

# Protokol k semestrální práci z BI-ZUM

FIT ČVUT, LS 2019/2020

Jméno studenta: David Omraai

Username: omraida

Název semestrální práce: Aproximace Sinu pomocí genetického algoritmu

## OSNOVA

1. Zadání semestrální práce.
2. Stručný rozbor, analýza problému/ zadání.
3. Výběr metody.
4. Popis aplikace metody na daný problém.
5. Implementace.
6. Reference - odkaz na literaturu/ zdroj, ze kterého čerpáte. Tento bod do protokolu zahrňte pouze v případě, že se rozhodnete v rámci semestrální práce implementovat algoritmus/ metodu nad rámec přednášek z BI-ZUM nebo při tvorbě semestrální práce čerpáte z doporučené literatury.

## Zpracování

### 1. Účetní počítá sinus

1. Uvažujme účetní, která chce spočítat sinus. Ponechme stranou, proč by tak měla činit, ale může se třeba vzdělávat v trigonometrii (během pracovní doby). Má ovšem jen účetnickou kalkulačku s číslicemi a tlačítky „+“, „-“, „\*“, „/“ a nějakými dalšími, přičemž „+“ je třikrát větší než ostatní tlačítka. Velký plus asi nepomůže a celkově to vypadá jako dost prekerní situace, váš úkol je pomoci. Pokuste se tedy najít výraz, který bude approximovat funkci sinus alespoň na intervalu (-10pi +10pi) s co nejvyšší přesností (techniku měření zvolte a zdůvodněte)

### 2. Stručný rozbor, analýza problému/zadání

#### 1. Rozbor

1. Účetní nemůže využít žádné jiné operace než ty, které jsou povoleny. Tedy operacemi jsou sčítání, odčítání, násobení a dělení. Násobení nám přináší i další možnou operaci o to celočíselné umocnění čísel.
2. V zadání je taktéž zmíněno, že plus je třikrát větší než ostatní tlačítka. Tedy plus je operací, kterou bude účetní nejspíše preferovat.
3. Je vyžadována, co nejvyšší přesnost
4. Je preferován kratší výraz před delším

### 3. Výběr metody

1. Pro řešení tohoto problému jsem použil dvě různé metody používající genetický algoritmus

#### 1. Polynomiální lineární regrese

1. Tuto metodu jsem zamýšlel implementovat od samého začátku. Líbila se mi myšlenka zkombinovat genetický algoritmus a polynomiální regresi. Jako geny chromozomu jsou voleny koeficienty jednotlivých polynomů. Stupeň použitého polynomu určuje délku chromozomu. U této implementace se snažím pro každý stupeň nalézt tu nejlepší approximaci.

#### 2. Náhodná funkce

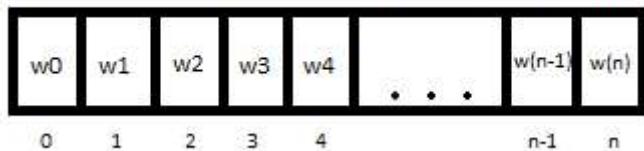
1. U této metody je opět volen genetický algoritmus společně s náhodnými funkcemi. Tyto funkce jsou generovány použitými operacemi uchovávanými v chromozomu. Gen může nabývat jednu ze čtyř hodnot, 1 až 4. V číslech jsou zakódovány operace. Tato metoda ignoruje priority operací.

#### **4. Popis aplikace metody na daný problému**

## 1. Polynomiální lineární regrese

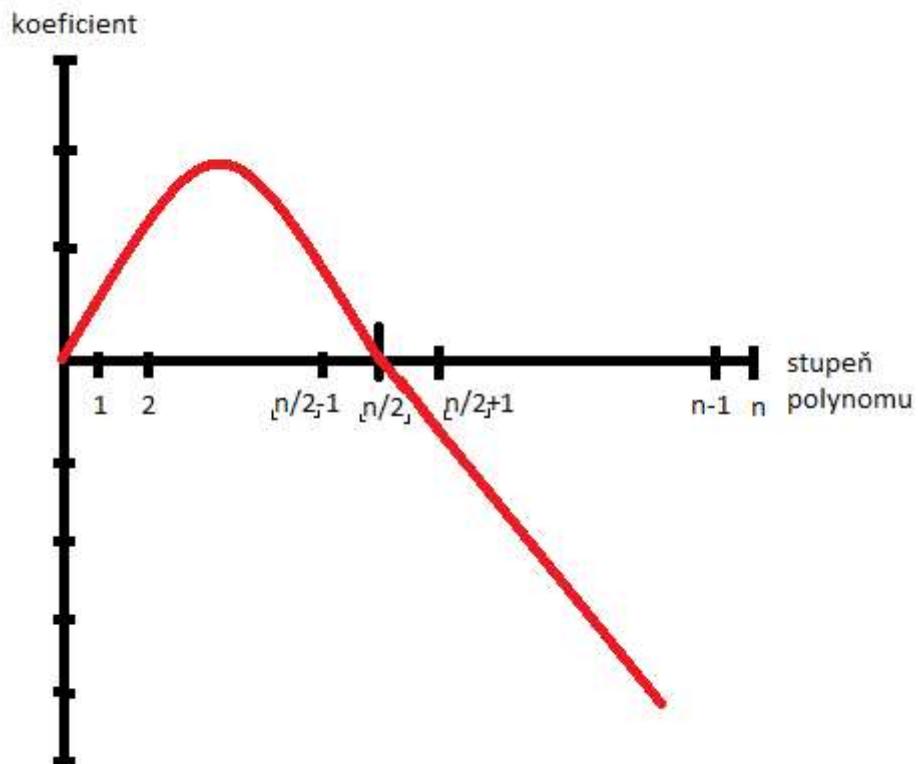
- U této metody je potřeba zjistit koeficienty polynomu, který bude sinus approximovat. Ty se s použitím genetického algoritmu zakódují do chromozomu jedinců z populace. Na následujícím obrázku lze vidět chromozom mající v genech koeficienty polynomu.

$$P(x) = w_0x^0 + w_1x^1 + w_2x^2 \dots w_{(n-1)}x^{(n-1)} + w_nx^{(n)}$$



### Obrázek 1: Chromozom

2. Při nastavování genů chromozomu lze využít generátor náhodných čísel. Avšak nejde o optimální řešení, neboť stavový prostor, který se bude pro koeficienty prohledávat nemusí nutně obsahovat požadované řešení, nebo jeho nalezení v přijatelné době je nepravděpodobné. Proto přicházíme s náhodným generátorem, který bere v potaz stupně polynomů a dle toho nastavuje jejich hodnotu. Při zkoumání podoby řešení polynomiální regrese jsem si všiml, že první polovina koeficientů, pro nižší stupeň polynomů, je větší než druhá. Taktéž, že v první polovině se postupně koeficienty do poloviny zvětšují a po polovině postupně klesají. Tuto informaci tedy používám v generátoru pro čísla. Koeficient pro daný stupeň polynomu je generován z okolí.



Graf 38: Hodnota koeficientů pro stupně polynomů

### **3. Použité operátory**

#### **1. Mutace**

1. Pro zajištění malé změny v genetickém kódu jedince je použita tato mutace. Předchozí hodnota jednotlivých genů chromozomu jedince je sečtena s náhodným číslem z generátoru, popsaného výše. Výsledek je následně vydělen dvěma, aby se zajistila právě ona menší změna.

#### **2. Křížení**

1. Křížení dvou potomků populace je zařízeno pomocí náhodného jednobodového rozdělení chromozomu. V raných fázích implementace tohoto operátoru bylo používáno uniformní, ale to nepřinášelo požadované zlepšení, proto bylo nahrazeno.

#### **3. Výběr**

1. Pro tento projekt je volen turnajový operátor. Náhodně se vybere N jedinců z rozmezí  $\frac{1}{4}$  až  $\frac{1}{3}$ .

### **4. Fitness funkce**

1. Funkce reprezentuje průměrnou chybu mezi testovanými daty. Na obrázku  $I(x)$  značí hodnotu polynomu jedince na daný bod a  $y$  hodnotu sinu na daný bod. Funkce iteruje přes body funkce. Pro zajištění, co největší přesnosti se fitness vypočítává z cca. sta bodů  $x$  na intervalu  $-10\pi$  až  $+10\pi$ .

$$f(I) = \left( \sum_{(x,y) \in V'} (I(x) - y)^2 \right) / |V'| .$$

Obrázek 2: Fitness funkce

## **2. Náhodná funkce**

1. Tato metoda je inspirována zadáním této úlohy. Účetní má kalkulačku a pomocí binárních operací chce dospět k co nejjednodušší funkci jako je sinus. Jelikož se mi zalíbil genetický algoritmus, tak jsem jej chtěl aplikovat i na tuto metodu. Náhodné funkce by asi bylo lepší reprezentovat náhodnými stromy a ty pak používat u genetického programování. Nicméně toto byl větší oříšek. Pro vytvoření podoby chromozomu vycházíme z povolených operací. Tedy geny chromozomu mohou nabývat čtyř hodnot. Následně podoba funkce, které reprezentují je následující. (((... $(x)+x)*x)/x...)).$  do této podoby funkce se postupně vkládají operace z chromozomu. Aby dokázal ga nalézt nejkratší nejlepší řešení, je brána velikost chromozomu v potaz. Projevuje se společně s velikostí chyby ve fitness jedince.
2. Pro lepší výsledky tedy umožňuji proměnnou velikost chromozomu. Délka je pouze omezována shora uživatelem, který na vstupu zadá maximální velikost. Aby nebyly počáteční jedinci příliš malí, tak je omezována zdola polovinou vstupu uživatele.

### **3. Použité operátory**

#### **1. Selekce**

1. Selekcí je opět stejná jako u polynomiální regrese. Je volena turnajová.

#### **2. Křížení**

1. Pro tuto metodu je třeba míti operátor, který může zkracovat a který může prodlužovat chromozom. Křížení zajišťuje prodlužování. Náhodně se zvolí body pro prvního a druhého jedince. První část prvního jedince je ponechána a k tomu je přidána druhá část druhého jedince. Podobně je tomu u druhého jedince. Výsledkem je tedy prodloužený nebo zkrácený chromozom.

### 3. Mutace

1. Chromozom mutace zajišťuje převážně zkracování chromozomů. Vybere se náhodně bod. S padesáti procentní pravděpodobností se vybere zda se zachová přední část, či poslední část. Po zkrácení se s pravděpodobností od uživatele poupraví geny.

### 4. Opravný operátor

1. Jelikož použitou logikou zpracovávání funkce se může stát, že nula vynuluje celý výraz, tak je použit tento opravný operátor. Ten hledá možnou existenci nul a operátory v těchto situacích nahrazuje inverzí.

### 5. Fitness funkce

1. Je použita podobná jako u předchozí metody s tou změnou, že je brána v potaz i velikost chromozomu.

$$\text{Fitness} = \frac{1}{n} \sum_{\text{abs}} (\text{approximated} - \text{actual}) \cdot |\text{Chromosome}|$$

Obrázek 3: Fitness funkce

## 5. Implementace

### 1. Genetický algoritmus

#### 1. Evolution

1. Program po spuštění a získání parametrů od uživatele vytvoří třídu reprezentující evoluci a následně zavolá její spuštěním metodou run.
2. V konstruktoru této třídy se nastavuje velikost populace, počet generací, pravděpodobnost mutace a křížení. Mimo tyto parametry přímo využívané genetickým algoritmem se zde uchovává i třída pro okno s vizuální reprezentací výsledků.

```
def __init__(self, generations, population_size, crossover_prob, mutation_prob, poly, window):  
    self.population_size = population_size  
    self.mutation_prob = mutation_prob  
    self.crossover_prob = crossover_prob  
    self.generations = generations  
    self.window = window  
    self.poly_level = poly
```

Kód 1: Konstruktor třídy evoluce

3. Po zavolání metody run se nastaví potřebné proměnné věci pro přípravu plátna v okně, pro zobrazení vizuální podoby funkcí. Dále se vytvoří funkce sinus na intervalu  $-10\pi$  a  $+10\pi$ . Ke kterému se bude aproximace vygenerována a porovnávat. Vypisování grafu funkce je poměrně nákladné, proto si pamatuji fitness nejlepšího jedince a graf funkce aktualizují při jeho změně. Pro uchování této informace si zde vytvářím proměnnou prev\_fitness.

```
f = Figure(figsize=(5,5))  
a = f.add_subplot(111)  
canvas = FigureCanvasTkAgg(f, master=self.window)  
x = arange(-10*(pi), 10*(pi), 0.1)  
  
sine_y = list()  
for i in x:  
    sine_y.append(sin(i))  
  
prev_fitness = 0
```

Kód 2: Konfigurace pro vizuální výstup

- Reprezentace populace je zajištěna třídou Population, kterou si před spuštěním generací vytvořím

```
#initial population
population = Population(self.population_size, self.poly_level)
```

*Kód 3: Vytvoření populace*

- Populace nyní obsahuje náhodnou množinu jedinců a je možné spustit generace

```
#iterate through generations
for generation in range(0, self.generations):
```

*Kód 4: Iterace generacemi*

- Vytvoří se pole pro nové jedince reprezentováno třídou Numpy, která přidává operace nad polem z knihovny numpy. Do konstruktoru se volají dva parametry, velikost pole a typ prvků v poli
- Jako první se do pole vloží nejlepší z populace, jeho kopie

```
#new population individuals
new_population = NumpyArray(self.population_size, object)
#get best individual
new_population.append(population.getBestIndividual().deepCopy())
```

*Kód 5: Pole nové populace*

- Po získání nejlepšího z populace se v cyklu volá operátor selekce, křížení a mutace. A to do té doby, než je v poli nové populace dostatek jedinců. Metoda selectIndividual se volá s parametrem udávající počet jedinců k vrácení. Na základě pravděpodobnosti pro křížení se bud' jako noví potomci pro novou generaci zvolí rodiči, nebo křížením dvojice jejich dětí. Nad jednotlivými dětmi se následně vykoná mutace.

```
#append individuals to new population
while new_population.getRealSize() != self.population_size:
    #get parents
    parents = population.selectIndividuals(2)
    #get children with prob
    if random.random() <= self.crossover_prob:
        children = parents[0].deepCopy().crossover(parents[1].deepCopy())
    else:
        children = parents
    #mutate first child with prob
    children[0].mutate(self.mutation_prob)
    #compute fitness
    children[0].computeFitness()
    #add first child to population
    new_population.append(children[0])
    #append second child if there is place
    if new_population.getRealSize() < self.population_size:
        children[1].mutate(self.mutation_prob)
        children[1].computeFitness()
        new_population.append(children[1])
```

*Kód 6: Naplnění nové populace*

9. Po dokončení získávání následující generace tito noví jedinci nahradí stávající. Na to slouží metoda třídy Population replacePopulation, která si jako parametr bere pole nové populace.

```
#update population
population.replacePopulation(new_population.getArr())
```

*Kód 7: Nahrazení aktuální populace*

10. Program obsahuje dva vizuální výstupy. Prvním je terminál ze kterého byl program spuštěn. Zde se zobrazuje nejlepší jedinec, jeho funkce a informace o fitness.

```
#best fitness
best_fitness = population.getBestFitness()
#average fitness
avg_fitness = population.getAvgFitness()
#best individual
best_individual = population.getBestIndividual()
print("-----")
print(generation, " generetion")
print("best fitness: ", best_fitness, "| avarge fitness: ", avg_fitness)
print(best_individual.toString())
```

*Kód 8: Vizuální výpis do konzole*

11. Jako druhý vizuální výstup je okno s grafickou reprezentací funkce. Pokud je nový nejlepší fitness jiný než předchozí aktualizuje se graf. Nejdříve se vyčistí pomocí metody clear. Uloží se data pro reprezentaci aktuálně nejlepší approximace. Nastaví se barvy pro jednotlivé funkce. A na závěr se konečně graf vypíše a aktualizuje. A nejlepší fitness se uloží pro další porovnání.

```
if prev_fitness != best_individual.getFitness():
    a.clear()

    y = list()
    for i in x:
        y.append(best_individual.useFunction(i))

    a.set_ylim(-2, 2)

    a.plot(x, sine_y, color="red", label="sine")
    a.plot(x, y, color="green", label="approx")

    a.axhline(y=0, color="black", linestyle=":")
    a.axvline(color="black", linestyle=":")

    #canvas.show()
    canvas.get_tk_widget().grid(column=20, row=20)#side=BOTH)

    canvas.draw_idle()

    self.window.update_idletasks()
    self.window.update()

    prev_fitness = best_individual.getFitness()
```

*Kód 9: Vizuální výstup do grafu*

## 2. Population

1. Populace je reprezentována třídou Population. Ta se vytváří s parametry udávající velikost populace a velikostí polynomu. Pro jednotlivé implementace jedince má velikost polynomu rozdílné významy. Pro regresi je to velikost chromozomu každého jedince. Pro náhodné funkce je to maximální možná velikost chromozomu. V konstruktoru se vybere náhodný počet vybraných jedinců pro turnaj, pro operátor selekce. Následuje náhodné vytvoření populace. Náhodné vytvoření je dáno parametrem True u třídy individual.

```
def __init__(self, population_size, poly_size):  
    self.tournament_count = random.randint(int(population_size/4), int(population_size/3))  
    self.population_size = population_size  
    self.population = np.empty(population_size, dtype=object)  
    self.poly_size = poly_size  
    for i in range(0, population_size):  
        self.population[i] = Individual(True, poly_size)
```

Kód 10: Konstruktor třídy Population

## 2. Operátor selekce

1. Tento operátor implementuje metoda selectIndividuals, která je volaná s počtem jedinců k vrácení.
2. Na začátku se ověří zda požadovaný počet jedinců k navrácení je validní.

```
def selectIndividuals(self, number):  
    """  
    Select number of individuals from population  
    tournament selection  
    """  
    #check if parameters are correct  
    if number > self.tournament_count or number > self.population_size:  
        return None
```

Kód 11: Selekce

3. Vytvoří se pole pro uložení jedinců pro turnaj. Dále se tito jedinci vyberou pomocí náhodné permutace pole. Pro to je vytvořeno nové pole s náhodnou permutací indexů. Prvních number je pak vybráno pro turnaj.

```
#array for tournament  
tournament_individuals = NumpyArray(self.tournament_count, object)  
#get random permutation of population  
perm = np.random.permutation(self.population_size)  
for i in range(0, self.tournament_count):  
    tournament_individuals.append(self.population[perm[i]])
```

Kód 12: Vytvoření pole k turnaji

4. Následuje seřazení pole účastníků turnaje a vybrání nejlepších z nich

```
arr = tournament_individuals.getArr()  
sorted(arr, key=lambda x: x.getFitness())  
#get best elements  
return arr[-number-1:-1]
```

Kód 13: Získání a vrácení výherců turnaje

## 3. Nahrazení populace

1. Metoda pro nahrazení populace vezme pole jedinců a nahradí jej novým polem

```
def replacePopulation(self, new_population):  
    self.population = new_population
```

Kód 14: Nahrazení populace

#### 4. Získání průměrného fitness

1. Metoda getAvgFitness seče fitness všech jedinců a následně jej vydělí počtem jedinců

```
def getAvgFitness(self):  
    avg_fitness = 0  
    num = 0  
    for x in self.population:  
        num+=1  
        avg_fitness+= x.getFitness()  
  
    return avg_fitness/num
```

Kód 15: Výpočet průměrného fitness

#### 5. Získání nejlepšího jedince

1. V metodě getBestIndividual se postupně prohledají všichni jedinci a z nich vybrán a vrácen ten s nejnižší fitness.

```
def getBestIndividual(self):  
    best_individual = self.population[0]  
    for x in self.population:  
        if x.getFitness() < best_individual.getFitness():  
            best_individual = x  
  
    return best_individual
```

Kód 16: Vyhledání nejlepšího jedince

#### 2. **Polynomiální regrese**

1. V konstruktoru třídy implementující polynomiální regresi nad jedincem se vytvoří fitness nastavená na None. Vytvoří se chromozom s geny typu float. Počáteční a koncový bod pro approximaci sinu a v případě, že je rand\_chrom nastaven na True je chromozom naplněn náhodnými hodnotami.

```
def __init__(self, rand_chrom, data_len=3):  
    self.fitness = None  
    self.chromozome = np.empty(data_len, dtype=float)  
    self.chromozome_length = data_len  
  
    self.start_point = -10*pi  
    self.end_point = 10*pi  
  
    if rand_chrom == True:  
        self.randomChromo()
```

Kód 17: Konstruktor třídy Individual

## 2. Generátor náhodného chromozomu

1. Jak jsem uvedl již dříve v textu, náhodné číslo závisí na tom, jaký stupeň polynom má. Pro tuto reprezentaci jsem vytvořil generátor čísel pro první polovinu, kde je využit sinus. A pro druhou polovinu, kde je využita prostá lineární funkce. Náhodně se také zvolí znaménko genu.

```
def randomChromo(self):
    for i in range(0, int(self.chromosome_length/2)):
        neg = 1
        if random.randint(0,1) == 1:
            neg = -1
        self.chromosome[i] = neg*random.uniform(1, 9.9)*(10**(- random.randint(3*sin(pi*i/int(self.chromosome_length/2))-1,
                                                                           3*sin(pi*i/int(self.chromosome_length/2))+1) ))

    for i in range(int(self.chromosome_length/2), self.chromosome_length):
        neg = 1
        if random.randint(0,1) == 1:
            neg = -1
        self.chromosome[i] = neg*random.uniform(1, 9.9)*(10**(-1*( random.randint(i-int(self.chromosome_length/2), i) ) )

    self.computeFitness()
```

Kód 18: Generace náhodného chromozomu

## 2. Převedení chromozomu na funkci

1. K operování s polynomem reprezentovaným chromozomem se používá metoda useFunction. Ta pro vstupní číslo použije informace z chromozomu k získání hodnoty.

```
def useFunction(self, x):
    result = 0
    for i in range(0, self.chromosome_length):
        result+=self.chromosome[i]*(x**i)

    return result
```

Kód 19: Použití funkce z chromozomu

## 3. Výpočet fitness

1. K výpočtu fitness slouží metoda computeFitness. Ta pro daný interval spočítá střední kvadratickou hodnotu.

```
def computeFitness(self):
    self.fitness = 0

    frm_index = self.start_point
    to_index = self.end_point
    im = 0
    N = 0
    for i in np.arange(frm_index, to_index, 0.55):
        N+=1
        self.fitness += (sin(i)-self.useFunction(i))**2

    self.fitness /= (N)
```

Kód 20: Výpočet fitness

## 4. Křížení

1. Křížení je implementováno pomocí jednobodového rozdělení chromozomu. Náhodně se vybere která část chromozomu se vymění. Stejně tak se náhodně zvolí bod ke kterému se bude tato část mezi dvěma jedinci měnit.

## 2. Mutace

1. Operátor mutace postupně prochází chromozom a s danou pravděpodobností mění geny. K hodnotě v genu se přičte nová a výsledek se vydělí dvěma

```
def mutate(self, mut_prob):  
    for i in range(0, int(self.chromosome_length/2)):  
        #tmp = 0.5**i  
        if mut_prob < random.random():  
            neg = 1  
            if random.randint(0,1) == 1:  
                neg = -1  
            self.chromosome[i] = (neg*random.uniform(1, 9.9)*(10**(* random.randint( int(3*sin(pi*i/int(self.chromosome_length/2))-1) ,  
                                            int(3*sin(pi*i/int(self.chromosome_length/2))+1) ))/1000  
                                            + self.chromosome[i] )/2  
  
    for i in range(int(self.chromosome_length/2), self.chromosome_length):  
        if mut_prob < random.random():  
            neg = 1  
            if random.randint(0,1) == 1:  
                neg = -1  
            self.chromosome[i] = (neg*random.uniform(1, 9.9)*(10**(* -1*( random.randint(i-int(self.chromosome_length/2), i) ))/1000  
                                            +self.chromosome[i])/2
```

Kód 21: Mutace

## 5. Opravný operátor

1. Tento operátor se volá na nejlepšího potomka z populace. Operátor porovná jestli změna znamínek genů nezlepší fitness.

```
def repairOperator(self):  
    plus_first = Individual(False, self.chromosome_length)  
    minus_first = Individual(False, self.chromosome_length)  
    for i in range(0, self.chromosome_length):  
        if i % 2 == 0:  
            plus_first.chromosome[i] = +abs(self.chromosome[i])  
            minus_first.chromosome[i] = -abs(self.chromosome[i])  
        else:  
            plus_first.chromosome[i] = -abs(self.chromosome[i])  
            minus_first.chromosome[i] = +abs(self.chromosome[i])  
    plus_first.computeFitness()  
    minus_first.computeFitness()  
    if plus_first.getFitness() < minus_first.getFitness():  
        if plus_first.getFitness() < self.fitness:  
            self.chromosome = plus_first.chromosome  
            self.fitness = plus_first.getFitness()  
    else:  
        if minus_first.getFitness() < self.fitness:  
            self.chromosome = minus_first.chromosome  
            self.fitness = minus_first.getFitness()
```

Kód 22: Opravný operátor

### 3. Náhodná funkce

1. V konstruktoru třídy pro jedince z populace této metody se skrývá nastavení fitness hodnoty. Zprvu je nastavena na None. Průměrná chyba mse, taktéž nastavena na None. Vytvoří se chromozom, jež je prezentován třídou ChromosomeTree. Dále se nastavuje rozsah který tato metoda approximuje.

```
def __init__(self, rand_chrom, data_len=3):
    self.fitness = None
    self.mse = None
    if rand_chrom == False:
        self.chromosome_length = data_len#data_len
    else:
        self.chromosome_length = data_len#random.randint(int(data_len/2))
    self.chromosome = ChromosomeTree(self.chromosome_length, rand_chrom)

    self.start_point = -10*pi
    self.end_point = 10*pi

    if rand_chrom == True:
        self.computeFitness()
        self.repairOperator()
```

Kód 23: Konstruktor jedince

2. Chromozom je reprezentován třídou ChromosomeTree, který si v sobě uchovává chromozom, a k tomu operace nad tímto chromozomem. Třída je připravena na možné rozšíření náhradou neznámých x ve výrazu konstantami. Avšak pro mou implementaci je to nevyužito. Číslo -3 v self.parameters představuje x.

```
def __init__(self, length, random_tree):
    self.length = length
    self.chromosome = np.empty(length, dtype=int)
    # random number from range -1 to 1, -3 represent 'x'
    self.parameters = np.full(shape=(length+1), fill_value=-3,dtype=float)

    if random_tree == True:
        self.generateChromosome()
```

Kód 24: Konstruktor třídy ChromosomeTree

1. Pro generování náhodného chromozomu je použita metoda třídy ChromosomeTree generateChromosome.

```
def generateChromosome(self):
    for i in range(0, self.length):
        self.chromosome[i] = random.randint(1, 4)
        self.parameters[i] = -3
    self.parameters[self.length] = -3
```

Kód 25: Generace náhodného chromozomu

2. Pro výpočet hodnoty pro daný vstup funkce je použita metoda countValue. Dle toho jakou hodnotu má se vykoná příslušná operace.

```
def countValue(self, x): #num, i, x):
    result = x
    for i in range(0, self.length):
        if self.chromozome[i] == 1:#+
            result+=x
        elif self.chromozome[i] == 2:#-
            result-=x
        elif self.chromozome[i] == 3:#*
            result*=x
        elif self.chromozome[i] == 4:#/
            if x != 0:
                result/=x

    return result
```

Kód 26: Výpočet funkce z chromozomu

3. Výpočet fitness se provede metodou computeFitness, která spočítá průměrnou chybu a to vynásobí délkou chromozomu

```
def computeFitness(self):
    self.fitness = 0

    frm_index = self.start_point
    to_index = self.end_point
    im = 0
    N = 0
    for i in np.arange(frm_index, to_index, 0.55):
        N+=1
        self.fitness += abs(sin(i)-self.useFunction(i))

    self.fitness /= (N)
    self.fitness *= self.chromozome_length
```

Kód 27: Výpočet fitness

#### 4. Mutace

1. Jak je popsáno výše, tento operátor zkracuje chromozom a navíc mění s danou pravděpodobností od uživatele geny. Po úpravě chromozomu je třeba jej opravit.

```
def mutate(self, mut_prob):  
    #remove gene with prob  
    if (mut_prob < random.random()) and self.chromozome.length >=64:  
        random_len = random.randint(int(3*self.chromozome.length/4), self.chromozome.length-2)  
        new_chromozome = NPArray(random_len, int)  
        new_param = NPArray(random_len+1, float)  
  
        if random.random() < 0.5:  
            #from end  
            for i in range(0, random_len):  
                new_chromozome.append(self.chromozome.chromozome[i])  
                new_param.append(self.chromozome.parameters[i])  
                new_param.append(self.chromozome.parameters[random_len])  
        else:  
            for i in range(self.chromozome.length-random_len, self.chromozome.length):  
                new_chromozome.append(self.chromozome.chromozome[i])  
                new_param.append(self.chromozome.parameters[i])  
                new_param.append(self.chromozome.parameters[self.chromozome.length])  
  
        self.chromozome.chromozome = new_chromozome.getArr()  
        self.chromozome.parameters = new_param.getArr()  
        self.chromozome.length = random_len  
        self.chromozome_length = random_len  
  
    #change op with prob  
    for i in range(0, self.chromozome.length):  
        if random.random() < mut_prob:  
            self.chromozome.chromozome[i] = random.randint(1, 4)  
  
    self.repairOperator()
```

Kód 28: Operátor mutace

#### 2. Křížení

1. Nejdříve se vytvoří dvojice náhodných bodů, podle kterých se budou dělit chromozomy.

```
#make new random points in both array  
first_random = random.randint(1, self.chromozome_length-1)  
second_random = random.randint(1, individual.chromozome_length-1)
```

Kód 29: Generace dvou náhodných bodů

2. Následně se připraví délky výsledných chromozomů jedinců. Noví jedinci a pole ve kterém se budou během zpracovávání uchovávat geny pro chromozomy nově vzniklých jedinců.

```
#calculate length of both arrays  
first_len = first_random + (individual.chromozome_length - second_random)  
second_len = second_random + (self.chromozome_length - first_random)  
#make arrays  
first_array = NPArray(first_len, int)  
second_array = NPArray(second_len, int)  
#make new individuals  
first_one = IndividualRandomFunction(False, first_len)  
second_one = IndividualRandomFunction(False, second_len)
```

Kód 30: Vytvoření dvou nových jedinců

3. Nyní proběhne získání požadovaných genů z rodičů. Po získání se uloží jednotlivým jedincům a proběhne oprava nově vzniklých jedinců.

```
#change chromosomes first part
for i in range(0, first_random):
    first_array.append(self.chromosome.chromosome[i])
for i in range(0, second_random):
    second_array.append(individual.chromosome.chromosome[i])
#change chromosomes second part
for i in range(second_random, individual.chromosome_length):
    first_array.append(individual.chromosome.chromosome[i])
for i in range(first_random, self.chromosome_length):
    second_array.append(self.chromosome.chromosome[i])
#get arrays
first_one.chromosome.chromosome = first_array.getArr()
second_one.chromosome.chromosome = second_array.getArr()

first_one.repairOperator()
second_one.repairOperator()
```

Kód 31: Získání chromozomů nových jedinců

4. Nakonec se vrátí nově vzniklé jedinci

```
return np.array([first_one, second_one])
```

Kód 32: Navrácení dvou jedinců

3. Opravný operátoru

1. U opravy se prohledává výraz, zda-li neobsahuje nulu, které by výraz ovlivnila. Postupně se tedy hledá místo, kde by mohla vzniknout a v tomto místě se změní operace na opačnou.

```
def repairOperator(self):
    result = 1
    for i in range(0, self.chromosome.length):
        if self.chromosome.chromosome[i] == 1:#+
            if result == -1:
                self.chromosome.chromosome[i] = 2
                result+=1
        elif self.chromosome.chromosome[i] == 2:#-
            if result==1:
                self.chromosome.chromosome[i] = 1
                result+=1
```

Kód 33: Opravný operátor

## **6. Reference**

1. Baqais, Abdulrahman. (2016). Genetic Algorithm for Function Approximation : An Experimental Investigation. International Journal of Artificial Intelligence & Applications. 7. 01-09. 10.5121/ijaia.2016.7301.
2. Genetic Programming Theory and Practice II - Google Books. Knihy Google [online]. Dostupné z: [https://books.google.cz/books?id=H1qLx3CCwpwC&pg=PA183&lpg=PA183&dq=genetic+function+approximation+size&source=bl&ots=gULCnvY2TU&sig=ACfU3U0IrMfGtYnmu07Nl8fLchhMQ5-d2g&hl=en&sa=X&ved=2ahUKEwjYzKqO6qTpAhWKY8AKHTS\\_As4Q6AEwCXoECAgAQ#v=onepage&q&f=false](https://books.google.cz/books?id=H1qLx3CCwpwC&pg=PA183&lpg=PA183&dq=genetic+function+approximation+size&source=bl&ots=gULCnvY2TU&sig=ACfU3U0IrMfGtYnmu07Nl8fLchhMQ5-d2g&hl=en&sa=X&ved=2ahUKEwjYzKqO6qTpAhWKY8AKHTS_As4Q6AEwCXoECAgAQ#v=onepage&q&f=false)