Project Report for Weierstrass function approximation with genetic algorithm

Clovis Carlier TD-P

Introduction

For this project, our goal was to approach the triplet (a, b, c) of the following Weierstrass function:

$$t(i) = \sum_{n=1}^{c} a^n cos(b^n \pi i)$$

With the following constraints:

$$A = \{a \in \mathbb{R} \mid a \in]0, 1[\}$$
$$B = \{b \in \mathbb{N} \mid b \in [1, 20]\}$$
$$C = \{c \in \mathbb{N} \mid c \in [1, 20]\}$$

Let's first discuss the scope and goals the algorithm by answering the mandatory questions.

1. What is the size of the researchable space for elements of the genetic algorithm?

To answer this question, let's first define an element: an element in the scope of this report and in the python script will be considered as a 3-dimension vector (a, b, c) (to lighten the notation, we will call the "Initialisation vector" that we are looking for, IV or iv during this report). Thus the size for our vector is the composition of the 3 spaces available for our 3 variables, but since one of our variable takes its values in $\mathbb{R}$, python will round this number to a 16 significant figures after the decimal point. So our size is: $A * B * C$ so $\mathbb{R}$ in theory, but only $1.10^{16} * 20 * 20$ in our python algorithm, so 4.10^{18} possible IVs.

2. What is our fitness function?

Our fitness function is based on the samples we have for the exercise, we have 20 sample values of i and $t(i)$ that have been generated with the secret IV that we are looking for. Our fitness function is simply the sum of the absolute value of the differences between our calculated $t(i)$ and the correct $t(i)$ as seen below.

```
#This code is within the scope of the initialisation vector class
def Weierstrass(self,i):
    sum=0
    for n in range(self.c+1):
        sum+=pow(self.a,n)*cos(pow(self.b,n)*pi*i)
    return sum

def fitness(self,samples):  #The samples are loaded into a dictionnary where i
are the keys and the equivalent t the values
    score=0
    for i,t in samples.items():
        estimation=self.Weierstrass(i)   # Calculation of our estimate of t(i)
        score+=abs(estimation-t)
    return score #The total score is the sum of all the absolute value of the
differences
```

3. What are the implementations of mutations and cross-overs?

Firstly, we will here use the term "parent" as an element part of the population in generation n that will create (alone in the case of a mutation or with another parent in the case of a cross-over) a "child" element for it to become an element in generation $n+1$. At the end of a generation, the population and the parents within it are discarded and the child array of elements becomes the current population.

Mutations and crossovers are both implemented in the program, cross-overs choose randomly 2 parents in the most fit part of the population (most of my results were done by choosing the parents population size to be a tenth of the full population), but since mutations are here to give major changes to an element, we apply then to the whole population.

Cross-overs

```
def cross_over(pop_parents):
    #We choose 2 random parents in the most fit elements of the pop
    mum,dad = pop_parents[randint(0, len(pop_parents)-1)],
pop_parents[randint(0, len(pop_parents)-1)]
    child=iv((mum.a+dad.a)/2,round((mum.b+dad.b)/2),round((mum.c+dad.c)/2))
    #For continuous values (since mutations will only be discretes)
    return child
```

As shown in the code above, the 2 parents for a cross-over are chosen at random in the `pop_parent` array, which is the most fit part of the current population. In this cross-over, the main goal is to search for the secret IV's a value in a dichotomic manner. Indeed, our mutations being largely discrete, in the sense that mutations occurring on the a value are addition or subtraction of small percentages of a, thus rough and inaccurate. Using dichotomy here will yield a correct a much faster than hoping for mutations to do the job and simply cross-over creating the child IV by crossing it's parents parameters (such as `child=iv(mum.a, dad.b, mum.c)` for example).

The fact that there is a probability `1/len(pop_parents)` (especially with small populations) that both parents are the same element does not matter, keeping `1/len(pop_parents)*100` % of parents with no change applied is completely without consequence, since we already keep all the most fit parents anyway.

Mutations

```
def mutation(pop, gen, samples):
    while True:
        parent=pop[randint(0, len(pop)-1)]
        child=iv(uniform(0, 1),randint(1, 20),randint(1, 20)) #Completly
randomized new point
        if parent.fitness(samples)<0.736:
            child=iv(parent.a+
((randint(0,2)-1)*parent.a/(gen**2)),parent.b,parent.c)
        if child.fit: #Check for correct a,b,c
            return child
```

My implementation of mutations is based on the fact that I'm not looking for efficiency, I'm looking for the best a possible. It was earlier concluded that a fitness of around 0.735 will be the next we can do. But I wanted to search for the best one! So whenever the global fitness went down under 0.736, I decided that choosing a random new point for the mutation was useless. Instead, a (since it's the only parameter that really matters here, b and c will always be 15 and 2 in the case of an IV fitness that low) will now take values that differs ever so slightly, with the following formula:

$$a = a * (1 + k/gen^2) \text{ with } k \in [-1, 0, 1]$$

4. What is the selection process?

The selection as discussed above chooses parents with the following function:

```
def parents(pop,n_parents,samples):
    sorted_pop=sorted(pop, key=lambda iv: iv.fitness(samples))
    print("Best parent fitness=",sorted_pop[0].fitness(samples),sorted_pop[0])
    #print(pop_dict,pop_dict.keys())
    #We only take a certain amount of the best parents
    return [sorted_pop[i] for i in range(len(sorted_pop)) if i<n_parents]
```

It returns only a precise amount of parents, the one who are the most fit in the eyes of the fitness function.

All of these parents are automatically added to the children population with the `childs=pop_parents`. The child population is then filled with around 50% of mutated random individuals of the population, and 50% of the most parents' cross overs. Meaning for a population of 100 and a parent size of 10%, the $n+1$ population will contains the 10 most fit parent, and around 45 (90*50%) mutants and 45 cross-overs.

The choice of 10% parent size and 50%/50% of mutant/cross-overs ratio are very disputable, increasing a lot the 10% for the parent size and we would inevitably fall into a local minima whenever a few good first parents would pop up, and more it would give to much disparity and never really give a value to the cross-overs. The 50/50 is also critical but disputable, these are simply the value that worked best for me for this dataset, however I feel like giving less

importance to the mutations would make it hard to climb up or even fell into another local minima.

5. What are the global parameters? (generations count, population count...)

This question is tricky, because it was asked with a "before you converge to a stable solution", and since it is insanely hard to converge to the right solution, we will as we mentioned earlier accept any solution converging with a fitness below 0.8. For this we will use a few different population: 100, 200, 500, 1000 and 10000. We will also stopwatch the average time that it take of each.

The result is as expected: generations decrease while time increase when we increase the initial population count.

Gen	Times	Gen mean	Time mean	Gen	Times	Gen mean	Time mean	Gen	Times	Gen mean	Time mean	Gen	Times	Gen mean	Time mean	Gen	Times	Gen mean	Time mean
208	5.579082966	56.85201702	1.680862135	33	1.75360775	30.17063753	1.78215671	4	0.680831432	17.86396881	2.763731088	6	2.042362	8.136671484	2.670914395	2	6.309054613	1.379514305	4.447548207
96	2.854609489			9	0.57952714	Population=200		4	0.670610428	Population=500		7	2.431713	Population=1000		1	3.206429958		
2	0.079071999			53	3.492150545			9	1.504257917			2	0.654595			3	9.66788888		
101	2.883380651			21	1.239126921			18	2.685947657			12	3.976132			2	6.457775593		
105	3.112341404			34	1.91432786			11	1.835669518			13	4.313436			2	6.461611986		
25	0.757970572			55	3.474933147			55	7.666917801			1	0.329299			1	3.234900475		
70	1.957780361			52	3.118837357			35	5.933418036			18	5.758667			2	6.411334038		
53	1.571429729			75	4.08406949			11	1.7295928			52	15.06567			1	3.154008389		
79	2.685442209			41	2.329129219			4	0.639086246			7	2.409201			1	3.244952202		
5	0.161146402			17	1.000910282			7	1.204095364			4	1.316713			1	3.296194077		
8	0.266242027			6	0.417885065			71	9.090635777			19	6.63171			1	3.221866846		
158	3.780364037			29	1.510435581			57	8.502010822			16	5.134767			3	9.775563955		
124	3.826969862			21	1.139069796			2	0.32229352			21	6.778501			1	3.216926098		
92	3.116352558			22	1.231195688			52	7.12066412			15	5.20276			3	9.717810392		
12	0.437903881			6	0.40715909			27	4.020779133			2	0.682186			2	6.468784094		
141	3.265137434			33	1.724543571			23	3.296685696			22	6.847215			1	3.214441299		
11	0.40937233			45	2.725111008			5	0.840764523			15	4.765013			1	3.164751053		
84	2.244757175			7	0.394358873			21	3.360907078			6	2.055603			3	9.726052761		
167	4.112406492			174	9.315826178			14	2.00333643			5	1.666448			1	3.176161528		
102	2.697395563			56	3.084596157			45	6.860315084			3	1.073396			3	9.517877579		
99	3.172453642			87	5.625904322			9	1.481347084			7	2.351138			1	3.233941555		
217	5.967427731			21	1.241128683			41	6.463694572			26	8.607151			1	3.272754908		
41	1.102507114			6	0.402376413			17	2.546560526			14	4.654003			1	3.257962704		
16	0.543494463			86	5.7402215			25	3.689969301			11	3.667252			1	3.236684799		
53	1.478953838			30	1.748749256			15	2.421202183			58	15.56587			1	3.223423958		
51	1.29006958			66	3.914592743			29	4.66770649			1	0.336306			1	3.202917337		
176	4.469074965			86	4.91623497			39	6.00039196			24	7.589051			1	3.235259056		
144	4.911238909			42	2.623369694			37	5.256476641			3	0.986414			1	3.239555902		
109	2.776748419			49	2.731989861			35	5.218586206			9	2.892982			2	6.447826624		
70	2.216670036			9	0.520473242			36	5.608910084			1	0.324295			1	3.278403282		

6. Discuss different solutions and parameters

Basically, I've design my algorithm to be precise, not efficient. This is highly supported by the fact that I'm doing a dichotomy on a and not trying to change b and c whenever I go near the solution in mutations.

The question is, what is the solution? The solution was probably chosen by hand so we can expect 2 decimal places only, taking into account the noise, it is perhaps around $(0.35, 15.2)$ or nowhere near it, depending on the noise intensity. However, let's not take into account the noise, indeed, we won't be able to reach a fitness of 0, it is impossible with noise, but how do we find the closest IV to match the sample, how do we find the lowest fitness?

Well, dichotomy. I basically brute-forced with another dichotomy program the best a, so I know where I should go from here. But after doing all I could, which consisted in implementing the dichotomy in the cross overs and adding a condition to jump out of local minima in the mutations part by changing a small percentage of a and taking into account the generation, I ran it a few times and it would never find the number I assumed to be the best in less then a not acceptable period of time (a few minutes). The best possible fitness I calculated was 0.7351051480069459 and often got within 10^{-11} approximation (which is really nice considering our a are 10^{-16}) in less then a minute for a pop=1000. I assume my mutation is highly disputable and could be improved with using logs and power of gen instead of gen^2 but it didn't take the time to do it.