MSE 426:

Introduction to Engineering Design Optimization

Lab 3

Introduction to Genetic Algorithms

Selikem Kwadzovia | 301303898 | 

Sepehr Rezvani | 301291960 | 

Tuesday, February 16, 2021

Table of Contents

Introduction

Genetic algorithms (GA) in MATLAB solve smooth (problem functions that are at least twice differentiable) or non-smooth optimization problems regardless of their constraints. In this lab, GA Toolbox was used to find the global optimum, number of iterations and number of function evaluations for four different questions. GA uses three rules in each step to find the optimal solution: Selection Rules, Crossover Rules, and Mutation Rules, that includes selecting parents, combining two parents, and applying random changes to individual parents to form children, respectively. In order to use GA for optimization problems, first, Fitness Function must be defined, similar to objective function that was created in Lab 2.

Results and Analysis

Q1)

Problem of discrete variables: compound Gear Train (GT) design problem

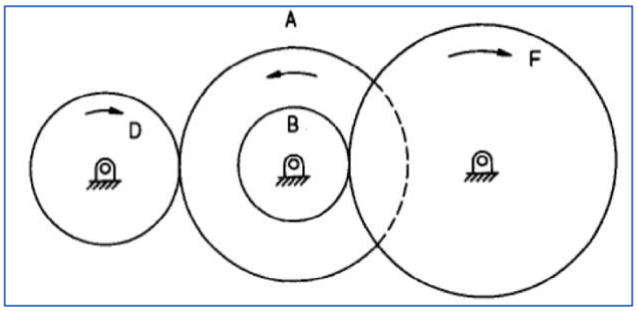

Figure 1 – Compound Gear Train

Four variables x1, x2, x3, and x4 are introduced, each representing number of teeth on gear D, B, A, and F, respectively.

Goal: Gear ratio as close as possible to $\frac{1}{6.931}$.

Constraint: Number of teeth for all gears in ranging from 14-60.

Result: For [x1, x2, x3, x4], optimum teeth number is **[15.0000, 26.0000, 53.0000, 51.0000] with function value 0**, for gears D, B, A, F, respectively.

Note: The optimum differs each time the program is run and gives 4 different values each time (see Detailed Results section for more examples). However, function value is always the same.

Q2)

A 16x16 matrix (a.mat) was created for simplicity as shown in Figure 2.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	0	0	1	0	0	1	1	0	0	0	0	0	0	0	1
2	0	1	1	0	0	0	1	0	0	1	0	0	0	0	0	0
3	0	0	1	0	0	0	1	0	1	1	0	0	0	1	0	0
4	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0
5	0	0	0	0	1	1	0	0	0	1	0	1	0	0	0	1
6	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0
7	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0
8	0	0	0	0	0	0	0	1	0	1	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0
11	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Figure 2 – Matrix a

Goal: Optimize 16 variable fitness function.

Constraint: Variable range is: [-1,1].

Result: Optimal was found at**: [-0.4999 -0.5000 -0.5000 -0.4992 -0.5000 -0.5000 -0.4999 -0.5001 -0.4994 -0.5001 -0.5001 -0.5005 -0.4999 -0.4998 -0.4997 -0.5001] with function value 25.8750.**

Q3)

Goal: Optimize multi-objective functions.
Constraint: none.
Result: Optimal was found at: **[0.7427,0.0853,0.2670] with function value 0.9430.**

Q4)

Goal: Optimize multi-objective functions.
Constraint: constraints are embedded in "func1.m" → Refer to: Appendices, Code, Sub-functions, Fitness Function – "func1.m".
Result: Optimal was found at: **[0.0815 0.9942 0.9883 0.9910 0.9844 0.9806 0.9815 0.9897 0.9777 0.9848] with function value 0.2808.**

Conclusion

This lab showed the power of genetic algorithms through Matlab. Optima were found for various population sizes, crossover, and mutation rates. The results showed that increasing the population size produced lower minima, due to the function approaching the global minimum. The detailed results in the appendix also show that the crossover and mutation rates tested did not significantly impact the end result. With more practice, we will gain greater experience in choosing the genetic algorithms settings.

Bibliography

[1] G. Wang, “Introduction to Optimum Design: MSE 426/726 Lab Manual.” Canvas.sfu.ca.

Appendix

Detailed Results

To find number of iterations and number of function evaluations, use the following codes respectively:

- `iterations= output.generations`
- `function_eval = output.funccount`

Q1)

The following table includes results after 10 random runs:

Run	Optimum Point(s) → 4 variables	Function Value	# of Iterations	# of func.eval
1	[26.0000,15.0000,53.0000,51.0000]	0.0000	51	1981
2	[19.0000,16.0000,49.0000,43.0000]	0.0000	57	2209
3	[22.0000,17.0000,54.0000,48.0000]	0.0000	54	2095
4	[14.0000,17.0000,30.0000,55.0000]	0.0000	51	1981
5	[18.0000,20.0000,58.0000,43.0000]	0.0000	52	2019
6	[15.0000,24.0000,52.0000,48.0000]	0.0000	54	2095
7	[26.0000,15.0000,51.0000,53.0000]	0.0000	57	2209
8	[22.0000,17.0000,48.0000,54.0000]	0.0000	53	2057
9	[23.0000,20.0000,55.0000,58.0000]	0.0000	51	1981
10	[23.0000,16.0000,50.0000,51.0000]	0.0000	53	2057

Analysis of results:

- It appears after multiple runs, results are consistent. Optimum points are number of teeth for gear D, B, A, and F, respectively. Note that all these points satisfy question's constraints, therefore, either of these 10 combinations can be used. Notice that for each run, the order from highest to lowest number of teeth are:x3, x4, x1, x2, which is consistent with their equivalent sizes shown in Figure 1 (gears A, F, D, B).

Q2)

The following table includes results after 5 random runs:

Run	Optimum Point(s) → 16 variables	Function Value	# of Iterations	# of func.eval
1	-0.4999 -0.5000 -0.5000 -0.4992 -0.5000 -0.5000 -0.4999 -0.5001 -0.4994 -0.5001 -0.5001 -0.5005 -0.4999 -0.4998 -0.4997 -0.5001	25.8750	5	49700
2	-0.5000 -0.4994 -0.5003 -0.4997 -0.5000 -0.5002 -0.5000 -0.4998 -0.4994 -0.5000 -0.4990 -0.5002 -0.5001 -0.4999 -0.4998 -0.4996	25.8750	5	49700
3	-0.5000 -0.5000 -0.5000 -0.5000 -0.4998 -0.5000 -0.5000 -0.5000 -0.5000 -0.4999 -0.5000 -0.4998 -0.5000 -0.5001 -0.5000 -0.5001	25.8750	5	49700
4	-0.5000 -0.4999 -0.5000 -0.4999 -0.4998 -0.5000 -0.5001 -0.5000 -0.4999 -0.4997 -0.4997 -0.4999 -0.4997 -0.5000 -0.4997 -0.5001	25.8750	5	49700
5	-0.4999 -0.4999 -0.4999 -0.4999 -0.5000 -0.5000 -0.4999 -0.5000 -0.5000 -0.4998 -0.5000 -0.5000 -0.5001 -0.4999 -0.5000 -0.4999	25.8750	5	49700

Changes to Option settings by adding one of the following:

- `options = gaoptimset ('PopulationSize', number of desired population size)` → different **population size**
- `options = optimoptions('ga','MutationFcn', {@mutationadaptfeasible, desired value in [0.01-0.1] range})` → different **mutation rate**
- `options = optimoptions('ga','CrossoverFraction', range from 0-1)` → different **crossover rate**

Change (only 1 variable at a time)	Optimum Point(s) → 16 variables	Function Value	# of Iterations	# of func.eval
Population size = 10	-0.1906 -0.3836 -0.4384 -0.2879 -0.5531 -0.1788 -0.2850 -0.4746 -0.5480 -0.4502 -0.2770 -0.2547	27.7586	5	3638

	-0.4462 -0.4634 -0.4985 -0.4803			
Population size = 50	-0.4988 -0.4975 -0.4957 -0.4987 -0.4988 -0.5008 -0.4996 -0.4969 -0.5004 -0.4955 -0.4972 -0.4998 -0.4995 -0.4998 -0.4989 -0.4980	25.8753	5	12300
Population size = 200	-0.5000 -0.5000 -0.5000 -0.4998 -0.4996 -0.5002 -0.5000 -0.4996 -0.4998 -0.5000 -0.4999 -0.4997 -0.5000 -0.4999 -0.4999 -0.5000	25.8750	5	49700
Population size = 500	-0.4998 -0.5000 -0.5000 -0.5001 -0.5000 -0.5000 -0.5001 -0.4999 -0.4999 -0.5001 -0.5000 -0.5000 -0.5000 -0.4999 -0.4999 -0.4999	25.8750	5	124250
mutation rate = 0.005	-0.4990 -0.4997 -0.4997 -0.5000 -0.5002 -0.4998 -0.4998 -0.4999 -0.4997 -0.5000 -0.4999 -0.4995 -0.5000 -0.4999 -0.4999 -0.4996	25.8750	5	49700
mutation rate = 0.05	-0.5000 -0.4999 -0.5000 -0.5000 -0.4998 -0.5000 -0.5000 -0.4998 -0.5000 -0.5000 -0.4999 -0.5000 -0.5000 -0.4999 -0.5000 -0.4999	25.8750	5	49700
mutation rate = 0.5	-0.4996 -0.5002 -0.5001 -0.4999 -0.5000 -0.5000 -0.4997 -0.4996 -0.4999 -0.5000 -0.4996 -0.4997 -0.5001 -0.4999 -0.5000 -0.4999	25.8750	5	49700
crossover rate = 0.1	-0.4988 -0.4988 -0.4989 -0.4991 -0.4993 -0.4989 -0.4992 -0.5000 -0.4982 -0.4990 -0.5002 -0.4988 -0.4983 -0.5001 -0.4985 -0.4994	25.8751	5	49700
	-0.4999 -0.4999 -0.5000 -0.5000 -0.4999 -0.5000			

crossover rate = 0.5	-0.5000 -0.4999 -0.4998 -0.5000 -0.5000 -0.5001 -0.5000 -0.5000 -0.5000 -0.5000	25.8750	5	49700
crossover rate = 0.8	-0.4998 -0.5000 -0.5000 -0.4999 -0.4999 -0.5000 -0.4999 -0.5000 -0.5000 -0.5000 -0.5000 -0.5001 -0.4999 -0.4999 -0.5000 -0.4999	25.8750	5	49700

Analysis of results:

- It appears that only Population Size affected function values, and changing Crossover or Mutation Rates did not have an impact.
- The higher the population size, the higher chance of finding the global minimum rather than a local minimum. That is consistent with findings shown in the table, as population size is increases to 500, the function value is decreased, which is the purpose of minimization.
- In addition to function value being constant, changing mutation rate or crossover rate, had no impact on number of iterations or number of function evaluations.

Q3)

The following table includes results after 10 random runs:

Run	Optimum Point(s) → 3 variables	Function Value	# of Iterations	# of func.eval
1	[0.3936,0.1594,-0.1856]	0.2369	140	7000
2	[0.3603,0.8183,0.5598]	0.9614	406	20300
3	[0.8881,0.1165,0.5838]	0.3047	150	7500
4	[0.4328,0.0000,0.7790]	0.6895	258	12900
5	[0.6122,0.7921,0.1729]	0.9344	208	10400
6	[0.9703,0.8954,0.8353]	0.3829	188	9400
7	[0.7427,0.0853,0.2670]	0.9430	161	8050
8	[0.7021,0.8586,0.7978]	0.9821	142	7100
9	[0.4979,0.5893,0.6556]	0.4378	203	10150
10	[0.9252,0.1236,0.0382]	0.9621	173	8650

Analysis of results:

- This question was treated differently due to being multi-objective (3 variables with 2 objective functions).
- 'gamultiobj' function was used instead of 'ga' function, and in the 'func1.m' MATLAB file, both functions are separately introduced.

- Compared to other three questions, number of function evaluations were scattered for each run. One reason could be due to it being multi-objective.
- Multi-objective problems cannot have inequality constraints, therefore, A, b, Aeq, beq, lb, and ub are left blank (see 'code' section).

Q4)

The following table includes results after 5 random runs:

Run	Optimum Point(s) → 10 variables				Function Value	# of Iterations	# of func.eval
1	0.0815	1.0000	1.0000	1.0000	0.2808	306	61200
	1.0000	0.9999	1.0000	0.9999			
		1.0000	1.0000				
2	0.0815	0.9803	0.9955	0.9994	0.2808	416	83200
	0.9918	0.9988	0.9979	0.9956			
		0.9946	0.9963				
3	0.0815	0.9942	0.9883	0.9910	0.2808	714	142800
	0.9844	0.9806	0.9815	0.9897			
		0.9777	0.9848				
4	0.0815	1.0000	0.9999	1.0000	0.2808	364	72800
	1.0000	0.9999	0.9999	1.0000			
		0.9999	0.9989				
5	0.0815	0.9999	1.0000	1.0000	0.2808	375	75000
	1.0000	1.0000	1.0000	1.0000			
		0.9999	1.0000				

Analysis of results:

- This is also a multi-objective function similar to question 3, with the difference of the second objective function including the first and third. All objective functions are defined in 'func1.m'.
- Again, a major difference between questions 3 and 4 versus questions 1 and 2 is the precision of number of function evaluations. It appears the multi-objective problems have more scattered number of function evaluations.
- The populations size, mutation rate and cross over rate will not have an effect on this problem. The third column from left indicates that number of function evaluations are exactly the same for all runs.

Code

Note: Original codes included in this section. Any specific changes (e.g. GA option changes) are included Detailed Results section.

Sub-functions:

Fitness Function – "func1.m"

```
function [f,g] = func1(x)
```

```
global problem_number
```

```
if problem_number == 1
    f = ((1/6.931) - ((x(1)*x(2))/(x(3)*x(4))))^2;
end
```

```
if problem_number == 2
    a = load('a.mat').a;
    t = 0;
    for i=1:16
        for j=1:16
        t = t + (a(i,j) * (x(i)^2 + x(i) + 1) * (x(j)^2 + x(j) +1));
        end
    end
    f = t;
end
```

```
if problem_number == 3
    t = 0;
    q = 0;
    for i=1:3
        t = (x(i) - (1/sqrt(3)))^2 + t;
    end
    f(1) = 1 - exp(-1*t);

    for i=1:3
        q = (x(i) + (1/sqrt(3)))^2 + q;
    end
    f(2) = 1 - exp(-1*q);
end
```

```
if problem_number == 4
    f(1) = 1 - (exp(-4*x(1)) * sin(6*pi*x(1))^6);
    t = 0;
    for i=2:10
        t = x(i)+t;
    end
    t = t/9;
    g = 1 + (9 * (t)^0.25);
    f(2) = (f(1)/g)^2;

end
```

Constraints – “func2.m”

```
function [c,ceq] = func2(x)
global problem_number

if problem_number == 1
    c = [];
    ceq = [];
end

if problem_number == 2
    c = [];
    ceq = [];
end

if problem_number == 3
    c = [];
    ceq = [];
end

if problem_number == 4
    c = [];
    ceq = [];
end
end
```

Q1)

Main Code – “Lab3.m”

```
if problem_number == 1
    A=[];
    b=[];
    Aeq=[];
    beq=[];
    lb=[14, 14, 14, 14];
    ub=[60, 60, 60, 60];
    Intcon = [1,2,3,4];

    nvar = 4; % number of variables
    [x,fval,exitflag] = ga(@func1,nvar,A,b,Aeq,beq,lb,ub, @func2, Intcon);

    fprintf("\n For Question 1: \n Optimum found is [%.4f,%.4f,%.4f,%.4f] where the function value is %.4f \n", x(1), x(2), x(3), x(4), fval)
end
```

Q2)

Main Code – “Lab3.m”

```
if problem_number == 2
    A=[];
    b=[];
    Aeq=[];
    beq=[];
    lb=[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1];
    ub=[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1];
    Intcon = [];

    nvar = 16; % number of variables
    [x,fval,exitflag] = ga(@func1,nvar,A,b,Aeq,beq,lb,ub, @func2, Intcon);

    fprintf("\n For Question 2: \n Optimum found is [%.4f,%.4f,%.4f,%.4f] where the function value is %.4f \n", x(1), x(2), x(3), x(4), fval)
end
```

Q3)

Main Code – "Lab3.m"

```
if problem_number == 3
    A=[];
    b=[];
    Aeq=[];
    beq=[];
    lb=[];
    ub=[];

    nvar = 3; % number of variables
    [x,fval,exitflag] = gamultiobj(@func1,nvar,A,b,Aeq,beq,lb,ub, @func2);  % allowed multiple onj
    fprintf("\n For Question 3: \n Optimum found is [%.4f,%.4f,%.4f] where the function value is %.4f \n", x(1), x(2), x(3), fval)
end
```

Q4)

Main Code – "Lab3.m"

```
if problem_number == 4
    A=[];
    b=[];
    Aeq=[];
    beq=[];
    lb=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0];
    ub=[1, 1, 1, 1, 1, 1, 1, 1, 1, 1];

    nvar = 10; % number of variables
    [x,fval,exitflag] = gamultiobj(@func1,nvar,A,b,Aeq,beq,lb,ub, @func2);  % allowed multiple onj
    fprintf("\n For Question 4: \n Optimum found is [%.4f,%.4f,%.4f] where the function value is %.4f \n", x(1), x(2), x(3), fval)
end
```