EVOLUTIONARY AND FUZZY SYSTEMS

COURSEWORK 2

M.Sc. Data Science and Computational Intelligence
Mohammad Huzefa Shaikh
8012085

Table of Contents

TASK 1

GENETIC IMPLEMENTATION OF THE INPUT MEMBERSHIP FUNCTION:

Genetic algorithm (GA) is a search technique that takes inspiration from natural selection and genetics. It updates a population of solutions to acquire a global optimum. The new solutions generated by the GA, also referred to as offspring, from the parental solutions by applying the crossover and mutation operation. Said operations should be applied in such a way that offspring inherit important factors of parents.

In this report, the GA is applied on the initial Fuzzy System design of the Ambient Assisted Living. The temperature and humidity system has been taken into consideration; the genotypes are implemented from its membership functions. Please observe the below-mentioned implementation.

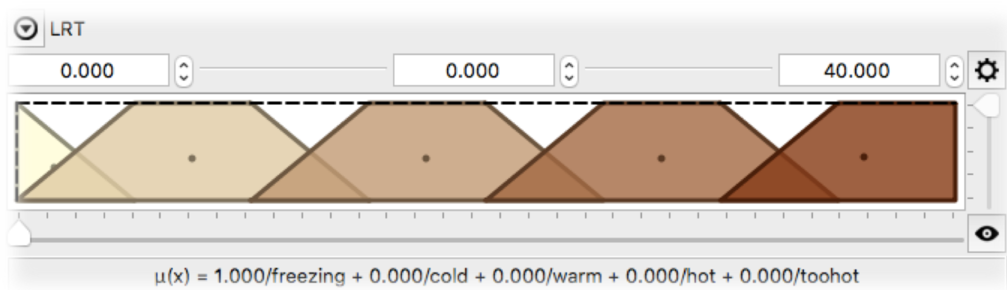


Figure 1: Living Room Temperature

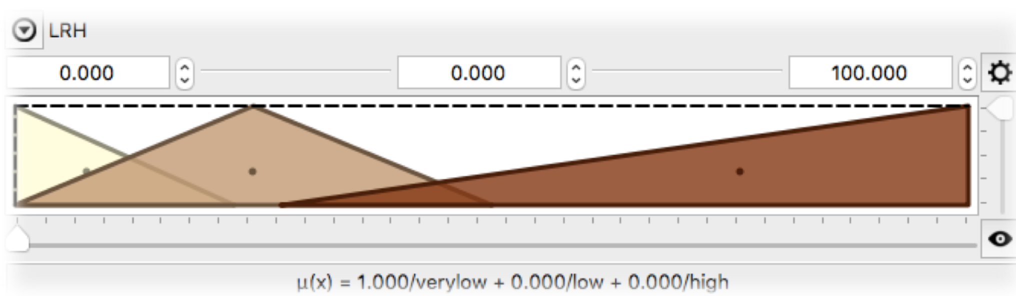


Figure 2: Living Room Humidity

5	5	10	15	10	15	20	25	20	25	30	35	30	35

Table 1: Genotype of The Temperature Membership Functions (pop1)

6	8	11	17	13	17	22	27	22	28	33	37	34	38

Table 2: Population Increment (pop2)

3	9	9	18	15	16	21	26	23	27	31	36	35	39

Table 3: Population Increment (pop3)

2	7	8	19	13	18	23	28	23	26	30	34	35	37

Table 4: Population Increment (pop4)

Algorithm Structure**:**

As the selection operation, the Minimal Generation Gap (MGG). The GA incorporating the MCG.

The operation is as follows:

1. Create the initial population, and set g←1. (g = generation).

   Created the population of 4 members (genotypes), pop1, pop2, pop3, and pop4.

   The first gene in the genotype (pop1) represents the first initiation point (crisp value) of the first membership function unless it's as same as the primary initiator (the starting point of the universe of discourse) of the particular MF. However, if it is the same then the next distinct turning point of that MF is considered to be the first gene, we move on to the next MF and store its initiator, the turning points, and the end of the membership function.

   The same implementation can be applied to the other inputs, however, there will be very small changes. Since we are using trapezoidal MFs in the current implementation, there are 4 genes representing each point of the trapezoid but the triangular MFs would probably generate 3 genes and similarly, Gaussian MFs would produce 2 genes (representing its height and the curve's width).

2. Chose 2 solutions, say *par1* and *par2* randomly from the created population as parents.

   Picked pop1 and pop3 as par1 and par2, respectively.

3. Create two further solutions *par3* and *par4* using the crossover operation. This operation is applied to every generation. *Picked non-similar (distinct) genes from the par1 and par2 and generated the following genotypes.*

3	25	15	9	30	9	30	18	20	16	25	21	35	26

Table 5: par3

5	5	10	15	10	15	21	21	26	23	27	31	36	39

Table 6: par4

4. Derive a solution *par5* from *par1* using the mutation operation. Do the same to generate solution *par6* from *par2*. This operation is also applied to every generation with the probability of *P.*

5. From the set of generated solutions: *{par1, par2, ⋯, par6}* select 2 best possible solutions. Remove *par1* and *par2* (parents) from the population, and replace with the best possible selected solutions in the population.

   As mentioned in the task description, there are n examples of the mapping (Xi, Yi), to expect output Yi for each Xi. Mean Square Error (MSE) was employed as a fitness function to evaluate the accuracy between the given defuzzified output (Y) and the genetically generated output (Y') of each population member.

   $$MSE = 1/n \sum_{k=1}^{n} (Yik - Y'ik)^2$$

6. Given If g=G, end the algorithm. (Where G =final generation; and its value is initially defined).

7. Given that g ≠ G, increment g (g←g+1) and continue from step 2.

Genetic implementation of the output membership function:

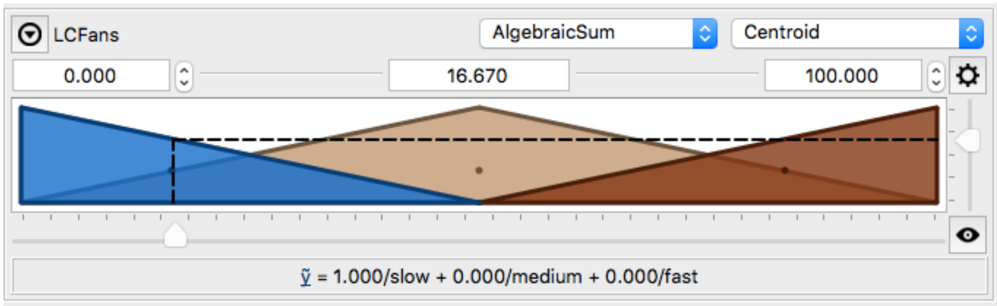


Figure 3: Cooling Fans (fuzzy output)

H/T – Cooling Fans	freezing	cold	warm	hot	Too hot
Very-low	No change	No change	slow	medium	fast
low	No change	slow	medium	fast	fast
high	No change	slow	fast	fast	fast

Table 7: Matrix- Temperature and Humidity output

Every output of each input membership function's combination represents each gene in the genotype. Please observe the below-mentioned genetic value assignation and the genotype.

Output Membership Function	Genetic value
No change	0
slow	1
Medium	2
fast	3

Table 7: Genetic values of each MF

0	0	1	2	3	0	1	2	3	3	0	1	3	3	3

Table 8: Genotype of the cooling fan output (pop1)

The above-mentioned genotype is generated by taking the first combination's rule base output, from the matrix table, as the first gene in genotype, similarly, the second gene is generated by the second the next input combination (row-wise), and so on. The same algorithm structure that is employed by the input MFs genetic population, would be applied to the output genotype. There are n examples of the combination and their mapped rules (Xi, Yi), representing the combination (Xi) and its consecutive rule (Yi). The same fitness function (MSE) would be applied to assess the accuracy of the current rule base and the predicted rule base.

Hybrid Implementation:

The above-mentioned Fuzzy Logic Control System is implemented by the hybrid functionalities of the evolutionary computing. The MFs are represented by the classes in hybrid systems, which consecutively produces a probable output of the distinct generated combinations with the respect of Q, the rule-base is developed by the continues reductions and increments of the size (space on the universe of discourse) of MFs.

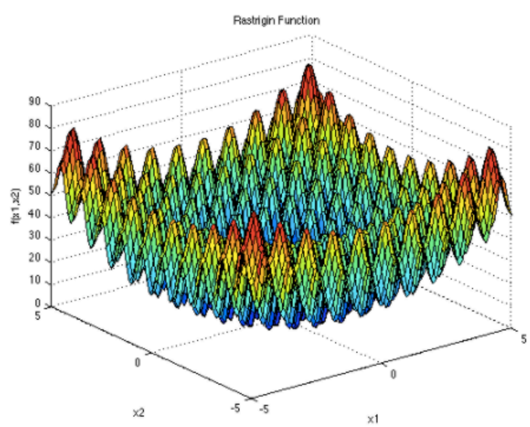
The same fitness function, that is used in the first question, is employed to extract the changes in the accuracy of the probable rule base, in each iteration of the increment and decrement with the mutation probability of P_u. The probability policy is updated by the count of the combination classes representing membership functions as sets, the update in the probability is represented by $_{u.}$

TASK 2

The employed optimization techniques on the used benchmark functions, Rastrigin and Sphere, are Particle Swarm Optimization (PSO), Genetic Algorithm (GA), and Differential Evolution (DE). The optimization experiment is implemented in Python and certain libraries, such as PyEvolve and SwarmPackagePy, were adopted.

However, not all the optimizers and functions were available as pre-defined methods, for which the methods had to be created from scratch for some of the optimizations' implementations.

RASTRIGIN FUNCTION:



$$f(\mathbf{x}) = 10d + \sum_{i=1}^{d} \left[x_i^2 - 10\cos(2\pi x_i)\right]$$

Figure 4: Rastrigin Function

$$f(\mathbf{x}) = An + \sum_{i=1}^{n} \left[x_i^2 - A\cos(2\pi x_i)\right]$$

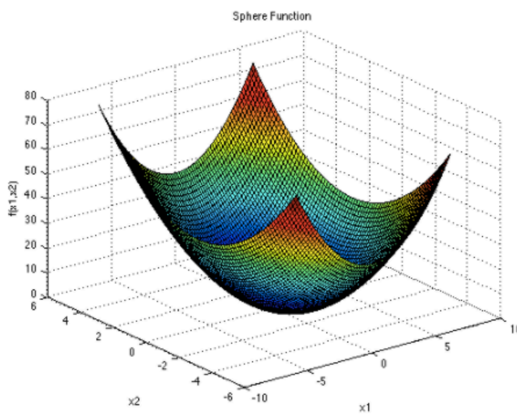
Dimensions: d; Global minimum at: x = 0; Function of x: f(x) = 0

In other way:

$f(\mathbf{x}^*) = 0$, at $\mathbf{x}^* = (0, \ldots, 0)$

The Rastrigin function (non-convex function) is exercised for optimization algorithms as a performance analysis problem. It contains various local minima, and it is a multimodal function with non-linearity. It was first introduced as a 2-dimensional problem, however, later upgraded to be used as a multi-dimensional performance scale. It is quite to difficult to optimize this function because of it having an extensive search space.

Sphere Function:



$$f(\mathbf{x}) = \sum_{i=1}^{d} x_i^2$$

Figure 5: Sphere Function

Dimensions: d; Global minimum: $f(\mathbf{x}^*) = 0$, at $\mathbf{x}^* = (0, \ldots, 0)$

The Sphere function contains local minima in multi-dimensional, however, its global minima lacks multi-dimensionality. The sphere function is convex, unimodal, continuous, and it produces twofold flat areas for the problem.

OPTIMIZATION:

Swarm Particle Optimization on Rastrigin Function-2-Dimensions:

PERFORMANCE	DIM-1	DIM-2	ITERATIONS	TIME (seconds)
1.004192582	-0.00160681	-1.0015896	977	215
0.012112632	-0.0072168	0.00299681	959	230
1.015899847	0.99732551	0.00999926	861	176
0.064382089	0.00111298	-0.0179895	959	170
1.018474241	0.01077565	-0.9933902	928	215
0.55338468	0.02742473	-0.0453083	986	181
0.244838892	0.01869077	0.02979509	856	201
0.288417203	-0.0208675	-0.0319743	903	180
1.003870576	-0.00629967	-0.9926692	910	166
1.050438822	0.00212645	1.01154807	902	170
1.051295318	1.00514006	0.0134323	948	207
1.159172635	-0.98827127	0.02801902	949	222
0.022748219	-0.00926288	-0.0053748	862	176
0.022668252	0.00219415	0.0104635	815	212
0.025847545	-0.01042455	0.00465339	988	179

BEST	0.01211263
WORST	1.15917264
AVERAGE	0.5691829
STD. DEV	0.48059819

Figure 6: SPO on Rastrigin-2-D

Swarm Particle Optimization on Rastrigin Function-10-Dimensions:

PERFORMANCE	DIM-1	DIM-2	DIM-3	DIM-4	DIM-5	DIM-6	DIM-7	DIM-8	DIM-9	DIM-10	ITERATIONS	TIME (seconds)
59.74209516	0.7455502	0.15149253	1.18772628	0.85675471	-0.9390943	2.11810809	2.04362635	1.99168213	1.76455825	0.88917185	949	359
24.14615125	2.91857432	1.00390016	-0.0105653	-0.9894297	0.06957866	0.89751705	-1.0152195	0.95001802	-1.0299541	-1.1386055	881	316
35.24321559	0.07250526	-0.9563747	0.33357897	1.03500326	0.83380287	0.92756116	0.91296069	-0.0482235	-1.1561055	0.04628638	916	300
30.65634975	-0.9420317	0.89730957	-1.931357	-0.0716403	1.06526886	-1.1079125	-0.8478117	-0.8833012	-1.9326297	1.07759694	881	333
53.46449465	-1.9540961	0.05100605	-3.0622195	-0.9555019	-2.0518409	1.8363265	-0.168963	-0.888214	-0.8342563	0.2551175	867	313
48.85057665	-1.0730292	-1.0573429	-0.8797572	0.12773846	0.30101046	0.9431046	0.04462942	1.76350913	1.85532653	0.13963424	844	346
57.88164692	2.00265357	-0.0200613	-1.0049266	2.2736744	1.9141751	-0.4853551	-1.9812959	1.90143945	-0.9560065	-0.0209142	983	348
42.97520653	-1.1218788	1.17382466	0.21813025	-1.9441568	1.00215391	0.13516592	-1.9722584	1.92911028	-1.0382685	-1.1450163	856	338
45.53016948	0.90751361	1.92256551	1.00587621	-0.9807461	-0.8317487	-0.0661847	0.20200891	-1.0262217	0.34589828	-1.90141	949	330
29.6494794	1.1041806	0.11920537	-1.0138623	0.13039794	2.04842242	2.11218316	1.92303642	1.01074121	0.0074751	-1.0672105	848	359
42.71606126	-2.0120012	0.25967043	-0.1163909	-0.8940425	-0.0412763	-0.9231801	2.17555135	-0.7826744	-0.9896435	-0.0409069	995	338
67.75746997	-4.1493348	1.04943963	-3.1499954	-0.3026761	-0.049156	0.07756884	0.06338698	-0.7665195	-0.9826522	1.93940054	1000	344
37.62397177	1.1678789	-0.0823742	-0.0101856	-1.0147182	-0.8600182	0.89503753	3.01562326	0.07685083	-1.0806452	-0.2301042	800	344
49.36993636	0.24585042	0.98672709	-0.7960229	1.155631	-0.9779238	0.94097048	0.26273539	-0.7547027	-0.0278722	-1.0198972	911	307
21.63106794	-1.0793954	-2.0087013	-0.9695615	-0.9592511	-2.9966618	1.04553361	-1.0314006	0.0082979	0.01390037	-1.0095856	899	348

BEST	21.6310679
WORST	67.75747
AVERAGE	43.1491928
STD. DEV	13.4613838

Figure 7: SPO on Rastrigin-10-D

Differential Evolution on Rastrigin Function-2-Dimensions:

PERFORMANCE	DIM-1	DIM-2	ITERATIONS	TIME (seconds)
8.847069706	-2.60742214	-1.4312301	932	155
8.847069706	-4.80395713	2.73234694	961	199
5.140096766	-1.60313704	-1.603137	858	197
0.127745934	-0.08740604	-0.3465633	961	209
8.847069706	4.220687413	-2.5837963	826	195
5.140096766	4.050964728	3.42059323	879	199
0.127745934	-5.90068888	-5.8062172	813	189
8.847069706	-3.98088785	1.02487408	926	224
5.140096766	-3.11981156	2.75717971	915	154
0.127745934	-3.11981156	2.75717971	907	189
6.585744794	-2.52660504	-1.1182457	817	201
8.847069706	-1.05323074	-2.3401816	885	222
5.140096766	3.063906155	3.41642919	992	186
0.127745934	1.101649384	-5.7585351	887	169
4.770919102	0.503390167	-2.55975	902	220

BEST	0.127745934
WORST	8.847069706
AVERAGE	5.110892215
STD. DEV	3.469207256

Figure 8: DE on Rastrigin-2-D

Differential Evolution on Rastrigin Function-10-Dimensions:

PERFORMANCE	DIM-1	DIM-2	DIM-3	DIM-4	DIM-5	DIM-6	DIM-7	DIM-8	DIM-9	DIM-10	ITERATIONS	TIME (seconds)
130.8344353	1.97469293	2.0397964	3.36103054	-4.8185669	-1.6290748	2.57606268	-4.5063734	-5.5876368	-3.8422993	-3.5604505	923	338
101.9256436	0.22789955	-1.1760037	-4.5542214	5.80835256	1.10131726	1.10131726	4.21670953	1.99005756	4.65811644	-0.3847234	815	312
109.4465424	-3.4342784	-0.5129366	-5.4725421	-3.9720492	-1.5890964	0.44586041	3.37646259	1.68963424	5.80318616	-1.0034571	817	321
112.3744508	-1.9554116	-4.3061299	2.64818168	-3.7598188	-1.73354	-2.5013733	-4.1918966	-4.8544498	-2.0552954	-3.7729312	815	313
112.1907942	1.27100072	-1.8990895	-5.1199023	-1.6871479	-3.0677096	-1.5732467	4.96470558	4.74099447	3.32295095	2.80279553	954	306
94.44079632	3.37206109	3.37206109	2.46248568	5.31675122	-3.9404475	-1.0185174	3.11143624	1.15518141	1.1785364	2.89803271	941	349
130.8344353	4.10253867	-2.4998395	-5.6687808	5.83712566	4.98511816	0.34291741	4.46527617	3.78291626	-4.9026733	5.72241283	818	351
101.9256436	-4.7354331	-0.8214756	3.52166839	-5.8630522	1.72743795	1.41879715	-2.7980326	-4.2450934	-3.102583	3.77576147	957	320
107.4454036	-2.4823708	0.56184674	-2.0525742	1.8224971	-0.9166007	-5.8209749	5.78163547	-1.6916259	4.63775984	0.95466279	880	312
109.4465424	-4.3605712	1.38079341	-1.8337172	3.0288714	-2.8832798	0.11438546	-0.9307129	-4.4015638	-5.2363862	4.69753351	844	302
112.3744508	-4.9090954	-0.5735374	2.74605799	-2.1078902	-1.4751279	-2.3068021	5.48732259	-5.7457803	-5.6627132	0.41749508	963	339
112.1907942	5.74672586	-4.4040517	3.43792366	-1.5139241	4.27410306	5.80474567	-0.9999863	-0.7075368	4.93092844	3.47115989	943	317
117.1305335	-0.7410481	-4.6596414	-4.4845706	-4.0425095	1.74806926	3.76725207	5.85682069	5.80637023	5.2676639	-0.9655178	951	359
111.0248688	-0.3068435	-4.953961	5.71093444	-4.7997597	1.6536106	-2.36435	0.77890405	-0.932229	1.60546853	4.28470182	991	311
93.94481935	0.36107743	4.27146367	-3.3209099	-2.2836741	3.86185999	-0.9707687	-0.9707687	0.76725157	4.33051527	4.81504381	953	333

BEST	93.9448194
WORST	130.834435
AVERAGE	110.50201
STD. DEV	9.32980055

Figure 9: DE on Rastrigin-10-D

Genetic Algorithm on Rastrigin Function-2-Dimensions:

	A	B	C	D	E
1	PERFORMANCE	DIM-1	DIM-2	ITERATIONS	TIME (seconds)
2	1.958874	0.00176782	-0.0012622	816	197
3	2.486346	-0.0030262	-0.0037378	890	218
4	1.129793	-0.0032606	-0.0037211	955	217
5	2.777357	-0.001082	-0.0010592	890	185
6	4.13996	0.00053875	2.64E-05	822	170
7	1.769212	0.00236801	0.00289143	919	195
8	3.46181	-0.0022134	0.00337133	851	202
9	3.313474	-0.0008425	0.00399817	939	213
10	2.623246	-0.0011595	0.00277111	834	213
11	2.62E+00	4.93E-05	0.00116189	911	224
12	3.23E+00	-2.40E-05	-0.0015999	823	156
13	2.32E+00	0.00093021	0.00076735	843	200
14	4.61E+00	0.00074259	-0.0022746	997	226
15	5.52E+00	0.00251189	0.00249516	913	157
16	4.49E+00	0.00307271	-9.81E-05	876	219
17					
18					
19					
20					
21					
22	BEST	1.129793			
23	WORST	5.519475			
24	AVERAGE	3.0955934			
25	STD. DEV	1.19132217			

Figure 10: GA on Rastrigin-2-D

Genetic Algorithm on Rastrigin Function-10-Dimensions:

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	PERFORMANCE	DIM-1	DIM-2	DIM-3	DIM-4	DIM-5	DIM-6	DIM-7	DIM-8	DIM-9	DIM-10	ITERATIONS	TIME (seconds)
2	8.681161	-0.0014604	0.00168668	-0.0006174	-0.0002466	-0.0034098	0.01242843	-0.0019799	0.00161948	0.00023292	3.38E-05	883	357
3	9.70115	-0.0003266	0.00073622	-0.0002208	-0.0046612	7.03E-05	-0.0012891	-0.0001338	-0.0013084	0.00405733	0.00057884	998	347
4	10.011021	0.00402525	-0.0019159	0.00421829	-0.0024461	-0.0002892	-0.0007936	0.00343196	0.00167755	-0.0013568	0.00048736	812	333
5	8.428581	-0.0014572	0.00109327	0.00356602	-0.000817	0.00069985	-0.0119845	-0.0061881	-0.0014418	-0.0002398	0.00137891	998	341
6	14.672402	-0.00067328	0.000371183	-0.00476729	0.003101358	-0.00428250	-0.00160715	0.001216673	-0.00047426	-0.00260493	0.0038611	912	332
7	7.819232	0.00225299	-0.0019269	-0.0005181	-0.0018101	0.00020078	0.00138634	0.00165216	-0.0007073	0.00036249	-0.0011897	827	338
8	8.553895	-0.0002145	0.00084042	0.00285496	0.00144556	-0.0025222	0.00025819	-1.10E-06	0.00026945	-0.0017338	0.00192033	954	335
9	8.503633	-0.0015655	-0.0017252	-3.52E-07	-0.0013862	0.00086374	-0.0012088	-0.000833	-0.0011981	0.0005849	0.00079751	892	345
10	8.651429	0.00196597	-0.0008416	-0.0004032	0.00093253	0.00142819	0.00224571	-0.0009413	0.00148084	-0.0013745	-0.0025136	904	357
11	15.326374	0.01161616	-0.0097905	0.00397153	-0.0074136	-0.0216327	0.01056218	0.00205672	-0.0151496	0.00114655	-0.0138126	894	354
12	15.914818	0.00096961	-0.0080839	0.0135418	-0.0040755	0.01734025	0.00011151	-0.0021082	0.01377067	-0.0228252	0.01077409	905	349
13	15.256202	-0.0034835	-0.0196033	0.00597235	-0.0012016	0.01153685	-0.0189034	0.01219082	0.00281446	-0.0030839	0.00140296	937	355
14	16.297696	0.00025128	7.12E-05	0.00096613	-0.0104107	-0.0087294	-0.0093768	-0.0023332	0.02531903	0.00430573	0.01359114	958	317
15	16.41473	0.00878268	0.01469177	0.00353502	-0.002196	0.00570496	-0.0091243	-0.0157576	0.0013688	-0.0012593	0.00502424	845	305
16	15.868718	-0.0030779	-0.0090301	-0.0011687	-0.0185432	0.00125603	0.00288537	-0.0051849	-0.0004303	-0.0132245	0.00564603	956	343
17													
18													
19													
20													
21													
22	BEST	7.819232											
23	WORST	16.41473											
24	AVERAGE	12.0067361											
25	STD. DEV	3.62697705											

Figure 11: GA on Rastrigin-10-D

Particle Swarm Optimization on Sphere Function-2-Dimensions:

PERFORMAN	DIM-1	DIM-2	ITERATIONS	TIME (seconds)
0.01033317	-0.0141447	0.0031242	951	204
0.01033317	-0.0141447	0.0031242	816	227
0.00453538	0.00626378	0.00125962	819	226
0.00453538	0.00626378	0.00125962	816	170
0.00453538	0.00626378	0.00125962	859	189
0.01033317	-0.0141447	0.0031242	951	189
0.00453538	0.00626378	0.00125962	910	177
0.00141418	-0.0019975	-6.32E-06	874	160
0.00453538	0.00626378	0.00125962	974	185
0.00141418	-0.0019975	-6.32E-06	859	174
0.00141418	-0.0019975	-6.32E-06	835	180
0.00453538	0.00626378	0.00125962	928	229
0.00141418	-0.0019975	-6.32E-06	969	169
0.00453538	0.00626378	0.00125962	980	186
0.00141418	-0.0019975	-6.32E-06	915	176

BEST	0.00141418
WORST	0.01033317
AVERAGE	0.00465454
STD. DEV	0.00297152

Figure 12: PSO on Sphere-2-D

Particle Swarm Optimization on Sphere Function-10-Dimensions:

PERFORMANCE	DIM-1	DIM-2	DIM-3	DIM-4	DIM-5	DIM-6	DIM-7	DIM-8	DIM-9	DIM-10	ITERATIONS	TIME (seconds)
-52.3261573	-0.2886796	-0.4110168	-1.2807931	-0.8328798	-0.0760306	-0.2470847	0.28790082	-1.280172	-0.0636256	-0.0015219	893	314
-77.03102941	-0.3522932	-0.5212038	0.40919157	-0.204329	1.13048629	-0.054802	-0.1213445	-0.7217496	-0.4226325	-0.1240379	935	325
-77.03102941	-0.3522932	-0.5212038	0.40919157	-0.204329	1.13048629	-0.054802	-0.1213445	-0.7217496	-0.4226325	-0.1240379	947	354
-119.0253656	0.1920824	0.1681016	-0.0322747	0.12873423	0.01382064	0.35666254	0.30758713	-0.1228919	-0.6629622	-0.1894011	935	300
-119.0253656	0.1920824	0.1681016	-0.0322747	0.12873423	0.01382064	0.35666254	0.30758713	-0.1228919	-0.6629622	-0.1894011	858	303
-77.03102941	-0.3522932	-0.5212038	0.40919157	-0.204329	1.13048629	-0.054802	-0.1213445	-0.7217496	-0.4226325	-0.1240379	951	319
-119.0253656	0.1920824	0.1681016	-0.0322747	0.12873423	0.01382064	0.35666254	0.30758713	-0.1228919	-0.6629622	-0.1894011	949	331
-119.0253656	0.1920824	0.1681016	-0.0322747	0.12873423	0.01382064	0.35666254	0.30758713	-0.1228919	-0.6629622	-0.1894011	810	307
-52.3261573	-0.2886796	-0.4110168	-1.2807931	-0.8328798	-0.0760306	-0.2470847	0.28790082	-1.280172	-0.0636256	-0.0015219	880	351
-77.03102941	-0.3522932	-0.5212038	0.40919157	-0.204329	1.13048629	-0.054802	-0.1213445	-0.7217496	-0.4226325	-0.1240379	998	321
-119.0253656	0.1920824	0.1681016	-0.0322747	0.12873423	0.01382064	0.35666254	0.30758713	-0.1228919	-0.6629622	-0.1894011	845	351
-52.3261573	-0.2886796	-0.4110168	-1.2807931	-0.8328798	-0.0760306	-0.2470847	0.28790082	-1.280172	-0.0636256	-0.0015219	861	334
-77.03102941	-0.3522932	-0.5212038	0.40919157	-0.204329	1.13048629	-0.054802	-0.1213445	-0.7217496	-0.4226325	-0.1240379	853	315
-119.0253656	0.1920824	0.1681016	-0.0322747	0.12873423	0.01382064	0.35666254	0.30758713	-0.1228919	-0.6629622	-0.1894011	894	344
-77.03102941	-0.3522932	-0.5212038	0.40919157	-0.204329	1.13048629	-0.054802	-0.1213445	-0.7217496	-0.4226325	-0.1240379	877	319

BEST	-119.02537
WORST	-52.326157
AVERAGE	-88.887789
STD. DEV	27.1285253

Figure 13: PSO on Sphere-10-D

Differential Evolution on Sphere Function-2-Dimensions:

PERFORMANCE	DIM-1	DIM-2	ITERATIONS	TIME (seconds)
0.007636838	0.07418719	-0.0461855	958	176
0.004316778	-0.0467298	-0.0461855	991	220
0.004316778	-0.0467298	-0.0461855	970	150
0.004316778	-0.0467298	-0.0461855	991	171
0.004316778	-0.0467298	-0.0461855	854	215
0.001533246	0.02590879	-0.0293595	997	223
0.001533246	0.02590879	-0.0293595	913	220
0.001533246	0.02590879	-2.94E-02	976	223
0.001533246	0.02590879	-0.0293595	896	183
8.58E-05	0.00191364	-9.07E-03	890	175
8.58E-05	0.00191364	-9.07E-03	883	193
8.58E-05	0.00191364	-0.0090656	987	225
8.58E-05	0.00191364	-9.07E-03	998	177
5.04E-05	0.00315999	-0.0063579	898	209
5.90E-06	0.00224791	-9.21E-04	915	161

BEST	5.9015E-06
WORST	0.00763684
AVERAGE	0.00209578
STD. DEV	0.00182955

Figure 14: DE on Sphere-2-D

Differential Evolution on Sphere Function-10-Dimensions:

PERFORMANCE	DIM-1	DIM-2	DIM-3	DIM-4	DIM-5	DIM-6	DIM-7	DIM-8	DIM-9	DIM-10	ITERATIONS	TIME (seconds)
2.028101258	0.6704921	0.0499496	0.42481341	0.09746	-0.735143	-0.3501047	-0.351139	-0.0694075	0.63506003	0.43778525	894	351
1.718307703	-0.1532146	-0.4773569	0.20734944	-0.2914934	-0.3550482	0.00330094	-0.8717375	-0.2688465	-0.0709955	-0.6129323	986	336
1.084643109	-0.2896526	-0.4773569	0.20734944	0.08226337	0.08871291	0.19010649	-0.3880115	-0.2656239	-0.2868944	-0.6129323	975	304
0.872803942	0.03192185	0.1485549	0.16366031	-0.0530223	-0.3667164	-0.0932166	0.3881018	-0.3504147	-0.6200957	0.13790538	986	317
0.872803942	0.03192185	0.1485549	0.16366031	-0.0530223	-0.3667164	-0.0932166	0.3881018	-0.3504147	-0.6200957	0.13790538	938	325
0.318260893	0.03192185	-0.0958423	0.16366031	-0.0530223	0.2498688	-0.0932166	0.3881018	-0.1556111	0.06548261	0.16795994	843	312
0.318260893	0.03192185	-0.0958423	0.16366031	-0.0530223	0.2498688	-0.0932166	0.3881018	-0.1556111	0.06548261	0.16795994	964	304
0.310722997	-0.3188868	-0.0958423	0.16366031	-0.0530223	0.32759557	-0.0932166	0.04069905	-0.1556111	0.01274605	0.16795994	961	317
0.310722997	-0.3188868	-0.0958423	0.16366031	-0.0530223	0.32759557	-0.0932166	0.04069905	-0.1556111	0.01274605	0.16795994	931	310
0.310722997	-0.3188868	-0.0958423	0.16366031	-0.0530223	0.32759557	-0.0932166	0.04069905	-0.1556111	0.01274605	0.16795994	921	332
0.310722997	-0.3188868	-0.0958423	0.16366031	-0.0530223	0.32759557	-0.0932166	0.04069905	-0.1556111	0.01274605	0.16795994	949	309
0.310722997	-0.3188868	-0.0958423	0.16366031	-0.0530223	0.32759557	-0.0932166	0.04069905	-0.1556111	0.01274605	0.16795994	874	308
0.310722997	-0.3188868	-0.0958423	0.16366031	-0.0530223	0.32759557	-0.0932166	0.04069905	-0.1556111	0.01274605	0.16795994	914	357
0.310722997	-0.3188868	-0.0958423	0.16366031	-0.0530223	0.32759557	-0.0932166	0.04069905	-0.1556111	0.01274605	0.16795994	944	320
0.310722997	-0.3188868	-0.0958423	0.16366031	-0.0530223	0.32759557	-0.0932166	0.04069905	-0.1556111	0.01274605	0.16795994	828	348

BEST	0.310723
WORST	2.02810126
AVERAGE	0.64659771
STD. DEV	0.43186217

Figure 15: DE on Sphere-10-D

Genetic Algorithm on Sphere Function-2-Dimensions:

PERFORMAN	DIM-1	DIM-2	ITERATIONS	TIME (seconds)
0.006031	0.02731402	0.00314857	918	186
0.098876	0.00325733	-0.0086354	872	166
0.040693	0.00094705	0.00090878	812	180
0.171818	0.00090562	0.00032712	872	167
0.19003	-0.0009501	-1.77E-03	981	208
0.138657	0.00278058	0.00156539	813	174
0.080647	-0.0010343	4.64E-05	873	190
0.188584	0.00054731	-0.0006641	1000	194
0.143247	0.00028613	3.81E-06	856	219
2.68E-01	-3.13E-04	-4.40E-05	974	154
2.13E-01	-2.75E-04	-0.0001502	926	176
1.60E-01	-0.0002437	-5.12E-05	956	207
4.08E-01	-0.0005358	0.00121683	974	151
3.50E-01	-0.0001549	-0.0003624	804	221
2.52E-01	6.41E-05	5.45E-04	863	221

BEST	0.006031
WORST	0.407679
AVERAGE	0.180567
STD. DEV	0.10068446

Figure 16: GA on Sphere-2-D

Genetic Algorithm on Sphere Function-10-Dimensions:

PERFORMANCE	DIM-1	DIM-2	DIM-3	DIM-4	DIM-5	DIM-6	DIM-7	DIM-8	DIM-9	DIM-10	ITERATIONS	TIME (seconds)
0.087727	0.00182658	0.01290853	-0.0151298	-0.0094092	-0.0010616	0.00250581	0.01016298	0.0016643	0.00966509	5.30E-03	857	325
0.204866	-0.001058	-0.0007276	-0.0027715	-8.30E-05	-1.08E-04	-0.0034242	0.00015948	0.0109706	0.00088952	-0.0013607	937	345
0.343519	-0.0003547	-0.0083622	0.00235162	0.00024575	0.00042267	-2.26E-05	-0.0156445	0.00019998	0.00151647	-0.0045632	801	359
0.404751	0.00531359	-0.0030129	0.00019103	0.00136348	-0.000665	0.00388142	-0.0022225	-0.0013534	0.0017474	0.00023757	937	337
0.454151	-0.0028068	0.00019637	-0.0061675	-0.0026464	-0.004315	-0.0053645	-0.000226	-0.0004893	0.0011228	0.00068394	921	335
0.757962	-0.0007825	0.00035243	0.00091255	-0.000205	0.00131581	-0.001967	-0.0005462	0.00044681	-0.0003686	0.00117289	978	300
0.617397	0.00041891	0.00389759	0.00218346	-0.0006974	2.81E-05	0.00222709	-7.36E-04	0.00057261	-0.0013093	0.00091975	846	337
0.721225	-0.0018865	-0.0013676	3.70E-04	-0.0004961	0.00107292	0.00045815	2.05E-05	-0.0008153	-0.00671	-0.000202	845	359
0.776884	0.00443542	-0.0001643	0.00574092	-0.0017829	-0.0002341	0.00364681	0.0011437	0.00196533	0.0019568	0.00064957	806	337
0.704821	0.00642068	-0.0003529	-0.0025204	-0.0036534	-0.0022993	0.00015817	-0.0039185	0.00065201	-0.0004073	-0.0040132	910	357
0.723286	0.00038748	0.00091514	0.00065016	0.00058306	-0.0002441	0.00026402	0.00045791	0.00015439	0.00147392	7.62E-05	833	354
0.723286	0.00038748	0.00091514	0.00065016	0.00058306	-0.0002441	0.00026402	0.00045791	0.00015439	0.00147392	7.62E-05	973	328
0.723286	0.00038748	9.15E-04	0.00065016	0.00058306	-0.0002441	0.00026402	0.00045791	0.00015439	0.00147392	7.62E-05	966	314
0.723286	0.00038748	0.00091514	0.00065016	0.00058306	-0.0002441	0.00026402	0.00045791	0.00015439	0.00147392	7.62E-05	925	333
1.914622	-0.003499	-0.0099031	-0.0010169	-0.0025625	0.00314751	-0.009056	0.00250579	0.00186382	-0.0014836	-0.0038827	957	347

BEST	0.087727
WORST	1.914622
AVERAGE	0.65873793
STD. DEV	0.39359196

Figure 17: GA on Sphere-10-D

RESULT ANALYSIS:

	A	B	C	D	E	F	G	H	I
1	perf/opt	PSO-rast-2D	DE-rast-2D	GA-rast-2D		perf/opt	PSO-sph-2D	DE-sph-2D	GA-sph-2D
2	BEST	0.01211263	0.12774593	1.129793		BEST	0.00141418	5.9015E-06	0.006031
3	WORST	1.15917264	8.84706971	5.519475		WORST	0.01033317	0.00763684	0.407679
4	AVERAGE	0.5691829	5.11089222	3.0955934		AVERAGE	0.00465454	0.00209578	0.180567
5	STD. DEV	0.48059819	3.46920726	1.19132217		STD. DEV	0.00297152	0.00182955	0.10068446
6		BEST	WORST					BEST	WORST
7									
8	perf/opt	PSO-rast-10D	DE-rast-10D	GA-rast-10D		perf/opt	PSO-sph-10D	DE-sph-10D	GA-sph-10D
9	BEST	21.6310679	93.9448194	7.819232		BEST	-119.02537	0.310723	0.087727
10	WORST	67.75747	130.834435	16.41473		WORST	-52.326157	2.02810126	1.914622
11	AVERAGE	43.1491928	110.50201	12.0067361		AVERAGE	-88.887789	0.64659771	0.65873793
12	STD. DEV	13.4613838	9.32980055	3.62697705		STD. DEV	27.1285253	0.43186217	0.39359196
13			WORST	BEST			BEST	WORST	
14									

Figure 18: Result Analysis

The code and methods' implementation is provided in *Appendix-A (SPO-Rastrigin), Appendix-B (DE-Rastrigin), Appendix-C (GA-Rastrigin), Appendix-D (SPO-Sphere), Appendix-E (DE-Sphere),* and *Appendix-F (GA-Sphere)*. After executing each optimization algorithm for *30-50* generations, on Rastrigin and Sphere function, the below-mentioned analysis was composed.

RASTRIGIN FUNCTION:

Best Performance on the Rastrigin Function-2-D: Particle Swarm Optimization performed really well with the fitness of 0.01211263 (Standard Dev: 0.48059819).

Best Performance on the Rastrigin Function-10-D: Genetic Algorithm performed really well with the fitness of 7.819232 (Standard Dev: 3.62697705).

Worst Performance on the Rastrigin Function-2-D: Differential Evolution performed really bad with the fitness of 8.84706971 (Standard Dev: 3.46920726).

Worst Performance on the Rastrigin Function-10-D: Differential Evolution performed really bad with the fitness of 130.834435 (Standard Dev: 9.32980055).

SPHERE FUNCTION:

Best Performance on the Sphere Function-2-D: Differential Evolution performed really well with the fitness of 5.9015E-06 (Standard Dev: 0.00182955).

Best Performance on the Sphere Function-10-D: Particle Swarm Optimization performed really well with the fitness of -119.02537 (Standard Dev: 27.1285253).

Worst Performance on the Sphere Function-2-D: Genetic Algorithm performed really bad with the fitness of 0.407679 (Standard Dev: 0.10068446).

Worst Performance on the Sphere Function-10-D: Differential Evolution performed really bad with the fitness of 2.02810126 (Standard Dev: 0.43186217).

Task 3

Bin Packing-1-Dimensional:

Let bin capacity be C, N is the number of bins, M is the set of items, where items = {i1, i2, iM}, and {S1, S2, ⋯, SM} is the integer sizes.

Definition of the problem:

The bin packing problem (BPP) is defined as follows:

Given a non-negative number C and a set of M items of integer size S1, the bin packing problem refers to packing given items into the bins such that the total size of items in any given bin doesn't exceed the capacity C and the number of bins N used is kept to a minimum.

Complexity Class Membership:

This problem resides in NP-Hard complexity class.

Existing Solutions:

The existing solutions to the BPP include the first-fit decreasing method (FFD) and best-fit decreasing method (BFD) which are popular heuristic methods. The computation of these methods works in such a way that the items are sorted in descending order, and then the methods applied for the items in turn.

Implementation:

The implemented Genetic Algorithm (GA) the set of item numbers (brick sizes) in a bin was made to act as a "gene". The genotype is expressed by sequencing the item sets for all the bins. For instance, a situation where *In = 15*, an example genotype *g1* would look as follows:

$$g_1 : (1, 3, 10)(2, 9, 11)(5, 7, 13, 15)(4, 6, 14)(8, 12)$$

The above *g1* means that (In1, In3, In10), (In2, In9, In11), (In5, In7, In13, In15), (In4, In6, In14) and (In8, In12) are assigned to B1, B2, B3, B4, and B5, respectively. The number of genes is variable.

Even if the order of the genes is changed, the genotype decodes to the same solution. For example, a genotype

$$g_2 : (8, 12)(2, 9, 11)(5, 7, 13, 15)(4, 6, 14)(1, 3, 10)$$

The above genotype shows the exact same genotype with the same set of genes as in *g1* but in a variable order, despite this, they produce the same solution.
The solution is independent of the order of the genes like in the case of sets.

Solution Proposal:

Existing solutions in GA are random generations specifically aimed at optimization problems. To obtain better initial solutions, the FF (First-Fit) was incorporated into the GA. The items are first sequenced, and then the FF applied for each item. The sequencing is done at random in this generation process to generate unprobeable, unexpected solutions.

Crossover**:**

This operation is designed in a way that offspring inherit important factors of the parents. An imperative aspect of BPP is the set of items. To incur a better solution, local optimization and heuristic rules such as FF (First-Fit) and MBS (Minimum Bin Slack)' should be introduced to the crossover operation of the GA.

In this crossover operation, offspring c1 and c2 are generated from parents P1 and P2. Initially, a few bins (genes) are randomly chosen from P1 and the items inside them are replicated in c1, then from P2, only those bins are picked which don't contain the same items as P1's bins that are replicated in c1. The remaining sets of P2 are replicated to St (temporary set), excluding the items similar to the items of the P1's sets that were copied to c1. The sets in St are not replicated in c1 because the quantity of the bricks (items) in St is quite less. The St sets are divided into two sets, St_A and St_B, St_A contains all the sets which just have one item in them and rest of the sets are stored in $St_{B.}$

Mutation**:**

This procedure works in such a way that 2 to 3 bins (gene equivalents) in the parent are randomly chosen. The items in the chosen bins are copied to *Stm* individually, that is, as item-sets containing a single item.

The remaining bins are copied to the offspring. Then, the replacement procedure between the offspring and *Stm* is executed such that is uses in the crossover operation. Lastly, a two-phase procedure is executed to get the items remaining in the *Stm* to be used in the crossover operation.

Loss Function:

$$\sum_{i=0}^{n} Ri + S$$

R: Remaining size of the i^{th} bin

S: Total size of the bin (new bin)

The solution should be considered good if the output of above-mentioned function gets smaller with each iteration. To calculate the Ri, the below-mentioned formula is used.

$$Ri = B - \sum_{i=0}^{n} Oi$$

B: Current size of the particular bin

O: i^{th} Brick's (item) size

Note: As long as O >= B > 0, new bin won't be added

The proposed fitness function should work in each situation since the number of bins doesn't affect the solution, it recommences with each bin-packing iteration. For each bin, the remaining size is calculated after a new bin is created and added to the size of the new bin, which, as a result, should give an approximate sum of total remaining space.

Bin Packing-2-Dimensional:

The same implementation can be applied to the 2-D BPP, however, some changes need to be made with the current genotypes. The purpose is to pack 2D bins (rectangles) with random 2D shapes while avoiding extensions amongst the shapes and large unused spaces on rectangular bounds.

Instead of a simple item number (brick sizes), the sets should contain the coordinates of the bricks with the consideration of (0,0) bottom-left-corner, so the figure is definable with respect to three parameters, coordinates (x, y) and orientation angle (Q). Furthermore, the mutation should be enhanced in such a way that it considers the distance between shapes and their corresponding sizes.

References

Benchmarks — DEAP 1.2.2 Documentation (n.d.) available from <http://deap.readthedocs.io/en/master/api/benchmarks.html#deap.benchmarks.sphere> [15 April 2018]

Iima, H. and Yakawa, T. (2003) A New Design Of Genetic Algorithm For Bin Packing [online] available from <https://ieeexplore.ieee.org/document/1299783/> [22 April 2018]

Rastrigin Function (n.d.) available from <https://www.sfu.ca/~ssurjano/rastr.html> [17 April 2018]

Sphere Function (n.d.) available from <https://www.sfu.ca/~ssurjano/spheref.html> [24 April 2018]

List Of Figures

Appendices

Appendix-A

```python
# Created By Huzefa Shaikh

import sys
import random
import math
import numpy as np

'''dimension'''
dim = 2
perfRes = []
#Particle-count-in-Swarm
particle_no = 15

# Bounds-on-positions-velocities
v_max = 20
v_min = -20
p_min = -32
p_max = 32

# updates_no
cmax = 1000

# amt-of-dampen-velocity-on-updates
dampener = 1
dampen_rate = 1

# Leaving original variables distinct
orig_dampen_rate = dampen_rate
orig_dampener = dampener

# Function-to-optimize (minimize)
def F(x):
  global dim
  D = dim
  summation = 0
  # D-dimensional Rastrigin Function

  i = 0
  summation = D*10
  while i < D:
    summation += x[i]**2 - 10 * math.cos(2 * math.pi * x[i])
    i = i + 1
  return summation

# main-function-construction-of-swarm-optimization

def main():


```

```
  for x in range(15):
    global cmax, dampener, dampen_rate, dim

    #running-multiple-iterations-with-while-loop
    dampen_rate = orig_dampen_rate
    dampener = orig_dampener
    # Construct the swarm
    swarm = []
    i = 0
    while i < particle_no:
      swarm.append(Particle())
      i = i + 1

    # Init-best-position-velocity-error
    best_pos = []
    worst_pos = []
    all_pos = []
    AvgX = 0
    AvgXc = []
    best_velocity = []
    best_err = -1
    worst_err = -1
    nerr = []
    # Run-updates-swarm-and-output-best-position/error
    i = 0
    while i <= cmax:
      # Iterate-swarm-and-evaluation-of-swarm-positions-on-the-function
      j = 0
      while j < len(swarm):
        err = swarm[j].Evaluate()
        #nerr.append(err)
        # If-particle-perform-better
        # Save-particle-position-velocity-error
        if err < best_err or best_err == -1:
          best_pos = []
          best_velocity = []
          k = 0
          while k < dim:
            best_pos.append(swarm[j].pos[len(swarm[j].pos)-1][k])

            best_velocity.append(swarm[j].velocity[len(swarm[j].velocity)-1][k])
            k = k + 1
          best_err = err
        j = j + 1

      # Update-swarm-based-on-new-positions
      j = 0
      while j < len(swarm):
```

```
      while j < len(swarm):
        swarm[j].UpdateVelocity(best_pos)
        swarm[j].UpdatePosition()
        j = j + 1

      dampener = dampener * dampen_rate # Dampen the velocity
      i = i + 1

    # Output-stats

    #print 'performance of 15 run'
    perfRes.append(best_err)

    #print 'Run: ', x
    #print '\nBest-performance: ', best_err,' ---BEST-Positions: ', best_pos
    print best_err, best_pos

    #print '\nworst-performance: ', worst_err,' ---WORST-Positions: ', worst_pos


  # the below attributes are instances of each particle:

  # minimization

  '''
  err: current position's error
  best_pos: location of particle seen lowest error
  best_err: lowest error
  pos: particle seen location list(past positions are recyclable)
  velocity: The list of particle velocities
  '''
  nRess = np.array(perfRes)
  print '\nOver all best performance: ', min(perfRes)
  print '\nOver all worst performance: ', max(perfRes)
  print '\nAverage performance: ', np.mean(nRess)
  print '\n'

  return

class Particle:
  def __init__(self):
    global dim
    # this function sets up each particle
    # we can initialize the position and velocity of the particles
    # using the InitPosition() and InitVelocity() functions
    self.err = 0
```

```
    self.err = 0
    self.best_pos = []
    self.best_err = -1 # this is set to -1 so we update after the first step
    self.pos = []
    self.velocity = []

    # Since we are operating in a potentially multi-dimensional space
    # we have to run through each of the positions, initializing the
    # positions and velocities for each dimension
    temp_pos = []
    temp_velocity = []
    j = 0
    while j < dim:
      temp_pos.append(self.InitPosition())
      temp_velocity.append(self.InitVelocity())
      self.best_pos.append(0) # initialize the best position array
      j = j + 1
    self.pos.append(temp_pos)
    self.velocity.append(temp_velocity)

  # Evaluate the performance of each particle
  # The current position of the particle is the last
  # array in the position array.
  def Evaluate(self):
    global dim
    # The F function that we are trying to minimize
    self.err = F(self.pos[len(self.pos)-1])
    if self.best_err == -1 or self.err < self.best_err:
      self.first_update = False
      self.best_err = self.err
      self.best_pos = []
      j = 0
      while j < dim:
        self.best_pos.append(self.pos[len(self.pos)-1][j])
        j = j + 1
    return self.err
  # Initialize the position of the particle between -30 and 30
  # for each dimension
  def InitPosition(self):
    temp = 30*random.random()
    if random.random() > 0.5:
      temp = -1 * temp
    if temp > p_max:
      return p_max
    elif temp < p_min:
      return p_min
    return temp
```

```python
# Initialize the velocity of the particle between 1 and -1
# for each dimension
def InitVelocity(self):
  if random.random() > 0.5:
    return random.random()
  return -1*random.random()

# A function that is used to randomize the cognitive term
def RandomizeCognitive(self):
  return random.random()

# A function that is used to randomize the social term
def RandomizeSocial(self):
  return random.random()

# A function that is used to update the velocity
# of the particle the particle's past and the global best position seen
def UpdateVelocity(self, global_best_pos):
  global v_max, dampener, dim
  # w is a control parameter that tells the particle
  # how much to discount the previous velocity
  w = 1
  # c1 is a control parameter that tells the particle
  # how much to weight its own previous positions
  c1 = 2
  # c2 is a control parameter that tells the particle
  # how much to weight the swarms best best position
  c2 = 2
  # r1 and r2 are random numbers that weight the
  # cognitive and social terms
  r1 = self.RandomizeCognitive()
  r2 = self.RandomizeSocial()

  t = len(self.velocity)

  # Construct the new velocity for the particle
  new_velocity_arr = []
  j = 0
  while j < dim:
    # Apply the control parameters to the particle's previous velocity
    # in the direction that we are working on
    v_term = dampener*w*self.velocity[t-1][j]

    # Create the cognitive and social terms
    own_term = c1 * r1 * (self.best_pos[j] - self.pos[t-1][j])
    social_term = c2 * r2 * (global_best_pos[j] - self.pos[t-1][j])
    # Add the velocities together to make the new velocity
    new_velocity = v_term + own_term + social_term
```

```python

            # If the velocity is larger than the max velocity, decrease it
            # If the velocity is smaller than the min velocity, increase it
            if new_velocity > v_max:
                new_velocity = v_max
            elif new_velocity < v_min:
                new_velocity = v_min
            new_velocity_arr.append(new_velocity)
            j = j + 1

        self.velocity.append(new_velocity_arr)

    # Update the particle's position based on its previous velocity and position
    def UpdatePosition(self):
        global p_max, p_min, dim
        t1 = len(self.velocity)
        t2 = len(self.pos)

        new_position_arr = []

        j = 0
        while j < dim:
            new_position = self.pos[t2-1][j] + self.velocity[t1-1][j]
            # If the position is smaller or larger than the bounds, change them
            if new_position > p_max:
                new_position = p_max
            elif new_position < p_min:
                new_position = p_min
            new_position_arr.append(new_position)
            j = j + 1
        self.pos.append(new_position_arr)

main()
```

Appendix-B

```
# Created by Huzefa Shaikh

import math
import random
import numpy as np
#math functions & constants
sin=math.sin
cos=math.cos
sqrt=math.sqrt
pi=math.pi
fabs=math.fabs

#FIX (no globals)
dim=2
Xu=[]
Xl=[]
pop=[]
fvals=[]
num_fe=0 #Function-evaluation-count-total-number
max_gen=0 #number of generations
NP=15 # number of iterations (population as well)
cr=0.90 #crossover-probability
F=0.90 #Scaling-factor
U=[] #trial-vector

f_best=-1
f_worst=1
#util function- return a random real in (0.0,1.0)
def rand_n():

    return random.random()

#function objective
def func(X):
    global num_fe
    sum=0

    for i in range(0,dim):
        sum = sum + X[i]*X[i];

    num_fe=num_fe+1

    return sum

# Control parameters
def setup():

    global max_gen,dim,Xu,Xl,NP,f_best,f_worst

```

```
    max_gen = input("Enter the number of runs:: ")
    dim=input("Enter the dimension of the problem:: ")

#     for i in xrange(0,dim):
#         print "Enter the lower and upper bound of %d th variable" %i
#         Xl.insert(i,input())
#         Xu.insert(i,input())

    print ("Enter the lower and upper bound of variables: ")
    l = input()
    u = input()
    for i in range(0,dim):
        Xl.insert(i,l)
        Xu.insert(i,u)

    #NP=20*dim #population size

    # Open the file to store the best individual of every generation
    f_best=open("best-population.out","w")
    #f_worst=open("worst-population.out","w")

#Initialize-population
def initpop():

    global pop,fvals,num_fe

    pop=[]
    fvals=[]

    for i in range(0,NP):
        X=[]
        for j in range(0,dim):
            #fill-up-X-add-population
            X.insert(j,(Xl[j] + (Xu[j]-Xl[j])*rand_n()))


        #bounds-check
        for j in range(0,dim):
            while X[j] < Xl[j] or X[j] > Xu[j]:
                if X[j]<Xl[j]:
                    X[j]=2*Xl[j]-X[j]
                if X[j]>Xu[j]:
                    X[j]=2*Xu[j]-X[j]


        pop.insert(i,X)
        fvals.insert(i,func(X)) #function-evaluation
```

```
#DE/rand/1
def evolve_de_rand_1():

    global pop,fvals

    for i  in range(0,max_gen):

        #Write the best individual of this generation into a file

        #best_pop.out
        write_best()
        for j in range(0,NP):


            #MUTATION

            while 1:
                r1=random.randint(0,NP-1)
                if r1!=j:
                    break

            while 1:
                r2=random.randint(0,NP-1)
                if r2!=r1 and r2!=j:
                    break

            while 1:
                r3=random.randint(0,NP-1)
                if r3!=r2 and r3!=r1 and r3!=j:
                    break

            U=[]
            for k in range(0,dim):
                #if rand_n() <= cr and k == dim_rand:
                U.insert(k,(pop[r3])[k] + F*((pop[r1])[k]-(pop[r2])[k]))

            #CROSSOVER
            n = int(rand_n()*dim)
            L=0
            while 1:
                L=L+1
                if rand_n() > cr or L>dim:
                    break

            for k in range(0,dim):
                for kk in (n,n+L):
                    if k != (kk % dim):
                        U.insert(k,(pop[j])[k])
```

```

            #BOUNDS-CHECK
            for k in range(0,dim):
                while U[k] < Xl[k] or U[k] > Xu[k]:
                    if U[k]<Xl[k]:
                        U[k]=2*Xl[k]-U[k]
                    if U[k]>Xu[k]:
                        U[k]=2*Xu[k]-U[k]


            U.insert(dim,func(U)) #the function value (the-last-value)

            #SELECTION
            #Comparing the trial vector and past individual
            if U[dim] <= fvals[j]:
                for k in range(0,dim):
                  (pop[j])[k]=U[k]
                fvals.insert(j,func(pop[j]))
```

```
#Find the best objective func. value and write it to the file

#accessed all generation
def write_best():
    best_val=fvals[0]
    worst_val=fvals[0]
    best_index=0
    for i in range(0,NP):
        if fvals[i] < best_val:
            best_index=i
            best_val=fvals[i]
        else:
            best_index=i
            worst_val=fvals[i]

    f_best.write(str(best_val))
    f_best.write('\n')
    #f_worst.write(str(worst_val))
    #f_worst.write('\n')
#Report the best pop and save the population
#STATs
def report():

    #Save the final population to the file
    f=open("final_population.out","w")
    f.write("Final Population Data: Variable Values -- Objective Function Values\n")
    for i in range(0,NP):
        for j in range(0,dim):
            f.write(str((pop[i])[j]) + '\t')
        f.write('\t\t|| ')
        f.write(str(fvals[i]))
        f.write('\n')
    f.close()

    #Find the best individual and report
    best_val=fvals[0]
    best_index=0
    worst_index=0
    results = []
    popss = []
    for i in range(0,NP):
        popss.append(pop[i])
        results.append(fvals[i])
        if fvals[i] < best_val:
            best_index=i
            best_val=fvals[i]

```

```
    for wi in range(0,NP):
        if fvals[wi] > best_val:
            worst_index=wi
            best_val=fvals[wi]

    nres = np.array(results)


    print results,',',popss
    print '\nBest : ',fvals[best_index],'--:--',pop[best_index]
    print '\nWorst : ',fvals[worst_index],'--:--',pop[worst_index]
    print '\naverage : ',np.mean(nres)

    print 'function-evaluations-total-number : ', num_fe


if __name__ =='__main__':

    print("Differential-Evolution->>>")
    print("---------------------------------------------")
    setup()
    initpop()
    print("Evolution-process-running..")
    evolve_de_rand_1()
    print("\nSUMMARY")
    print("-----------------------\n")
    report()
```

Appendix-C

```
# Created by Huzefa Shaikh

from pyevolve import Mutators, Initializators
from pyevolve import Selectors
import math
from pyevolve import GSimpleGA
from pyevolve import G1DList
from pyevolve import Consts
import numpy as np
#Rastrigin function intitiation
def rast(gen):
    total_ret = 0
    len_genome = len(gen)
    for i in range(len_genome):
        total_ret += gen[i]**2 - 10*math.cos(2*math.pi*gen[i])
    return (10*len_genome) + total_ret

def algorith_run():

  lr = 0.06
  results = []
  for x in range(15):

    #Genome structure
    gen = G1DList.G1DList(2)
    gen.setParams(rangemin=-5.2, rangemax=5.30, bestrawscore=0.00, rounddecimal=2)
    gen.initializator.set(Initializators.G1DListInitializatorReal)
    gen.mutator.set(Mutators.G1DListMutatorRealGaussian)

    gen.evaluator.set(rast)

    #Genetic Algorithm
    gen_Algo = GSimpleGA.GSimpleGA(gen)
    gen_Algo.terminationCriteria.set(GSimpleGA.RawScoreCriteria)
    gen_Algo.setMinimax(Consts.minimaxType["minimize"])
    gen_Algo.setGenerations(300)
    gen_Algo.setCrossoverRate(0.8)
    gen_Algo.setPopulationSize(50)
    gen_Algo.setMutationRate(lr)
    lr += 0.01
    gen_Algo.evolve(freq_stats=40)

    #print 'Run: ',x,'\n'
    exec_best = gen_Algo.bestIndividual()
    print exec_best
    results.append(exec_best)
    print '\n'

  minn = min(results)
  maxx = max(results)
  nresult = np.array(results)
  meann = np.mean(nresult)

  print 'Best Performacne: ', minn
  print 'Worst Performacne: ', maxx
  print 'Average of all the performance', meann

  return

if __name__ == "__main__":
    algorith_run()
```

Appendix-D

```
# Created by Huzefa

import SwarmPackagePy
import numpy as np
from SwarmPackagePy import testFunctions as tf
from SwarmPackagePy import animation, animation3D

# configure swarm-particle-optimization

result = []

for x in range(15):

    spo = SwarmPackagePy.pso(50, tf.sphere_function, -10, 10, 2, 15, w=0.5, c1=1, c2=1) # define dimensions and iteration

    #print(spo.get_Gbest())

    result.append(spo.get_Gbest())

    best_agent = result[x]
    min =  tf.ackley_function(best_agent)
    for agent in result:
        current_value = tf.ackley_function(agent)
        if min > current_value:
            min = current_value
            best_agent = agent
            print(min,',', best_agent)

```

Appendix-E

```
# Created by Huzefa Shaikh

import random
import numpy as np

newRes = []
'''EXAMPLE COST FUNCTIONS'''

def func1(x):
    '''Sphere function-using-any-bounds'''
    return sum([x[i]**2 for i in range(len(x))])

def func2(x):
    # Beale's function, use bounds=[(-4.5, 4.5),(-4.5, 4.5)], f(3,0.5)=0.
    term1 = (1.500 - x[0] + x[0]*x[1])**2
    term2 = (2.250 - x[0] + x[0]*x[1]**2)**2
    term3 = (2.625 - x[0] + x[0]*x[1]**3)**2
    return term1 + term2 + term3

#--- FUNCTIONS ----------------------------------------------------------------+


def ensure_bounds(vec, bounds):

    vec_new = []
    # cycle through each variable in vector
    for i in range(len(vec)):

        # variable exceedes the minimum boundary
        if vec[i] < bounds[i][0]:
            vec_new.append(bounds[i][0])

        # variable exceedes the maximum boundary
        if vec[i] > bounds[i][1]:
            vec_new.append(bounds[i][1])

        # the variable is fine
        if bounds[i][0] <= vec[i] <= bounds[i][1]:
            vec_new.append(vec[i])

    return vec_new


#--- MAIN ---------------------------------------------------------------------+

def main(cost_func, bounds, popsize, mutate, recombination, maxiter):

```

```
    #--- INITIALIZE A POPULATION (step #1) ----------------+

    population = []
    for i in range(0,popsize):
        indv = []
        for j in range(len(bounds)):
            indv.append(random.uniform(bounds[j][0],bounds[j][1]))
        population.append(indv)

    #--- SOLVE --------------------------------------------+

    # cycle through each generation (step #2)
    for i in range(1,maxiter+1):
        #print('Run:',i)
        gen_scores = [] # score keeping

        # cycle through each individual in the population
        for j in range(0, popsize):

            #--- MUTATION (step #3.A) ---------------------+

            # select three random vector index positions [0, popsize), not including current vector (j)
            canidates = range(0,popsize)
            canidates.remove(j)
            random_index = random.sample(canidates, 3)

            x_1 = population[random_index[0]]
            x_2 = population[random_index[1]]
            x_3 = population[random_index[2]]
            x_t = population[j]     # target individual

            # subtract x3 from x2, and create a new vector (x_diff)
            x_diff = [x_2_i - x_3_i for x_2_i, x_3_i in zip(x_2, x_3)]

            # multiply x_diff by the mutation factor (F) and add to x_1
            v_donor = [x_1_i + mutate * x_diff_i for x_1_i, x_diff_i in zip(x_1, x_diff)]
            v_donor = ensure_bounds(v_donor, bounds)

            #--- RECOMBINATION (step #3.B) ----------------+

            v_trial = []
            for k in range(len(x_t)):
                crossover = random.random()
                if crossover <= recombination:
                    v_trial.append(v_donor[k])

```

```
                else:
                    v_trial.append(x_t[k])

            #--- GREEDY SELECTION (step #3.C) -------------+

            score_trial  = cost_func(v_trial)
            score_target = cost_func(x_t)

            if score_trial < score_target:
                population[j] = v_trial
                gen_scores.append(score_trial)
                #print('   >',score_trial, v_trial)

            else:
                #print('   >',score_target, x_t)
                gen_scores.append(score_target)

        #--- SCORE KEEPING --------------------------------+

        #gen_avg = sum(gen_scores) / popsize             # current generation avg. fitness
        gen_best = min(gen_scores)                      # fitness of the best individual
        newRes.append(gen_best)
        #gen_worst = max(gen_scores)                     # fitness of the worst individual

        gen_sol = population[gen_scores.index(min(gen_scores))]     # solution of best individual

        #print('\n      > GENERATION AVERAGE:',gen_avg)
        #print('      > GENERATION WORST:',gen_worst)
        print gen_best,',',gen_sol

    NnewRes = np.array(newRes)

    genn_sol = population[gen_scores.index(min(newRes))]
    #gennw_sol = population[gen_scores.index(max(newRes))]

    print '\nBest from over all run: ', min(newRes),'#####',genn_sol
    print '\nWorst from over all run: ', max(newRes)
    print '\nAverage: ', np.mean(NnewRes)
    print '\n'
    return gen_sol

#--- CONSTANTS -----------------------------------------------------------+

cost_func = func1                   # Cost function
bounds = [(-1,1),(-1,1),(-1,1),(-1,1),(-1,1),(-1,1),(-1,1),(-1,1),(-1,1),(-1,1)]           # Bounds [(x1_min, x1_max), (x2_min
popsize = 15                        # Population size, must be >= 4, also number of iterations
mutate = 0.5                        # Mutation factor [0,2]
recombination = 0.7                 # Recombination rate [0,1]
maxiter = 15                        # Max number of generations (maxiter)

#--- RUN -----------------------------------------------------------------+

main(cost_func, bounds, popsize, mutate, recombination, maxiter)

#--- END -----------------------------------------------------------------+
```

Appendix-F

```
# created by Huzefa Shaikh

from pyevolve import G1DList
from pyevolve import Mutators, Initializators
from pyevolve import GSimpleGA, Consts
import numpy as np

# Sphere-Function
def sphere(xlist):
    total = 0
    for i in xlist:
        total += i**2
    return total

def run_main():

  lr = 0.01
  results = []
  for x in range(15):

    genome = G1DList.G1DList(2)
    genome.setParams(rangemin=-5.12, rangemax=5.13)
    genome.initializator.set(Initializators.G1DListInitializatorReal)
    genome.mutator.set(Mutators.G1DListMutatorRealGaussian)
    genome.evaluator.set(sphere) # Call sphere function

    # Configure Genetic Algorith
    ga = GSimpleGA.GSimpleGA(genome, seed=666)
    ga.setMinimax(Consts.minimaxType["minimize"])
    ga.setGenerations(300)
    ga.setMutationRate(lr)
    lr += 0.01
    ga.evolve(freq_stats=100)

    #print best individual
    #print 'Run: ',x,'\n'
    best = ga.bestIndividual()
    print best
    results.append(best)
    print '\n'

  minn = min(results)
  maxx = max(results)
  nresult = np.array(results)
  meann = np.mean(nresult)

  print 'Best Performacne: ', minn
  print 'Worst Performacne: ', maxx
  print 'Average of all the performance', meann
  return

if __name__ == "__main__":

    run_main()
```