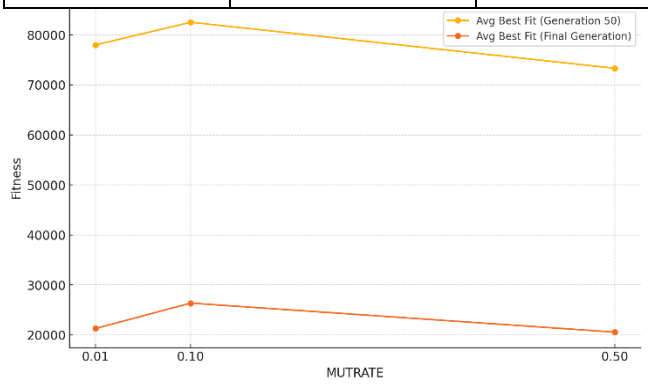

Figure 3: MUTRATE's effect on the average of best fitness at final generation vs generation 50 over 6 runs on function 1

In the case of the first function, it seems that individually, the mutation rate changing had a negligible effect on the fitness value, as seen in the table and graph. The difference between the highest best fitness value and the lowest best fitness value at the final generation is small, with a 0.5 mutation rate having the lowest best fitness value, of 20544.96. this number is a large value in terms of the fact that this is a minimisation function, which points to mutation rate not being a significant factor in isolation in exploring the problem space of function 1. Mutation rate primarily influences the exploration of the problem space, but other factors, such as the population size and the mutation step size also play a big role in that. This shows that in isolation, mutation rate for this functions' impact is diminished, as it needs to be used in tandem with the other parameters to have a meaningful effect.

MUTRATE	Avg best fit	Avg avg fit	Avg best fit (g1)	Avg best fit (g50)
.01	2611.673870	3984.298874	6540.282387	3843.889209
.1	2559.636893	3973.96899	6247.06323	3570.60501
.5	2617.18480	4188.189568	6388.954277	4210.96644

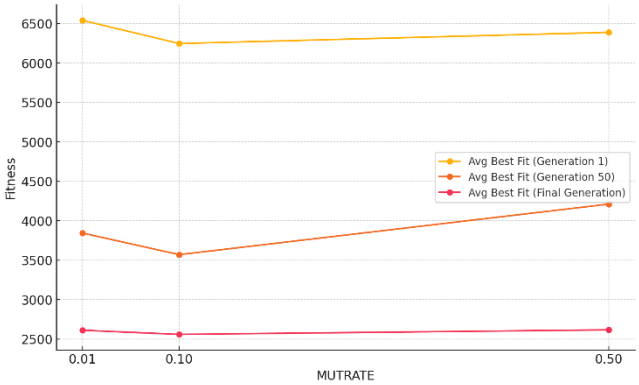

Figure 4: MUTRATE's effect on the average of best fitness at final generation vs generation 50 vs generation 1 over 6 runs on function 2

This is also shown in function 2, where the difference between the best fitness is minute, with a 0.1 mutation rate slightly edging the others out. While small, the difference in the best fitness' between results did show that a mutation rate of 0.1 did allow for a better level of balance between exploration and exploitation. A mutation rate of 0.01 being worse indicates that it resulted in premature convergence and not allowing for thorough exploration of the problem space, whereas a mutation rate of 0.5 introduced too much randomness within the results, not allowing the function to converge on an adequate solution. These results differ from those found with the first function, where a higher mutation rate performed better, which points to the first problem needing more aggressive exploration to find a good solution.

Function 1

Population	Avg best fit	Avg avg fit	Avg best fit (g1)	Avg best fit (g50)
20	49660.7226	54383.92901	542424.667	78197.3703
50	16790.85106	147464.4278	448035.2033	67042.72066
100	30327.0344	486908.4526	374050.3413	78091.09071

Function 2:

Population	Avg best fit	Avg avg fit	Avg best fit (g1)	Avg best fit (g50)
20	3187.02097	3306.647195	6744.822717	3811.218105
50	2899.151429	4184.148339	6108.737298	4106.405603
100	3051.893367	5664.128993	6106.39468	4105.939057

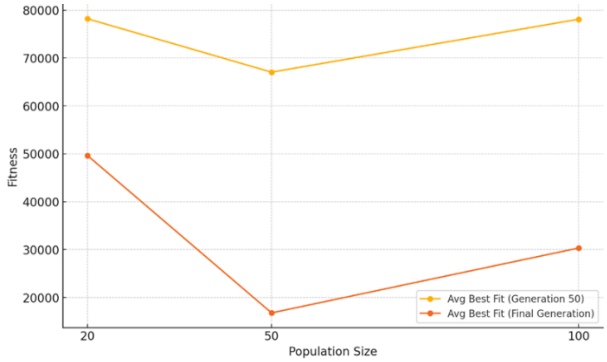

Figure 5: population sizes' effect on the average of best fitness at final generation vs generation 50 over 6 runs on function 1

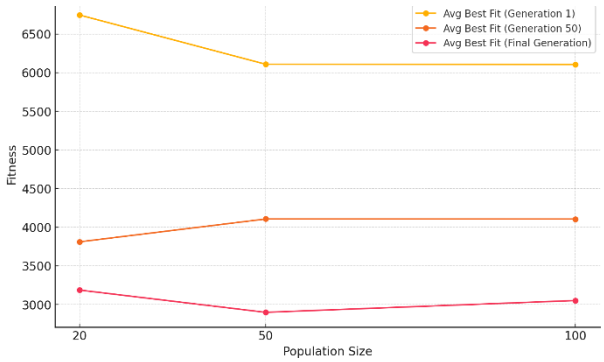

Figure 6: population sizes' effect on the average of best fitness at final generation vs generation 50 vs generation 1 over 6 runs on function 2

The population size on both functions had a similar result of a population size of 50 being the best. The population size is a significant parameter and works closely with the MUTRATE to balance exploration vs exploitation, and increasing it can increase diversity, but also computational costs. In function 1, the difference is quite large between the 3 values, with P=50 being significantly lower than the other 2, 20 especially. This shows that function 1 needed a moderate amount of diversity, but too much (P=100) didn't lead to improvements at all and thus isn't worth the extra time and effort for the increased diversity. On the other hand, function 2's results were much closer, showing that for function 2, like MUTRATE, population size in isolation is also not a hugely significant parameter. This suggests that function 2 can be solved with a range of population sizes, and that a population size of 50 is the optimal balance between diversity and computational efficiency. Function 2 appears to be less dependent on diversity, pointing to a potentially simpler problem space than function 1.

MUTSTEP	Avg best fit	Avg avg fit	Avg best fit (g1)	Avg best fit (g50)
0.01	31923.6636	200351.9021	619958.5653	108127.2212
0.1	36365.2746	187703.2227	465840.3275	69586.8715
0.5	12613.1524	114812.8570	328274.6386	60151.1525

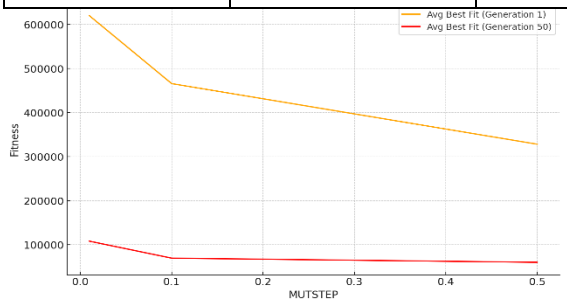

Figure 7: MUTSTEP's effect on the average of best fitness at final generation vs generation 50 over 6 runs on function 1

The mutation step size significantly affects the convergence of the function in that it affects how big the changes are to individuals. A MUTSTEP of 0.5 provided the best results, majorly outperforming the other values. This allowed the algorithm to explore the complex problem space effectively and made sure it didn't get stuck in local optima. The big jump between the best fitness at generation 1 vs generation 50 shows that a large mutation step size improves the solution heavily in the early stages. Both of the other values were too small to properly explore the problem space and couldn't converge on a more effective solution, and possible got stuck in local optima. This highlights the complex problem space that function 1 has, and shows the significance of the mutation step size in comparison to mutation rate and population size.

MUTSTEP	Avg best fit	Avg avg fit	Avg best fit (g1)	Avg best fit (g50)
0.01	2895.557788	4486.601737	6231.841436	4058.25791
0.1	3096.852681	4248.999936	6321.957488	3906.868407
0.5	2549.463244	4231.548853	6269.265062	4069.403122

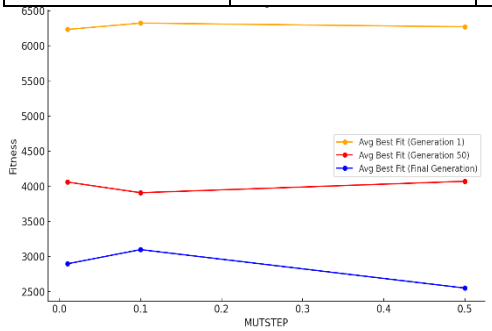

Figure 8: MUTSTEP's' effect on the average of best fitness at final generation vs generation 50 vs generation 1 over 6 runs on function 2

similarly, a mutation step size of 0.5 provided the best results for function 2, while not as large of a difference. It also had the worst best fitness result at generation 50, showing that for function 2, a large mutstep seemingly works better at a larger generation number. Although a mutation step size of 0.5 is the best result out of the experiments, it is still much higher than the optimal result of this function, showing that, like population size and mutation rate before it, mutation step size is not as significant on its own.

Generations	Avg best fit	Avg avg fit	Avg best fit (g1)	Avg best fit (g50)
50	74080.58999	560620.7663	441454.5091	74080.58999
100	19041.3223	171873.3396	487803.1499	72928.55007
200	4176.854871	10431.69209	323651.2374	52376.92608

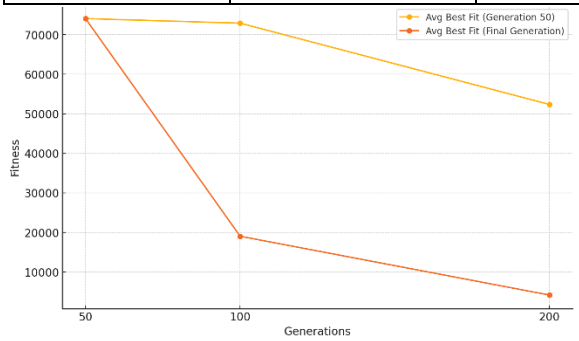

Figure 9: number of generations effect on the average of best fitness at final generation vs generation 50 over 6 runs on function 1

For function 1, the number of generations had a seemingly very significant effect on the fitness values. From 50 generations to 100 generations, there was a massive jump, from 74000 to 19000. From 100 to 200 generations, there was still a big jump, but definitely less impactful than the jump from 50 to 100, going from 19000 to 4000. Depending on the necessities of the person doing these experiments, a generations value of 100 may be good enough, seeing the huge jump, but for the purposes of this report, a generations value of 200 performed the best, and in isolation, had the most impact out of any of the parameters.

Generations	Avg best fit	Avg avg fit	Avg best fit (g1)	Avg best fit (g50)
50	4023.184983	6001.50355	5907.735552	4023.184983
100	2876.696227	4340.742271	6212.578331	4186.121582
200	2052.312613	2458.469469	6638.755706	3991.246974

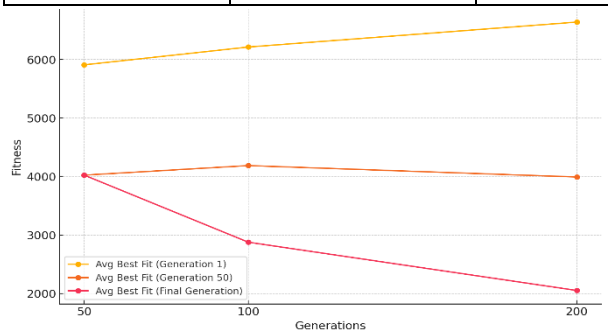

Figure 10: number of generations effect on the average of best fitness at final generation vs generation 50 vs generation 1 over 6 runs on function 2

like function 1, function 2 also had the best result at generations = 200, but a slower improvement from 100 to 200, compared with 50-100. On the other hand, the 'slower' improvement was only slight, maybe suggesting that function 2 would have a better result to computational cost ratio at even higher generations. in the case of both of these functions, a higher number of generations inherently will increase the best fitness result, due to the fact that a higher number of generations only gives the function more chances to improve, but at a certain point, it is not worth the resources to do so. In the case of function 1, that point seems to be at around 200, potentially less with more testing, whereas in the case of function 2, it could potentially be higher.

After conducting individual experiments on each parameter, an optimal set of values was found for each function. In the case of function 1, the experiments found the best values that provided the best result, but for function 2, different values provided better results than what was predicted through the experimentation. In the initial experiments, the best performance for function 1 was seen with a mutation rate ranging from 0.1 to 0.5 providing relatively similar performances and a mutation step size of 0.5 giving the best average fitness, indicating that larger jumps were beneficial for exploration. A population size of 50 provided a good balance

between diversity and convergence speed, compared to smaller and larger populations and increasing generations from 50 to 200 led to better convergence, but with diminishing returns. However, during combined parameter testing, it was found that the mutation rate of 0.5 and mutation step size of 0.5 performed the best when used together. This combination allowed both effective exploration and refined exploitation of the problem space. Additionally, while a population size of 50 was retained, the use of 200 generations provided enough evolutionary time for convergence without introducing significant inefficiencies or unnecessary computational costs. For function 2, a mutation rate of 0.5 was preferred, allowing sufficient exploration while maintaining some stability, with a mutation step size of 0.5 working well in isolation, balancing exploration and exploitation. A population size: of 50 was found to be optimal for balancing diversity and convergence and increasing to 200 generations produced the best results, but with diminishing return. however, combined testing showed that 250 generations and a slightly higher mutation step size of 0.7 performed better for function 2. the larger number of generations provided more opportunities for gradual improvement, while the higher mutation step size allowed for broader exploration, which appeared to be beneficial in this more rugged and extensive search space.

Algorithm	Best fitness function 1	Best fitness function 2
PSO	130.2555249	4604.728254
EA	164.4392905	1469.745481

Particle Swarm Optimisation (PSO) (Kennedy and Eberhart, 1995) was chosen to compare against the evolutionary algorithm for these two functions, because of its similarity to Evolutionary algorithms. It is also a population based algorithm, but relies on the swarm's social and cognitive behaviour to guide the search for optimal solutions. The Particle Swarm Optimisation algorithm was implemented using the PySwarms library (Miranda, 2018), which provides a convenient interface for configuring behaviour and managing parameter tuning. This comparison aims to evaluate how different population-based methods perform on the given minimisation functions. Like EA's, the PSO uses 'options', which are like EA's parameters, to guide the search through the problem space. To make sure that both algorithms were equal, PSO's options were set to be similar, where possible considering they don't match exactly, with c1 being set to 0.5 to mimic the mutation rate for both functions. C1 controls the PSO's exploration, with a higher C1 having more exploration and vice versa, like mutation rate. The inertia rate, or w, is functionally similar to the MUTSTEP in an EA, which is why is was set to 0.7, and 0.9, in function 1 and 2 respectively, mimicking the higher mutation steps of 0.5 and 0.7 used in the evolutionary algorithms. Iterations, being like generations, were set to 200 and 250 respectively, and n_particles was 50 for both functions, acting like population size in the evolutionary algorithm. As seen in the table above, the particle swarm optimisation algorithm outperformed the evolutionary algorithm in function 1, having an average best fitness of around 130, whereas the EA's was around 165. PSO's parameters may have allowed it to more effectively explore and converge to a better solution for function 1, which suggests that the landscape of Function 1 is more suitable for PSO's behaviour. The PSO could balance exploration and exploitation more

dynamically than EA, which may have led to a more efficient search in Function 1's problem space. On the other hand, the two results were very close, seemingly showing that they were both very effective with the search through the problem space, showing that both are strong candidates for solving function 1. For function 2, EA performed significantly better, achieving an average best fitness of roughly 1470, while PSO reached 4600. This could indicate that Function 2's landscape is more complex or contains more local optima, and the random nature of EA's mutation may have been more effective in avoiding these local traps. The swarm-based approach of PSO may have caused it to converge too quickly to suboptimal regions in this case, suggesting that the parameter settings for PSO may need further tuning to adapt to the characteristics of Function 2.

The experiments revealed that Particle Swarm Optimisation and the Evolutionary Algorithm perform differently across different types of minimisation functions. PSO showed better results for Function 1, highlighting its strong ability to balance exploration and exploitation. In contrast, EA outperformed PSO for Function 2, likely due to its randomness, which helped avoid local optima. These results suggest that the choice of algorithm should be informed by the characteristics of the problem space—PSO is more effective for simpler landscapes, whereas EA excels in more rugged problem spaces. Overall, these findings demonstrate the importance of choosing the right optimisation algorithm depending on the problem type and further highlight the value of parameter tuning. Future work could involve more rigorous parameter testing for PSO, to further finetune the results.

The experiments conducted on these functions for this report have provided a clear understanding of evolutionary algorithms, their parameters and how to solve functions using them. It has also provided an understanding of the strengths and weaknesses of evolutionary algorithms compared to particle swarm optimisation. After careful and systematic testing on the two functions using evolutionary algorithms, a set of parameters were found, that came back with the most optimised results. These were then tested against a particle swarm optimisation algorithm, using the same parameters, to see which algorithm would be best suited for each function. The conclusion was that evolutionary algorithms are better suited for more complex problem spaces, shown by the significantly better results in function 2, whereas a PSO may perform marginally better in a less complex problem space, shown by the close results in function 1. One limitation of this study was the limited parameter tuning performed for PSO, which may have affected its performance in the more complex Function 2. Additionally, the number of runs for each algorithm was limited to five, which may not fully capture the variance in performance due to the random nature of these algorithms. Another limitation was the systematic testing of parameters. The parameters were tested in isolation, to see how they each individually affected the function and the results that came out, but testing of parameters together to get a

bigger idea of the overall picture may have created more finetuned parameters and results for both of the functions. These results are based on a limited number of runs. Additional runs or statistical tests (e.g., standard deviations) would be required to confirm the statistical significance of the observed results. Future work could include more comprehensive parameter testing for all algorithms involved. In summary, for Function 1 the EA was slightly outperformed by PSO, suggesting that the more cooperative search strategy of PSO suited this problem's landscape. Conversely, for Function 2, the EA delivered better results, likely due to its randomised mutations helping avoid local optima in a more complex search space. These outcomes highlight the importance of tailoring the choice of optimisation method and parameter settings to the nature of the problem.

Miranda, L. J. (2018) *PySwarms: a Python library for Particle Swarm Optimization*. Available at: https://github.com/ljvmiranda921/pyswarms (Accessed: 1 December 2024).

Eiben, A.E. and Smith, J.E. (2003) *Introduction to Evolutionary Computing*. Berlin: Springer

Kennedy, J. and Eberhart, R.C. (1995) 'Particle swarm optimization', *Proceedings of the IEEE International Conference on Neural Networks*, Perth, Australia, 27 November–1 December. Piscataway, NJ: IEEE, pp. 1942–1948

The appended code includes comments explaining the evolutionary loop, fitness evaluation, mutation, and crossover steps.

Function 1

```
import random

import copy

import matplotlib.pyplot as plt


P = 50

N = 20
```

```
MUTRATE = 0.5
GENERATIONS = 200
MIN = -10
MAX = 10
MUTSTEP = 0.5

class Individual:
    def __init__(self):
        self.gene = [0]*N
        self.fitness = 0

# initialise the population with random individuals
population = []
for x in range (0, P):
    tempgene=[]
    for y in range (0, N):
        tempgene.append(random.uniform(MIN,MAX))
    newind = Individual()
    newind.gene = tempgene.copy()
    population.append(newind)

def test_function(ind):
    d = 20
    utility=(ind.gene[0]-1)**2
    for i in range(1,d):
        utility += i * ((2*(ind.gene[i]**2)-ind.gene[i-1])**2)
    return utility
```

```
avg_fit_off_list = []
best_fit_off_list = []

# main evolutionary loop
for generation in range(GENERATIONS):
  for ind in population:
    ind.fitness = test_function(ind)

  offspring = []

  # apply mutation to create offspring
  for i in range(0, P):
    newind = Individual()
    newind.gene = []
    for j in range(0, N):
      gene = population[i].gene[j]
      mutprob = random.random()
      if mutprob < MUTRATE:
        alter = random.uniform(-MUTSTEP, MUTSTEP) # using mutstep and mutrate to
mutate
        gene = gene + alter
        gene = max(MIN, min(MAX, gene))
      newind.gene.append(gene)
    offspring.append(newind)

  # apply crossover between pairs of offspring
  toff1 = Individual()
  toff2 = Individual()
```

```
temp = Individual()
for i in range( 0, P, 2 ):
   toff1 = copy.deepcopy(offspring[i])
   toff2 = copy.deepcopy(offspring[i+1])
   temp = copy.deepcopy(offspring[i])
   crosspoint = random.randint(1,N)
   for j in range (crosspoint, N):
      toff1.gene[j] = toff2.gene[j]
      toff2.gene[j] = temp.gene[j]
   offspring[i] = copy.deepcopy(toff1)
   offspring[i+1] = copy.deepcopy(toff2)

#fitness evaluation
for ind in offspring:
   ind.fitness = test_function(ind)

total_fit_pop = sum(ind.fitness for ind in population)
total_fit_off = sum(ind.fitness for ind in offspring)

avg_fit_pop = total_fit_pop / P
avg_fit_off = total_fit_off / P

best_fit_pop = min(population, key=lambda ind: ind.fitness)
best_fit_off = min(offspring, key=lambda ind: ind.fitness)
best_fit_pop_num = min(ind.fitness for ind in population)
best_fit_off_num = min(ind.fitness for ind in offspring)

# replace worst offspring with best individual
```

```
    worst_fit_off = max(offspring, key=lambda ind: ind.fitness)
    worst_index_off = offspring.index(worst_fit_off)
    if avg_fit_pop < avg_fit_off:
        offspring[worst_index_off] = copy.deepcopy(best_fit_pop)

    #keep the best solution in the population
    best_index_off = offspring.index(best_fit_off)
    if best_fit_pop.fitness<best_fit_off.fitness:
        offspring[best_index_off] = copy.deepcopy(best_fit_pop)
    population = copy.deepcopy(offspring)

    avg_fit_off_list.append(avg_fit_off)
    best_fit_off_list.append(min(ind.fitness for ind in offspring))

    print(f"Generation {generation + 1}:")
    print(f"total population fitness: {total_fit_pop}, total offspring fitness: {total_fit_off}")
    print(f"avg population fitness: {avg_fit_pop}, avg offspring fitness: {avg_fit_off}")
    print(f"best population fitness: {best_fit_pop_num}, best offspring fitness:
{best_fit_off_num}\n")

print(f"Avg fitness list length: {len(avg_fit_off_list)}")
print(f"Best fitness list length: {len(best_fit_off_list)}")
plt.plot(range(1, GENERATIONS + 1), avg_fit_off_list, label='Average Offspring Fitness')
plt.plot(range(1, GENERATIONS + 1), best_fit_off_list, label='Best Offspring Fitness')
plt.title('Offspring Fitness over Generations')
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.legend()
```

```
plt.grid()
plt.show()
```

function 2

```
import random
import copy
import matplotlib.pyplot as plt
import math

P = 50
N = 20
MUTRATE = 0.5
GENERATIONS = 250
MIN = -500
MAX = 500
MUTSTEP = 0.7

class Individual:
  def __init__(self):
    self.gene = [0]*N
    self.fitness = 0

# initialise the population with random individuals
population = []
for x in range (0, P):
  tempgene=[]
```

```
    for y in range (0, N):
        tempgene.append(random.uniform(MIN,MAX))
    newind = Individual()
    newind.gene = tempgene.copy()
    population.append(newind)

def test_function(ind):
    d = 20
    utility=418.9829*d
    for i in range(d):
        utility-=ind.gene[i]*math.sin(math.sqrt(abs(ind.gene[i])))
    utility+=31
    return utility

avg_fit_off_list = []
best_fit_off_list = []

# main evolutionary loop
for generation in range(GENERATIONS):
    for ind in population:
        ind.fitness = test_function(ind)

    offspring = []

    # apply mutation to create offspring
    for i in range(0, P):
        newind = Individual()
        newind.gene = []
```

```
    for j in range(0, N):
      gene = population[i].gene[j]
      mutprob = random.random()
      if mutprob < MUTRATE:
        alter = random.uniform(-MUTSTEP, MUTSTEP)  # using mutstep and mutrate to
mutate
        gene = gene + alter
        gene = max(MIN, min(MAX, gene))
      newind.gene.append(gene)
    offspring.append(newind)

    # apply crossover between pairs of offspring
  toff1 = Individual()
  toff2 = Individual()
  temp = Individual()
  for i in range( 0, P, 2 ):
    toff1 = copy.deepcopy(offspring[i])
    toff2 = copy.deepcopy(offspring[i+1])
    temp = copy.deepcopy(offspring[i])
    crosspoint = random.randint(1,N)
    for j in range (crosspoint, N):
      toff1.gene[j] = toff2.gene[j]
      toff2.gene[j] = temp.gene[j]
    offspring[i] = copy.deepcopy(toff1)
    offspring[i+1] = copy.deepcopy(toff2)

  #fitness evaluation
  for ind in offspring:
```

```
    ind.fitness = test_function(ind)

total_fit_pop = sum(ind.fitness for ind in population)
total_fit_off = sum(ind.fitness for ind in offspring)

avg_fit_pop = total_fit_pop / P
avg_fit_off = total_fit_off / P

best_fit_pop = min(population, key=lambda ind: ind.fitness)
best_fit_off = min(offspring, key=lambda ind: ind.fitness)
best_fit_pop_num = min(ind.fitness for ind in population)
best_fit_off_num = min(ind.fitness for ind in offspring)

 # replace worst offspring with best individual
worst_fit_off = max(offspring, key=lambda ind: ind.fitness)
worst_index_off = offspring.index(worst_fit_off)
if avg_fit_pop < avg_fit_off:
    offspring[worst_index_off] = copy.deepcopy(best_fit_pop)

    #keep the best solution in the population
best_index_off = offspring.index(best_fit_off)
if best_fit_pop.fitness<best_fit_off.fitness:
    offspring[best_index_off] = copy.deepcopy(best_fit_pop)
population = copy.deepcopy(offspring)

avg_fit_off_list.append(avg_fit_off)
best_fit_off_list.append(min(ind.fitness for ind in offspring))
```

```
    print(f"Generation {generation + 1}:")
    print(f"total population fitness: {total_fit_pop}, total offspring fitness: {total_fit_off}")
    print(f"avg population fitness: {avg_fit_pop}, avg offspring fitness: {avg_fit_off}")
    print(f"best population fitness: {best_fit_pop_num}, best offspring fitness: {best_fit_off_num}\n")

print(f"Avg fitness list length: {len(avg_fit_off_list)}")
print(f"Best fitness list length: {len(best_fit_off_list)}")
plt.plot(range(1, GENERATIONS + 1), avg_fit_off_list, label='Average Offspring Fitness')
plt.plot(range(1, GENERATIONS + 1), best_fit_off_list, label='Best Offspring Fitness')
plt.title('Offspring Fitness over Generations')
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.legend()
plt.grid()
plt.show()
```