Introduction

Two main problems were to be solved using genetic algorithms. The first being a simple optimiser that has the function of finding a specified number. The second is to create an optimiser to optimise a set of parameters, the coefficients, for a 5th-order polynomial. Multiple methods have been created to solve each problem, each with their own advantages and disadvantages that are discussed in this report. Holland's Scheme Theorem is used and discussed throughout the report and some ideas were used to create the new methods and optimisers. Each method and its entire code can be found in the appendix referenced in corresponding the title.

Exercise 1

The goal of this exercise was to create a simple optimiser that finds a specified number with the shortest number of iterations.

Method 1 - Adaption of Example code – (Appendix A)

From the example code provided, a few alterations can be made to achieve this. There were two key modifications to make the optimiser search for a single integer. The first was to change `i_length` and set that equal to 1 (as shown in the code below). This was because the target is a single integer, and so each individual within the population must be of the same structure.

```
p_count = 100
i_length = 1
i_min = -1000
i_max = 1000
target = randint(i_min, i_max)
generations = 500
```

Code snippet 1

The second modification was to change the fitness function. The fitness function is shown below. The fitness is a measure of how 'close' an individual is to the target. For this situation, the fitness is measured as the absolute difference between the target and current individual. It is common in genetic algorithms to maximise a fitness function but in this situation, minimising the fitness function was more intuitive.

```
def fitness(individual, target):
    # Determine the fitness of an individual. Lower                 is better #

    # individual:   The individual to evaluate
    # target:       The target number individuals are aiming for
    return abs(target - individual[0])
```

Code snippet 2

As code snippet 1 shows, `i_min` and `i_max` are set manually and the `target` is set randomly between that range. Using the code in snippet 3, this genetic algorithm was run 1000 times and a counter was used to check when the target was met as well as the number of iterations required. If the current `fitness_history` value is 0, the target is said to be met.

```
Found = 0
for k in range(1000):
    p = population(p_count, i_length, i_min, i_max)
    fitness_history = [grade(p, target), ]
    for i in range(generations):
        p = evolve(p, target)
        fitness_history.append(grade(p, target))
        if fitness_history[i] == 0:
            print("It took " + str(len(fitness_history) - 1) + " iterations")
            found += 1
            break
print(found)
```

Code snippet 3

From 1000 tests, the target was only found 38 times which gives a success rate of only 3.8%. However, for the tests that did find the solution, the number of iterations, on average, was only 5. This shows that the algorithm can find the target very quickly but is prone to being stuck in a local minima rather than finding the global minima. This is expected as each population is not 'evolving' as much as it could be.

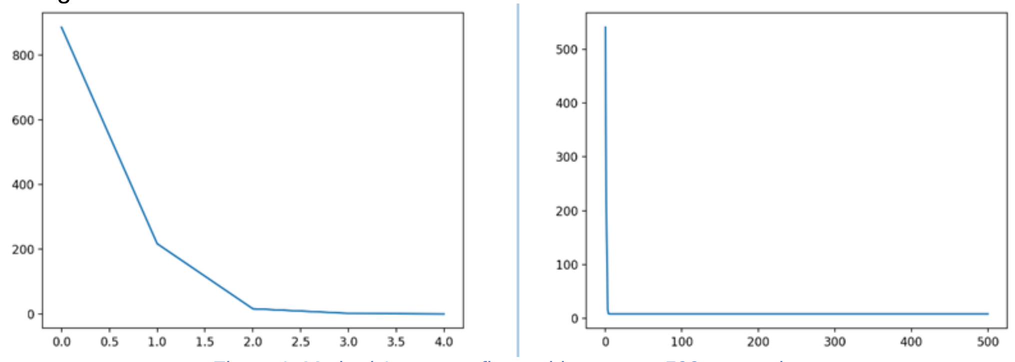

Figure 1: Method 1 average fitness history over 500 generations

Figure 1 shows two graphs, the left showcases the fitness history when a target was found after 5 iterations (generations). The right shows when the target was not found. From the graph on the left, it is clear to see there is optimisation occurring as it gradually gets to 0, which is when the target is met. The right graph shows a steady drop which indicates a local minima was found. As the graph never fully reaches 0, the global minima was not found and the optimiser was stuck in that local minima.

Method 2 – Binary representation – (Appendix B)

The second method explored was to use a binary representation for the target and individuals. Binary representation is commonly used in genetic algorithms. This would overcome the issue previously stated as the crossover and mutation can have an affect on binary numbers. The full code can be found in appendix B

For this implementation, there is a need to convert between decimal and binary representations of an integer. To account for positive and negative numbers, the two’s complement system is needed. A function was created using the libraries `‘math’` and `‘numpy’` that can be used to convert from decimal to binary.

```
Def dec2bin(x):
    # Convert decimal number to two’s complement binary
    if target != 0:
        return np.binary_repr(x, math.ceil(math.log2(abs(x))+2))
    else:
        # Cannot do log(0) so set manually to 0
        return [0]
```

Code snippet 4

Another function was created to convert from binary to decimal and is shown below. This function takes a two’s complement binary number (as a string or array) and returns the integer value.

```
def bin2dec(x):
    # Convert two's complement binary number to integer
    val = 0
    # Check if binary value is negative
    if x[0] == '1' or x[0] == 1:
        # First value is negative
        val = -(2**(len(x)-1))
    # Loop through bits and multiply by 2^n (n is index) then increment val
    for i in range(1, len(x)):
        val = val + (int(x[i]) * 2**(len(x)-1-i))
    return val
```

Code snippet 5

As before, 1000 tests were done using code snippet 6. This method proved to be far more reliable as it reached the target 870 times. A success rate of 87%. The number of iterations needed to find the target ranged from 4 to 450 which shows that this method has a better chance of finding the global minima, instead of getting stuck in a local minima.

```
found = 0
for k in range(1000):
    target = randint(-1000, 1000)
    if target != 0:
        # Get twos complement binary string of target
        target = dec2bin(target)
    else:
        # Cannot do math.log2(0) so manually set target = 0
        target = [0]
    # Length of individual is equal to number of bits of target
    i_length = len(target)
    p = population(p_count, i_length, i_min, i_max)
    fitness_history = [grade(p, target), ]
    for i in range(generations):
        p = evolve(p, target)
        fitness_history.append(grade(p, target))
        if fitness_history[i+1] == 0:
            print("The target was found in " + str(i+1) + " iterations.")
            # Increment found if target was found
            found += 1
            break
print(found)
```

Code snippet 6

Another thing to note is that the target was set randomly with each test. The range was from -1000 to 1000 as show in code snippet 6. During experimentation, it was clear that reducing the range that the target could be within, made the algorithm find the target more often. When the range was set to be between -10 and 10, the target was almost always met. However, having a bigger range gives a better indicator on how the algorithm performs in a more generalised sense.

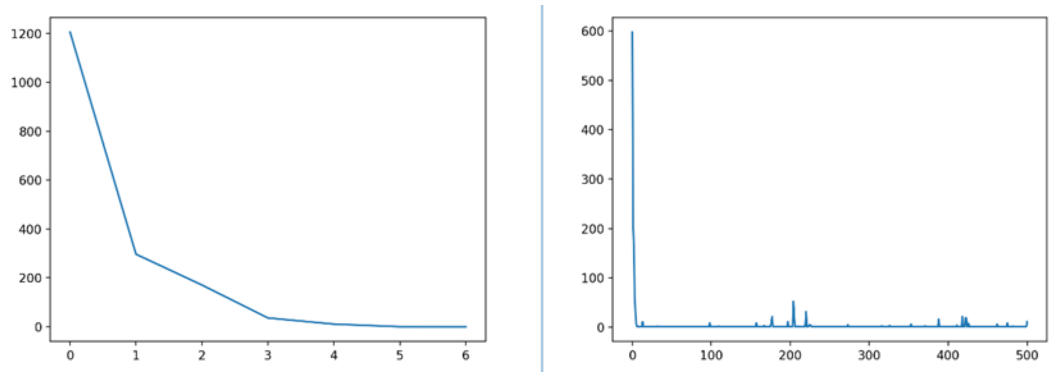

Figure 2: Method 2 average fitness history over 500 generations

Figure 2 shows the results of using a binary representation. The first is the results from finding the target after 6 iterations and the second is not finding the target. The left is very similar to figure 1 in that it clearly converges to 0. The second graph is slightly different compared to the second graph in figure 1. The second graph in figure 2 drops initially and then spikes up somewhat randomly throughout the generations. This shows that there are some attempts from the optimiser to leave the local minima.

Method 3 - Binary representation + roulette wheel selection (Appendix C)

While still using the binary representation, another genetic algorithm was designed which used the roulette wheel method for selecting parents during the evolve process. This is where each individual within the current population is given a certain amount of area on a figurative roulette wheel. The wheel is then spun, and a parent is chosen. The amount of area given is dependent on the fitness of that individual. The better the fitness, the more area that individual has, and so they have a higher chance of being a parent. This means that fitter individuals are chosen more often and so the children have better 'genes'. Other individuals with lower fitness still have a chance to be chosen and this is important allow for more genetic diversity and to minimise the chance that a good 'gene' is lost.

The new evolve function is shown here in code snippet 7. The function returns a new population that is made from the top 20% of the existing population and the rest are created by using crossover from two parents.

```
def evolve(population):
    # Get the population in ranked order
    rankedPop = rank_population(population)
    # Create the wheel
    wheel = make_roulette_wheel(rankedPop)
    children = []
    # Choose top 20% from existing population
    for i in range(int(0.2*len(population))):
        children.append(rankedPop[i][1])
    # Fill the rest of the population
    # Choose a father and mother from wheel
    # Child is first half of father and second half of mother
    while len(children) < len(population):
        father = spin_wheel(wheel)
        mother = spin_wheel(wheel)
        splitPoint = len(father) / 2
        child = father[:int(splitPoint)] + mother[int(splitPoint):]
        child = random_mutate(child)
        children.append(child)
    return children
```

Code snippet 7

The choice of keeping the top 20% of the existing population was decided during experimentation and seemed to produce the best results. The method of crossover is to take the first half of the father and the second half of the mother to produce a new child individual. There did not seem to be a difference in results when changing the `splitPoint`. There are many other methods of crossover but due to the fact that each individual could have a different length of bits, the halfway point is a simple and effective point to use. Another mutate function was also created that simply randomly changes a bit. The full code can be found in appendix C

Running the code in code snippet 8, there was a success rate of 91.2%. Similar to method 2, the number of iterations required when the target was found varied heavily. This again shows that the process of 'natural selection' and 'evolution' was taking place as the target could still be found after many generations. The graph in figure 3 also shows this as some points of the graph start to have a positive gradient (leaving a local minima), and then find the global minima.

```
popSize = 200
iMin = 0
iMax = 1
generations = 200
history = []
found = 0
for k in range(1000):
    target = randint(-1000, 1000)
    target = dec2bin(target)
    iLength = len(target)
    p = create_population(popSize, iLength, iMin, iMax)
    history = []
    history.append(grade_population(p, target))
    for g in range(generations):
        p = evolve(p)
        history.append(grade_population(p, target))
        if history[g+1] == 0:
            print("Solution found in " + str(g+1) + " generations")
            found += 1
            break
print(found)
```

Code snippet 8

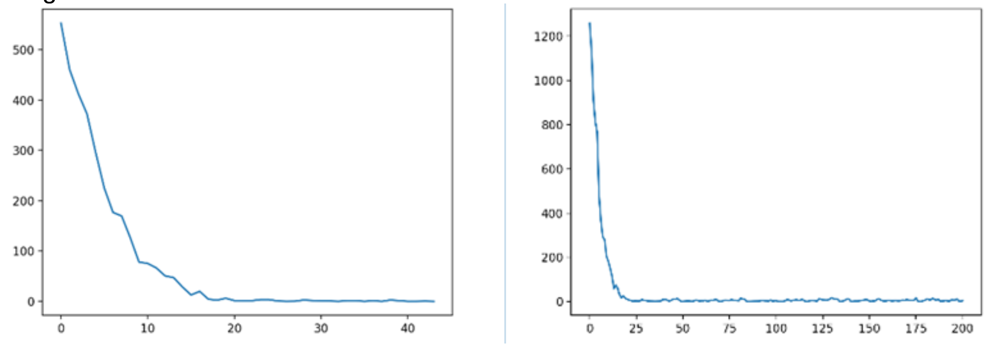

Figure 3: Method 3 average fitness history over 200 generations

The roulette wheel method seems to have a less steady convergence when compared to the graphs in figures 1 and 2. The first graph of figure 3 shows the target being found in 46 iterations. Similar to the second graph from figure 2, this method also makes some attempts at leaving the local minima which can be seen from the wobbly characteristic shown.

Exercise 2

The default value `generations` and the parameters used within the `evolve` function all have a direct impact on how quickly a solution can be found. The number of `generations` specifies the number of times the crossover and mutation process should occur. Genetic algorithms need a sufficient number of iterations (generations) for a solution to converge. A genetic algorithm is said to have converged once the current population is all identical. This is when each individual within the population is the same, as any offspring would be identical and so more iterations would not produce a new solution. This can also be seen when the average fitness of the population is 0 (for this situation a lower fitness it better).

When using the single integer (method 1 for exercise 1), changing the `mutation` and `crossover` parameters did not make a difference. This was expected as only a single integer was being used. For method 2 in exercise 1, the binary representation allowed the `mutation` and `crossover` parameters to have more of an affect. Increasing `retain` caused more of the current population to be kept in the next population. Having this too high would reduce the amount of 'evolving' the next population does. It was found that keeping this at 0.2 (20%) was the ideal value. The `mutate` value was also kept the same. This causes a random bit to change and allows for the chance of a better 'gene' to be found. However, from experimentation it was clear that having this occur too often prevented the population to converge as the good 'gene' was often replaced.

For method 3 in exercise 1, the roulette wheel method had similar parameters. In the evolve function (Appendix C), the top 20% of the current population is kept. This is slightly different to before as the previous method kept 20% randomly, whereas this method ranks the current population and keeps the top 20%. This is a key difference as it ensures that the best of the current population is used to create the next generation. The next 80% of the population is created by using the roulette selection which again, provides a higher chance for better individuals to pass on their 'genes'. There is a concern that this could lead to less genetic diversity but for this problem and situation it produced the best results.

```
def random_mutate(child):
    # Set a 10% chance a random bit is changed
    mutate = random()
    if mutate < 0.01:
        # Random bit within the child
        mutateBit = randint(0, len(child)-1)
        # Flip the bit
        if child[mutateBit] == 0:
            child[mutateBit] = 1
        else:
            child[mutateBit] = 0
    return child
```

Code snippet 9

There is also a new `random_mutate` function which behaves similarly to the mutate part of the `evolve` function in the example code. In this function, there is a 1% chance for the current individual to have a random bit flipped. This chance value was also experimented with however, it was found that having less chance for a mutation produced better results. This is not usually good design as the random mutation plays a key part in finding new solutions that may have been lost or new solutions that may not have been found. The case could be that this exercise is not complex enough for the mutation to have a big impact. For this method, the chance of mutation was kept at 1% as any significant increase reduced the performance.

Exercise 3

The number of generations needed for the genetic algorithm to find a working solution is usually unknown. This can cause redundant iterations and an increase in computational power if a solution is found before reaching the number of generations specified. To prevent this, a condition can be created to check if the current population has an individual that meets the required solution. If the case is true, then the algorithm may be stopped as no more iterations are needed. In all the methods explored in this report, the average fitness of the current population is used to determine if the current target has been met. For example, code snippet 10 has a condition

```
if fitness_history[i+1] == 0:
```

Code snippet 10

Here, `i` is the current generation (the + 1 is to account for the initial population). `fitness_history` in this solution stores the average fitness of each generation. As stated previously, the target is met once the average fitness of the current generation is 0 and so a break statement is used to exit the loop once this condition is met.

Exercise 4

The 5th-order polynomial provided below is the target.

$$y = 25x^5 + 18x^4 + 31x^3 - 14x^2 + 7x - 19$$

This can be expressed as a set of coefficients in an array as:

$$[25, 18, 31, -14, 7, -19]$$

Method 1 – Binary representation + Loop for each coefficient (Appendix D)

The first method tried is similar to exercise 1. The full code can be found in appendix D. The algorithm runs 6 times where each coefficient is the target of the current run. After each run, the coefficient is stored, and once 6 runs are done, the coefficients are compared to the target. The solution is said to be found if each coefficient is the same as the target. Code snippet 10 shows how the tests were achieved.

```
targets = [25, 18, 31, -14, 7, -19]
found = 0
for k in range(1000):
    # Make empty array to hold the solution
    solution = []
    for i in range(len(targets)):
        # Fill with 'N/A' (Used to check if coefficient is found or not)
        solution.append("N/A")
    # Loop through each coefficient in targets
    for k in range(len(targets)):
        # The current target (coefficient to find)
        target = targets[k]
        target = dec2bin(target)
        # Length of individual is equal to number of bits of target
        i_length = len(target)
        p = population(p_count, i_length, i_min, i_max)
        fitness_history = [grade(p, target), ]
        for i in range(generations):
            p = evolve(p, target)
            fitness_history.append(grade(p, target))
            # Check if solution has already been reached
            if fitness_history[i] == 0:
                # Add current coefficient to solution
                solution[k] = bin2dec(p[0])
                break
    if target == solution:
        found += 1
```

Code snippet 11

From 1000 tests, the solution was found 974 times. A success rate of 97.5%. This is much higher than the success rate found in exercise 1. A possible reason for this is that the target is first converted into its binary representation, and the number of bits for the individuals is set to be the same length. This drastically limits the number of possible values each individual can be and so it is easier for the algorithm to find the target. This solution is not the most optimal way as it is running the same algorithm 6 times (for each coefficient) and scaling this up to more coefficients or coefficients that are much larger may show that this solution is very slow. However, for the purpose of this exercise this solution performs very well.

Figure 4 shows the fitness history for each coefficient when they were found. Each of the graphs show a similar trend and were found in a similar number of iterations. This shows that this method is quite consistent and reliable. For the cases where the target was not found, the graphs would look very similar to figure 2.

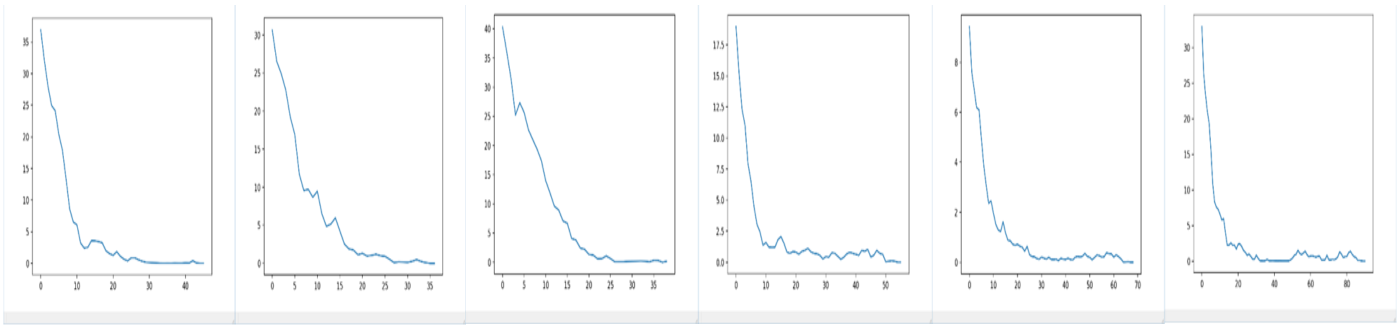
Figure 4: Average fitness for each coefficient

Method 2 – Binary representation + Loop for each coefficient + roulette wheel selection (Appendix E)

This is almost identical to the previous method covered but the new roulette wheel selection in the evolve function was used instead, similar to method 3 from exercise 1. The full code can be found in appendix E.

```
targets = [25, 18, 31, -14, 7, -19]
popSize = 200
iMin = 0
iMax = 1
generations = 200
found = 0

for k in range(1000):
    history = []
    # Create array to hold solution
    solution = []
    for i in range(len(targets)):
        solution.append('N/A')
    # Loop through each coefficient in targets
    for i in range(len(targets)):
        # Get current coefficient and set as the target
        target = targets[i]
        # Convert to binary representation
        target = dec2bin(target)
        # Individuals have the same number of bits
        iLength = len(targets)
        p = create_population(popSize, iLength, iMin, iMax)
        history.append(grade_population(p, target))
        for g in range(generations):
            p = evolve(p)
            history.append(grade_population(p, target))
            # Check if solution has already been reached
            if history[g+1] == 0:
                # Add individual to the solution
                solution[i] = bin2dec(p[0])
                break
    if targets == solution:
        found += 1

print(found)
print(solution)
```

Code snippet 12

Running the code above, this method was tested 1000 times and scored a success rate of 100%. Every coefficient was found on every run. This is remarkably higher than all other methods. Method 3 from exercise 1 did not score 100% but this is most likely due to the fact that the target range was far greater compared to the coefficients required in this exercise. Each coefficient is far more likely to be found as during the decimal to binary conversion, the number of bits of the target (coefficient) is used to characterise the new individual, which restricts the range of the individual.

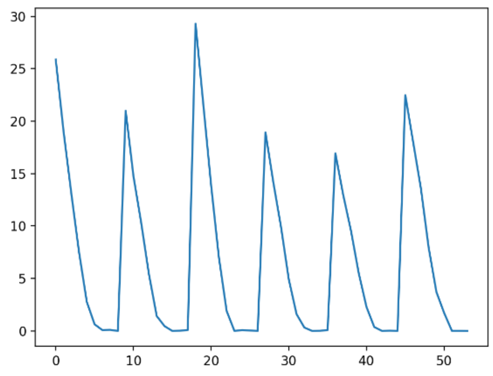

Figure 5 Average fitness for each coefficient

Due to the way this method was coded, the average fitness of each coefficient can be plotted on the same graph. Each spike is when the optimiser finds the current target (coefficient) and moves on to the next. Like before in figure 5, each coefficient is found within roughly 10 iterations. Another note is that the tip of the spikes indicates the start of the current coefficient, and their heights reflect how far away the initial population is from the target. As expected, the largest coefficients have a bigger spike as the population created has a wider range of possible values, thus decreasing the likelihood that the individuals are initially closer to the target. This also reinforces the conclusions stated in exercise 1's method 2. That increasing the possible range of each individual makes it harder for the optimiser to find the target.

Exercise 5 – Holland's Theorem

Throughout the design of each of these methods, Holland's theorem was kept in mind. The use of a binary representation was one of the key factors when designing each method. In the roulette wheel selection process designed, there was a choice to keep the top 20% of the current population. By ranking each of the individuals, the population and finding the average fitness of the population, each individual has a specific chance of being selected by the roulette wheel. This is stated in Holland's Theorem that the schema with a better than average fitness will have a higher chance of being selected. The code from appendix C and E both contain the roulette wheel implementation, where this theorem can be seen in practice. The net effect of this process is that the best individuals (highest ranked in terms of fitness) are more likely to be in the next population. This causes the individuals within the population to get closer to the desired target with each generation.

The use of a specific 'schema' was not used in this implementation as the binary bits were not representing some other value, they were being used as normal binary bits. For this reason, the method of selection is less complex to Holland's. As Holland suggests checking each individual for a certain patten of bits, whereas this implementation converts the bits into decimal when computing the fitness.

The mutation idea was implemented but again, in a simpler form. Rather than having a specific schema to search for, a random bit is just flipped. There are concerns when using these types of mutations and crossover techniques, they can cause new solutions to be found but can also destroy good 'genes' and cause the solution to not be found. This was a core reason as to why 20% of the current population was carried over to the next, to ensure that the current best 'genes' were not lost during mutation or crossover. And that the chance of mutation was set to 1%, to avoid mutating too much and destroying a good 'gene'. As stated in exercise 4, having a high change of mutation produced poor results as the mutation has a very big impact on these problems where each bit corresponds to a value.

The technique of crossover also discussed in exercise 1 method 3 is also very simple, in that the child becomes the first half of the father and second half of the mother. There are many different methods this crossover can be implemented as. This simple choice as used as the length of the current individual (number of bits) is unknown and so having a more complex scheme may not work for individuals of varying length. Having a half of each parent can work for an individual of any length as it is calculated relative to the current individual. Some other ratios were explored such as having the first 25% of the father and last 75% of the mother but they did not have much of an affect on the results, and so the half point was used for simplicity.

With Holland's theorem, there are more techniques that can be explored but due to the simplicity of the problems within this report, they would not have drastically produced different results.

Appendix

Here every method's code can be found. There are many repetitions due to each method building from the previous but for completeness and for ease of reading, the entire code is given each time. The code snippets used in the reports have some alterations to explain a specific purpose at that point in the report and so are not included in the appendix.

Appendix A – Ex1 – Method 1

```python
from functools import reduce
from random import randint, random
from operator import add
import matplotlib.pyplot as plt


def individual(length, min, max):
    # Create a member of the population #
    return [randint(min, max) for x in range(length)]


def population(count, length, min, max):
    # Create a number of individuals (i.e. a population)

    # count:    The number of individuals in a population
    # length:   The number of values per individual
    # min:      The minimum possible value in an individual's list of values
    # max:      The maximum possible value in an individual's list of values
    return [individual(length, min, max) for x in range(count)]


def fitness(individual, target):
    # Determine the fitness of an individual. Lower is better #

    # individual:   The individual to evaluate
    # target:       The target number individuals are aiming for
    return abs(target - individual[0])


def grade(pop, target):
    # Find average fitness for a population #
    summed = reduce(add, (fitness(x, target) for x in pop))
    return summed / (len(pop) * 1.0)


def evolve(pop, target, retain=0.2, random_select=0.05, mutate=0.01):
    graded = [(fitness(x, target), x) for x in pop]
    graded = [x[1] for x in sorted(graded)]
    retain_length = int(len(graded) * retain)
    parents = graded[:retain_length]

    # randomly add other individuals to promote genetic diversity
    for individual in graded[retain_length:]:
        if random_select > random():
            parents.append(individual)

    # mutate some individuals
    for individual in parents:
        if mutate > random():
            pos_to_mutate = randint(0, len(individual) - 1)
            # this mutation is not ideal, because it
            # restricts the range of possible values,
            # but the function is unaware of the min/max
            # values used to create the individuals,
            individual[pos_to_mutate] = randint(
                min(individual), max(individual))

    # crossover parents to create children
    parents_length = len(parents)
    desired_length = len(pop) - parents_length
    children = []
    while len(children) < desired_length:
        male = randint(0, parents_length - 1)
        female = randint(0, parents_length - 1)
        if male != female:
            male = parents[male]
            female = parents[female]
            half = len(male) / 2
            child = male[:int(half)] + female[int(half):]
            children.append(child)
    parents.extend(children)
    return parents


# Example usage
p_count = 100
i_length = 1
i_min = -100
i_max = 100
target = randint(i_min, i_max)
generations = 500

p = population(p_count, i_length, i_min, i_max)
fitness_history = [grade(p, target), ]
for i in range(generations):
    p = evolve(p, target)
    fitness_history.append(grade(p, target))
    if fitness_history[i+1] == 0:
        print("It took " + str(len(fitness_history) - 1) + " iterations")
        break

for datum in fitness_history:
    print(datum)

plt.plot(fitness_history)
plt.show()
```

Appendix B – Ex1 – Method 2

```python
from functools import reduce
from random import randint, random
from operator import add
import numpy as np
import math
import matplotlib.pyplot as plt


def individual(length, min, max):
    # Randomly choose a number between min and max
    # Loop length number of times
    return [randint(min, max) for x in range(length)]


def population(count, length, min, max):
    # Create the population

    # count:    The number of individuals in a population
    # length:   The number of values per individual
    # min:      The minimum possible value in an individual's list of values
    # max:      The maximum possible value in an individual's list of values
    return [individual(length, min, max) for x in range(count)]


def fitness(individual, target):
    # Determine the fitness of an individual. Lower is better

    # individual:   The individual to evaluate
    # target:       The target number individuals are aiming for

    # Convert target and individual to decimal and find the difference
    return abs(bin2dec(target)-bin2dec(individual))


def grade(pop, target):
    # Find average fitness for a population #

    summed = reduce(add, (fitness(x, target) for x in pop))
    return summed / (len(pop) * 1.0)


def evolve(pop, target, retain=0.2, random_select=0.05, mutate=0.01):
    graded = [(fitness(x, target), x) for x in pop]
    graded = [x[1] for x in sorted(graded)]
    retain_length = int(len(graded) * retain)
    parents = graded[:retain_length]

    # randomly add other individuals to promote genetic diversity
    for individual in graded[retain_length:]:
        if random_select > random():
            parents.append(individual)

    # mutate some individuals
    for individual in parents:
        if mutate > random():
            pos_to_mutate = randint(0, len(individual) - 1)
            # this mutation is not ideal, because it
            # restricts the range of possible values,
            # but the function is unaware of the min/max
            # values used to create the individuals,
            individual[pos_to_mutate] = randint(
                min(individual), max(individual))

    # crossover parents to create children
    parents_length = len(parents)
    desired_length = len(pop) - parents_length
    children = []
    while len(children) < desired_length:
        male = randint(0, parents_length - 1)
        female = randint(0, parents_length - 1)
        if male != female:
            male = parents[male]
            female = parents[female]
            half = len(male) / 2
            child = male[:int(half)] + female[int(half):]
            children.append(child)

    parents.extend(children)
    return parents


def bin2dec(x):
    # Convert two's complement binary number to integer
    val = 0
    # Check if binary value is negative
    if x[0] == '1' or x[0] == 1:
        # First value is negative
        val = -(2**(len(x)-1))
    # Loop through bits and multiply by 2^n (n is index) then increment val
    for i in range(1, len(x)):
        val = val + (int(x[i]) * 2**(len(x)-1-i))
    return val


def dec2bin(x):
    # Convert decimal number to two's complement binary
    if target != 0:
        return np.binary_repr(x, math.ceil(math.log2(abs(x))+2))
    else:
        return [0]


p_count = 100
generations = 500
i_min = 0   # Binary min value is 0
i_max = 1   # Binary max value is 1

target = randint(-1000, 1000)
target = dec2bin(target)
# Length of individual is equal to number of bits of target
i_length = len(target)
p = population(p_count, i_length, i_min, i_max)
fitness_history = [grade(p, target), ]
for i in range(generations):
    p = evolve(p, target)
    fitness_history.append(grade(p, target))
    if fitness_history[i+1] == 0:
        print("The target was found in " + str(i+1) + " iterations.")
        break

for datum in fitness_history:
    print(datum)

plt.plot(fitness_history)
plt.show()
```

Appendix C – Ex1 – Method 3

```python
import numpy as np
import math
import matplotlib.pyplot as plt
from random import randint, random


def create_individual(length, min, max):
    return [randint(min, max) for i in range(length)]


def create_population(popSize, length, min, max):
    return [create_individual(length, min, max) for i in range(popSize)]


def fitness(target, individual):
    # Get fitness of current individual
    a = bin2dec(target)
    b = bin2dec(individual)
    diff = abs(a-b)
    # Lower is better
    return diff


def make_roulette_wheel(rankedPop):
    # Get the total sum of fitness for the population
    sumFitness = sum([x[0] for x in rankedPop])
    wheel = []

    if sumFitness == 0:
        # Every individual is a correct solution, return wheel of one individual
        wheel.append(rankedPop[0][1])
        return wheel

    # Calculate the percentage of the wheel each individual should take up
    allFitness = [math.ceil((rankedPop[i][0] / sumFitness) * 100) for i in range(len(rankedPop))]
    # Reverse as we want to minimise the fitness function
    allFitness = sorted(allFitness, reverse=True)

    wheel = []
    # Create wheel - Array of 100 elements, each individual repeated depending on fitness
    for i in range(len(allFitness)):
        # Add current individual and repeat according to percentage calculated
        for j in range(allFitness[i]):
            wheel.append(rankedPop[i][1])
    return wheel


def spin_wheel(wheel):
    # Random number (spin the wheel)
    x = randint(0, len(wheel) - 1)
    # Parent is the random index within the wheel
    parent = wheel[x]
    return parent


def random_mutate(child):
    # Set a 10% chance a random bit is changed
    mutate = random()
    if mutate < 0.01:
        # Random bit within the child
        mutateBit = randint(1, len(child)-1)
        # Flip the bit
        if child[mutateBit] == 0:
            child[mutateBit] = 1
        else:
            child[mutateBit] = 0
    return child


def rank_population(population):
    # Rank the population in order of fitness (lower fitness is better)
    fitnessPop = []
    for i in range(len(population)):
        fitnessPop.append((fitness(target, population[i]), population[i]))
    return sorted(fitnessPop
def evolve(population):
    # Get the population in ranked order
    rankedPop = rank_population(population)
    # Create the wheel
    wheel = make_roulette_wheel(rankedPop)

    children = []

    # Choose 20% from existing population
    for i in range(int(0.2*len(population))):
        children.append(spin_wheel(wheel))

    # Fill the rest of the population
    # Choose a father and mother from wheel
    # Child is first half of father and second half of mother
    while len(children) < len(population):
        father = spin_wheel(wheel)
        mother = spin_wheel(wheel)
        splitPoint = len(father) / 2
        child = father[:int(splitPoint)] + mother[int(splitPoint):]
        child = random_mutate(child)
        children.append(child)

    return children


def grade_population(population, target):
    # Get average fitness of current population
    grade = 0
    for i in range(len(population)):
        grade = grade + fitness(target, population[i])
    return grade / len(population)


def bin2dec(x):
    # Convert two's complement binary number to integer
    val = 0
    # Check if binary value is negative
    if x[0] == '1' or x[0] == 1:
        # First value is negative
        val = -(2**(len(x)-1))
    # Loop through bits and multiply by 2^n (n is index) then increment val
    for i in range(1, len(x)):
        val = val + (int(x[i]) * 2**(len(x)-1-i))
    return val


def dec2bin(x):
    # Convert decimal number to two's complement binary
    if target != 0:
        return np.binary_repr(x, math.ceil(math.log2(abs(x))+2))
    else:
        # Cannot do log(0) so set manually to 0
        return [0]

popSize = 200
iMin = 0
iMax = 1
generations = 200
found = 0

target = randint(-1000, 1000)
print("The target is: " + str(target))
target = dec2bin(target)
iLength = len(target)
history = []

p = create_population(popSize, iLength, iMin, iMax)
history.append(grade_population(p, target))
for g in range(generations):
    p = evolve(p)
    history.append(grade_population(p, target))
    # Check if solution has already been reached
    if history[g+1] == 0:
        print("Solution found in " + str(g+1) + " generations")
        break

for datum in history:
    print(datum)
```

Appendix D – Ex4 – Method 1

```
from functools import reduce
from random import randint, random
from operator import add
import numpy as np
import math
import matplotlib.pyplot as plt


def individual(length, min, max):
    # Randomly choose a number between min and max
    # Loop length number of times
    return [randint(min, max) for x in range(length)]


def population(count, length, min, max):
    # Create the population

    # count:    The number of individuals in a population
    # length:   The number of values per individual
    # min:      The minimum possible value in an individual's list of values
    # max:      The maximum possible value in an individual's list of values
    return [individual(length, min, max) for x in range(count)]


def fitness(individual, target):
    # Determine the fitness of an individual. Lower is better

    # individual:   The individual to evaluate
    # target:       The target number individuals are aiming for
    return abs(bin2dec(target)-bin2dec(individual))


def grade(pop, target):
    # Find average fitness for a population #
    summed = reduce(add, (fitness(x, target) for x in pop))
    return summed / (len(pop) * 1.0)


def evolve(pop, target, retain=0.2, random_select=0.5, mutate=0.01):
    graded = [(fitness(x, target), x) for x in pop]
    graded = [x[1] for x in sorted(graded)]
    retain_length = int(len(graded) * retain)
    parents = graded[:retain_length]

    # randomly add other individuals to promote genetic diversity
    for individual in graded[retain_length:]:
        if random_select > random():
            parents.append(individual)

    # mutate some individuals
    for individual in parents:
        if mutate > random():
            pos_to_mutate = randint(0, len(individual) - 1)
            # this mutation is not ideal, because it
            # restricts the range of possible values,
            # but the function is unaware of the min/max
            # values used to create the individuals,
            individual[pos_to_mutate] = randint(
                min(individual), max(individual))

    # crossover parents to create children
    parents_length = len(parents)
    desired_length = len(pop) - parents_length
    children = []
    while len(children) < desired_length:
        male = randint(0, parents_length - 1)
        female = randint(0, parents_length - 1)
        if male != female:
            male = parents[male]
            female = parents[female]
            half = len(male) / 2
            child = male[:int(half)] + female[int(half):]
            children.append(child)

    parents.extend(children)
    return parents
```

```
def bin2dec(x):
    # Convert two's complement binary number to integer
    val = 0
    # Check if binary value is negative
    if x[0] == '1' or x[0] == 1:
        # First value is negative
        val = -(2**(len(x)-1))
    # Loop through bits and multiply by 2^n (n is index) then increment val
    for i in range(1, len(x)):
        val = val + (int(x[i]) * 2**(len(x)-1-i))
    return val


def dec2bin(x):
    # Convert decimal number to two's complement binary
    if target != 0:
        return np.binary_repr(x, math.ceil(math.log2(abs(x))+2))
    else:
        # Cannot do log(0) so set manually to 0
        return [0]

# Example usage
targets = [25, 18, 31, -14, 7, -19]
p_count = 200
i_min = 0
i_max = 1
generations = 200
found = 0


# Make empty array to hold the solution
solution = []
for i in range(len(targets)):
    # Fill with 'N/A' (Used to check if coefficient is found or not)
    solution.append("N/A")
# Loop through each coefficient in targets
for k in range(len(targets)):
    # The current target (coefficient to find)
    target = targets[k]
    target = dec2bin(target)
    # Length of individual is equal to number of bits of target
    i_length = len(target)
    p = population(p_count, i_length, i_min, i_max)
    fitness_history = [grade(p, target), ]
    for i in range(generations):
        p = evolve(p, target)
        fitness_history.append(grade(p, target))
        # Check if solution has already been reached
        if fitness_history[i] == 0:
            # Add current coefficient to solution
            solution[k] = bin2dec(p[0])
            break

print("The coefficients are: ")
print(str(solution))

plt.plot(fitness_history)
plt.show()
```

Appendix E – Ex4 – Method 2

```
import numpy as np
import math
import matplotlib.pyplot as plt
from random import randint, random


def create_individual(length, min, max):
    return [randint(min, max) for i in range(length)]


def create_population(popSize, length, min, max):
    return [create_individual(length, min, max) for i in range(popSize)]


def fitness(target, individual):
    # Get fitness of current individual
    a = bin2dec(target)
    b = bin2dec(individual)
    diff = abs(a-b)
    # Lower is better
    return diff


def make_roulette_wheel(rankedPop):
    # Get the total sum of fitness for the population
    sumFitness = sum([x[0] for x in rankedPop])
    wheel = []

    if sumFitness == 0:
        # Every individual is a correct solution, return wheel of one individual
        wheel.append(rankedPop[0][1])
        return wheel

    # Calculate the percentage of the wheel each individual should take up
    allFitness = [math.ceil((rankedPop[i][0] / sumFitness) * 100) for i in range(len(rankedPop))]
    # Reverse as we want to minimise the fitness function
    allFitness = sorted(allFitness, reverse=True)

    wheel = []
    # Create wheel - Array of 100 elements, each individual repeated depending on fitness
    for i in range(len(allFitness)):
        # Add current individual and repeat according to percentage calculated
        for j in range(allFitness[i]):
            wheel.append(rankedPop[i][1])
    return wheel


def spin_wheel(wheel):
    # Random number (spin the wheel)
    x = randint(0, len(wheel) - 1)
    # Parent is the random index within the wheel
    parent = wheel[x]
    return parent


def random_mutate(child):
    # Set a 10% chance a random bit is changed
    mutate = random()
    if mutate < 0.01:
        # Random bit within the child
        mutateBit = randint(0, len(child)-1)
        # Flip the bit
        if child[mutateBit] == 0:
            child[mutateBit] = 1
        else:
            child[mutateBit] = 0
    return child


def rank_population(population):
    # Rank the population in order of fitness (lower fitness is better)
    fitnessPop = []
    for i in range(len(population)):
        fitnessPop.append((fitness(target, population[i]), population[i]))
    return sorted(fitnessPop)


def evolve(population):
    # Get the population in ranked order
    rankedPop = rank_population(population)
    # Create the wheel
    wheel = make_roulette_wheel(rankedPop)

    children = []

    # Choose top 20% from existing population
    for i in range(int(0.2*len(population))):
        children.append(rankedPop[i][1])

    # Fill the rest of the population
    # Choose a father and mother from wheel
    # Child is first half of father and second half of mother
    while len(children) < len(population):
        father = spin_wheel(wheel)
        mother = spin_wheel(wheel)
        splitPoint = len(father) / 2
        child = father[:int(splitPoint)] + mother[int(splitPoint):]
        child = random_mutate(child)
        children.append(child)

    return children


def grade_population(population, target):
    # Get average fitness of current population
    grade = 0
    for i in range(len(population)):
        grade = grade + fitness(target, population[i])
    return grade / len(population)


def bin2dec(x):
    # Convert two's complement binary number to integer
    val = 0
    # Check if binary value is negative
    if x[0] == '1' or x[0] == 1:
        # First value is negative
        val = -(2**(len(x)-1))
    # Loop through bits and multiply by 2^n (n is index) then increment val
    for i in range(1, len(x)):
        val = val + (int(x[i]) * 2**(len(x)-1-i))
    return val


def dec2bin(x):
    # Convert decimal number to two's complement binary
    if target != 0:
        return np.binary_repr(x, math.ceil(math.log2(abs(x))+2))
    else:
        # Cannot do log(0) so set manually to 0
        return [0]


targets = [25, 18, 31, -14, 7, -19]
popSize = 200
iMin = 0
iMax = 1
generations = 200
found = 0
history = []
# Create array to hold solution
solution = []
for i in range(len(targets)):
    solution.append('N/A')
# Loop through each coefficient in targets
for i in range(len(targets)):
    # Get current coefficient and set as the target
    target = targets[i]
    # Convert to binary representation
    target = dec2bin(target)
    # Individuals have the same number of bits
    iLength = len(targets)
    p = create_population(popSize, iLength, iMin, iMax)
    history.append(grade_population(p, target))
    for g in range(generations):
        p = evolve(p)
        history.append(grade_population(p, target))
        # Check if solution has already been reached
        if history[g+1] == 0:
            # Add individual to the solution
            solution[i] = bin2dec(p[0])
            break
print(solution)
```