Weaponizing Favorite Test Functions for Testing Global Optimization Algorithms: An illustration with the Branin-Hoo Function

Charles F. Jekel[*] and Raphael T. Haftka[†]
University of Florida, Gainesville, Florida 32611

Some popular functions used to test global optimization algorithms, such as the Branin-Hoo and Himmelblau functions, have multiple local optima, all with the same value of the objective function. That is all local optima are also global optima. This renders them easy to optimize, because it is impossible for the algorithm to get stuck in a local optimum that is not the global one. Such functions actually present an opportunity to create challenging problems for optimization algorithms, because, as illustrated here, it is easy to convert them to functions with competitive local optima by adding a localized bump at the location of one of the optima. This process is illustrated here for the Branin-Hoo function, which has three global optima. We use the popular Python SciPy differential evolution (DE) optimizer for the illustration, because its wide use is likely to imply a well written code. DE also allows the use of the gradient-based BFGS local optimizer for final convergence. By making a large number of replicate runs we establish the probability of reaching a global optimum with the original and weaponized Branin-Hoo. With the original function we find 100% probability of success with a moderate number of function evaluations. With the weaponized version, we found that the probability of getting trapped in a non-global optimum can be made small only with a much larger number of function evaluations.

I. Nomenclature

ε	=	Bump width parameter
n	=	Number of optimization runs
p	=	Fraction of failure to find global optimum
r	=	Radial distance for RBF
e	=	Euler's constant
$\boldsymbol{x}$	=	Design vector
x_1	=	Design variable 1
x_2	=	Design variable 2

II. Introduction

There is substantial interest in global optimization algorithms. These include nature inspired stochastic algorithms like simulated annealing, genetic and differential evolution (DE) algorithms, particle swarm optimization, and ant colony optimization. They include also deterministic global optimizers like DIRECT [1] and SHGO [2], and more recently, there has been substantial development in surrogate based adaptive sampling algorithms such as EGO [3, 4].

Many test functions are used for tuning these algorithms, and a substantial list may be found in Wikipedia `https://en.wikipedia.org/wiki/Test_functions_for_optimization`. While some of these functions have complex shapes, few have local optima that are competitive with the global optimum. That means that the probability that the algorithm will be trapped in a local optimum is low. In fact, some of the popular ones, like the Branin-Hoo function, the Himmelblau function, and the Hölder table function even more extreme situation. They have multiple local optimal, but all with the same value. That is, all their local optimal are global! This means that the probability of

[*]PhD Candidate, Department of Mechanical and Aerospace Engineering.
[†]Emeritus Distinguished Professor, Department of Mechanical and Aerospace Engineering, AIAA Fellow.

being trapped in a local non-global optimum is zero, and in addition finding a global optimum is easy because there are several to choose from.

One objective of this abstract is to provide an easy-to use tool with its Python code to convert these functions to more challenging problems with the global optimum being accompanied by competing local optima. This is done by adding a radial-basis-function (RBF) bump to one of the global optima with the Branin-Hoo function used for illustration. A second objective is to illustrate how this makes the optimization much more challenging for a popular Python differential evolution optimizer in SciPy.

III. The Branin-Hoo Function

Using the information from `https://www.sfu.ca/~ssurjano/branin.html` the Branin-Hoo function is defined as

$$f(\boldsymbol{x}) = a(x_2 - bx_1^2 + cx_1 - d)^2 + s(1-t)\cos(x_1) + s. \tag{1}$$

We use the recommended values $a = 1, b = 5.1/(4\pi^2), c = 5/\pi, d = 6, s = 10, t = 1/(8\pi)$. The domain is here as usual, $x_1 \in [-5, 10]$, $x_2 \in [0, 15]$, and the function is shown in Fig. 1.

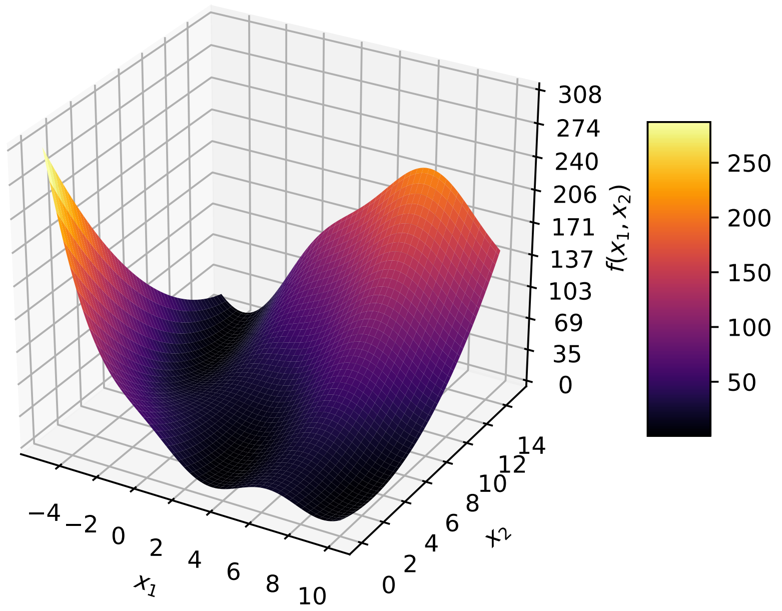

Fig. 1 Branin-Hoo function.

The function has three global optima. At each optimum the function value is $f(\boldsymbol{x}) = 0.398$. The locations of the optima are

$$\text{First: } \boldsymbol{x} = (-\pi, 12.275) \tag{2}$$

$$\text{Second: } \boldsymbol{x} = (\pi, 2.275) \tag{3}$$

$$\text{Third: } \boldsymbol{x} = (9.42478, 2.475) \tag{4}$$

where the first optimum is referred to as optimum 1 in this text.

IV. The Bump

It is desirable to add (for maximization) or subtract (for minimization) a bump to the original function that will not change the location of an optimum. Radial basis functions (RBF) [5] are selected because they depend only on the distance from the optimum. In addition, it is desirable that the bump will affect only one optimum, and for that the RBF bump function `https://en.wikipedia.org/wiki/Radial_basis_function` is selected. The bump function is defined as

$$\varphi(r) = \begin{cases} \exp\left(-\frac{1}{1-(\varepsilon r)^2}\right) & r < \frac{1}{\varepsilon} \\ 0 & r \geq \frac{1}{\varepsilon} \end{cases} \tag{5}$$

Here r is the radial distance from the center of the bump, and its maximum value at $r = 0$ is $1/e = 0.3679$. The width of the bump is determined by ε. Figure 2 shows a one-dimensional slice of the Branin-Hoo function with $10\varphi(r)$ subtracted at the location of one of its global optima.

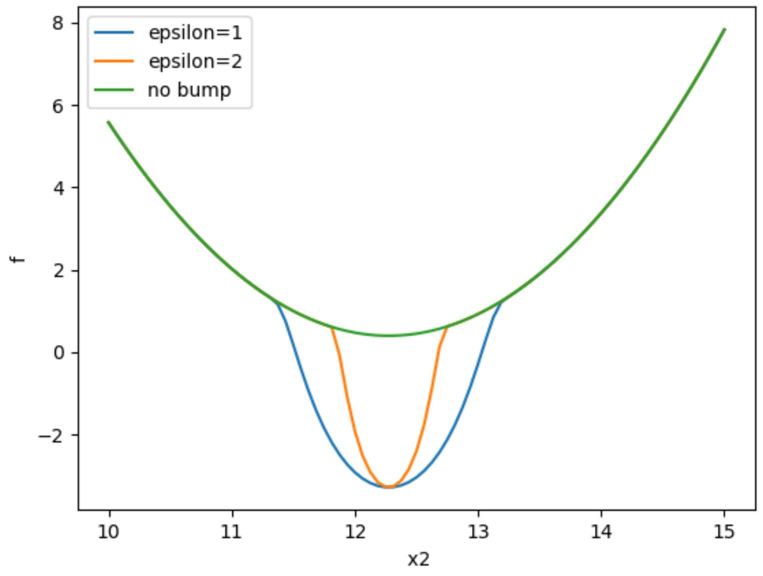

Fig. 2 A 1D slice through the Branin-Hoo function for $x_1 = -\pi$ with the bump defined by Eq. 5 multiplied by 10 subtracted at the location of location of one of its global optima at $x_1 = -\pi, x_2 = 12.275$. The width parameters are $\varepsilon = 1$ and $\varepsilon = 2$.

V. Differential Evolution Algorithm with BFGS Follow-up in SciPy

Differential evolution (DE) [6] is a popular global optimization algorithm loosely inspired by evolutionary processes observed in nature. The algorithm attempts to find a design vector of real numbers that minimizes a blackbox function. The objective function does not need to be differentiable or continuous. The implementation of DE used in this work is available in the Python SciPy ecosystem [7]. This implementation was chosen for rich features (parallelization, LHS initial design, L-BFGS-B follow up, etc.), simplicity, and wide availability.

Like most global search algorithms, once the algorithm gets close to the global optimum the final convergence is slow. SciPy therefore offers the option of switching to a local gradient based optimizer, BFGS, for this 'polish' phase. The number of function evaluations in the DE search is determined by the maximum number of iterations, *max_iter* and the population size *pop*. Once this phase is finished, the BFGS algorithm is run to convergence. Note that by specifying a small population size and a small maximum number of iterations, the SciPy function will be using BFGS for most of the search.

A. Performance Measurement

DE is a stochastic algorithm, so that every time it is run one can get a different result. Therefore, when comparing the performance with and without a bump, we make multiple runs and compare the fraction of failure deduced from the percentage of runs that fail to reach the optimum with some given tolerance. Here we need to choose the number of runs, which is designated as n needed for some desired accuracy in the estimated fraction of failure.

The accuracy of the fraction of failure estimate is measured here by the standard deviation of the number of runs that failed. It is easy to show that the standard deviation σ_{fail} of the number of failures in n runs is related to the probability of failure p as

$$\sigma_{\text{fail}} = p\sqrt{pn(1-p)}. \tag{6}$$

If we want accuracy to one percent of the total number of runs, we set $\sigma_{\text{fail}} = 0.01n$ and calculate the required number of runs for a given p.

$$(0.01n)^2 = p^3 n(1-p) \Rightarrow n = 10^4 p^3 (1-p) \tag{7}$$

Since we do not know p ahead of time, we observe that the maximum required n is attained for $p = 0.75$, which gives $n = 1055$ runs. Since $p = 0.75$ is the worst case, we rounded n to $1,000$ for ease of translating the number of failures to percentages.

B. Performance with No Bump

The DE algorithm was run first with the original Branin-Ho function for large number of times with for different population sizes (*pop*) and different numbers of maximum iterations (*max_iter*) with and without the BFGS polish. No convergence criteria were applied, so the optimization stopped with a number of function evaluations equal to *2*pop*(max_iter+1)*. The optimization run was deemed successful if the optimized function value was within 0.01 of the true optimum. We recorded the fraction p of failed runs, as well as how many were at or near (within a distance of one) from each optimum. This would help understand the behavior with a bump later on. Note that the closest optima, Optimum 2 and Optimum 3 are at a distance of about 6.3 from each other.

We also made the same runs with the BFGS polish turned on to improve local convergence. This time all runs were successful, since all optima have the same value, but again we recorded how many were at each optimum.

Table 1 provides a summary of the results. As expected by Equation 1.4 and the discussion following the equation, with 1,000 runs the percentage results were mostly repeatable with a different random seed to about 1-2% accuracy. That is, if the percentage of failures is reported for example as 46%, when repeated with a different seeds it may range from 44% to 48%.

Table 1 Performance of algorithms for the original Branin-Hoo function without a bump. Results are based on 1,000 replicate runs, and repeating the runs with different random seed indicates that the numbers are good to about 1% accuracy, as predicted by Eqs. 6 and 7.

Algorithm	Pop	Max_iter	Percent failures (%)	Average number of function evaluations	Percentage at or near each optimum (%)
DE	10	20	0.1	420	30, 44, 25
DE	10	10	46	220	27, 49, 24
DE/BFGS	10	10	0	240	27, 49, 24
DE	5	5	98	60	15, 37, 22
DE/BFGS	5	5	0	84	27, 45, 29
DE	2	2	100	15	7, 10, 6
DE/BFGS	2	2	0	42	30, 35, 35

The first row in Table 1 shows that we can reach very low probability of missing a global optimum with 420 function evaluations, as only one optimization out of 1,000 failed. We also see that the DE favors the centrally positioned Optimum 2 compared to the other two. The second row shows that if we reduce the number of iterations to 10, almost half of the runs fail, but they all get within a distance of 1 from an optimum. That is, the failure is in the local convergence. Failed local convergence is supported by row 3, which shows that no failures occur if we add the BFGS follow-up. The average number of function evaluations is 240, with 220 coming from the DE part and 20 from the BFGS follow-up. With a population of 5 and maximum number of iterations equal to 5, 98% of the runs fail if we use only DE. Still 74% (15+37+22) make it to the vicinity of one of the optima. Adding BFGS reduces the number of failures to zero for an average cost of 84 function evaluations. Sixty of these come from the DE and 24 from the BFGS. This still may be viewed as a case where the two algorithms share the work.

The last row in Table 1 looks at a case where we use the BFGS to do almost all the work. With a population of 2 and max number of iterations equal to 2 all runs fail, and only 23% get near one of the optima. When we add BFGS, they all converge with an average cost of 42 function evaluations. This time we see that when BFGS does most of the work the three optima have similar probabilities of being selected, while Optimum 2 was preferred in the previous runs where DE performed most of the work.

C. Performance with Bump at the First Optimum

The procedure described with no bump was repeated with the wider bump shown in Fig. 2, that is for an amplitude of $10/e$, with $\varepsilon = 1$. The results when the bump was added to Optimum 1 are summarized in Table 2.

Table 2 Performance of algorithms when a bump is added to Optimum 1 Results are based on 1,000 replicate runs and repeating the runs with different random seed indicates that the numbers are good to about 1% accuracy as predicted by Eqs. 6 and 7.

Algorithm	Pop	Max_iter	Percent failures (%)	Average number of function evaluations	Percentage at or near each optimum (%)
DE	10	20	64	420	36, 39, 25
DE	20	10	89	440	46, 34, 20
DE/BFGS	20	10	54	458	46, 34, 20
DE	40	5	98	480	56, 27, 17
DE/BFGS	40	5	43	500	56, 27, 17
DE	50	4	99	500	57,22, 21
DE/BFGS	50	4	43	522	57,22,21
DE	80	5	96	960	71, 14, 15
DE/BFGS	80	5	29	980	71, 14, 15
DE/BFGS	100	4	26	1021	74, 14, 12
DE/BFGS	125	3	19	1022	82, 8, 10
DE/BFGS	165	2	16	1013	84, 7, 9
DE/BFGS	330	2	4	2002	96, 2, 2

The first row of Table 1, with a population of 10 and 20 iterations with DE alone had only one failure out of a thousand runs when run without a bump. This was due to the fact that with 20 iterations DE can achieve final convergence, and it did not matter which optimum it converged to. With the bump, Table 2 shows 64% failures, because the percentage of runs that converged to Optimum 1 was 36%. Note that without the bump, as can be seen from Table 1, only 30% of the runs converged to Optimum 1.

Increasing the number of iterations, or adding BFGS does not help, because the runs are already converged, so we next increase the population size to 20 and reduced the number of iterations to 10 to get a similar cost. The second row in Table 2 shows that this increased the percentage of runs that went to the bump (Optimum 1) to 46%. The percent failure increased to 89% because 10 iterations were not close enough for final convergence. However, here adding BFGS (row 3) caused all of these 46% runs to converge to Optimum 1, reducing the percent failures to 54%.

Based on the results in Table 1, even 5 iterations were enough to get DE close to the optima, so we next tried a population of 40 with 5 iterations. This increased the percentage of runs near Optimum 1 to 56%. With DE alone 98% of the runs failed because of lack of final convergence, but adding BFGS we realized the 56% rate of success, or 44% failure. The average number of function evaluations, shows that adding BFGS increases the cost by about 20 evaluations, in line with what we saw without the bump in Table 1.

Additional experimentation showed that as the population size increases, the optimum number of iterations decreases. This is due to the two dimensional nature of the problem that creates a high probability that the initial population, placed randomly, will have a member inside the bump.

The total area of the design space is $15 \times 15 = 225$. The bump covers a circle of radius 1, so that its area is equal to π. The probability that a member of the population will be inside the bump area is $\pi/225$ or about 1.4%. The probability that it will outside is about 98.6%. However, with a population size of *pop*, the probability that not even one member of the population will fall inside the bump is 0.986^{pop}. When pop = 165, this probability is 9.8%, so there is more than 90% chance that one member of the initial population will be inside the bump, and several other members very close to the bump.

Comparing Table 1 to Table 2 demonstrates that adding the bump increases the required number of function evaluations by more than one order of magnitude. It may be also reasonable to speculate that for functions with large number of variables, the effect will be larger, because the bump will cover a much smaller percentage of the volume of the design space.

VI. Concluding Remarks

Global optimization algorithms are often tested on easy functions with multiple identical-valued optima. This abstract suggests that these functions could be weaponized to become much harder by adding or subtracting a bump to one of the optima. This is illustrated for the Branin-Hoo function, which has three identical-valued global optima by examining the performance of the Python SciPy popular differential evolution (DE) optimizer with a follow up by the BFGS local gradient based algorithm. It is shown that adding the bump to the first optimum increases the required computational effort by about a factor of 16. The Python code to perform this study is available online at `https://github.com/cjekel/weaponizing_test_functions`.

Acknowledgments

C. F. Jekel thanks the University of Florida Graduate Preeminence Award and the U.S. Department of Veterans Affairs Educational Assistance for funding his PhD research.

References

[1] Jones, D. R., Perttunen, C. D., and Stuckman, B. E., "Lipschitzian optimization without the Lipschitz constant," *Journal of Optimization Theory and Applications*, Vol. 79, No. 1, 1993, pp. 157–181. https://doi.org/10.1007/BF00941892, URL http://link.springer.com/10.1007/BF00941892.

[2] Endres, S. C., Sandrock, C., and Focke, W. W., "A simplicial homology algorithm for Lipschitz optimisation," *Journal of Global Optimization*, Vol. 72, No. 2, 2018, pp. 181–217. https://doi.org/10.1007/s10898-018-0645-y, URL https://doi.org/10.1007/s10898-018-0645-y.

[3] Jones, D. R., Schonlau, M., and Welch, W. J., "Efficient Global Optimization of Expensive Black-Box Functions," *Journal of Global Optimization*, Vol. 13, No. 4, 1998, pp. 455–492. https://doi.org/10.1023/A:1008306431147, URL https://doi.org/10.1023/A:1008306431147.

[4] Zhang, Y., Kristensen, J., Ghosh, S., Vandeputte, T., Tallman, J., and Wang, L., "Finding Maximum Expected Improvement for High-Dimensional Design Optimization," *AIAA Aviation 2019 Forum*, AIAA AVIATION Forum, American Institute of Aeronautics and Astronautics, 2019. https://doi.org/doi:10.2514/6.2019-2985, URL https://doi.org/10.2514/6.2019-2985.

[5] Broomhead, D. S., and Lowe, D., "Radial basis functions, multi-variable functional interpolation and adaptive networks," Tech. rep., Royal Signals and Radar Establishment Malvern (United Kingdom), 1988.

[6] Storn, R., and Price, K., "Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces," *Journal of Global Optimization*, Vol. 11, No. 4, 1997, pp. 341–359. https://doi.org/10.1023/A:1008202821328, URL https://doi.org/10.1023/A:1008202821328.

[7] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Jarrod Millman, K., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C., Polat, İ., Feng, Y., Moore, E. W., Vand erPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, Vol. 17, 2020, pp. 261–272. https://doi.org/https://doi.org/10.1038/s41592-019-0686-2.