Natural Computing Coursework

Patryk Kuchta – s2595201 – B248125

University of Edinburgh

Problem 1 – preliminary steps

In this section, I will cover the use of the Particle Swarm Optimisation (PSO) algorithm for minimising the Rastrigin function in d-dimensions, where all dimensions are constrained to values within -5.12 to 5.12.

It is important to note that an inspection of this function on a graph quickly reveals that this function can easily 'trap' any algorithm into a local optimum. Therefore, the expectations for the performance of the algorithm cannot be too high in this case.

Fitness Function

In order to solve this problem using the PSO algorithm, a suitable fitness function had to be selected. This is the code for the fitness function chosen:

```
def evaluate_fitness(self, solution: np.ndarray) -> float:
    penalties = 0

    # apply a penalty for values over max
    penalties += np.sum(solution[solution > self.solution_boundary[1]] ** 2)
    # apply a penalty for values below min
    penalties += np.sum(solution[solution < self.solution_boundary[0]] ** 2)

    # we are trying to minimise penalties and the function therefore fitness will
be * -1
    return (NDimensionalRastriginProblem.__variable_rastrigin_function(solution) +
penalties) * -1
```

The output of this fitness function is the value of the Rastrigin function for that solution, with the added penalties, which are the summed squared difference between the search space boundary and the value for each dimension that has left the search boundary. This penalty is introduced to discourage the algorithm from leaving the search space. Penalties will always equal 0 if all dimensions are within the search space. This number is multiplied by * -1, because the algorithm I have written is aimed at maximising fitness.

Acceptable solutions.

Acceptable solutions have been selected based on running the PSO algorithm for the same problem but with a generous population of 100, 300 iterations, and this algorithm has been run for 10 times to select the best output of all 10 of the tests.

Using this methodology, the best optimum found for 5 dimensions had the fitness of negative 3.233616069943885 therefore the fitness that is required for the solution to be deemed as good enough will be -9.700848209831655, as this will be 3 times the best solution found, and this allows the testing to observe and reward 'quite good' solutions not just best possible.

Parameter selection

My methodology for finding the parameters will be for each population size test (starting from 1) to run the model and record the number of iterations before reaching an acceptable solution, until reaching the population of 60. Please note that for most tests the behaviour for small populations is quite random and luck-based, due to the behaviour of the PSO algorithm in those cases.

Because there is a high chance that PSO will be unable to find an optimum this good (especially for small populations), the algorithm is run 100 times, and if it is able to find a good enough optimum, its number of iterations will be taken into the overall average. Otherwise, the failure to find the optimum will be recorded in the success rate.

The search for a set of optimum parameters has been started by researching the effects of different parameters in terms of particle behaviour and divergence insert the paper here, assuming no prior knowledge of the problem the behaviour of both zigzagging and oscillating

could be useful to the problem. To achieve this behaviour, the values of inertia will be kept close to 1, whilst the sum of the forces kept equal to 4. Running the algorithm for a set of very generic parameters (inertia = 0.7, both forces = 2), quickly revealed that the algorithm makes rather slow progress, the global best would be stuck on the current perceived global best for a while.

Alpha values

This points to the issue that in this problem there is a need to de-emphasize exploitation and focus more on exploration. This pointed first to increasing the personal best force, whilst sacrificing the global best. The findings of these tests can be seen in Figure 1 and 2.

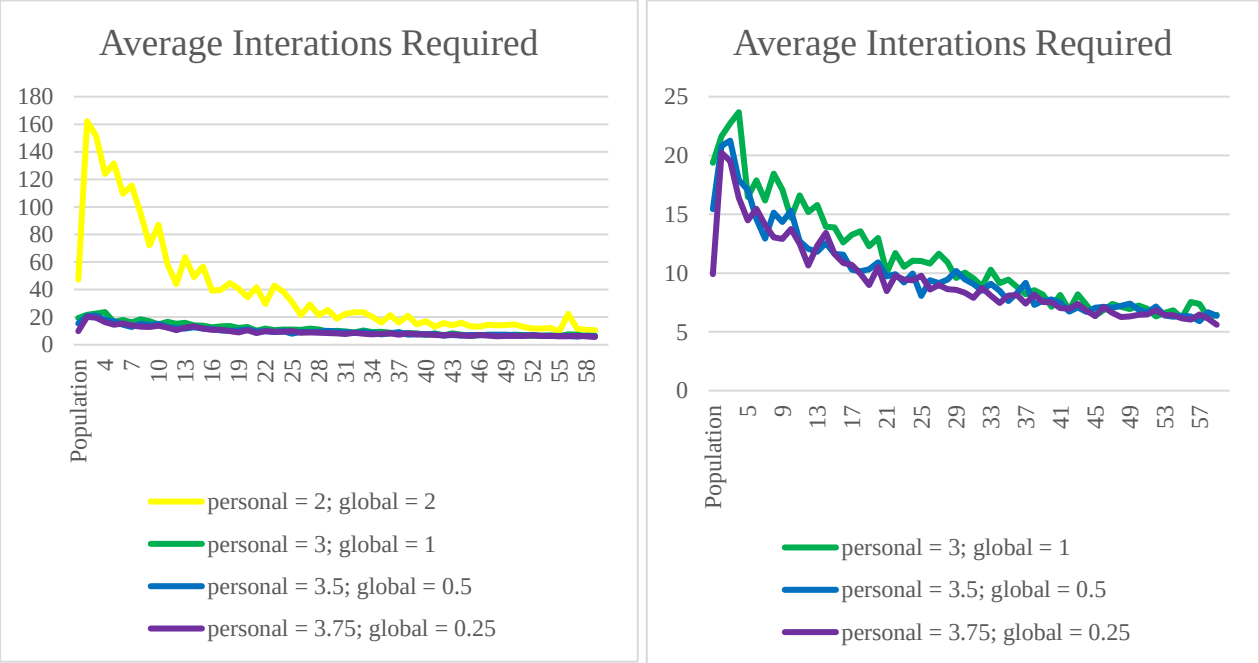

Figure 1: Average Iterations required for finding an optimal solution, for different population sizes, given different values for the personal and global best forces. The figure on the left includes the entry for personal = 2; and global = 2; whilst the figure on the right shows the other 3 entries in more detail.

These tests show that increasing the importance of the personal_best has led to large improvements in the algorithm, but they are coming at the expense of the success rate. Based on these findings, all considerations going forward will use the personal_best_force = 3.5 and

global_best_force = 0.5, as these values give a good balance between relatively reliably getting an answer and optimising the number of iterations.

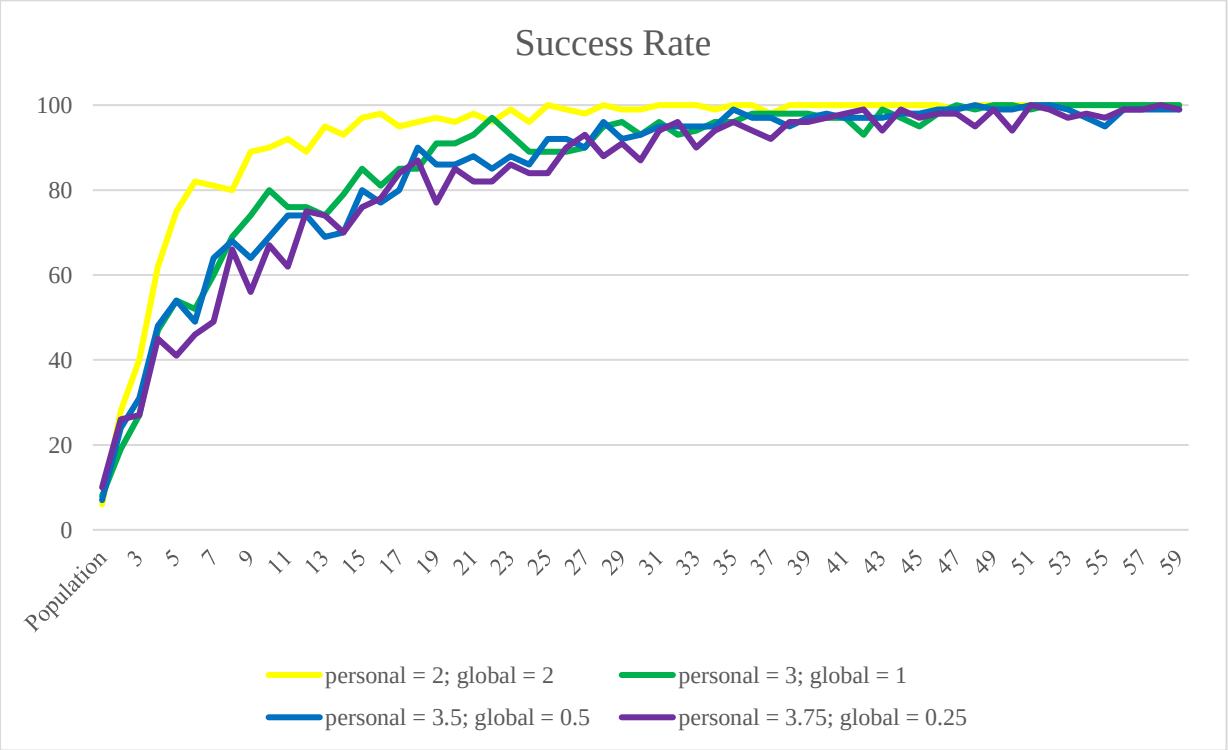

Figure 2: The percentage of successful runs of the algorithm over different population sizes, and different values for the personal and global best forces. A success rate of 100 means the algorithm has found an acceptable solution every time, whilst 0 indicates that the algorithm has never reached an acceptable solution.

Inertia

Similar tests have been performed for different inertia; the findings shown in Figure 3.

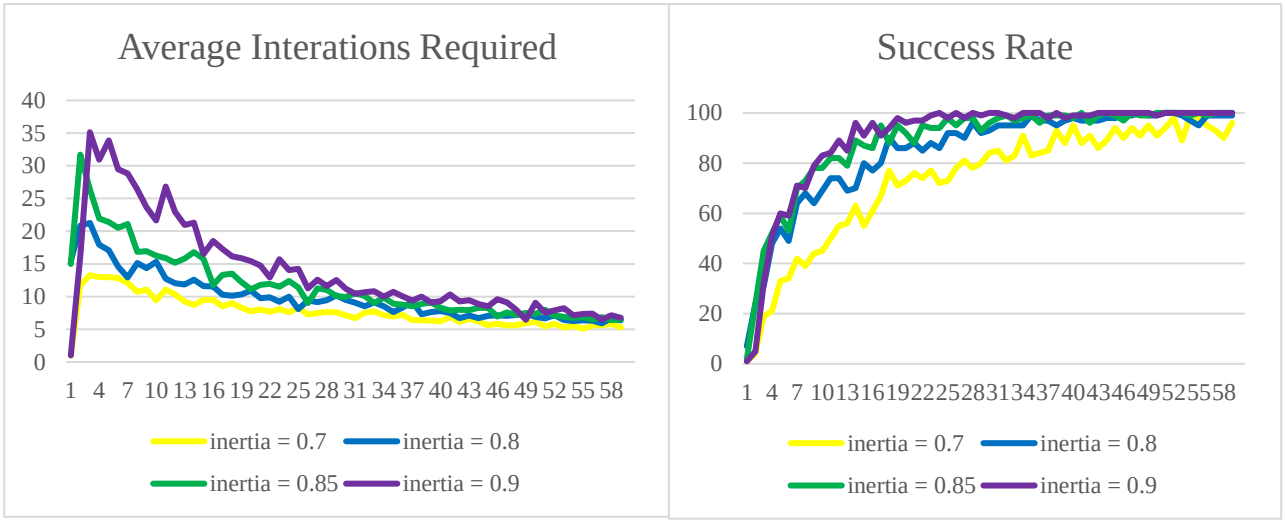

Figure 3: The average iterations and success rate in reaching an optimal solution for different population sizes given different inertia values.

In the case of inertia, there is no more clear benefit in changing the value as any improvement in the success rate comes at the expense of the average required number of iterations. The inertia of 0.85 will be used as it has shown good performance in terms of average iterations required and did not cause the success rate to go too low like in the case of inertia = 0.7.

Problem 1 – Subtask a)

Modified test set-up

These tests will be run slightly differently as it will be preferable to produce one metric for each population size rather than a more nuanced two values in the tests for the parameters. In these tests each test of a given population size (still searched between 1 and 100), will be given an equal budget for the maximum number of evaluations of the fitness it can perform. Additionally, the model will only be restarted if the progress in the algorithm stops improving* without reaching an acceptable optimum or as soon as the acceptable optimum is reached. The metric for the comparison will be how many times the model was able to reach the acceptable optimum within its evaluation budget. This metric will neatly combine computational cost and the costs associated with not converging with the perceived performance. For simplicity, this metric will be referred to as optimal convergences given k evaluations. The budget has been set to k = 1 000 000 as this is the largest number feasible with my limited computational power.

*Small note about going over budget, for simplicity going over budget within an iteration of the algorithm is allowed, but the algorithm will be stopped right after. Although it does introduce as small advantage for bigger populations, it should not be significant to the testing.

The asterisk* next to stop improving is because it is difficult to determine when a model is not making progress. Therefore, for the purposes of these tests 'stops improving' will be approximated as the model has not made any improvements to the fitness in more than 25 iterations.

Findings

For the Rastrigin problem represented in five dimensions, the result of the tests is shown in Figure 4.

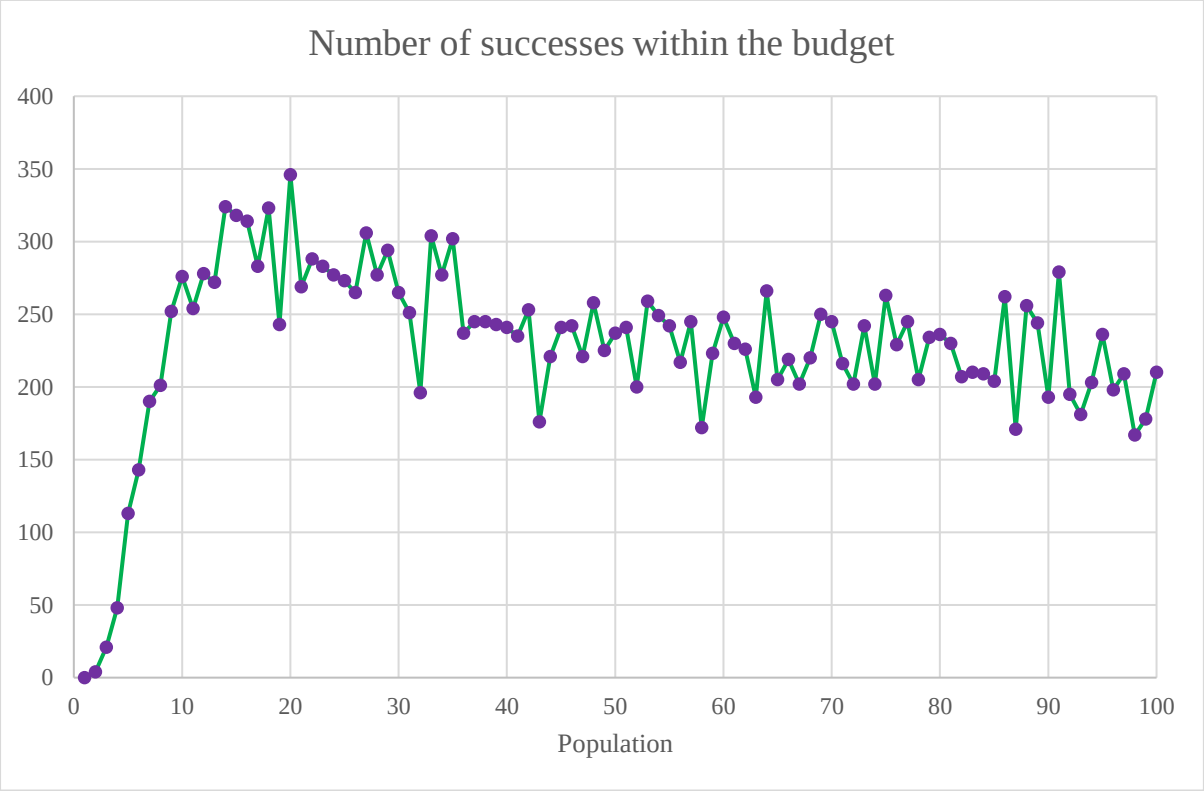

Figure 4: Number of times the PSO algorithm was able to find an optimal solution given a limited number of fitness evaluations.

These findings suggest that values between 10 and 34 can be considered optimal. The data produced is quite noisy, and drawing a single point as definitely the optimal would be ill-judged. If tasked with choosing only one value, the choice of value 20, which is where the maximum performance has been reached, would be a sensible value. However, there is it would be hard to concretely prove that it is better than any other values from 10 to 34.

Problem 1 – Subtask b)

To answer this task, I will examine what are the optimal values for the following problem complexities (numbers of dimensions of the solution): 3, 4, 5, 6, 7.

Based on a run of the PSO algorithm with generous computational power (just like in task a), the fitness of the best solution and the smallest acceptable fitness value are shown in Table 1.

Number of Dimensions	Best Fitness Found	Worst Fitness Acceptable
3	-0.7462195287025324	-2.238658586107597
4	-0.994959371603386	-2.984878114810158
5	-3.233616069943885	-9.700848209831655
6	-3.4823559128459323	-10.447067738537797
7	-5.721012612624037	-17.16303783787211

Table 1: The values of the best fitness found, and the worst fitness deemed as acceptable for each of the tested number of dimensions.

Please note that as the method for selecting the value is not perfect, it cannot be assumed that thresholds are all representing a proportional or equivalent difficulty for the program. It may even be argued that the selection method for these values could be a bit more lenient for the higher dimensionality problems.

Testing

The tests performed for this section are the same as for the previous task. The findings are shown in Figure 5. Due to the noisy nature of the data, a smoothed representation of the data is shown in Figure 6.

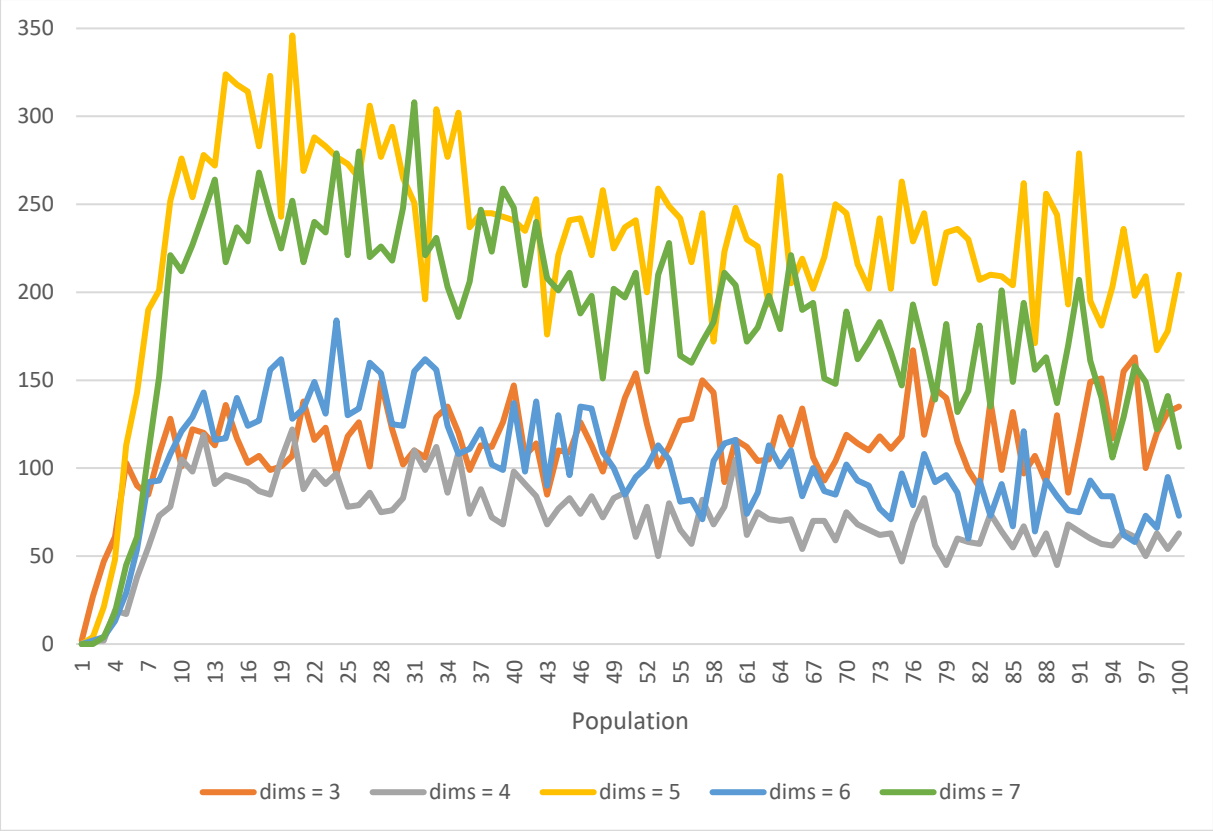

Figure 5: Number of times the PSO algorithm was able to find an optimal solution given a limited number of fitness evaluations, for each of the numbers of dimensions of the given problem.

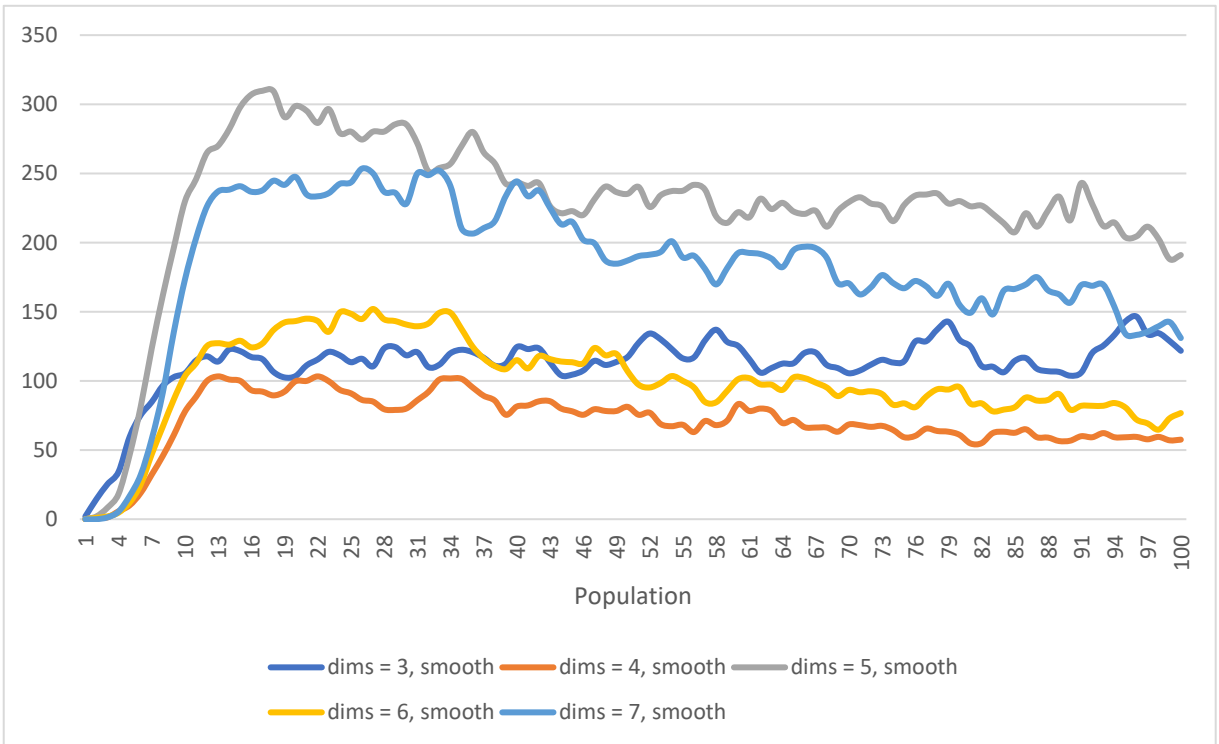

Figure 6: Data from Figure 5 represented in a smoothed form, each entry showing the moving average over the last 3 values.

For all dimensions, except for the dimensionality of three, it is visible that the set of optimal population sizes stays relatively stable at around populations between 10 and 30. This is because the dynamics of the group, which rely on each of the particles to be drawn to 2 previous optima's, will over time particles will become very similar over time if the set of particles is large enough. Therefore, we can observe the slow decrease in the metric as we raise the population as each new particle has a higher chance of not contributing by virtue of becoming too similar to other particles. This means at one point each new particle becomes less of a benefit and more of a computational burden. This behaviour is kept with the number of dimensions rising because the impact of the group dynamics and the force does not increase nor decrease with respect to the number of dimensions.

The exception is 3 dimensions, where I believe the added computation burden of a growing population was offset by purely increasing the chances of landing on a solution by luck rather than by utilising group dynamics, which makes sense given the whole search space is (10.24)^3, which makes getting a lucky solution quite a frequent occurrence. However, it's worth noting that even in those peculiar circumstances, the choice of a population from the set 10–30 would still yield good performance in comparison to the rest of the options (although not best).

Question 2 – preliminary steps

Encoding

Encoding will be achieved using a k^2 binary vector where each scalar denotes whether a given value in the games' matrix should be deleted or not (deletion = 0, retention = 1), and ought to be read as:

$$DeletionOf(x, y) = EncodingVector(k^y + x)$$

This makes the mutations simple, as they could simply be based on bit flipping, and the fitness evaluation is also quite computationally simple, which is great as this operation will be performed often.

Parameter selection

For this problem, the ranges for possible parameters have been constrained into the following ranges:

- population_size from 100 to 400, with step 10 (i.e. 100, 110, 120..., 390, 400)
- mutation_rate from 0.1 to 0.9, with step 0.1 (i.e. 0.1, 0.2, 0.3..., 0.8, 0.9)
- crossover_rate from 0.1 to 0.9, with step 0.1 (i.e. 0.1, 0.2, 0.3..., 0.8, 0.9)

For each pair of parameters, the genetic algorithm will be run for several iterations, until exhausting the given budget of evaluations, restarting after finding a correct solution or when the number of iterations goes above 20. The final performance metric will be the number of times that the genetic algorithm has reached the correct solution.

This is quite constraining, but it should be able to produce at least meaningful information pointing to better parameters or in the best cases, a good set of parameters.

Mutation Scheme

The mutation scheme has been kept quite simple as it will randomly select a value from the encoding vector and flip it (0 will become 1’s, and vice versa).

Selection Scheme

The selection scheme used in this algorithm will be the 'Roulette Wheel Selection' where the chance of survival for each genome is based on its relative fitness which can be expressed as:

$$SurvialProbability(x) = \frac{fitness(x)}{\sum fitness(x')}$$

Based on that probability distribution, the new population is built with each of the slots being allocated randomly based on that probability distribution. This selection scheme has been selected as it still gives a chance for the less-fit solutions to make it into the next generation. This makes the generation diverse. The algorithm written for this functionality also includes the optional `elitism` parameter which can be set to tell the function to keep a few best solutions by default. Elitism has been kept at 0 for all these experiments as this encourages diversity which is highly needed for this problem.

Crossover Scheme

The crossover scheme is based on slicing one of the solution vectors and inserting that 'slice' into the same part of the vector inside the other vector. The placing of where the slicing begins and ends is selected randomly.

Problem 2 – Question a)

Firstly, selecting the parameters for this problem will be useful. To select the parameters, the 3 x 3 version of the problem will be used so that the computations are a bit faster. The First observations are that given a budget of 100 000 evaluations, the algorithm was still struggling to find the solutions for many parameters. Table 2 shows the top 40 best-performing parameters tuples.

population	mutation	crossover	successes
370	0.2	0.9	**29**
330	0.1	0.9	**27**
330	0.2	0.9	**27**
330	0.3	0.9	**27**
370	0.4	0.9	**27**
380	0.1	0.9	**25**
340	0.2	0.9	**23**
370	0.3	0.8	**23**
340	0.1	0.9	**22**
390	0.1	0.9	**22**
340	0.1	0.8	**21**
340	0.3	0.8	**21**
380	0.2	0.9	**20**
400	0.1	0.9	**20**
330	0.5	0.9	**19**
340	0.3	0.6	**19**
350	0.5	0.9	**19**
370	0.1	0.7	**19**
400	0.2	0.8	**19**
370	0.2	0.7	**18**
340	0.3	0.7	**17**
350	0.6	0.9	**17**
350	0.7	0.9	**17**
360	0.1	0.8	**17**
380	0.1	0.7	**17**
400	0.6	0.9	**17**
340	0.1	0.5	**16**
350	0.8	0.9	**16**
370	0.7	0.9	**16**

Table 2: Top 30 parameters tuples with the number of successful runs shown for each of them.

Table 2 shows that there are a lot of variances in the mutation_rate for tuples that still performed well. This points to the fact that in the current state, the algorithm doesn't get much of a benefit from mutations. One thing that is quite consistent is that the larger population sizes have performed better and that the algorithm performs better for larger crossover rates.

But this testing done to produce an optimal tuple of parameters shows the issues with this fitness function in general, which is that the genetic algorithm has no way of knowing which solutions are better if they aren't the correct solution (if one of them is, then the algorithm is done regardless). This makes it so that a larger population size, which necessitates more random initialisations, is better as the algorithm is basically reduced in its functionality to just a random search. Mutations even if they happen often are not useful as it is better to just create a new random solution, rather than mutate an existing one if the only thing we know about it is that it is incorrect. The tendency to favour high crossover rates can be explained as it introduces a lot of new random solutions and when used with a value close to one it will happen for all the genomes, which helps the algorithm, as it makes the current random search even more random.

In Table 4 are the findings for multiple dimensionalities of the problem (using parameters population: 350, mutation_rate: 0.5, crossover_rate: 0.9, and 100 generations, 100 times (each time with a new model (i.e. resetting)). The number represents the number of times the model has reached the correct solution.

Matrix size	Successes per 100 trials
2 by 2	100
3 by 3	100
4 by 4	36
5 by 5	0
6 by 6	0

Table 4: Number of successes per 100 trials for different matrix sizes, when running the genetic algorithm with a flawed fitness function.

This shows that as the dimensionality of the problem grows this fitness function becomes more of an issue as random search becomes less and less feasible as we increase the problems complexity.

Problem 2 – Subtask b)

Proposed fitness function

A fitness function better suited for larger instances of the problem will be a fitness function:

- that will tell the model when it is getting closer to the correct sums.
- that will reward getting the correct sum in each dimension.

This will allow it to select solutions that are in theory closer to the correct answer for future generations. This scheme can be represented using the code below:

```
def evaluate_fitness(self, solution: np.ndarray) -> float:
    solution_sums = self.game.calculate_the_sums_given_solution(solution)

    row_wise_errors_squared = (self.game.correct_sums['rows'] - solution_sums[0])
** 2
    column_wise_errors_squared = (self.game.correct_sums['cols'] -
solution_sums[1]) ** 2

    reaching_zero_reward = (len(row_wise_errors_squared[row_wise_errors_squared ==
0]) +

len(column_wise_errors_squared[column_wise_errors_squared == 0])) * 4

    # because we are minimising the error, we multiply by -1
    return (np.sum(row_wise_errors_squared) + np.sum(column_wise_errors_squared))
* -1 + reaching_zero_reward
```

The fitness in this function is defined as the negative sum of square differences between the correct sums and the sums that the given solution gives, plus a +4 reward for each sum that the solution gets correctly.

Parameter selection

To choose the parameters, the analysis was the same as in the previous subtask. However, it after a quick test this fitness function has proven that the number of allowed interactions before the default restarting to 200, because with this fitness function the algorithm is able to make progress for longer.

The best performance has been reached for population_size 150, mutation rate 0.1, and crossover_rate 0.9, and many of the best parameter tuples it mutation_rate was between 0.1 - 0.3 and the crossover_rate 0.7–0.9, and the population_size within 120 to 190. Therefore, to limit the effect of noise which might have been the reason for the best tuple, the final parameters chosen will be:

- population_size = 150
- mutation_rate = 0.2
- crossover_rate = 0.8

It's worth noting that this population size means that this algorithm is significantly less computationally costly than the previously tested algorithm. The findings are shown in Table 5.

Matrix size	Successes per 100 trials
2 by 2	91
3 by 3	92
4 by 4	89
5 by 5	33
6 by 6	18

Table 5: Number of successes per 100 trials for different matrix sizes, when running the genetic algorithm with a better fitness function.

This fit ness function scales much better with growth of the matrix, and was able to find solutions in cases of 5 by 5 and 6 by 6 whereas the previous fitness function did not yield any solutions for these dimensions.

Problem 2 – Subtask c)

Different parameter pairs affect the solution. Crossover rate will tend to create solutions that are combinations of low deletion of small values that iteratively get closer to the correct solution. One could consider these solutions as 'not optimal' as they will have more deletions than other solutions (this is assuming that there is more than one solution which is not always the case for a given game). Population size and mutation rate will tend to create more random solution that can be both considered 'optimal' or 'not optimal'.

Problem 3 – Subtask a)

Selection Scheme

The selection scheme for this problem has been kept from the previous exercise. Please refer to Question 2 – Preliminary steps, Selection Scheme for more details.

Mutation and Initialization Scheme

The mutation and initialization scheme is based on random growth in a random part of the tree, whilst discarding the previous structure present at that node. This random growth is achieved by this function:

```
def grow_randomly(self, wanted_depth):
    def generate_terminal():
        if random.random() < 0.5:
            self.stored = 'constant-' + str(
                self.problem.possible_constants[
                    random.randint(0, len(self.problem.possible_constants) - 1)
                ])
        else:
            self.stored = 'input-' + str(random.randint(0,
self.problem.input_vector_length - 1))

    if wanted_depth == 1:
        generate_terminal()
    elif wanted_depth > 1:
        random_chance = random.random()
        if random_chance < 0.3:
            generate_terminal()
        else:
            self.stored = 'function-' + self.problem.function_labels[
                random.randint(0, len(self.problem.function_labels) - 1)]

            new_left = MathematicalFunctionTreeNode(self.problem, 'input-0')
            new_right = MathematicalFunctionTreeNode(self.problem, 'input-0')

            new_left.grow_randomly(wanted_depth - 1)
            new_right.grow_randomly(wanted_depth - 1)

            self.left = new_left
            self.right = new_right
```

This code for each Node will create a random structure, limiting itself to the wanted_depth parameter. If the wanted_depth is equal to 1 then just the value stored will be set to one of the terminal values, either ‘constant-X’ or ‘input-X’ with equal probabilities. If there is still depth left, the algorithm with a probability of 0.3, will still terminate a terminal value. Otherwise, a

function will be created ‘function-‘ with one of the function labels available. The left and right nodes will be created and the `grow_randomly` algorithm will be called upon them with depth adjusted down by one.

This algorithm is run for a random node of the tree when mutating. The selection of the node is done by recursively trying to mutate with probability set to mutation_probabilty, and otherwise running the mutation algorithm for both children of the current node. In terms of code, this looks as follows:

```
def mutate(self, mutation_prob):
    dice_roll = random.random()

    if dice_roll < mutation_prob:
        self.grow_randomly(2)
    else:
        if self.left:
            self.left.mutate(mutation_prob)
        if self.right:
            self.right.mutate(mutation_prob)
```

For the initialization a simple node is initialized and then the grow_randomly. This is done with this code:

```
def generate_random_solution(self):
    new_solution = MathematicalFunctionTreeNode(self, 'input-0')
    new_solution.grow_randomly(self.max_solution_depth)
    return new_solution
```

Crossover Scheme:

To perform crossover, a random index within the tree of each of the genomes omits the root (this is because replacing at the root is not really crossover but rather just replacement and it would be significantly more complicated computationally). Based on the selected index a part of the tree of the other genome gets copied and that copy is inserted into the other tree at the index. Below is the code for each of these operations:

```
def make_copy(self):
    copy = MathematicalFunctionTreeNode(self.problem, self.stored)
    if self.left:
        copy.left = self.left.make_copy()
    if self.right:
        copy.right = self.right.make_copy()
    return copy

def cut_and_return_copy(self, index):
    if index == 0:
        return self.make_copy()
    else:
        index -= 1
        if self.left:
            left_volume = self.left.get_volume()
            if index < left_volume:
                return self.left.cut_and_return_copy(index)
            else:
                index -= left_volume
        if self.right:
            right_volume = self.right.get_volume()
            if index < right_volume:
                return self.right.cut_and_return_copy(index)
        raise IndexError('index exceeds tree size')

# does not work for replacing the root
def insert_tree(self, index, inserted_tree):
    if index == 0:
        raise IndexError('index has reached zero')
    elif index == 1:
        self.left = inserted_tree
        return
    else:
        index -= 1
        if self.left:
            left_volume = self.left.get_volume()
            if index < left_volume:
                return self.left.insert_tree(index, inserted_tree)
            else:
                index -= left_volume
        if index == 0:
            self.right = inserted_tree
            return
        elif self.right:
            right_volume = self.right.get_volume()
            if index < right_volume:
                return self.right.insert_tree(index, inserted_tree)
        raise IndexError('index exceeds tree size')

def crossover(self, other):
    self_volume = self.get_volume()
    other_volume = other.get_volume()
    if self_volume == 1 or other_volume == 1:
        # crossover with just one node does not contribute much but makes
the algorithm more complicated
        return
    self_cut_index = random.randint(1, self.get_volume()-1)
    other_cut_index = random.randint(1, other.get_volume()-1)

    inserted_tree = other.cut_and_return_copy(other_cut_index)
    self.insert_tree(self_cut_index, inserted_tree)
```

Fitness function:

To evaluate the fitness of each of the solutions firstly a method that computes the output of the solution for each of the inputs had to be created. This was done using recursion using the following code:

```
def calculate(self, inputs):
    if 'input-' in self.stored:
        return inputs[int(self.stored[6:])]
    elif 'constant-' in self.stored:
        return float(self.stored[9:])
    elif 'function-' in self.stored:
        function_label = self.stored[9:]
        function = self.problem.name_to_function_map[function_label]
        return function(self.left.calculate(inputs),
self.right.calculate(inputs))
```

With the outputs for each of the inputs, the fitness can be simply evaluated as the distance between the outputs from the proposed solution and target outputs. This is done with this code:

```
def evaluate_fitness(self, solution) -> float:

    sum_of_errors = 0

    for inputs, target in self.training_data:
        output = solution.calculate(inputs)

        sum_of_errors += abs(target - output)**self.fitness_error_exponent

    return -1 * sum_of_errors
```

This code allows to adjust the exponent for the distance therefore, allowing to user to choose between the Euclidean distance, Manhattan Distance or more.

Problem 3 – Subtask b)