Electronics and Computer Science
Faculty of Engineering and Physical Sciences
University of Southampton

Peter Alexander

19/07/2021

COMP6202: Evolution of Complexity

Assignment 2

Module Leader: Richard Watson

Moderator: Danesh Tarapore

A Report submitted for the Award of MEng
Electronic Engineering

Abstract

This report details the reimplementation of a paper discussing the merits of Cooperative Coevolutionary Genetic Algorithms (CCGAs)[1]. The example algorithm given in the paper and its implementation in Python are discussed. An extension to the paper is proposed to rule out the granular crossover scheme enabled by the CCGA as a source of its increased performance over the standard algorithm. The extension is then implemented and the findings are examined.

Contents

1 Introduction

In this assignment, the paper titled: *A Cooperative Coevolutionary Approach to Function Optimisation*[1] was chosen for reimplementation. The paper presents a framework for a Cooperative Coevolutionary Genetic Algorithm (CCGA) and compares its performance with a standard Genetic Algorithm (GA) as a function optimiser.

The functions to be optimised were chosen for being *"highly multi-modal"*. These functions are referred to as the Rastrigin, Schwefel, Griewangk, and Ackley functions[2]–[4]. Each of these functions has a global minimum of zero provided that the set of parameters fed into it are bounded by set values. The optimisation task presented in the paper is to evolve the set of function parameters that give an output of zero. The performance of each algorithm is tested on each function individually.

In the standard GA, the individuals being evolved are bit-strings representing every parameter. The fitness of each individual is calculated by by splitting the bit-string into the parameters and running them through the function. In the CCGA, each parameter is represented by its own population of bit-strings. Each population is evaluated in a round-robin fashion. The fitness of each individual is calculated by combining it with the best individuals from each of the other populations and running this set through the function. An initial fitness value is assigned before the first generation by evaluating each individual with a random individual from each population. In both algorithms, the closer the function output is to zero, the fitter the individual.

A scaling window is used to translate this smaller-is-better fitness regime into a fitness proportionate selection scheme. When calculating the selection probability of an individual its fitness is subtracted from the fitness of the worst individual from the past 5 generations. This value is used when calculating the size of an individual's 'slice' on the fitness-proportionate roulette wheel. This also allows small variations in the population to standout relative to the rest of the population.

Table 1, reproduced from the original paper, contains the algorithm characteristics used in the experiments. Pseudo code for the algorithms implemented can be found in [1].

Characteristic	**Value**
Representation	Binary (16 bits per function parameter)
Selection	Fitness Proportionate
Fitness Scaling	Scaling Window Technique (width 5)
Elitist Strategy	Single copy of best individual preserved
Genetic Operators	Two-point crossover and bit-flip mutation
Mutation Probability	1/chromlength
Crossover Probability	0.6
Population Size	100
Simulation Length	100000 function evaluations

Table 1: *A table showing the characteristics of the algorithms used to perform the experiments. Reproduced from [1]*

2 Reimplementation

The reimplementation was written in Python v3.8 for flexibility and speed of development. The results generated by the Python script were saved to disk and plotted using MATLAB. This meant plots could be fine tuned without having to rerun the experiments.

Individuals are defined using an `Individual` class in `individual.py` (Appendix A.1). This class defines the genome as a BitArray object (defined in the third-party bistring library[5]), and that individual's fitness for convenience. The fitness functions are defined in `functions.py` (Appendix A.2). Each function is defined with identical interfaces and an accompanying Python dict containing details about the function such as the limits of the parameter values. Also included are functions to convert a set of BitArrays to function parameters. The original paper did not specify the conversion process used so the range of $[0, 2^{16} - 1]$ that the BitArray can represent is mapped to the lower and upper limits of the functions (e.g. $[-5.12, 5.12]$ for the Rastrigin function).

The algorithms are implemented in the classes `GAExperiment` and `CCGAExperiment` (see Appendices A.3, A.4). These classes implement generic versions of their algorithm and must be provided with the fitness function, the corresponding Python dict, and the number of parameters under test upon instantiation. The experiment can then be run using the `run_experiment()` method. The `CCGAExperiment` class shares the same overall structure as the `GAExperiment` class but has been upgraded with the infrastructure needed to store, evolve, and evaluate multiple populations.

The experiments are performed in the file `main_data_gather.py` (see Appendix A.5). This acts as the project's main file. The user specifies which algorithms should be tested and the number of runs the results should be averaged over with command line arguments. The results are saved to txt files for plotting. The computing resources available to this project were limited due to working from home so results of each experiment were averaged over 15 runs rather than the 50 in the original paper.

The MATLAB script `combined_plots.m` (see Appendix A.6) was used to produce the reimplemented figure.

3 Reimplementation Results

Figure 1 shows the results from the original paper and Figure 2 shows the reimplementation results. It can be seen that both plots show the same trajectory for the fitness over the series of function evaluations for each algorithm and fitness function combination. The reimplementation plots are slightly rougher due to the lower number of experiments used to calculate the average performance.

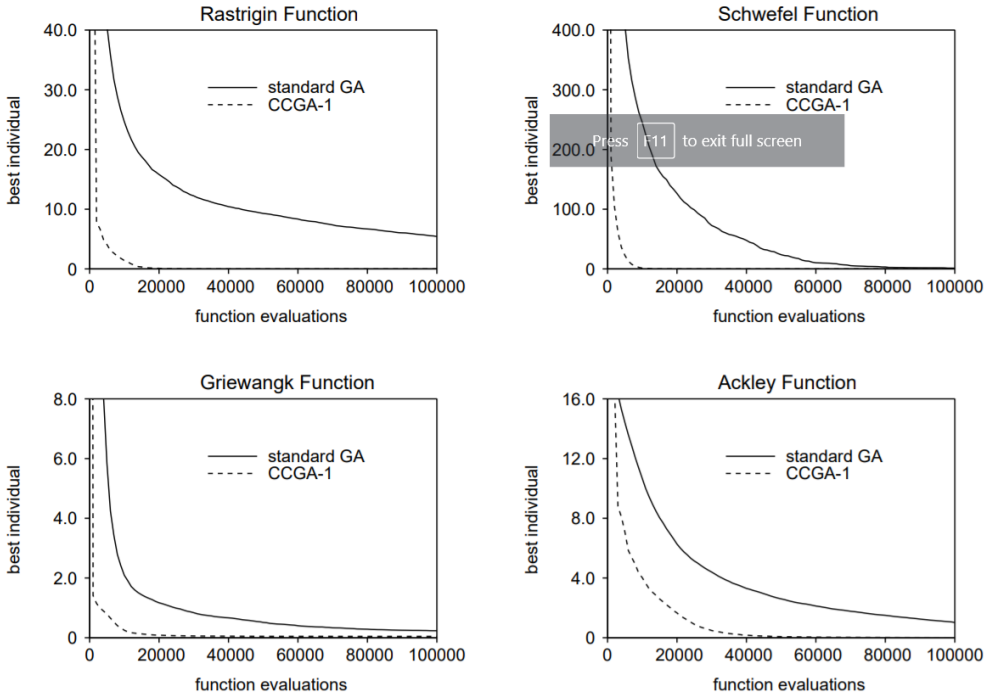

Figure 1: *The figure this assignment aims to reimplement. Reproduced from[1].*

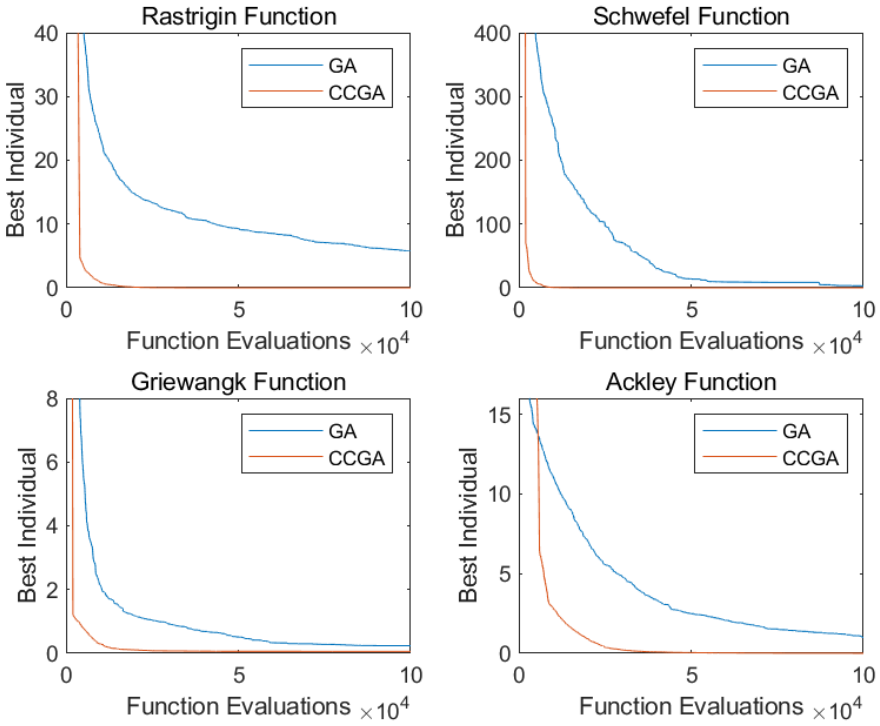

Figure 2: *The figure produced by this reimplementation.*

4 Extension

When reading the original paper and surrounding literature it was noted that while of the characteristics given in Table 1 impact each algorithm equally, there is one area where it

seems the CCGA has an unfair advantage. By performing a two point crossover on each individual in the CCGA it is effectively performing an N point crossover on the problem as a whole. Whilst the number of crossovers does not change between the two algorithms as the number of fitness evaluations are fixed, it still enables evolution to occur at a more granular level in the CCGA. It is thought that these properties have a small but measurable impact on the CCGA's peprformance, but the majority comes from how fitness data is measured and stored.

The hypothesis that is presented in this extension is as follows:

The performance increase of the CCGA over the standard GA comes from the ability to effectively measure the fitness of individual genes rather than entire genomes, not the more granular crossover scheme a CCGA enables.

Where *gene* refers to a 16 bit chunk that encodes a single function parameter. To test this, two new crossover functions were added to the GA:

1. **N Point/Chunk Crossover** - In this scheme, two parents are selected as is the case for two point crossover. However, rather than combining two contiguous halves from each parent to produce the offspring, each 16-bit chunk is taken from a randomly selected parent. This can be thought of like N chunk uniform crossover. This accounts for the situation where both parents have good and bad genes distributed throughout rather than just in one half.

2. **N Individual Crossover** - This is an extension of the scheme above. Here each individual has a number of parents equal to the number of function parameters. Like before, each 16-bit chunk is taken from a randomly selected parent. This means that each function evaluation is able to sample a wider range of the population. This crossover method is combined with standard two point crossover to allow genes to be split occasionally.

Both these schemes aim to equalise the playing field between the GA and the CCGA where crossover is concerned. They constitute reasonable drop-in improvements to a standard GA that aim to mimic the granularity of a CCGA without adding the infrastructure needed for multiple populations. The two schemes shall be referred to as the EXGA_1 and the EXGA_2.

A plot showing the results of this extension was produced by the MATLAB script `extension_plot.m` (see Appendix A.7).

5 Extension Results

Figure 3 shows the results produced by the EXGA_1 and EXGA_2 algorithms plotted on top of the reimplementation results.

It can be seen that the EXGA_1 and EXGA_2 algorithms do indeed perform slightly better on the Rastrigin and Schwefel functions, with EXGA_2 performing slightly better

than EXGA_1 in both cases. However, they perform worse that the standard GA on the Griewangk and Ackley Functions.

It is thought this difference in performance comes from the interdependency between parameters in the Griewangk and Ackley functions. These functions have a higher epistasis than the Rastrigin or Schwefel functions and using two point crossover preserves more of the genetic background along with any good genes developed.

The CCGA maintains a relative performance increase over all other tested algorithms as each gene is evaluated in the context of the other best performing genes. The performance of each new gene is measured in this context so sudden shifts in the genetic background that would spoil previously fit genes are less likely than in EXGA_1 and EXGA_2.

It is felt that the hypothesis presented in Section 4 has been partially confirmed. The results of EXGA_1 and EXGA_2 show that the crossover scheme enabled by the CCGA is not the main driving factor behind its performance and that, instead, it is the ability to effectively assign fitness values to individual genes within the context of the prevailing genetic background. However, it has also been shown that in problems with high epistasis these properties may even work against the GA by removing high-fitness genes from the context in which their fitness was achieved.

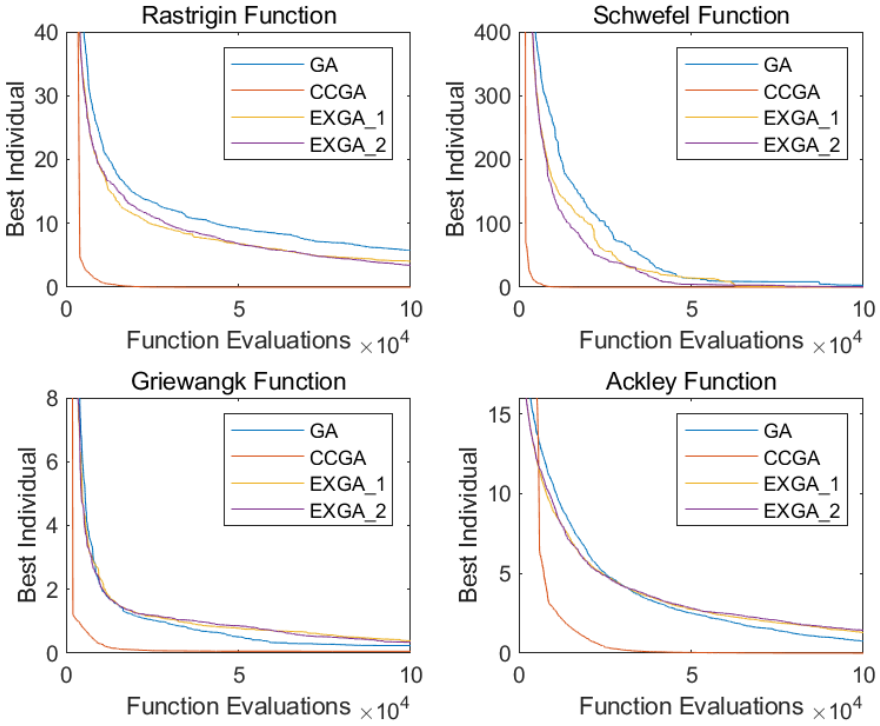

Figure 3: *The figure produced to demonstrate the extension produced for this assignment.*

6 Conclusion

In this assignment the paper titled "A cooperative coevolutionary approach to function optimization"[1] was reimplemented successfully in Python. An extension was added to

try to determine the source of the performance increase in the Cooperative Coevolutionary Genetic Algorithm (CCGA). This extension attempted to imitate the crossover scheme enabled by the CCGA in a standard Genetic Algorithm (GA) to rule that out as a significant performance booster. This attempt was partially successful, in that the extended algorithms proved marginally better in some problems and marginally worse in others.

References

[1] M. A. Potter and K. A. De Jong, "A cooperative coevolutionary approach to function optimization," in *Parallel Problem Solving from Nature — PPSN III*, Y. Davidor, H.-P. Schwefel, and R. Männer, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 249–257, ISBN: 978-3-540-49001-2.

[2] H. Mühlenbein, M. Schomisch, and J. Born, "The parallel genetic algorithm as function optimizer," *Parallel Computing*, vol. 17, no. 6, pp. 619–632, 1991, ISSN: 0167-8191. DOI: https://doi.org/10.1016/S0167-8191(05)80052-3. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819105800523.

[3] V. Gordon and D. Whitley, "Serial and parallel genetic algorithms as function optimizers," *Serial and Parallel Genetic Algorithms As Function Optimizers*, Jul. 1998.

[4] T. Bäck and H.-P. Schwefel, "An overview of evolutionary algorithms for parameter optimization," *Evolutionary Computation*, vol. 1, no. 1, pp. 1–23, 1993. DOI: 10.1162/evco.1993.1.1.1.

[5] S. Griffiths, *Bitstring python module - release bitstring-3.1.7*, GitHub Repo, 2020. [Online]. Available: https://github.com/scott-griffiths/bitstring.

A Source Code Listing

Listings

A.1 individual.py

```
"""
Individual
==========

Simple class holding just a BitArray and the corresponding fitness for that
array. If Python had structs I'd use one of those.
"""

from bitstring import BitArray

class Individual:
    def __init__(self, bit_arr: BitArray):

        self._bit_arr = bit_arr
        """The BitArray defining this individuals chromasome"""

        self._fitness = None
        """The fitness of this individual. Computed externally"""

    @property
    def bit_arr(self):
        return self._bit_arr

    @bit_arr.setter
    def bit_arr(self, bit_arr):
        self._bit_arr = bit_arr

    @property
    def fitness(self):
        return self._fitness

    @fitness.setter
    def fitness(self, fitness):
        self._fitness = fitness
```

Listing 1: *individual.py*

A.2 functions.py

```python
"""
Functions
=========

This file implements the four functions that are to be optimised and contains
dictionaries listing parameters for the default implementations of each
function.
"""
from math import exp, sin, cos, sqrt, pi
from numpy.testing import assert_almost_equal
from scipy.interpolate import interp1d
from bitstring import BitArray


def range_map(value: float, interpolator: interp1d) -> float:
    """Maps a value from one range to another using a scipy interpolator

    :param value: The value to map
    :param interpolator: The interpolator used to perform the map
    :returns: The mapped value
    """

    return interpolator(value)


def bin_str_to_int(bin_str: BitArray) -> int:
    """Converts bin_str to an integer between 0 and (2^len(bin_str)) - 1.

    :param bin_str: Binary number represented in a string.
    """

    return bin_str.uint


rast_dict = {
    "n": 20,
    "min": -5.12,
    "max": 5.12,
    "interp": interp1d([0, (2 ** 16) - 1], [-5.12, 5.12]),
}


def rastrigin(chrom_list: list, n=rast_dict["n"], interp=rast_dict["interp"]) -> float:
    """Run the Rastrigin function for the chromasome list given.

    :param chrom_list: A list of BitArrays representing the chromosomes
        for each parameter.
    :param n: The number of parameters
    :returns: The output of the function
    """

    # Convert all chromosomes to decimal values
    dec_chroms = [bin_str_to_int(bin_str) for bin_str in chrom_list]

    # Map those numbers to the function range to get the parameters
    func_params = [range_map(chrom, interp) for chrom in dec_chroms]

    # Run the function
    # Non-sum term
    func_out = 3 * n

    # Sum term
    for x in func_params:
        func_out += (x ** 2) - 3 * cos(2 * pi * x)

    return func_out


schwe_dict = {
    "n": 10,
    "min": -500,
    "max": 500,
    "interp": interp1d([0, (2 ** 16) - 1], [-500, 500]),
    "inv_interp": interp1d([-500, 500], [0, (2 ** 16) - 1]),
```

```
}


def schwefel(chrom_list: list, n=schwe_dict["n"], interp=schwe_dict["interp"]) -> float:
    """Run the Schwefel function for the chromasome list given.

    :param chrom_list: A list of BitArrays representing the chromosomes
        for each parameter.
    :param n: The number of parameters
    :returns: The output of the function
    """

    # Convert all chromosomes to decimal values
    dec_chroms = [bin_str_to_int(bin_str) for bin_str in chrom_list]

    # Map those numbers to the function range to get the parameters
    func_params = [range_map(chrom, interp) for chrom in dec_chroms]

    # Run the function
    # Non-sum term
    func_out = 418.9829 * n

    # Sum term
    for x in func_params:
        func_out -= x * sin(sqrt(abs(x)))

    return func_out


grie_dict = {
    "n": 10,
    "min": -600,
    "max": 600,
    "interp": interp1d([0, (2 ** 16) - 1], [-600, 600]),
    "inv_interp": interp1d([-600, 600], [0, (2 ** 16) - 1]),
}


def griewangk(chrom_list: list, n=grie_dict["n"], interp=grie_dict["interp"]) -> float:
    """Run the Griewangk function for the chromasome list given.

    :param chrom_list: A list of BitArrays representing the chromosomes
        for each parameter.
    :param n: The number of parameters
    :returns: The output of the function
    """

    # Convert all chromosomes to decimal values
    dec_chroms = [bin_str_to_int(bin_str) for bin_str in chrom_list]

    # Map those numbers to the function range to get the parameters
    func_params = [range_map(chrom, interp) for chrom in dec_chroms]

    # Run the function
    sum = 0
    product = 1

    for i, x in enumerate(func_params):
        sum += (x ** 2) / 4000
        product *= cos(x / sqrt(i + 1))

    return 1 + sum - product


ackl_dict = {
    "n": 30,
    "min": -30,
    "max": 30,
    "interp": interp1d([0, (2 ** 16) - 1], [-30, 30]),
    "inv_interp": interp1d([-30, 30], [0, (2 ** 16) - 1]),
}


def ackley(chrom_list: list, n=ackl_dict["n"], interp=ackl_dict["interp"]) -> float:
```

```
    """Run the Ackley function for the chromasome list given.

    :param chrom_list: A list of BitArrays representing the chromosomes
        for each parameter.
    :param n: The number of parameters
    :returns: The output of the function
    """

    # Convert all chromosomes to decimal values
    dec_chroms = [bin_str_to_int(bin_str) for bin_str in chrom_list]

    # Map those numbers to the function range to get the parameters
    func_params = [range_map(chrom, interp) for chrom in dec_chroms]

    # Calculate sum terms
    sum1 = 0
    sum2 = 0

    dpi = 2 * pi

    for x in func_params:
        sum1 += x ** 2
        sum2 += cos(dpi * x)

    # Calculate exponential terms
    exp_term_1 = exp(-0.2 * sqrt((1 / n) * sum1))
    exp_term_2 = exp((1 / n) * sum2)

    return 20 + exp(1) - 20 * exp_term_1 - exp_term_2
```

Listing 2: *functions.py*

A.3 ga_experiments.py

```
"""
GA Experiment
=============

This file implements a class which is used to complete all standard GA experiments.
This includes the extension.
The CCGA Experiment class is based on this.
"""

import sys
import cProfile

from bitstring import BitArray
import numpy as np
import random
import matplotlib.pyplot as plt

from individual import Individual


class GAExperiment:
    def __init__(
        self,
        _fitness_func,
        _func_dict,
        _evaluations=100000,
        _func_param_num=-1,
        _extension=0,
    ):

        self.fitness_func = _fitness_func
        """The function used to evaluate the population, taken from function.py"""

        self.func_dict = _func_dict
        """A dict containing parameters for the chosen fitness function"""

        self.pop_width = None
        """How many bits contained within each pop.
```

```
        Determined from the number of function parameters * 16 bits.
        """

        self.param_num = None
        """The number of function parameters stored in the geonome"""

        if _func_param_num == -1:
            self.pop_width = 16 * self.func_dict["n"]
            self.param_num = self.func_dict["n"]
        else:
            self.pop_width = 16 * _func_param_num
            self.param_num = _func_param_num

        self.pop_size = 100
        """Number of individuals in a population"""

        self.evaluations = _evaluations
        """The number of function evaluations that should be completed during the
    experiment"""

        self.evaluations_completed = 0
        """The number of evaluations completed so far"""

        self.generation = 0
        """The current generation of population"""

        self.pop = self.get_starting_pop()
        """The population of BitArrays that are evolved during this experiment"""

        self.scaling_window = [0] * 5
        """A record of the worst fitness of the last 5 generations. Used as the
        baseline for comparing all other fitnesses.
        """

        self.fitness_data = []
        """The data to be exported. The lowest fitness of each generation is added
        to this array."""

        self.evaluation_data = []
        """An array containing the function evaluation number that each corresponding
        element in fitness_data was collected."""

        #### Extension ####

        self.crossover = None
        """A placeholder for the function which performs crossover.
        It is selected based on the value of the _extension argument to be either
        standart two_point_crossover or the extension function n_chunk_crossover.
        """

        if _extension == 0:
            self.crossover = self.two_point_crossover
        elif _extension == 1:
            self.crossover = self.n_chunk_crossover

        self.extension = _extension
        """Determins which extension to use.
        If 0: normal GA.
        If 1: n_chunk crossover
        If 2: n individual crossover"""

        #### End Extension ####

        # Update the fitnesses of the pops to prepare for the start of the algorithm
        self.update_fitnesses()

    def get_starting_pop(self):
        """Generate a starter population of random individuals"""

        # Define a BitArray format string
        def_format = "uint:{}={}".format(self.pop_width, "{}")

        # Generate an array of definition strings with random starting numbers
        def_strings = [
```

```
            def_format.format(random.randint(0, (2 ** (self.pop_width)) - 1))
            for i in range(self.pop_size)
        ]

        # Generate bitarrays/genomes from those definition strings
        pop_bitarrays = [BitArray(def_str) for def_str in def_strings]

        # Assign those bitarrays/genomes to individuals in a population
        population = [Individual(arr) for arr in pop_bitarrays]

        return population

    def update_fitnesses(self):
        """Update the fitness values for every individual in self.pop.

        Splits the pop into parameters and runs them through the function given in
        self.fitness_func.
        """

        # Iterate accross all individuals in the population
        for i, ind in enumerate(self.pop):

            # Split geonome into list of parameters
            bin_param_list = self.get_bin_params(ind)

            # Apply those parameters to the given fitness function
            ind.fitness = self.fitness_func(bin_param_list, self.param_num)

        # Update the track of the number of fitness evaluations completed
        self.evaluations_completed += self.pop_size

        # Add current best fitness to data capture
        self.evaluation_data.append(self.evaluations_completed)
        self.fitness_data.append(min(self.pop, key=lambda pop: pop.fitness).fitness)

    def get_bin_params(self, individual: Individual):
        """Split the individual given into 16 bit chunks for the parameters

        :param individual: The BitArray that defines the individual.
        :returns: A list 16-bit BitArrays, each implementing a parameter.
        """

        # Define a list of empty bit arrays to be overwritten with params
        bin_params = [BitArray("uint:16=0")] * self.param_num

        # Assign slices of individual to bin_params
        for i, ind_index in enumerate(range(0, self.param_num * 16, 16)):
            bin_params[i] = individual.bit_arr[ind_index : ind_index + 16]

        return bin_params

    def run_experiment(self):
        """Run the experiment up to a number of function evaluations given in
        evaluations.

        :param iterations: The number of function evaluations
        :returns:
        """

        while self.evaluations_completed < self.evaluations:

            self.generation += 1

            # Update scaling window from previous fitness update
            self.scaling_window.pop()
            self.scaling_window.insert(
                0, max(self.pop, key=lambda pop: pop.fitness).fitness
            )
            self.scaling_factor = max(self.scaling_window)

            # Generate new generation and replace the old one
            self.pop = self.breed_new_population()

            # Update fitness values
```

```
            self.update_fitnesses()

        return self.evaluation_data, self.fitness_data

    def breed_new_population(self):
        """Perform proportional fitness selection to generate the next generation.

        Apply crossover to get a new individual and apply mutation to that individual.
        Leave the most fit individual from the previous generation in the new generation.
        """

        # Declare a new array for the new population with space for the previous best
        new_population = []

        # Add the best pop from the previous generation
        new_population.append(min(self.pop, key=lambda ind: ind.fitness))
        # Generate the roulette wheel
        roulette_wheel = self.get_roulette_wheel()

        # Generate new individuals for the rest of the population
        for i in range(0, self.pop_size - 1):

            # Crossover chance of 0.6
            if random.random() < 0.6:

                # Perform normal 2 parent crossover
                if self.extension in [0,1]:
                    ind1 = self.select_new_ind(roulette_wheel)
                    ind2 = self.select_new_ind(roulette_wheel)
                    new_ind = self.crossover(ind1, ind2)

                # Perform either an N parent crossover or a standard two point crossover
                # This allows an individual to be made from genes of many parents
                # And for genes to be split with two point crossover.
                else:
                    if random.random() < 0.8:
                        parent_list = [self.select_new_ind(roulette_wheel) for i in range
(0, self.param_num)]
                        new_ind = self.n_ind_crossover(parent_list)

                    else:
                        ind1 = self.select_new_ind(roulette_wheel)
                        ind2 = self.select_new_ind(roulette_wheel)
                        new_ind = self.two_point_crossover(ind1, ind2)


            else:
                new_ind = self.select_new_ind(roulette_wheel)

            new_population.append(self.mutate(new_ind))

        return new_population.copy()

    def get_roulette_wheel(self):
        """Generate the proportional fitness roulette wheel for the current population

        Perform a cumulative sum of the population fitnesses.
        By seeing where a random number in that range falls individuals can be selected #
        proportionally to their fitness.

        Since a lower fitness value is better we use the scaling window in reverse,
        subtracting the pop fitness from it. This makes smaller pops take up
        bigger chunks of the wheel.
        """

        # Calculate the cumulative sum of all the population fitnesses
        wheel = np.cumsum([self.scaling_factor - pop.fitness for pop in self.pop])

        return wheel.tolist()

    def select_new_ind(self, roulette_wheel: list):
        """Select a new pop using proportional selection/roulette wheel.

        Generate a random number between 0.0 and 1.0 and find the index this
```

```
        falls within on the roulette wheel. Return the pop at this index.

        This value is found by finding all values greater than the selected
        value and finding the smallest of those. The index of that value is
        then found in the roulette wheel.
        """

        # Get random float between 0.0 and 1.0
        rand = random.random() * roulette_wheel[-1]

        # Get the index of that value in the wheel
        ind_idx = roulette_wheel.index(min([i for i in roulette_wheel if i >= rand]))

        # Create a new individual with the same genes to stop python assigning every
        # member of the population a link back to a single shared BitArray as their
        # genetic material.
        return Individual(self.pop[ind_idx].bit_arr.copy())

    def two_point_crossover(self, ind1: Individual, ind2: Individual) -> Individual:
        """Perform two point crossover on two individuals and return the child

        Two random numbers between 0 and pop_width are generated.
        ind1 is copied to the child. The slice of ind2 between the two points is
        spliced into the child.
        """

        cross_start = random.randint(0, self.pop_width)
        cross_end = random.randint(cross_start, self.pop_width)

        # Clone first individual
        child = Individual(ind1.bit_arr)

        # Splice in part of the second individual
        child.bit_arr[cross_start:cross_end] = ind2.bit_arr[
            cross_start:cross_end
        ].copy()

        return child

    def mutate(self, ind: Individual):
        """Mutate bits in ind with a chance of 1/len(ind)"""

        # Get mutation chance from array length
        mut_chance = 1 / len(ind.bit_arr)

        # For every bit position in ind
        for i in range(0, len(ind.bit_arr)):

            # If mutation chance met, invert bit at that position
            if random.random() < (mut_chance):
                ind.bit_arr.invert(i)

        return ind

    ####EXTENSION####

    def n_chunk_crossover(self, ind1: Individual, ind2: Individual) -> Individual:
        """Create a new individual by taking parameter chunks from each parent at
        random.

        A combination of n-point crossover and uniform crossover.
        It has property of randomly sampling each parent to make the child that
        uniform crossover uses but, it respects the boundaries of the parameters.
        """

        # First create a blank bitarray to store the new child in.
        new_bit_arr = BitArray("uint:{}=0".format(self.pop_width))

        # Get the bitarrays of the parents
        ind1_bit_arr = ind1.bit_arr
        ind2_bit_arr = ind2.bit_arr

        # Generate an N value array of random numbers to decide which
        # chunks come from which parent
```

```
        chunk_rands = np.random.rand(self.param_num).tolist()

        # Assign each chunk in the new bitarray with the equivalent chunk
        # from a randomly selected parent.
        for i, ind_idx in enumerate(range(0, self.param_num * 16, 16)):

            # Define the chunk start and ends
            start = ind_idx
            end = ind_idx + 16

            if chunk_rands[i] < 0.5:
                new_bit_arr[start:end] = ind1_bit_arr[start:end]

            else:
                new_bit_arr[start:end] = ind2_bit_arr[start:end]

        return Individual(new_bit_arr)

    def n_ind_crossover(self, ind_list) -> Individual:
        """Create a new individual by taking parameter chunks from a number of
        parents equal to the number of parameters in being evaluated.

        The is an extension of the n_chunk_crossover designed above.
        It is designed to mimic the use of multiple populations as seen in
        the CCGA as closely as possible.
        """

        # First create a blank bitarray to store the new child in.
        new_bit_arr = BitArray("uint:{}=0".format(self.pop_width))

        # Get the bitarrays of the parents
        bit_arrs = [ind.bit_arr for ind in ind_list]

        # Assign each chunk in the new bitarray with the equivalent chunk
        # from a randomly selected parent.
        for i, ind_idx in enumerate(range(0, self.param_num * 16, 16)):

            # Define the chunk start and ends
            start = ind_idx
            end = ind_idx + 16

            # Select a parent at random and take their parameter
            rand_ind_bit_arr = random.choice(bit_arrs)
            new_bit_arr[start:end] = rand_ind_bit_arr[start:end]

        return Individual(new_bit_arr)
```

Listing 3: *ga_experiment.py*

A.4 ccga_experiments.py

```
"""
CCGA Experiment
==============

This file implements a class that is used to perform all CCGA experiments.
It is based off the ga_experiments file.
"""

from bitstring import BitArray
import numpy as np
import random
import matplotlib.pyplot as plt

from individual import Individual


class CCGAExperiment:
    def __init__(
        self, _fitness_func, _func_dict, _evaluations=100000, _func_param_num=-1
    ):

```

```
        self.fitness_func = _fitness_func
        """The function used to evaluate the population, taken from function.py"""

        self.func_dict = _func_dict
        """A dict containing parameters for the chosen fitness function"""

        self.pop_width = 16
        """How many bits contained within each pop. 16 bits for all population"""

        self.param_num = None
        """The number of function parameters stored in the geonome"""

        if _func_param_num == -1:
            self.param_num = self.func_dict["n"]
        else:
            self.param_num = _func_param_num

        self.pop_num = self.param_num
        """The number of populations used to store all parameters.
        Defined by param_num"""

        self.pop_size = 100
        """Number of individuals in a population"""

        self.evaluations = _evaluations
        """The number of function evaluations that should be completed during the
    experiment"""

        self.evaluations_completed = 0
        """The number of evaluations completed so far"""

        self.generation = 0
        """The current generation"""

        self.pops = [self.get_starting_pop() for i in range(0, self.pop_num)]
        """A set of populations, each defining one function parameter"""

        self.best_ind_idxs = [0] * self.param_num
        """The indexes of the best individuals in each population"""

        self.pops_best_fitness = [10e50] * self.param_num
        """The fitness of the Individuals stored in best_ind_idxs, used for computing
        global fitness each generation"""

        self.scaling_windows = [[0] * 5 for i in range(0, self.pop_num)]
        """Records of the worst fitness in each pop for the last 5 generations.
        Used as the baseline for comparing all other fitnesses.
        """

        self.fitness_data = []
        """The data to be exported. The lowest fitness of each generation is added
        to this array."""

        self.evaluation_data = []
        """An array containing the function evaluation number that each corresponding
        element in fitness_data was collected."""

    def get_starting_pop(self):
        """Generate a starter population of random individuals"""

        # Define a BitArray format string
        def_format = "uint:{}={}".format(self.pop_width, "{}")

        # Generate an array of definition strings with random starting numbers
        def_strings = [
            def_format.format(random.randint(0, (2 ** (self.pop_width)) - 1))
            for i in range(self.pop_size)
        ]

        # Generate bitarrays/genomes from those definition strings
        pop_bitarrays = [BitArray(def_str) for def_str in def_strings]

        # Assign those bitarrays/genomes to individuals in a population
        population = [Individual(arr) for arr in pop_bitarrays]
```

```
95
96          return population
97
98      def set_starting_fitness(self):
99          """Define the fitness of each individual by selecting random individuals from
100         other populations and running them through the fitness function.
101         """
102
103         # Iterate through each member of each population
104         for pop_num, pop in enumerate(self.pops):
105             for ind_num, ind in enumerate(pop):
106
107                 # Get a random member from each other population
108                 # Slot the current one into it
109                 random_param_set = []
110
111                 for i in range(0, self.pop_num):
112
113                     # Add ind under test in the correct position
114                     if i == pop_num:
115                         random_param_set.append(ind.bit_arr)
116
117                     # Get random ind from the current pop, place it in set
118                     else:
119                         random_idx = random.randint(0, self.pop_size - 1)
120                         rand_pop = self.pops[pop_num][random_idx]
121                         random_param_set.append(rand_pop.bit_arr)
122
123                 ind.fitness = self.fitness_func(random_param_set, self.param_num)
124                 self.evaluations_completed += 1
125
126             # Update the best fitness found for every population set
127             self.update_best_inds(pop_num, pop)
128
129     def update_best_inds(self, pop_num, pop):
130         """Find the index of the best individual in each pop and save it and the fitness
131         to global arrays for calculating the fitness of other pops.
132         """
133
134         best_ind = min(pop, key=lambda ind: ind.fitness)
135
136         self.best_ind_idxs[pop_num] = pop.index(best_ind)
137         self.pops_best_fitness[pop_num] = best_ind.fitness
138
139         # Add current best fitness to data capture
140         self.evaluation_data.append(self.evaluations_completed)
141         self.fitness_data.append(min(self.pops_best_fitness))
142
143     def update_fitnesses(self, pop_num):
144         """Update the fitness values for every individual in each pop in self.pops
145
146         A list of the best performing pops from last generation is assembled.
147         Each ind in each pop is substituted into the relevant position and the
148         new set is evaluated.
149         This allows each ind to be compared against the best from each other
150         pop.
151         """
152
153         # Assemble a "Greatest Hits" list of the best ind bit_arrays from each pop
154         best_inds_copy = [
155             self.pops[p_num][i_num].bit_arr.copy()
156             for p_num, i_num in enumerate(self.best_ind_idxs)
157         ]
158
159         # Test every ind in this pop against the best from the previous generation
160         for ind in self.pops[pop_num]:
161             best_inds_copy[pop_num] = ind.bit_arr
162             ind.fitness = self.fitness_func(best_inds_copy, self.param_num)
163             self.evaluations_completed += 1
164
165         # Update the best fitness found for every population set
166         self.update_best_inds(pop_num, self.pops[pop_num])
167
168     def update_scaling_windows(self):
```

```
        """Update all scaling windows with the worst fitness value from each population
        from the last generation.
        """

        # Update each scaling window from previous fitness update
        for pop_num, pop in enumerate(self.pops):
            self.scaling_windows[pop_num].pop()
            self.scaling_windows[pop_num].insert(
                0, max(pop, key=lambda ind: ind.fitness).fitness
            )

    def run_experiment(self):
        """Run the experiment up to a number of function evaluations given in
        evaluations.
        """

        # Update the fitnesses of the pops to prepare for the start of the algorithm
        self.set_starting_fitness()

        try:
            while self.evaluations_completed < self.evaluations:

                self.generation += 1

                # print(
                #     "Generation:",
                #     self.generation,
                #     "\tEvaluation",
                #     self.evaluations_completed,
                # )

                self.update_scaling_windows()

                # Generate a new generation of each pop
                for pop_num, pop in enumerate(self.pops):
                    scaling_factor = max(self.scaling_windows[pop_num])
                    self.pops[pop_num] = self.breed_new_population(pop, scaling_factor)
                    self.update_fitnesses(pop_num)

        except KeyboardInterrupt:
            print("Halting simulation...")

        return self.evaluation_data, self.fitness_data

    def breed_new_population(self, pop, scaling_factor):
        """Perform proportional fitness selection to generate the next generation.

        Apply crossover to get a new individual and apply mutation to that individual.
        Leave the most fit individual from the previous generation in the new generation.
        """

        # Declare a new array for the new population with space for the previous best
        new_population = []

        # Add the best pop from the previous generation
        new_population.append(min(pop, key=lambda ind: ind.fitness))

        # Generate the roulette wheel
        roulette_wheel = self.get_roulette_wheel(pop, scaling_factor)

        # Generate new individuals for the rest of the population
        for i in range(0, self.pop_size - 1):

            # Crossover chance of 0.6
            if random.random() < 0.6:
                ind1 = self.select_new_ind(roulette_wheel, pop)
                ind2 = self.select_new_ind(roulette_wheel, pop)
                new_ind = self.two_point_crossover(ind1, ind2)

            else:
                new_ind = self.select_new_ind(roulette_wheel, pop)

            new_population.append(self.mutate(new_ind))

```

```
        return new_population.copy()

    def get_roulette_wheel(self, pop, scaling_factor):
        """Generate the proportional fitness roulette wheel for the current population

        Perform a cumulative sum of the population fitnesses.
        By seeing where a random number in that range falls individuals can be selected #
        proportionally to their fitness.

        Since a lower fitness value is better we use the scaling window in reverse,
        subtracting the pop fitness from it. This makes smaller pops take up
        bigger chunks of the wheel.
        """

        # Calculate the cumulative sum of all the population fitnesses
        wheel = np.cumsum([scaling_factor - ind.fitness for ind in pop])

        return wheel.tolist()

    def select_new_ind(self, roulette_wheel: list, pop: list):
        """Select a new pop using proportional selection/roulette wheel.

        Generate a random number between 0.0 and 1.0 and find the index this
        falls within on the roulette wheel. Return the pop at this index.

        This value is found by finding all values greater than the selected
        value and finding the smallest of those. The index of that value is
        then found in the roulette wheel.
        """

        # Get random float between 0.0 and 1.0
        rand = random.random() * roulette_wheel[-1]

        # Get the index of that value in the wheel
        try:
            ind_idx = roulette_wheel.index(
                min([i for i in roulette_wheel if i >= rand])
            )
        except ValueError as e:
            print(roulette_wheel)
            print("\n\n")
            print(self.scaling_windows)
            print("\n\n")
            print(pop[0].fitness)
            raise e

        # Create a new individual with the same genes to stop python assigning every
        # member of the population a link back to a single shared BitArray as their
        # genetic material.
        return Individual(pop[ind_idx].bit_arr.copy())

    def two_point_crossover(self, ind1: Individual, ind2: Individual) -> Individual:
        """Perform two point crossover on two individuals and return the child

        Two random numbers between 0 and pop_width are generated.
        ind1 is copied to the child. The slice of ind2 between the two points is
        spliced into the child.
        """

        cross_start = random.randint(0, self.pop_width)
        cross_end = random.randint(cross_start, self.pop_width)

        # Clone first individual
        child = Individual(ind1.bit_arr)

        # Splice in part of the second individual
        child.bit_arr[cross_start:cross_end] = ind2.bit_arr[
            cross_start:cross_end
        ].copy()

        return child

    def mutate(self, ind: Individual):
        """Mutate bits in ind with a chance of 1/len(ind)"""
```

```
317
318         # Precalculate mutation chance
319         mut_chance = 1 / len(ind.bit_arr)
320
321         # For every bit position in ind
322         for i in range(0, len(ind.bit_arr)):
323
324             # If mutation chance met, invert bit at that position
325             if random.random() < (mut_chance):
326                 ind.bit_arr.invert(i)
327
328         return ind
```

Listing 4: *ccga_experiment.py*

A.5 main_data_gather.py

```
1  """
2  Main Data Gather
3  ================
4
5  The project's main file. All data generated for this project is done so in here.
6  No plots are produced. The data is gathered and saved to disk to be plotted
7  in MATLAB.
8  """
9
10 import numpy as np
11 import os
12 import sys
13
14 from functions import (
15     rastrigin,
16     rast_dict,
17     schwefel,
18     schwe_dict,
19     griewangk,
20     grie_dict,
21     ackley,
22     ackl_dict,
23 )
24
25 import matplotlib.pyplot as plt
26
27 from ga_experiment import GAExperiment
28 from ccga_experiment import CCGAExperiment
29
30
31 def run_ga_experiment(
32     fitness_func, func_dict, iterations, num_experiments, extension=0
33 ):
34     """Runs all standard ga experiments and saves results to disk"""
35
36     GAs = [
37         GAExperiment(fitness_func, func_dict, iterations, _extension=extension)
38         for i in range(0, num_experiments)
39     ]
40
41     print("Finished Generating GAs")
42
43     iteration_data = []
44     sum_fitness_data = []
45
46     sum_fitness_data = np.array([0.0] * int(iterations / 100))
47
48     for i, GA in enumerate(GAs):
49         iteration_data, fitness_data = GA.run_experiment()
50         sum_fitness_data += np.array(fitness_data)
51
52         print("Experiment {} Complete".format(i))
53
54     avr_fitness_data = sum_fitness_data / num_experiments
55
```

```
    return iteration_data, avr_fitness_data


def run_ccga_experiment(
    fitness_func, func_dict, iterations, num_experiments, param_num=-1
):
    """Runs all standard ga experiments and saves results to disk"""

    if param_num == -1:
        param_num = func_dict["n"]

    GAs = [
        CCGAExperiment(fitness_func, func_dict, iterations, param_num)
        for i in range(0, num_experiments)
    ]

    print("Finished Generating GAs")

    # Compute the number of outputs the GA will generate
    output_size = 0
    while output_size < iterations:
        output_size += param_num * 100

    output_size = int(output_size / 100)

    iteration_data = []
    sum_fitness_data = np.array([0.0] * output_size)

    for i, GA in enumerate(GAs):
        iteration_data, fitness_data = GA.run_experiment()
        sum_fitness_data += np.array(fitness_data)

        print("Experiment {} Complete".format(i))

    avr_fitness_data = sum_fitness_data / num_experiments

    return iteration_data, avr_fitness_data


def write_to_file(iter_data, avr_fitness_data, filepath):
    """Output data given to a file on path given"""

    cwd = os.getcwd()

    # Generate output lines
    out_lines = [
        "{}, {}\n".format(itr, avr_fitness)
        for itr, avr_fitness in zip(iter_data, avr_fitness_data)
    ]

    with open(cwd + "\\" + filepath, "w+") as output_file:
        output_file.writelines(out_lines)


def run_ga_experiments(experiment_num):

    print("GA Experiments")
    output_data_path = "collected_data\\ga\\"

    # Run standard GA experiments
    print("Rastrigin Experiment")
    rast_iter, rast_avr_fitness = run_ga_experiment(
        rastrigin, rast_dict, 100000, experiment_num
    )
    write_to_file(rast_iter, rast_avr_fitness, output_data_path + "ga_rast.txt")

    print("Schwefel Experiment")
    schw_iter, schw_avr_fitness = run_ga_experiment(
        schwefel, schwe_dict, 100000, experiment_num
    )
    write_to_file(schw_iter, schw_avr_fitness, output_data_path + "ga_schw.txt")

    print("Griewangk Experiment")
    grie_iter, grie_avr_fitness = run_ga_experiment(
```

```
        griewangk, grie_dict, 100000, experiment_num
    )
    write_to_file(grie_iter, grie_avr_fitness, output_data_path + "ga_grie.txt")

    print("Ackley Experiment")
    ackl_iter, ackl_avr_fitness = run_ga_experiment(
        ackley, ackl_dict, 100000, experiment_num
    )
    write_to_file(ackl_iter, ackl_avr_fitness, output_data_path + "ga_ackl.txt")

    # Write data to disk


def run_ccga_experiments(experiment_num):

    print("CCGA Experiments")

    output_data_path = "collected_data\\ccga\\"

    print("Rastrigin Experiment")
    rast_iter, rast_avr_fitness = run_ccga_experiment(
        rastrigin, rast_dict, 100000, experiment_num
    )
    write_to_file(rast_iter, rast_avr_fitness, output_data_path + "ccga_rast.txt")

    print("Schwefel Experiment")
    schw_iter, schw_avr_fitness = run_ccga_experiment(
        schwefel, schwe_dict, 100000, experiment_num
    )
    write_to_file(schw_iter, schw_avr_fitness, output_data_path + "ccga_schw.txt")

    print("Griewangk Experiment")
    grie_iter, grie_avr_fitness = run_ccga_experiment(
        griewangk, grie_dict, 100000, experiment_num
    )
    write_to_file(grie_iter, grie_avr_fitness, output_data_path + "ccga_grie.txt")

    # Run standard GA experiments
    print("Ackley Experiment")
    ackl_iter, ackl_avr_fitness = run_ccga_experiment(
        ackley, ackl_dict, 100000, experiment_num
    )
    write_to_file(ackl_iter, ackl_avr_fitness, output_data_path + "ccga_ackl.txt")


def run_exga_experiments(experiment_num, extension):

    print("EXGA_{} Experiments".format(extension))

    expname = "exga_" + str(extension)

    output_data_path = "collected_data\\{}\\".format(expname)

    # Run standard GA experiments
    print("Rastrigin Experiment")
    rast_iter, rast_avr_fitness = run_ga_experiment(
        rastrigin, rast_dict, 100000, experiment_num, extension=1
    )
    write_to_file(rast_iter, rast_avr_fitness, output_data_path + expname + "_rast.txt")

    print("Schwefel Experiment")
    schw_iter, schw_avr_fitness = run_ga_experiment(
        schwefel, schwe_dict, 100000, experiment_num, extension=1
    )
    write_to_file(schw_iter, schw_avr_fitness, output_data_path + expname + "_schw.txt")

    print("Griewangk Experiment")
    grie_iter, grie_avr_fitness = run_ga_experiment(
        griewangk, grie_dict, 100000, experiment_num, extension=1
    )
    write_to_file(grie_iter, grie_avr_fitness, output_data_path + expname + "_grie.txt")

    print("Ackley Experiment")
    ackl_iter, ackl_avr_fitness = run_ga_experiment(
```

```
        ackley, ackl_dict, 100000, experiment_num, extension=1
    )
    write_to_file(ackl_iter, ackl_avr_fitness, output_data_path + expname + "_ackl.txt")


if __name__ == "__main__":

    if len(sys.argv) > 2:
        experiment_num = int(sys.argv[2])
    else:
        experiment_num = 15

    # Run GA experiments if requested
    if sys.argv[1] in ["ga", "all"]:
        run_ga_experiments(experiment_num)

    # Run CCGA experiments if requested
    if sys.argv[1] in ["ccga", "all"]:
        run_ccga_experiments(experiment_num)

    # Run EXGA experiments if requested
    if sys.argv[1] in ["exga", "all"]:

        # Set extension number from cmd, default to 1
        extension = 1

        if len(sys.argv) > 3:
            extension = int(sys.argv[3])

        run_exga_experiments(experiment_num, extension)
```

Listing 5: *main_data_gather.py*

A.6 combined_plots.m

```
load ..\ga\ga_rast.txt
load ..\ga\ga_schw.txt
load ..\ga\ga_ackl.txt
load ..\ga\ga_grie.txt

load ..\ccga\ccga_rast.txt
load ..\ccga\ccga_schw.txt
load ..\ccga\ccga_ackl.txt
load ..\ccga\ccga_grie.txt

combined_plot = figure();

% Rastrigin Function
subplot(2, 2, 1)
plot(ga_rast(:,1), ga_rast(:,2))
hold on
plot(ccga_rast(:,1), ccga_rast(:,2))
legend("GA", "CCGA")
subtitle("Rastrigin Function")
ylabel("Best Individual")
xlabel("Function Evaluations")
xlim([0 100000])
ylim([0 40])

% Schwefel Function
subplot(2, 2, 2)
plot(ga_schw(:,1), ga_schw(:,2))
hold on
plot(ccga_schw(:,1), ccga_schw(:,2))
legend("GA", "CCGA")
subtitle("Schwefel Function")
ylabel("Best Individual")
xlabel("Function Evaluations")
xlim([0 100000])
ylim([0 400])

% Griewangk Function
```

```
subplot(2, 2, 3)
plot(ga_grie(:,1), ga_grie(:,2))
hold on
plot(ccga_grie(:,1), ccga_grie(:,2))
legend("GA", "CCGA")
subtitle("Griewangk Function")
ylabel("Best Individual")
xlabel("Function Evaluations")
xlim([0 100000])
ylim([0 8])

% Ackley Function
subplot(2, 2, 4)
plot(ga_ackl(:,1), ga_ackl(:,2))
hold on
plot(ccga_ackl(:,1), ccga_ackl(:,2))
legend("GA", "CCGA")
subtitle("Ackley Function")
ylabel("Best Individual")
xlabel("Function Evaluations")
xlim([0 100000])
ylim([0 16])

saveas(combined_plot, "../../../Report/img/combined_plot.png")
```

Listing 6: *combined_plots.py*

A.7 extension_plots.m

```
load ..\ga\ga_rast.txt
load ..\ga\ga_schw.txt
load ..\ga\ga_ackl.txt
load ..\ga\ga_grie.txt

load ..\ccga\ccga_rast.txt
load ..\ccga\ccga_schw.txt
load ..\ccga\ccga_ackl.txt
load ..\ccga\ccga_grie.txt

load ..\exga_1\exga_1_rast.txt
load ..\exga_1\exga_1_schw.txt
load ..\exga_1\exga_1_ackl.txt
load ..\exga_1\exga_1_grie.txt

load ..\exga_2\exga_2_rast.txt
load ..\exga_2\exga_2_schw.txt
load ..\exga_2\exga_2_ackl.txt
load ..\exga_2\exga_2_grie.txt


extension_plot = figure();

% Rastrigin Function
subplot(2, 2, 1)
plot(ga_rast(:,1), ga_rast(:,2))
hold on
plot(ccga_rast(:,1), ccga_rast(:,2))
plot(exga_1_rast(:,1), exga_1_rast(:,2))
plot(exga_2_rast(:,1), exga_2_rast(:,2))
legend("GA", "CCGA", "EXGA\_1", "EXGA\_2")
subtitle("Rastrigin Function")
ylabel("Best Individual")
xlabel("Function Evaluations")
xlim([0 100000])
ylim([0 40])

% Schwefel Function
subplot(2, 2, 2)
plot(ga_schw(:,1), ga_schw(:,2))
hold on
plot(ccga_schw(:,1), ccga_schw(:,2))
plot(exga_1_schw(:,1), exga_1_schw(:,2))
```

```
plot(exga_2_schw(:,1), exga_2_schw(:,2))
legend("GA", "CCGA", "EXGA\_1", "EXGA\_2")
subtitle("Schwefel Function")
ylabel("Best Individual")
xlabel("Function Evaluations")
xlim([0 100000])
ylim([0 400])

% Griewangk Function
subplot(2, 2, 3)
plot(ga_grie(:,1), ga_grie(:,2))
hold on
plot(ccga_grie(:,1), ccga_grie(:,2))
plot(exga_1_grie(:,1), exga_1_grie(:,2))
plot(exga_2_grie(:,1), exga_2_grie(:,2))
legend("GA", "CCGA", "EXGA\_1", "EXGA\_2")
subtitle("Griewangk Function")
ylabel("Best Individual")
xlabel("Function Evaluations")
xlim([0 100000])
ylim([0 8])

% Ackley Function
subplot(2, 2, 4)
plot(ga_ackl(:,1), ga_ackl(:,2))
hold on
plot(ccga_ackl(:,1), ccga_ackl(:,2))
plot(exga_1_ackl(:,1), exga_1_ackl(:,2))
plot(exga_2_ackl(:,1), exga_2_ackl(:,2))
legend("GA", "CCGA", "EXGA\_1", "EXGA\_2")
subtitle("Ackley Function")
ylabel("Best Individual")
xlabel("Function Evaluations")
xlim([0 100000])
ylim([0 16])

saveas(extension_plot, "../../../Report/img/extension_plot.png")
```

Listing 7: *extension_plots.py*