Zadanie nr 3 - Prosty algorytm genetyczny

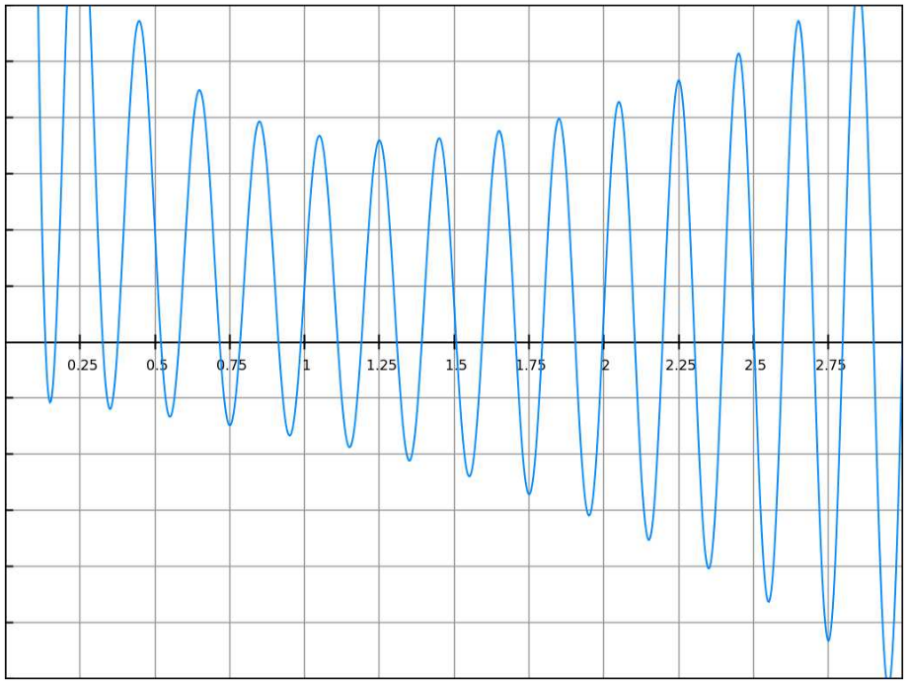
Inteligentne przetwarzanie danych

Mateusz Szczęsny, 233266 Dawid Wójcik, 233271

30.11.2019 r.

1 Cel zadania

Celem zadania jest znalezienie punktu x_0 z przedziału $[0.5, 2.5]$ maksymalizującego funkcję $f(x) = [\, e^x \cdot sin(10\pi x) + 1 \,] \,/\, x$, której wykres znajduje się na poniższym rysunku.



Rysunek 1: Wykres maksymalizowanej funkcji

Do poszukiwań wykorzystać samodzielnie zaimplementowany (w dowolnym języku programowania) algorytm genetyczny, maksimum globalne funkcji x_0 wyznaczyć z dokładnością do 3 miejsc po kropce dziesiętnej. W algorytmie genetycznym do selekcji wykorzystać metodę ruletki.

2 Wstęp teoretyczny

W algorytmie genetyczne rozróżniamy następujących pojęcia:

- *Populacja* - zbiór osobników, o określonej liczebności,
- *Osobniki* - zakodowane w postaci chromosomów zbiory parametrów zadania,
- *Chromosom* - uporządkowany ciąg genów,

- *Gen* - pojedynczy element genotypu, w szczególności chromosomu,
- *Genotyp* - zespół chromosomów danego osobnika,
- *Fenotyp* - zestaw wartości odpowiadających danemu genotypowi,
- *Allel* - wartość danego genu,
- *Funkcja przystosowania* - miara przystosowania danego osobnika w populacji. Pozwala ocenić stopień przystosowania poszczególnych osobników w populacji i na tej podstawie wybrać osobniki najbardziej przystosowane,
- *Generacja* - kolejna iteracja w algorytmie genetycznym,
- *Pokolenie* - kolejna nowo utworzona populacja.

Algorytm genetyczny przekształca populację osobników, z których każdy ma określoną wartość funkcji dopasowania, w następne pokolenie używając Darwinowskich zasad ewolucji, z wykorzystaniem operacji krzyżowania i mutacji. Każdy możliwy punkt w przestrzeni poszukiwań jest kodowany na potrzeby algorytmu genetycznego. W kolejnych pokoleniach poszukiwany jest najlepszy osobnik.

Etap przygotowawczy:

1. *R*eprezentacja i sposób kodowania – określany jest sposób przekształcenia przestrzeni poszukiwań w łańcuch ostałej długości (chromosom) oraz sposób zdekodowania konkretnego łańcucha w punkt z przestrzeni poszukiwań,
2. *O*kreślenie funkcji dopasowania, która przyporządkowuje liczbową wartość dopasowania do każdego osobnika z populacji. Wartość funkcji dopasowania decyduje o prawdopodobieństwie wyboru danego osobnika do następnego pokolenia,
3. *O*szacowanie parametrów startowych algorytmu (wielkość populacji, maks. liczba pokoleń),
4. *O*kreślenie kryterium zatrzymania i wyboru rozwiązania.

Ogólne zasady wyboru osobnika do następnego pokolenia:

- Osobnik lepiej dopasowany ma większe szanse wyboru,
- Możliwa jest reselekcja tzn. osobnik lepiej dopasowany może być wybrany kilkakrotnie,
- Selekcja jest losowa.

3 Przebieg eksperymentu

Eksperyment polega na przeanalizowaniu jakości oraz wydajności poszukiwania maksimum globalnego funkcji wymienionej w sekcji *1. Cel zadania.* Kolejne sekcje tego rozdziału opisują kolejne etapy eksperymentu.

3.1 Generacja populacji

Etap pierwszy polega na wygenerowaniu wstępnej populacji, która będzie służyła jako baza dla kolejnych iteracji. Poniższe dane określają zasady, według których, zostaną wylosowane osobniki.

Dane wejściowe:

- CHROMOSOMES_NUMBER - ilość chromosomów w populacji,
- NUM_OF_EPOCHS - ilość epok,
- MIN_RANGE - dolny zakres dziedziny funkcji,
- MAX_RANGE - górny zakres dziedziny funkcji,
- DECIMAL_PRECISION - dokładność dziesiętna,
- POPULATION - wstępna populacja.

Poniższy kod prezentuje implementację funkcji odpowiedzialnej za wygenerowanie początkowej populacji.

```
#gen.py
def generate_chromosomes():
    chroms = []
    for _ in range(CHROMOSOMES_NUMBER):
        chroms.append(
            Chromosome(
                random.uniform(MIN_RANGE, MAX_RANGE),
                (MIN_RANGE, MAX_RANGE),
                DECIMAL_PRECISION,
            )
        )
    return chroms
```

3.2 Tworzenie nowej populacji na podstawie poprzedniej

W tym etapie następuje zastąpienie starych zasobników nowymi. Chromosomy powstają w wyniku trzech metod: mutacji, krzyżowania i przeżywania. Każda z tych metod posiada własny parametr, który określa prawdopodobieństwo jej wystąpienia. Poniższy kod prezentuje wartości przyjęte w celu przeprowadzenia badania.

```
CROSSOVER = "crossover"
CROSSOVER_PROB = 0.7
MUTATION = "mutation"
MUTATION_PROB = 0.01
SURVIVE = "survive"
SURVIVE_PROB = 1 - MUTATION_PROB - CROSSOVER_PROB
OPERATIONS = [
    (CROSSOVER, CROSSOVER_PROB),
    (MUTATION, MUTATION_PROB),
    (SURVIVE, SURVIVE_PROB),
]
```

W celu użyciu wylosowanej metody konieczne jest dobranie odpowiednich osobników z populacji. Aby doprowadzić do tego że, osobniki osiągające lepsze wyniki w przyjętej przez Nas funkcji adaptacji, zastosowaliśmy metodę ruletki, która na podstawie wylosowanej wartości adaptacji, wybiera poszczególne chromosomy jako rodziców do następnej generacji. Poniższy kod przedstawia implementacje tej funkcjonalności.

```python
#gen.py
def generate_population(population: typing.List):
    chroms = []
    for i in range(CHROMOSOMES_NUMBER):
        pass
        # STEP1: choose operaion
        r = random.random()
        operation = None
        for op, prob in OPERATIONS:
            r -= prob
            if r <= 0:
                operation = op
                break
        # STEP2: do operaion
        result = None
        if operation is CROSSOVER:
            result = crossover(
                russian_roulette(population),
    russian_roulette(population)
            )
        if operation is MUTATION:
            result = russian_roulette(population).mutate()
        if operation is SURVIVE:
            result = survive(russian_roulette(population))
        # STEP3: append to new population
        if type(result) is tuple:
            chroms.append(result[0])
            chroms.append(result[1])
        else:
            chroms.append(result)
    return chroms
```

```python
#chromosome.py
def adaptationFunc(x):
    return (math.exp(x) * math.sin(10 * math.pi * x) + 1) / x
```

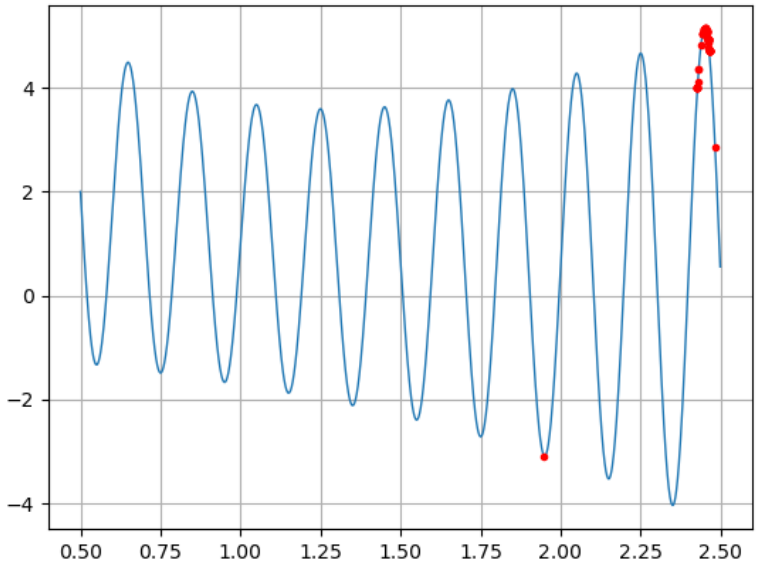
```python
#gen.py
def russian_roulette(population: typing.List[Chromosome]):
    chroms_with_adaptation = [(c, c.adapted) for c in
    population]
    s = sum([c[1] for c in chroms_with_adaptation])
    r = random.uniform(0, s)
    s_local = 0
    for chrom, adaptation in chroms_with_adaptation:
        s_local = s_local + adaptation
        if s_local > r:
            return chrom
```

Opisany wyżej algorytm pozwala Nam na zastąpienie starej populacji nowymi osobnikami, które osiągają lepsze wyniki w przyjętej przez Nas funkcji adaptacji.

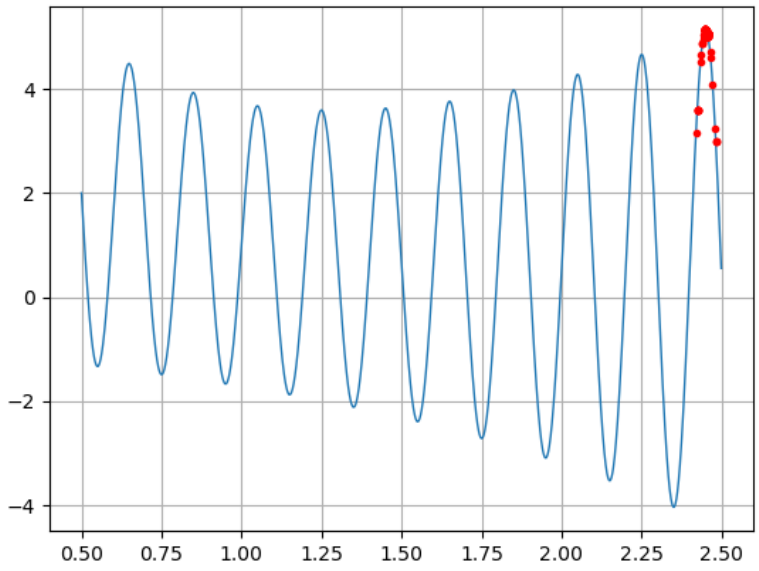
3.3 Rezultaty

Eksperyment wykonaliśmy w trzech wariantach: 25, 50 i 100 epok. Reszta danych wejściowych nie uległą zmianie. Poniżej prezentujemy otrzymane rezultaty.

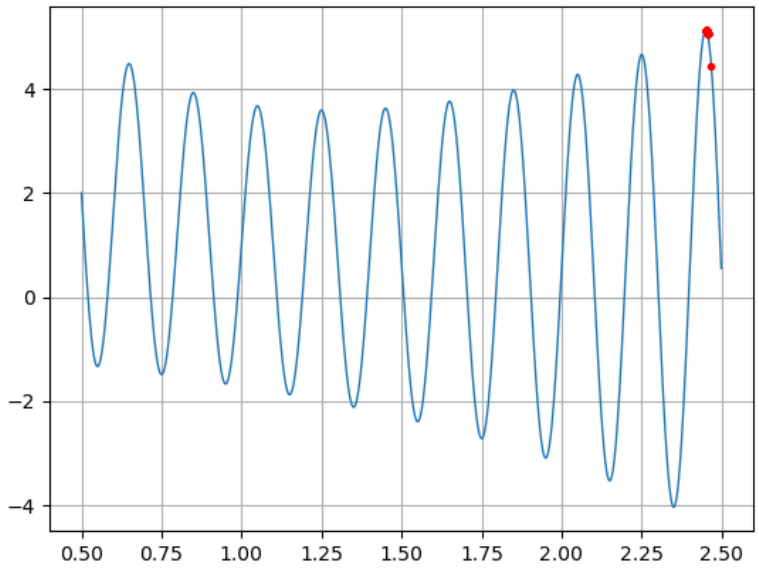
Rysunki 2, 3, 4 przedstawiają wykresy funkcji bazowej oraz punkty będące rezultatami uzyskanymi w najmłodszym pokoleniu.



Rysunek 2: Wykres maksymalizowanej funkcji dla 25 epok

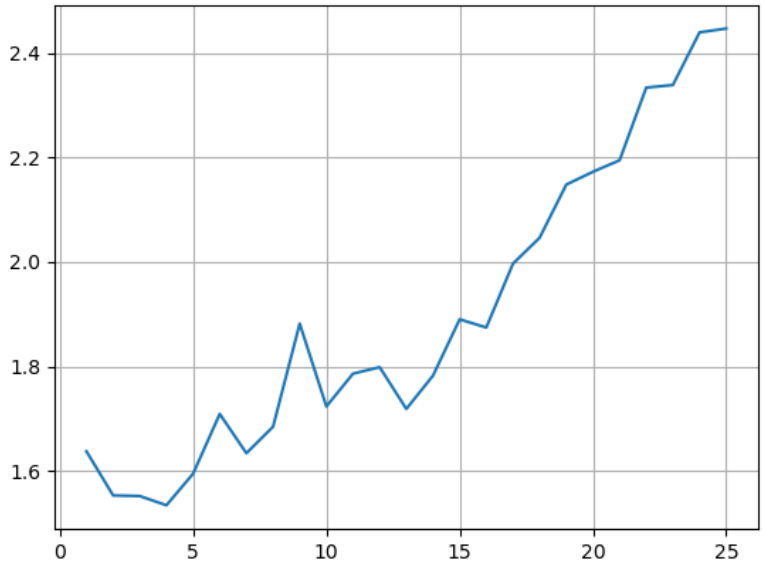


Rysunek 3: Wykres maksymalizowanej funkcji dla 50 epok



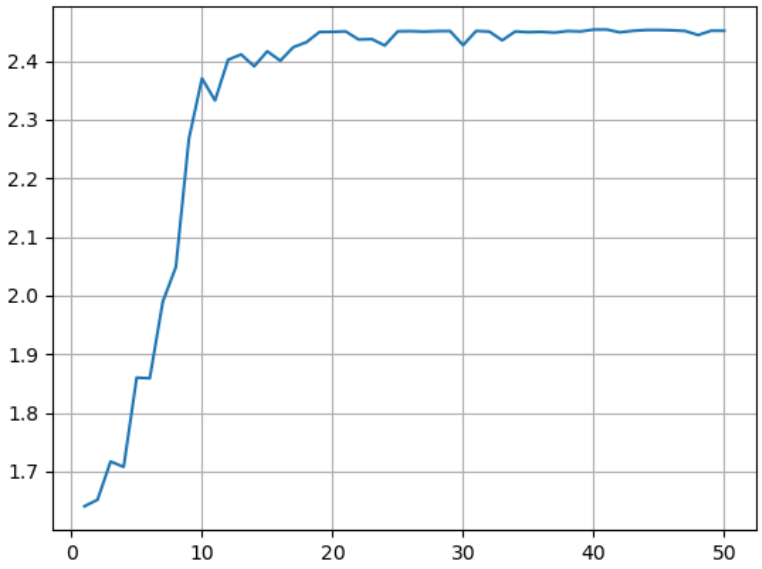
Rysunek 4: Wykres maksymalizowanej funkcji dla 100 epok

Rysunki 5, 6 i 7 przedstawiają przebieg średniej wartości funkcji dopasowania pokolenia w funkcji numeru pokolenia dla wszystkich badanych przypadków.



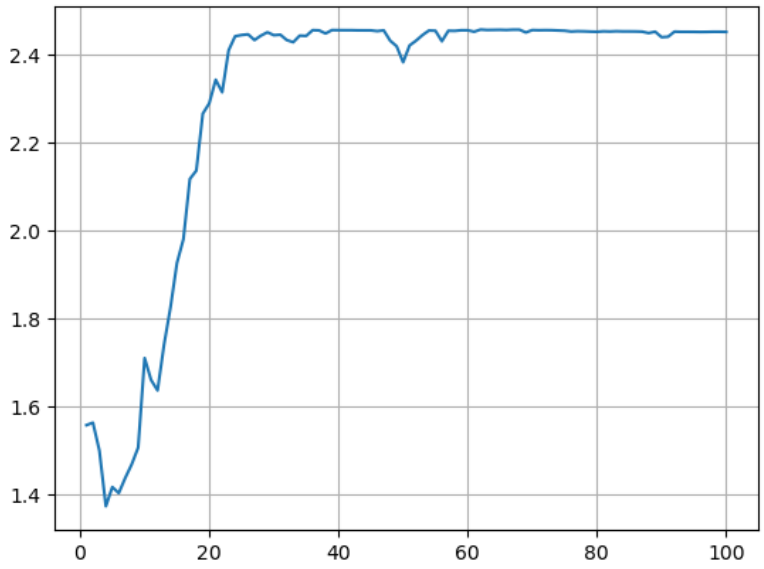
Rysunek 5: Przebieg średniej wartości funkcji dopasowania dla 25 epok

Średni wynik dla ostatniej populacji wyniósł $2,44$



Rysunek 6: Przebieg średniej wartości funkcji dopasowania dla 50 epok

Średni wynik dla ostatniej populacji wyniósł $2,45$
Średni wynik dla ostatniej populacji wyniósł $2,45$



Rysunek 7: Przebieg średniej wartości funkcji dopasowania dla 100 epok

Poniższe tabele 1, 2 oraz 3 prezentują wybiórcze wyniki uzyskane w najmłodszym pokoleniu (dane zostały pomniejszone o wartości powtarzalne oraz nieznaczące).

x_0	x_bin	f(x)
2.463	11110101011	4.781
2.452	11110100000	5.134
2.428	11110001000	4.009
2.429	11110001001	4.103
2.456	11110100100	5.07
2.455	11110100011	5.093
1.949	10110101001	-3.088
2.452	11110100000	5.134
2.483	11110111111	2.858
2.46	11110101000	4.932
...		

Tabela 1: Wartości uzyskane dla 25 epok

x_0	x_bin	f(x)
2.448	11110011100	5.124
2.45	11110011110	5.138
2.458	11110100110	5.01
2.45	11110011110	5.138
2.482	11110111110	2.986
2.424	11110000100	3.601
2.44	11110010100	4.882
2.456	11110100100	5.07
2.452	11110100000	5.134
2.458	11110100110	5.01
...		

Tabela 2: Wartości uzyskane dla 50 epok

x_0	x_bin	f(x)
2.452	11110100000	5.134
2.452	11110100000	5.134
2.452	11110100000	5.134
2.456	11110100100	5.07
2.452	11110100000	5.134
2.452	11110100000	5.134
2.456	11110100100	5.07
2.452	11110100000	5.134
2.452	11110100000	5.134
2.453	11110100001	5.125
...		

Tabela 3: Wartości uzyskane dla 100 epok

4 Wnioski

1. Metoda nie jest idealna, nie wszystkie otrzymane wyniki są prawidłowe. Ich uśrednienie pozwala oszacować prawidłowe rozwiązanie, niemniej jednak wynik nie zawsze jest jednoznaczny.

2. Łatwa modyfikacja parametrów takich jak rozmiar populacji, parametry mutacji oraz krzyżowania.

3. Możliwość użycia różnych metod dobierania osobników (metoda ruletki, turniejowa itp.) w zależności od analizowanego problemu.

4. Pewna losowość wyników zależna od populacji oraz rodzaju analizowanego problemu (np. funkcji).

5. Duża kontrola nad procesem "uczenia" z uwagi na możliwość doboru metod oraz parametrów.

6. Każdy z dokonanych eksperymentów przy założonej ilości iteracji dał satysfakcjonujący wynik, natomiast wielkość populacji miała znaczenie na prędkość "uczenia" osobników.

5 Załączniki

```python
# gen.py
import numpy as np
import random
import matplotlib.pyplot as plt
import math
from chromosome import Chromosome, map_range, adaptationFunc
import typing

CHROMOSOMES_NUMBER = 50
NUM_OF_EPOCHS = 25
MIN_RANGE = 0.5
MAX_RANGE = 2.5
DECIMAL_PRECISION = 3
POPULATION = []
AVERAGE_RESULTS = {}


def mutate(c: Chromosome) -> Chromosome:
    c.mutate()
    return c


def crossover(c1: Chromosome, c2: Chromosome) -> typing.List[
    Chromosome]:
    child = Chromosome.crossover(c1, c2)
    return child


def survive(c: Chromosome) -> Chromosome:
    return c


CROSSOVER = "crossover"
CROSSOVER_PROB = 0.7
MUTATION = "mutation"
MUTATION_PROB = 0.01
SURVIVE = "survive"
SURVIVE_PROB = 1 - MUTATION_PROB - CROSSOVER_PROB
OPERATIONS = [
    (CROSSOVER, CROSSOVER_PROB),
    (MUTATION, MUTATION_PROB),
    (SURVIVE, SURVIVE_PROB),
]


def generate_chromosomes():
```

```
    chroms = []
    for _ in range(CHROMOSOMES_NUMBER):
        chroms.append(
            Chromosome(
                random.uniform(MIN_RANGE, MAX_RANGE),
                (MIN_RANGE, MAX_RANGE),
                DECIMAL_PRECISION,
            )
        )
    return chroms


def russian_roulette(population: typing.List[Chromosome]):
    chroms_with_adaptation = [(c, c.adapted) for c in
    population]
    s = sum([c[1] for c in chroms_with_adaptation])
    r = random.uniform(0, s)
    s_local = 0
    for chrom, adaptation in chroms_with_adaptation:
        s_local = s_local + adaptation
        if s_local > r:
            return chrom


def generate_population(population: typing.List):
    chroms = []
    for i in range(CHROMOSOMES_NUMBER):
        pass
        # STEP1: choose operaion
        r = random.random()
        operation = None
        for op, prob in OPERATIONS:
            r -= prob
            if r <= 0:
                operation = op
                break
        # STEP2: do operaion
        result = None
        if operation is CROSSOVER:
            result = crossover(
                russian_roulette(population),
    russian_roulette(population)
            )
        if operation is MUTATION:
            result = russian_roulette(population).mutate()
        if operation is SURVIVE:
            result = survive(russian_roulette(population))
        # STEP3: append to new population
        if type(result) is tuple:
```

```
            chroms.append(result[0])
            chroms.append(result[1])
        else:
            chroms.append(result)
    return chroms


def dump_average(i: int, population: typing.List):
    AVERAGE_RESULTS[i + 1] = sum([c.value for c in population
    ]) / len(population)


def main():
    plt.figure()

    # draw base function
    xs = [x for x in np.arange(MIN_RANGE, MAX_RANGE, 0.001)]
    ys = [adaptationFunc(x) for x in xs]
    plt.plot(xs, ys, linewidth=1)

    # deal with population
    POPULATION = generate_chromosomes()
    for i in range(NUM_OF_EPOCHS):
        POPULATION = generate_population(POPULATION)
        dump_average(i, POPULATION)

    # visualize results
    plt.plot(
        [c.value for c in POPULATION],
        [c.adapted for c in POPULATION],
        "ro",
        markersize=3,
    )
    plt.grid()

    plt.figure()
    plt.plot(
        [x + 1 for x in range(NUM_OF_EPOCHS)], [v for _, v in
    AVERAGE_RESULTS.items()]
    )
    plt.grid()

    plt.show()

    # dump results to file
    with open("result.txt", "w+") as f:
        for chromosome in POPULATION:
            f.write(
```

```
139                 f"value: {round(chromosome.value, 3)} \t bin:
    {''.join([str(x) for x in chromosome.binary])} \t adapted
    : {round(chromosome.adapted, 3)} \n"
140             )
141
142
143 if __name__ == "__main__":
144     main()
145
146 #chromosome.py
147 PROTOTYPE = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
148
149
150 def adaptationFunc(x):
151     return (math.exp(x) * math.sin(10 * math.pi * x) + 1) / x
152
153
154 class Chromosome:
155     def __init__(self, value: float, r: typing.Tuple[float,
    float], precision: int):
156         self.value = value
157         self.range = r
158         self.precision = precision
159
160     @classmethod
161     def from_value(cls, chromosome, value):
162         return cls(value, chromosome.range, chromosome.
    precision)
163
164     @staticmethod
165     def bin2dec(binary):
166         bin_temp_string = "".join(list(map(str, binary)))
167         decimal = int(bin_temp_string, 2)
168         return decimal
169
170     @staticmethod
171     def crossover(c1, c2):
172         p = random.randint(0, len(c1.binary) - 1)
173         child1_bin = c1.binary[:p] + c2.binary[p:]
174         child2_bin = c2.binary[:p] + c1.binary[p:]
175
176         return (
177             Chromosome.from_value(
178                 c1,
179                 map_range(
180                     Chromosome.bin2dec(child1_bin),
181                     0,
182                     (c1.range[1] - c1.range[0]) * (10 ** c1.
    precision),
```

```
                    c1.range[0],
                    c1.range[1],
                ),
            ),
            Chromosome.from_value(
                c2,
                map_range(
                    Chromosome.bin2dec(child2_bin),
                    0,
                    (c2.range[1] - c2.range[0]) * (10 ** c2.precision),
                    c2.range[0],
                    c2.range[1],
                ),
            ),
        )
        print(child1_bin)

    @property
    def adapted(self) -> float:
        return adaptationFunc(self.value)

    @property
    def decimal(self) -> int:
        return int(
            map_range(
                self.value,
                self.range[0],
                self.range[1],
                0,
                (self.range[1] - self.range[0]) * (10 ** self.precision),
            )
        )

    def mutate(self):
        p = random.randint(0, len(self.binary) - 1)
        temp = self.binary
        temp[p] = 1 if self.binary[p] is 0 else 0

        self.value = map_range(
            Chromosome.bin2dec(temp),
            0,
            (self.range[1] - self.range[0]) * (10 ** self.precision),
            self.range[0],
            self.range[1],
        )
        return self
```

```
229
230     @property
231     def binary(self) -> typing.List[int]:
232         return [
233             sum(x)
234             for x in zip_longest(
235                 list(map(int, list(bin(self.decimal)[2:])))
    [::-1],
236                 PROTOTYPE,
237                 fillvalue=0,
238             )
239         ][::-1]
240
241
242 def map_range(s: float, a1: float, a2: float, b1: float, b2:
    float) -> float:
243     return b1 + ((s - a1) * (b2 - b1) / (a2 - a1))
```