Metaheuristic

Assignment 2 23/06/19

Contents

Exercise 1: minimization and sensitivity analysis (in Python)

This problem is to use sensitivity analysis on revenue management for a very simplified airline pricing model. We will assume that an airline has one flight per day from Boston to Atlanta and they use an airplane that seats 150 people. The airline wishes to determine three prices, p_i (i=1,2,3), one for seats in each of the three fare buckets it will use. The fare buckets are designed to maximize revenue by separating travelers into groups, for instance 14 days advance purchase, leisure travelers, and business travelers. The airline models demand for seats in each group using the formula:

$$D_i = a_i \exp\left(-\frac{1}{a_i} p_i\right).$$

Where D_i is the people that want to fly given price p_i, the remaining parameters are a_1=100, a_2=150, and a_3=300. Please note, for simplicity you may assume that each D_i is a continuous variable.

(a) Formulate the revenue maximization problem for this flight as an optimization problem.
(b) What are the optimal prices and how many people are expected to buy a ticket in each fare bucket?
(c) Using sensitivity analysis, if the airline were to squeeze three additional seats onto this flight,
   a. How much do you expect revenue to change?
   b. By how much should the airline change each price?

1. Formulate the revenue maximization problem

I used Python to solve this problem. As the problem is maximization, but in Scipy we are going to deal with the minimization problem, therefore we transform it with (-1). The I define 4 constraints and transform them into the form >=0. Exercise there's no bound so no bound was defined.

Code

```
def objective(p):
    result=0
    a=[100,150,300]
    d=[]
    for i in range(3):
        di=a[i]*exp(-p[i]/a[i])
        d.append(di)
        result=result+d[i]*p[i]
    return -result
```

```
#D1>=0
def cons1(p):
    result=100*exp(-p[0]/100)
    return result
#D2>=0
def cons2(p):
    result=150*exp(-p[1]/150)
    return result
#D3>=0
def cons3(p):
    result=300*exp(-p[2]/300)
    return result
#the same as constraints above
#(D1+D2+D3)<=150  --> change to 150-(D1+D2+D3))>=0
def cons4(p):
    result=cons1(p)+cons2(p)+cons3(p)
    return 150-result
```

2. Optimize point

As no bound is required, I just take three 0 as initial point to apply to the minimization algorithm. Then I run Scipy to minimize the function with constraints.

Code

```
#no bounds for pricing (p)
#now give a initial
p0=np.zeros(3)
print(objective(p0))
```

```
con1={'type': 'ineq', 'fun':cons1}
con2={'type': 'ineq', 'fun':cons2}
con3={'type': 'ineq', 'fun':cons3}
con4={'type': 'ineq', 'fun':cons4}
cons=[con1,con2,con3,con4]

sol=minimize(objective,p0,method='SLSQP',constraints=cons)
```

```
print('\nExpective revenue:' , -round(sol.fun))
print('Seat[D1,D2,D3]: ',
round(cons1(sol.x)),',',round(cons2(sol.x)),',',round(cons3(sol.x)))
print('Price[P1,P2,P3]: ',
round(sol.x[0],2),',',round(sol.x[1],2),',',round(sol.x[2],2))
```

The result I get is below as the optimal result

```
Expective revenue: 43671
Seat[D1,D2,D3]:  21 , 38 , 91
Price[P1,P2,P3]:  156.75 , 206.75 , 356.75
```

3. Sensitivity analysis

I've define an function to run the scipy minimize that we just have to call the function then we would know directly how the increase of seats would have impact on the result and revenue. Once we apply it to the increase of 3 seats we get the result of 164 increase in the revenue.

```
New Expective revenue: 43835
New seat[D1,D2,D3]:  22 ,  39 , 93
New price[P1,P2,P3]:  152.87 ,  202.87 , 352.87
```

Code

```
def sensiAna(increase):
    def new_cons4(p):
        return cons4(p)+increase
    con1={'type': 'ineq', 'fun':cons1}
    con2={'type': 'ineq', 'fun':cons2}
    con3={'type': 'ineq', 'fun':cons3}
    con4={'type': 'ineq', 'fun':new_cons4}
    new_cons=[con1,con2,con3,con4]
    newsol=minimize(objective,p0,method='TNC',constraints=new_cons)
    return newsol
```

```
newsol=sensiAna(3)
print('\nwhen we increase the seat by 3\n')
print('New Expective revenue:' , -round(newsol.fun))
print('New seat[D1,D2,D3]: ', round(cons1(newsol.x)),',
',round(cons2(newsol.x)),',',round(cons3(newsol.x)))
print('New price[P1,P2,P3]: ', round(newsol.x[0],2),',
',round(newsol.x[1],2),',',round(newsol.x[2],2))
```

Exercise 2: Gradient based (in Python)

1. Banana (Rosenvrock) function

Consider the following three optimization problems:

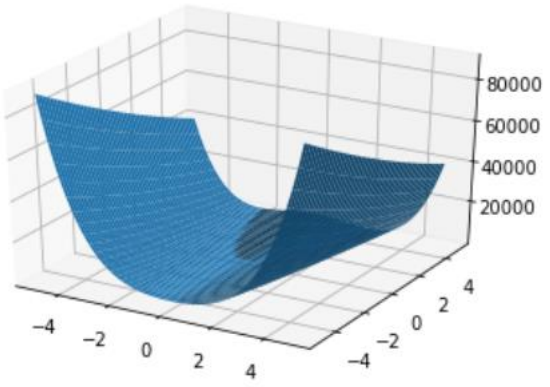
The Banana (Rosenbrock) Function

This function is known as the "banana function" because of its shape; it is described mathematically in Equation (1). In this problem, there are two design variables with lower and upper limits of [-5, 5]. The Rosenbrock function has a known global minimum at [1, 1] with an optimal function value of zero.

$$\text{Minimize } f(\mathbf{x}) = 100\left(x2 - x1^2\right)^2 + (1 - x1)^2 \qquad (1)$$

Code

```
fig = plt.figure()
ax = fig.gca(projection='3d')

xmesh, ymesh = np.mgrid[-5:5:50j,-5:5:50j]
fmesh = ex21objective(np.array([xmesh, ymesh]))

surf=ax.plot_surface(xmesh,ymesh,fmesh)
```



1.1. One run with fix starting point

First to try out the function I used a fix point x1=0 and x2=0 as an initial point to see if it would converge to the good result. In the end it converge to **x1: 1.0 x2: 1.0 result: 0.0,** the same as the problem above describe. (here we used TNC method, which uses a truncated Newton algorithm to minimize a function with variables subject to bounds. This algorithm uses gradient information; it is also called Newton Conjugate-Gradient. We can also use method 'L-BFGS-B' for bound constrained problem, see more)

Code

```
#exercise 2-1:
#Banana (Rosenvrock) function
def ex21objective(x):
    x1=x[0]
    x2=x[1]
    return 100*(x2-x1**2)**2+(1-x1)**2
#define bound
```

```
b=(-5,5)
bnds=(b,b)

#starting point
x0=np.zeros(2)

#apply Scipy
ex21sol=minimize(ex21objective,x0,method=' TNC',bounds=bnds)
print('x1: ',round(ex21sol.x[0]),' x2: ' ,round(ex21sol.x[1]))
print('result:',round(ex21objective(ex21sol.x)))
```

1.2. One run with random starting point and iteration result

Then I try with a random starting point to see if it would also converge, here I also present the iteration point for this optimization process. (with the defined callback function)

Code

```
#define callback function
Nfeval = 1
def callbackF(Xi) :
    global Nfeval
    print(Nfeval, Xi[0],Xi[1],ex21objective(Xi))
    Nfeval += 1

x0=[random.uniform(-5,5),random.uniform(-5,5)]
ex21sol=minimize(ex21objective,x0,method='TNC',bounds=bnds, callback=callbackF)

 1 2.105700646814523 -2.451141365977696 4741.705605901995
 2 -0.7837177551479008 -0.26527203113638903 80.53113244902045
 3 0.9803163111387305 0.8457787355346301 1.3284439618844548
 4 0.9353280744717986 0.8649263483409373 0.014007744916026792
 5 0.931369642249137 0.8671188821564662 0.004721050903816549
 6 0.9315729765284225 0.8680708452486284 0.004688144698547167
 7 0.9748947829885539 0.9486299879709897 0.000950628197420929
 8 0.9978991222644856 0.9955136765656047 1.2764726699437057e-05
 9 0.998231994520288 0.9964861484075037 3.162070878170172e-06
 10 1.0001333685647564 1.0002617060632377 2.033626618023583e-08
 11 1.000131076894153 1.000262715323505 1.7210784313620442e-08
```

1.3. 10 runs with random starting points with converge plot

Then I do a loop to run it for 10 times to see if every time it converges to the optimal points, also draw all the optimal process for 10 runs. In the data frame I've also kept each run's iteration numbers can memory running time.

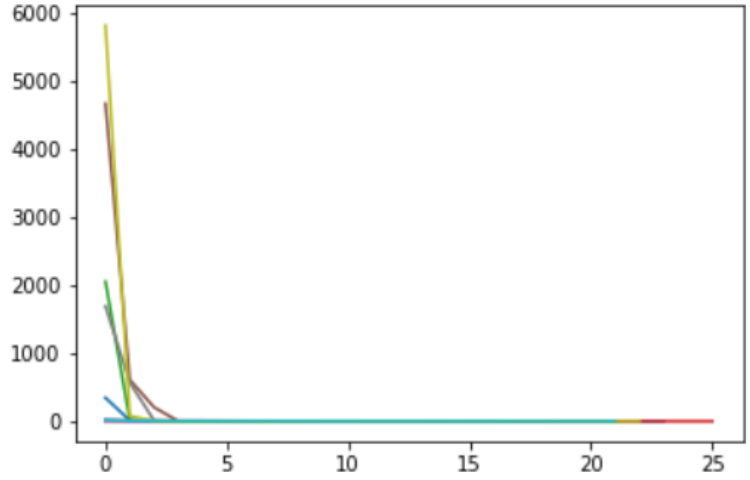
Eventually the result show us it all converge close to point (1,1) to reach the global minimum 0.

Code

```
#define callback function
Nfeval1 = 1
def callbackF1(Xi) :
    global resultlist1
    global Nfeval1
    print(Nfeval1, Xi[0],Xi[1],ex21objective(Xi))
```

```
    resultlist1.append(ex21objective(Xi))
    Nfeval1 += 1

#x0=[random.uniform(-5,5),random.uniform(-5,5)]
#ex21sol=minimize(ex21objective,x0,method='TNC',bounds=bnds, callback=callbackF1)

#Banana (Rosenvrock) function
def ex21objective(x):
    x1=x[0]
    x2=x[1]
    return 100*(x2-x1**2)**2+(1-x1)**2
#define bound
b=(-5,5)
bnds=(b,b)
result1=[]
for i in range(10):

    start_time = time.time()
    resultlist1=[]
    x0=[random.uniform(-5,5),random.uniform(-5,5)]
    ex21sol=minimize(ex21objective,x0,method='TNC',bounds=bnds, callback=callbackF1)
    end_time = time.time()
    a=(i+1,ex21sol.nit,ex21sol.x[0],ex21sol.x[1],ex21sol.fun,end_time-start_time)
    result1.append(a)

    nbiter1=[]
    for f in range(0,ex21sol.nit):
        nbiter1.append(f)
    plt.plot(nbiter1,resultlist1)

dfresult=pd.DataFrame(result1,columns=['Run','nbiter','x1','x2','f(x)','CPU(time)'])
print(dfresult)
```

	Run	nbiter	x1	x2	f(x)	CPU(time)
0	1	24	1.000251	1.000485	9.521498e-08	0.006981
1	2	22	0.999998	0.999996	4.698733e-12	0.010970
2	3	19	0.999428	0.998854	3.279151e-07	0.005984
3	4	26	1.000013	1.000027	1.967620e-10	0.008976
4	5	17	0.999867	0.999734	1.770955e-08	0.008977
5	6	18	0.999962	0.999924	1.430653e-09	0.008972
6	7	14	0.999988	0.999975	1.518854e-10	0.003989
7	8	17	1.000000	0.999999	3.900756e-13	0.004986
8	9	23	1.000003	1.000005	7.168263e-12	0.008975
9	10	22	0.959041	0.920177	1.694962e-03	0.009972



2. Eggcrate function

The Eggcrate Function

This function is described mathematically in Equation (2). In this problem, there are two design variables with lower and upper bounds of [-2π, 2π]. The Eggcrate function has a known global minimum at [0, 0] with an optimal function value of zero.

$$\text{Minimize } f(\mathbf{x}) = x1^2 + x2^2 + 25\left(\sin^2 x1 + \sin^2 x2\right) \qquad (2)$$

2.1. One run with fix starting point

First to try out the function I used a fix point x1=0 and x2=0 as an initial point to see if it would converge to the good result. In the end it converge to **x1: 0.0 x2: 0.0 result: 0.0,** the same as the problem above describe. (here we used TNC method, which uses a truncated Newton algorithm to minimize a function with variables subject to bounds. This algorithm uses gradient information; it is also called Newton Conjugate-Gradient. We can also use method 'L-BFGS-B' for bound constrained problem, see more)

Code

```
#exercise 2-2:
#Eggcrate function
def ex22objective(x):
    x1=x[0]
```

```
    x2=x[1]
    result= x1**2+x2**2+25*(sin(x1)**2+sin(x2)**2)
    return result
#define bound
b=(-2*pi,2*pi)
bnds=(b,b)

#starting point
x0=np.zeros(2)

#apply Scipy
ex22sol=minimize(ex22objective,x0,method='TNC',bounds=bnds)
print('x1: ',ex22sol.x[0],' x2: ' ,ex22sol.x[1])
print('result:',ex22objective(ex22sol.x))
```

2.2. One run with random starting point and iteration result

Then I try with a random starting point to see if it would also converge, here I also present the iteration point for this optimization process. (with the defined callback function)

Code

```
#define callback function
Nfeval = 1
def callbackF(Xi) :
    global Nfeval
    print(Nfeval, Xi[0],Xi[1],ex22objective(Xi))
    Nfeval += 1

x0=[random.uniform(-2*pi,2*pi),random.uniform(-2*pi,2*pi)]
ex21sol=minimize(ex22objective,x0,method='TNC',bounds=bnds, callback=callbackF)
```

```
1 -0.4829956726953333 -0.5456678877630021 12.657259734462436
2 -0.47814836141257205 -0.03584747184983465 5.555151420502396
3 -0.020737813040430167 0.18584966273648693 0.8993263170810185
4 0.0035181806956215015 0.020649103417393136 0.011406323522415508
5 -1.3783888007791283e-06 -2.584008590190978e-06 2.2300345808724473e-10
6 -4.999999987639253e-09 -4.999999984708784e-09 1.2999999928104898e-15
```

2.3. 10 runs with random starting points with converge plot

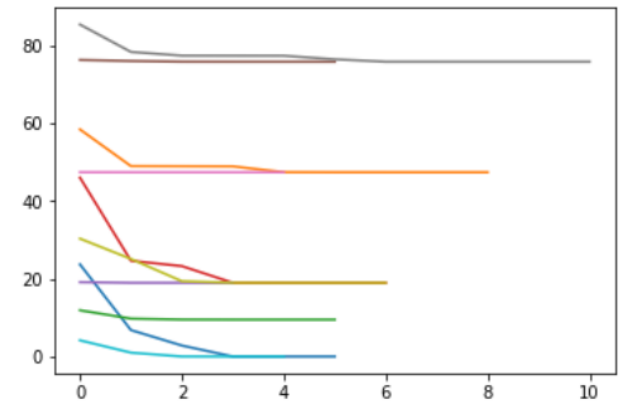
Then I do a loop to run it for 10 times to see if every time it converges to an optimal points, also draw all the optimal process for 10 runs. In the data frame I've kept each run's iteration numbers can memory running time. As the result, you can see easily from the graph that it most of the time stuck at the local minimum, and the gradient based minimization would take it as a final result if I only do one run.

Code

```
#define callback function
Nfeval2 = 1
def callbackF2(Xi) :
    global resultlist2
    global Nfeval2
    print(Nfeval2, Xi[0],Xi[1],ex22objective(Xi))
    resultlist2.append(ex22objective(Xi))
```

```
    Nfeval2 += 1

#x0=[random.uniform(-2*pi,2*pi),random.uniform(-2*pi,2*pi)]
#ex22sol=minimize(ex22objective,x0,method='TNC',bounds=bnds, callback=callbackF2)

#Eggcrate function
def ex22objective(x):
    x1=x[0]
    x2=x[1]
    result= x1**2+x2**2+25*(sin(x1)**2+sin(x2)**2)
    return result
#define bound
b=(-2*pi,2*pi)
bnds=(b,b)
result2=[]
#with random starting point
for i in range(10):

    start_time = time.time()
    resultlist2=[]
    x0=[random.uniform(-2*pi,2*pi),random.uniform(-2*pi,2*pi)]
    ex22sol=minimize(ex22objective,x0,method='TNC',bounds=bnds, callback=callbackF2)
    end_time = time.time()
    a=(i+1,ex22sol.nit,ex22sol.x[0],ex22sol.x[1],ex22sol.fun,end_time-start_time)
    result2.append(a)
    nbiter2=[]
    for f in range(0,ex22sol.nit):
        nbiter2.append(f)
    plt.plot(nbiter2,resultlist2)

print(pd.DataFrame(result2,columns=['Run','nbiter','x1','x2','f(x)','CPU(time)']))
```

	Run	nbiter	x1	x2	f(x)	CPU(time)
0	1	7	-3.019602e+00	-3.441339e-08	9.488197	0.002992
1	2	7	1.930122e-05	6.031423e+00	37.929472	0.001994
2	3	7	4.384638e-06	-6.031428e+00	37.929472	0.001994
3	4	6	-3.019602e+00	3.145288e-08	9.488197	0.001995
4	5	6	6.031424e+00	3.019615e+00	47.417669	0.001994
5	6	7	3.019598e+00	6.031416e+00	47.417669	0.001994
6	7	6	-3.762939e-09	-3.019602e+00	9.488197	0.002991
7	8	5	3.360371e-09	3.019602e+00	9.488197	0.000999
8	9	6	-3.019602e+00	-3.019602e+00	18.976395	0.000999
9	10	6	6.031424e+00	3.019601e+00	47.417669	0.000998



3. Golinski's Speed Reducer

Golinski's Speed Reducer

This hypothetical problem represents the design of a simple gearbox such as might be used in a light airplane between the engine and propeller to allow each to rotate at its most efficient speed.

The gearbox is depicted in Figure 2 and its seven design variables are labeled. The objective is to minimize the speed reducer's weight while satisfying the 11 constraints imposed by gear and shaft design practices. A full problem description can be found in Reference [1]. A known feasible solution obtained by a sequential quadratic programming (SQP) approach is a 2994.34 kg gearbox with the following values for the seven design variables: [3.5000 0.7000 17.0000 7.3000 7.7153 3.3502 5.2867].

This is a feasible solution with four active constraints, but is it an optimal solution?

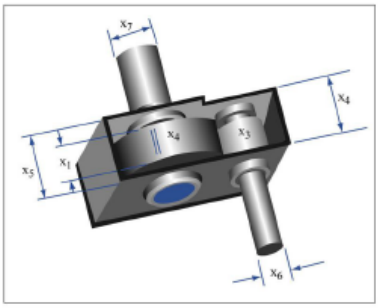


Image by MIT OpenCourseWare.

Figure 2: Golinski's Speed Reducer with 7 design variables

[1] Ray, T., "Golinski's Speed Reducer Problem Revisited," *the AIAA Journal*, Vol. 41, No. 3, 2003, pp. 556 -558.

3.1. 10 runs with random starting points with converge plot

We do the same technique as the last 2 functions, but here we need to define also the constraints, therefore we used the SLSQP method, which isSequential Least SQuares Programming to minimize a function of several variables with any combination of bounds, equality and inequality constraints.

Even though the success of the minimization is always False (means not reaching the global minimum) but all the tried (10 times) always reach to around 2995 that is to said, we can say this as a result. (converge close to the global minimum)

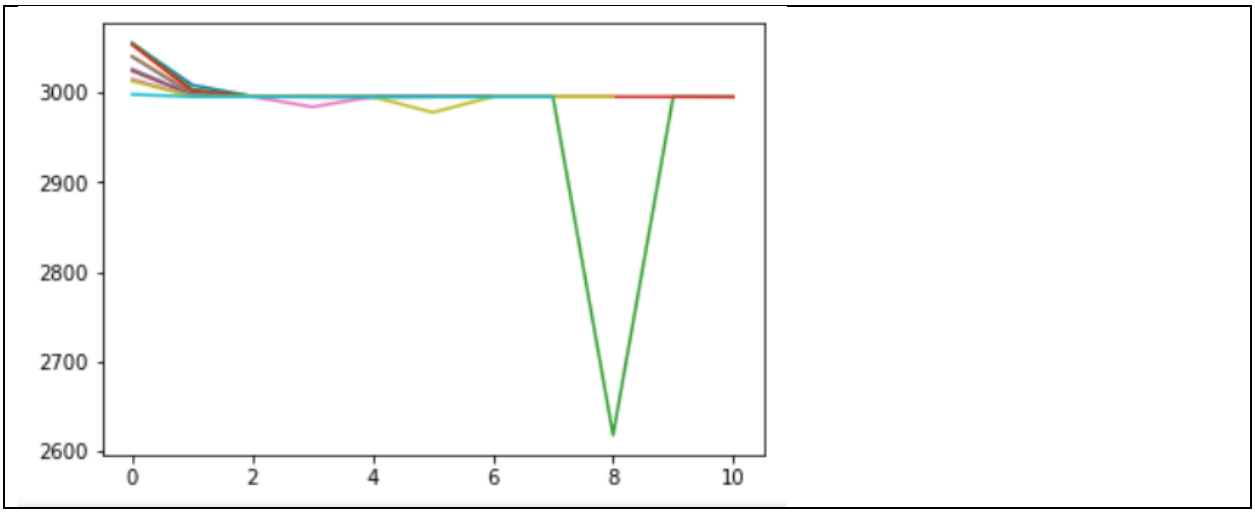
Code

```
#exercise 2-3 Speed Reducer
Nfeval3 = 1
def callbackF3(Xi) :
    global resultlist3
    global Nfeval3
    print(Nfeval3, objective(Xi))
    resultlist3.append(objective(Xi))
    Nfeval3 += 1

#Golinski

def objective(x):
    x1=x[0]
    x2=x[1]
    x3=x[2]
```

```
    x4=x[3]
    x5=x[4]
    x6=x[5]
    x7=x[6]
    a=(0.7858*3.3333)*x1*(x2**2)*(x3**2)
    b=(0.7854*14.9334)*x1*(x2**2)*x3
    c=(0.7854*43.0934)*x1*(x2**2)
    d=1.508*(x1*(x6**2)+x1*(x7**2))
    e=7.4777*(x6**3+x7**3)
    f=0.7854*((x4*(x6**2))+(x5*(x7**2)))
    result=a+b-c-d+e+f
    return result
#Constraints (change to >=0)
def cons1(x):
    result=(27/(x[0]*(x[1]**2)*x[2]))-1
    return -result
def cons2(x):
    result=(397.5/(x[0]*(x[1]**2)*(x[2]**2)))-1
    return -result
def cons3(x):
    result=((1.93*x[3]**3)/(x[1]*x[2]*(x[5]**4)))-1
    return -result
def cons4(x):
    result=((1.93*x[4]**3)/(x[1]*x[2]*(x[6]**4)))-1
    return -result
def cons5(x):
    result=(745**2)*(x[3]**2)/((x[1]**2)*(x[2]**2))-(110**2)*(x[5]**6)+16.9*(10**6)
    return -result
def cons6(x):
    result=(745**2)*(x[4]**2)/((x[1]**2)*(x[2]**2))-(85**2)*(x[6]**6)+157.5*(10**6)
    return -result
def cons7(x):
    result = (x[1]*x[2])-40
    return -result
def cons8(x):
    result = 5*x[1]-x[0]
    return -result
def cons9(x):
    result = x[0]-12*x[1]
    return -result
def cons10(x):
    result = 1.5*x[5]-x[3]+1.9
    return -result
def cons11(x):
    result = 1.1*x[6]-x[4]+1.9
    return -result

con1={'type': 'ineq', 'fun':cons1}
con2={'type': 'ineq', 'fun':cons2}
con3={'type': 'ineq', 'fun':cons3}
con4={'type': 'ineq', 'fun':cons4}
con5={'type': 'ineq', 'fun':cons5}
con6={'type': 'ineq', 'fun':cons6}
con7={'type': 'ineq', 'fun':cons7}
con8={'type': 'ineq', 'fun':cons8}
con9={'type': 'ineq', 'fun':cons9}
con10={'type': 'ineq', 'fun':cons10}
con11={'type': 'ineq', 'fun':cons11}
```

```
cons=[con1,con2,con3,con4,con5,con6,con7,con8,con9,con10,con11]

b1=(2.6, 3.6)
b2=(0.7, 0.8)
b3=(17, 28)
b4=(7.3, 8.3)
b5=(7.3, 8.3)
b6=(2.9, 3.9)
b7=(5, 5.9)
bnds=(b1,b2,b3,b4,b5,b6,b7)


result=[]
for i in range(10):
    start_time=time.time()
    resultlist3=list()
    x0=[random.uniform(2.6, 3.6),random.uniform(0.7, 0.8),random.uniform(17, 28),
        random.uniform(7.3, 8.3),random.uniform(7.3, 8.3),random.uniform(2.9,
3.9),random.uniform(5, 5.9)]
    sol3= minimize(objective,x0, method='SLSQP', constraints = cons, bounds = bnds,
callback=callbackF3)
    end_time=time.time()

a=(i+1,sol3.nit,sol3.x[0],sol3.x[1],sol3.x[2],sol3.x[3],sol3.x[4],sol3.x[5],sol3.x[6],s
ol3.fun,end_time-start_time)
    result.append(a)
    nbiter3=list()
    for f in range(0,sol3.nit-4):
        nbiter3.append(f)

    plt.plot(nbiter3,resultlist3)

print(pd.DataFrame(result,columns=['Run','nbiter','x1','x2','x3','x4','x5','x6','x7','f
(x)','CPU(time)']))
```

	Run	nbiter	x1	x2	x3	x4	x5	x6	x7	f(x)	CPU(time)
0	1	7	3.500000	0.700001	17.000000	7.300000	7.715333	3.350252	5.286669	2995.153655	0.009973
1	2	12	3.499998	0.700000	17.000000	7.300000	7.715320	3.350213	5.286651	2995.128263	0.033911
2	3	14	3.500000	0.700000	17.000000	7.300000	7.715320	3.350215	5.286654	2995.131658	0.045877
3	4	8	3.500020	0.700001	17.000356	7.300008	7.715550	3.350259	5.286492	2995.119147	0.012967
4	5	11	3.499998	0.700000	17.000000	7.300000	7.715320	3.350213	5.286651	2995.128588	0.013962
5	6	8	3.500000	0.700001	17.000000	7.300000	7.715320	3.350215	5.286655	2995.136296	0.006981
6	7	9	3.500000	0.700000	17.000000	7.300000	7.715320	3.350215	5.286654	2995.131859	0.007978
7	8	13	3.500007	0.700005	17.000000	7.300000	7.715302	3.350192	5.286643	2995.141393	0.019946
8	9	11	3.499999	0.700000	17.000000	7.300000	7.715320	3.350214	5.286653	2995.130516	0.024932
9	10	8	3.499995	0.700000	17.000260	7.300010	7.715358	3.350342	5.286662	2995.213096	0.013963



4. Conclusion for exercise 2

After running all 3 functions with gradient based method, based on Python Scipy package. We can see that only eggcrate function not always reach to a final good result. However the other 2 does. We can say the eggcrate function as multi-modal. So for exercise 3 we would move on with the same 3 functions but with heuristic technique in R.

Exercise 3: Genetic algorithm (GA) heuristic technique (in R)

Repeat the numerical experiments from exercise 2, but this time using a heuristic technique of your
choice (e.g. SA, GA …). Explain how you "tuned" the heuristic algorithm. Both SA and GA.

Compare your two algorithms (the gradient- search one and the heuristic one) from above quantitatively and qualitatively for the three problems as follows:

i. Dependence of answers on initial design vector (start point, initial population)
ii. Computational effort (CPU time [sec] or FLOPS)
iii. Convergence history
iv. Frequency at which the technique gets trapped in a local maximum

In order to answer this question, you *do not need* to implement your algorithms in some way *BUT* you MUST explain what algorithm is being used. Describe not just your conclusions, but also the process you followed. Do you think your conclusions would still apply for larger, more complex design optimization problems?

In this exercise I've used the GA package in R, within the latest GA package, I've tried both GA algorithm and DE (Differential Evolution via Genetic Algorithms), DE is a population-based evolutionary algorithm for optimization of fitness functions defined over a continuous parameter space.

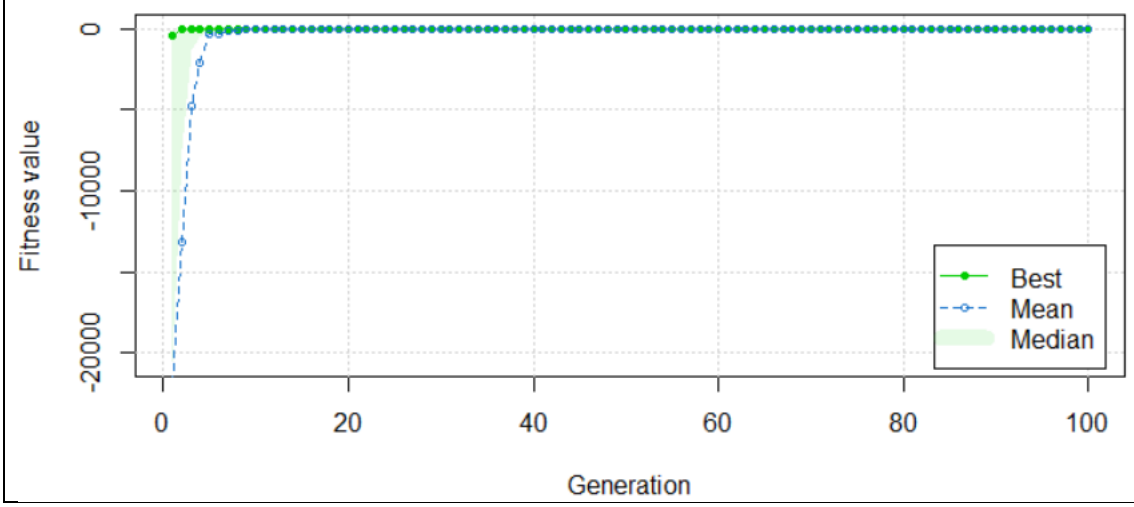
Even with tuning many different GA parameters. It turns out it is hard to get a stable result for the Golinski's function for all the runs, while DE can then satisfied all 3 functions, therefore I've used de for all the functions in the end.

1. Banana (Rosenvrock) function

1.1. One run with graph

Code

```
obj <- function(x) {
  x1=x[1]
  x2=x[2]
  return (100*(x2-x1**2)**2+(1-x1)**2)
}

#part 1: One run GA in R with graph
ga_res = de(type = "real-valued",
  fitness=function(x) -obj(x),
  keepBest = TRUE,
  lower = c(-5,-5),
  upper = c(5,5))
summary(ga_res)
plot(ga_res)

GAsum<- data.frame("iter"=ga_res@iter,"x"=ga_res@solution, "minf(x)"=-ga_res@fitnessValue)
GAsum
```



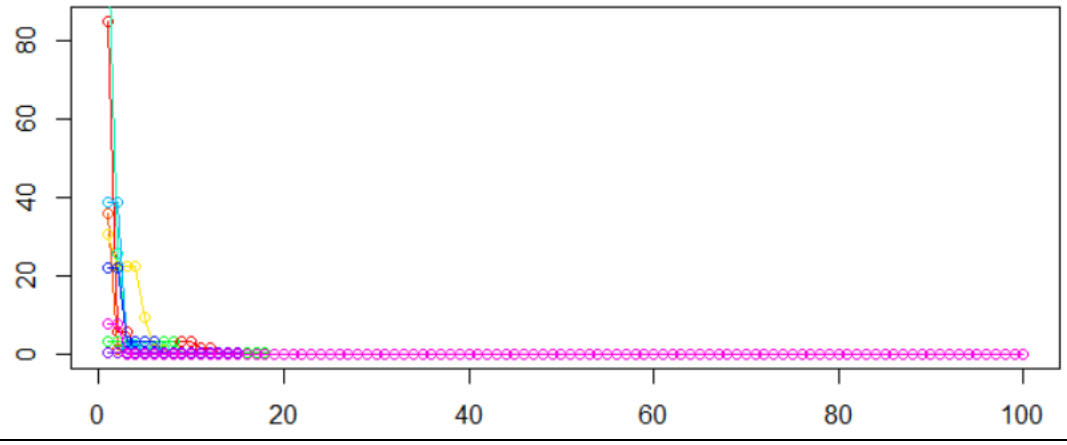
1.2. 10 runs with converge plot

Code

```
#part 2: 10 run table
df <- data.frame(matrix(ncol = 5, nrow = 0))
x1 <- c("iter", "x1","x2","minf(x)","CPU(time)")
val<-list()
for (i in 1:10){
  start_time <- Sys.time()
  ga_res = de(type = "real-valued",
          fitness=function(x) -obj(x),
```

```
          keepBest = TRUE,
          lower = c(-5,-5),
          upper = c(5,5))
  end_time <- Sys.time()
  tdif <- end_time - start_time
  df=rbind(df,c(ga_res@iter,ga_res@solution[1],ga_res@solution[2],-ga_res@fitnessValue,tdif))
  vec<-vector()
  for (j in 1:100){
    vec[j]=obj(ga_res@bestSol[[j]])
  }
  val[[i]]=vec
}
colnames(df) <- x1
name=c(1:10)
valdf <- as.data.frame(val)
colnames(valdf)<-name

colors<-rainbow(20)
plot(valdf[,10],type = "o",col=colors[1],ylab='')
for(i in 1:9){
  lines(valdf[,i], type = "o", col =colors[2*i])
}
```

	iter	x1	x2	minf(x)	CPU(time)
1	100	0.9986837	0.9973498	1.770235e-06	0.1486020
2	100	0.9999158	0.9998186	2.390578e-08	0.1725371
3	100	1.0003239	1.0006311	1.334400e-07	0.1166899
4	100	0.9999392	0.9998761	4.252638e-09	0.1126981
5	100	0.9998093	0.9995788	1.956216e-07	0.1226699
6	100	0.9998887	0.9998047	8.732426e-08	0.1515961
7	100	0.9997948	0.9995366	3.237235e-07	0.1266959
8	100	0.9998660	0.9997991	4.678519e-07	0.1286211
9	100	1.0002201	1.0004049	1.731314e-07	0.1675830
10	100	1.0003865	1.0007632	1.592339e-07	0.1665230

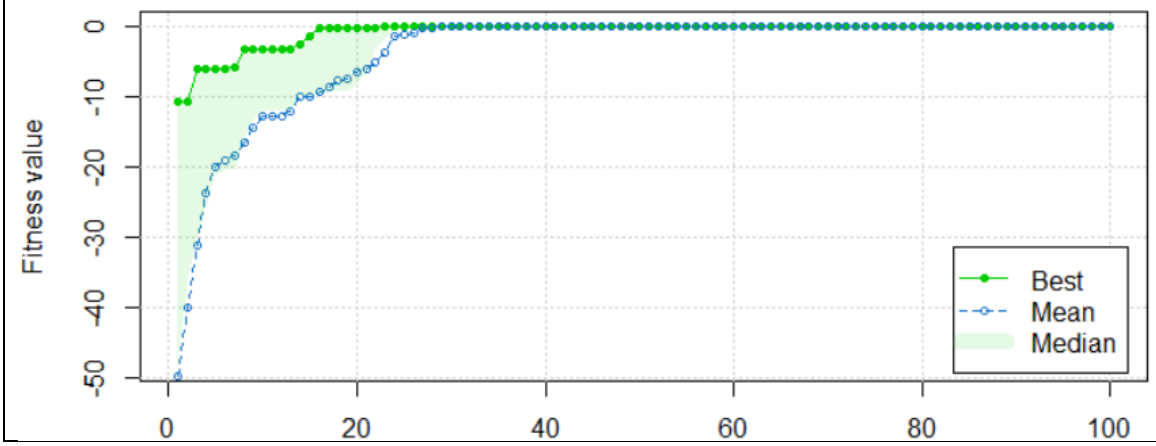


2. Eggcrate function

2.1. One run with graph

Code

```
obj <- function(x) {
  x1=x[1]
  x2=x[2]
  return (x1**2+x2**2+25*(sin(x1)**2+sin(x2)**2))
}

#part 1: One run GA in R with graph
ga_res = de(type = "real-valued",
  fitness=function(x) -obj(x),
  lower  = c(-2*pi,-2*pi),
  upper = c(2*pi,2*pi),
  keepBest = TRUE
)
summary(ga_res)
plot(ga_res)

GAsum<- data.frame("iter"=ga_res@iter,"x"=ga_res@solution, "minf(x)"=-ga_res@fitnessValue)
GAsum
```



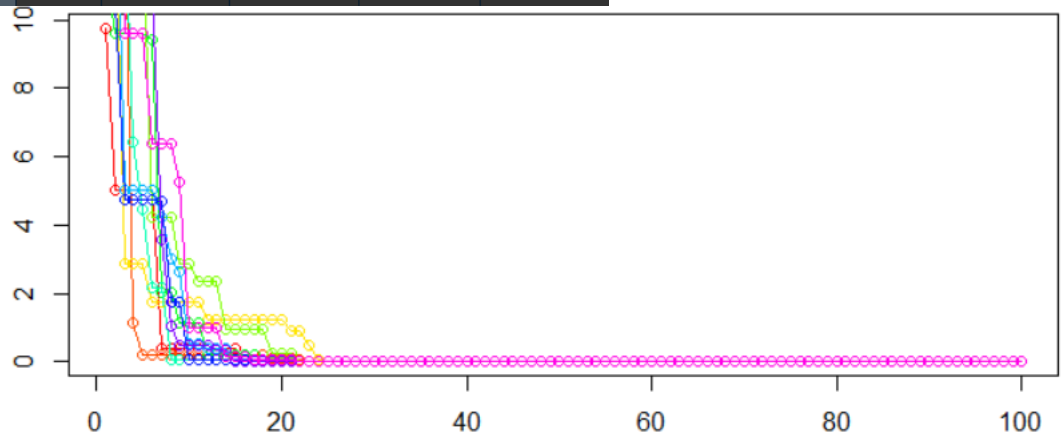
2.2. 10 runs with converge plot

Code

```
#part 2: 10 run table
df <- data.frame(matrix(ncol = 5, nrow = 0))
x1 <- c("iter", "x1","x2","minf(x)","CPU(time)")
val=list()
for (i in 1:10){
  start_time <- Sys.time()
  ga_res = de(type = "real-valued",
              fitness=function(x) -obj(x),
              lower  = c(-2*pi,-2*pi),
```

```
        upper = c(2*pi,2*pi),
        keepBest = TRUE)
 end_time <- Sys.time()
 tdif <- end_time - start_time
 df=rbind(df,c(ga_res@iter,ga_res@solution[1],ga_res@solution[2],-ga_res@fitnessValue,tdif))
 vec<-vector()
 for (j in 1:100){
  vec[j]=obj(ga_res@bestSol[[j]])
 }
 val[[i]]=vec
}
colnames(df) <- x1
name=c(1:10)
valdf <- as.data.frame(val)
colnames(valdf)<-name

colors<-rainbow(20)
plot(valdf[,10],type = "o",col=colors[1],ylab='')
for(i in 1:9){
 lines(valdf[,i], type = "o", col =colors[2*i])
}
```

	iter	x1	x2	minf(x)	CPU(time)
1	100	1.769682e-08	6.834011e-09	9.356911e-15	0.2194479
2	100	4.217439e-10	1.893010e-09	9.779527e-17	0.1320112
3	100	-2.630128e-09	-6.773603e-09	1.372781e-15	0.1416218
4	100	3.262828e-09	4.348106e-09	7.683538e-16	0.1186819
5	100	-6.296894e-09	1.245796e-08	5.066144e-15	0.1216772
6	100	3.070427e-09	9.650274e-11	2.453577e-16	0.1276610
7	100	5.830675e-08	-1.255541e-08	9.249020e-14	0.1366310
8	100	-6.054365e-10	9.869113e-10	3.485423e-17	0.1595740
9	100	-1.373639e-08	-1.274848e-09	4.948152e-15	0.1545880
10	100	-4.170816e-09	3.064913e-08	2.487589e-14	0.1655579



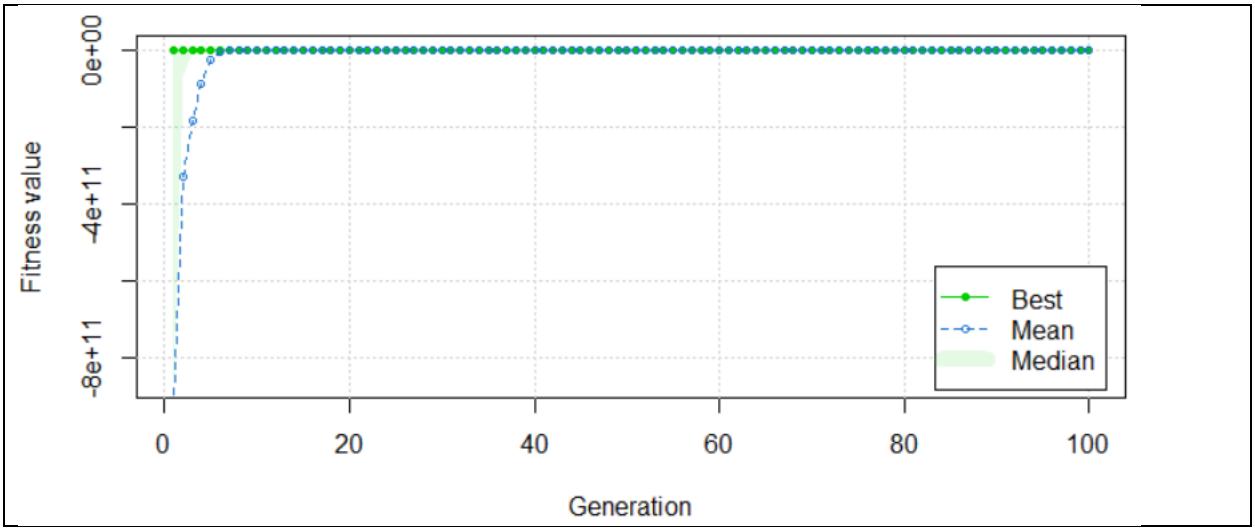
3. Golinski's Speed Reducer

3.1. One run with graph

Code

```
obj <- function(x) {
  x1=x[1]
  x2=x[2]
  x3=x[3]
  x4=x[4]
  x5=x[5]
  x6=x[6]
  x7=x[7]
  a=(0.7858*3.3333)*x1*(x2**2)*(x3**2)
  b=(0.7854*14.9334)*x1*(x2**2)*x3
  c=(0.7854*43.0934)*x1*(x2**2)
  d=1.508*(x1*(x6**2)+x1*(x7**2))
  e=7.4777*(x6**3+x7**3)
  f=0.7854*((x4*(x6**2))+(x5*(x7**2)))
  return (a+b-c-d+e+f)
}
# x=c(3.5,0.7,17,7.3,7.715333,3.350252,5.286669)
# x[7]
# obj(x)

c1<-function(x){
  (27/(x[1]*(x[2]**2)*x[3]))-1
}
c2<-function(x){
  (397.5/(x[1]*(x[2]**2)*(x[3]**2)))-1
}
c3<-function(x){
  ((1.93*x[4]**3)/(x[2]*x[3]*(x[6]**4)))-1
}
c4<-function(x){
  ((1.93*x[5]**3)/(x[2]*x[3]*(x[7]**4)))-1
}
c5<-function(x){
  (745**2)*(x[4]**2)/((x[2]**2)*(x[3]**2))-(110**2)*(x[6]**6)+16.9*(10**6)
}
c6<-function(x){
  (745**2)*(x[5]**2)/((x[2]**2)*(x[3]**2))-(85**2)*(x[7]**6)+157.5*(10**6)
}
c7<-function(x){
  (x[2]*x[3])-40
}
c8<-function(x){
  5*x[2]-x[1]
}
```

```
c9<-function(x){
  x[1]-12*x[2]
}
c10<-function(x){
  1.5*x[6]-x[4]+1.9
}
c11<-function(x){
  1.1*x[7]-x[5]+1.9
}


#penalised fitness function, <=0
fitness<-function(x)
{
  f<- -obj(x)
  #pen <- sqrt(.Machine$double.xmax)
  #pen<-0.00001
  pen<-50000
  cons1=max(c1(x),0)*pen
  cons2=max(c2(x),0)*pen
  cons3=max(c3(x),0)*pen
  cons4=max(c4(x),0)*pen
  cons5=max(c5(x),0)*pen
  cons6=max(c6(x),0)*pen
  cons7=max(c7(x),0)*pen
  cons8=max(c8(x),0)*pen
  cons9=max(c9(x),0)*pen
  cons10=max(c10(x),0)*pen
  cons11=max(c11(x),0)*pen
  f-cons1-cons2-cons3-cons4-cons5-cons6-cons7-cons8-cons9-cons10-cons11
}




#part 1: One run GA in R with graph
ga_res1 <- de(type = "real-valued", fitness = fitness,
           lower = c(2.6, 0.7, 17, 7.3, 7.3, 2.9, 5.0),
           upper = c(3.6, 0.8, 28, 8.3, 8.3, 3.9, 5.5),
           elitism = 2,keepBest=TRUE)

summary(ga_res1)
plot(ga_res1)

GAsum<- data.frame("iter"=ga_res1@iter,"x"=ga_res1@solution, "minf(x)"=-ga_res1@fitnessValue)
GAsum
```

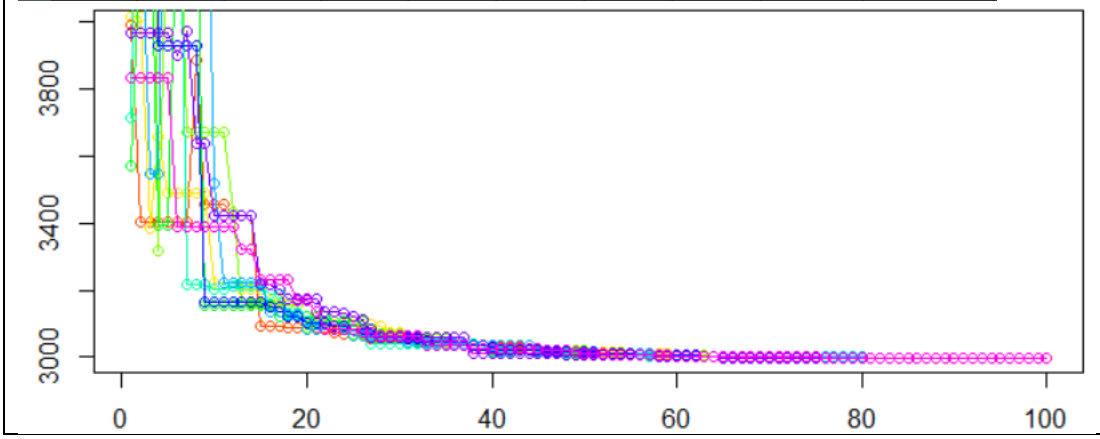


3.2. 10 runs with converge plot

Code

```
#part 2: 10 run table
df <- data.frame(matrix(ncol = 5, nrow = 0))
x1 <- c("iter", "x1","x2","x3","x4","x5","x6","x7","minf(x)","CPU(time)")
val=list()
for (i in 1:10){
 start_time <- Sys.time()
 ga_res <- de(type = "real-valued", fitness = fitness,
              lower = c(2.6, 0.7, 17, 7.3, 7.3, 2.9, 5.0),
              upper = c(3.6, 0.8, 28, 8.3, 8.3, 3.9, 5.5),
              elitism = 2,keepBest=TRUE)
 end_time <- Sys.time()
 tdif <- end_time - start_time
 df=rbind(df,c(ga_res@iter,ga_res@solution,-ga_res@fitnessValue,tdif))
 vec<-vector()
 for (j in 1:100){
  vec[j]=obj(ga_res@bestSol[[j]])
 }
 val[[i]]=vec
}
colnames(df) <- x1
name=c(1:10)
valdf <- as.data.frame(val)
colnames(valdf)<-name

colors<-rainbow(20)
plot(valdf[,1],type = "o",col=colors[1],ylab='')
for(i in 1:9){
 lines(valdf[,i], type = "o", col =colors[2*i])
}
```

	iter	x1	x2	x3	x4	x5	x6	x7	minf(x)	CPU(time)
1	100	3.500583	0.7000265	17.00081	7.313014	7.720022	3.350613	5.286689	2995.963	0.4931889
2	100	3.500449	0.7000365	17.00040	7.319565	7.717714	3.351838	5.286714	2996.219	0.5724940
3	100	3.500249	0.7000321	17.00113	7.311562	7.715536	3.352230	5.286678	2996.206	0.4832189
4	100	3.500117	0.7000057	17.00043	7.305163	7.721369	3.351090	5.286686	2995.699	0.6612310
5	100	3.500279	0.7000141	17.00015	7.300108	7.718104	3.350399	5.286692	2995.463	0.6965981
6	100	3.501468	0.7000103	17.00114	7.311481	7.716945	3.350497	5.287156	2996.480	0.6397972
7	100	3.500454	0.7000051	17.00031	7.310828	7.718437	3.351255	5.286810	2995.915	0.7091880
8	100	3.500159	0.7000181	17.00019	7.308865	7.722231	3.350676	5.286745	2995.714	0.6826839
9	100	3.500165	0.7000103	17.00013	7.320471	7.716106	3.350339	5.286833	2995.610	0.6592369
10	100	3.500518	0.7000611	17.00038	7.314725	7.715469	3.350415	5.286748	2995.921	0.6522100



4. Gradient search and heuristic optimization comparation

We will compare

i. Dependence of answers on initial design vector (start point, initial population)
ii. Computational effort (CPU time [sec] or FLOPS)
iii. Convergence history
iv. Frequency at which the technique gets trapped in a local maximum

function 1: Banana (Rosenvrock) function

	Gradient based	GA
Dependence of answers on initial design vector	Low	Low
Computational effort (CPU time [sec])	Lower	Higher
Convergence history	Always converge close to 0 (not exact)	Always converge close to 0 (not exact)
Technique gets trapped in a local maximum frequency	Never	Never

Function 2: Eggcrate function

	Gradient based	GA
Dependence of answers on initial design vector	High	Low
Computational effort (CPU time [sec])	Lower	Higher
Convergence history	Converge but some stop at local minim	Always converge to global minim

Technique gets trapped in a local maximum frequency	90%**	Never

**from trying 100 runs many time

Function 3: Golinski's Speed Reducer

	Gradient based	GA
Dependence of answers on initial design vector	Not much	Not much
Computational effort (CPU time [sec])	Lower	Higher
Convergence history	Always converge to 2995	All converge close to 2995
Technique gets trapped in a local maximum frequency	Never	Never