Review: Basin-hopping Algorithm: Combining Global Stepping with Local Minimization*

*ECE 506 Project, Fall 2018

Mustafa Salman
Dept. of Electrical & Computer Engineering
University of New Mexico
Albuquerque, NM
esalman@unm.edu

***Abstract*—Stochastic algorithms are well-suited for optimization of complex and non-deterministic functions, but comes at a price of increased computational cost. Basin-hopping is a hybrid optimization algorithm implemented in the SciPy library. It combines the advantages of both stochastic and deterministic algorithms. In this work I presented a comparison of the performance between Basin-hopping algorithm and several other deterministic algorithms. Results show that Basin-hopping algorithm has better accuracy and robustness while having acceptable efficiency.**

***Index Terms*—stochastic optimization, deterministic optimization, hybrid optimization, Basin-hopping**

I. Introduction

Typical deterministic optimization algorithms can optimize smooth continuous functions very well. Some of these algorithms can take advantage of gradients and Hessians (if exists) of such functions for faster convergence [1]. However, when it comes to continuous function with many local minima, these algorithms fall short. Furthermore, many real world problems are non-deterministic where these deterministic algorithms are not applicable. Stochastic algorithms are designed for solving such problems [3].

Some stochastic algorithms apply random perturbations to the local minima at each iteration in order to traverse the solution space. Markov Chain Monte Carlo (MCMC) criteria are techniques for establishing a stationary distribution using the acceptable solutions [4]. The global minimum can be envisioned as the highest point of such a distribution. Stochastic algorithms are able to sample the possible solutions and eventually converge to a global minimum using the Metropolis criterion. Metropolis is just one of various types of MCMC techniques which are useful for sampling from a probability distribution when direct sampling is difficult [11]. These algorithms has shown promise in finding the solution of many difficult problems. However, there is no free lunch and this success comes at a price.

A major drawback of the stochastic algorithms is that in order to find the global minimum of complex functions, they need to sample a very high number of possible solutions and can only accept a fraction of those using a criterion such as Metropolis. This high computation cost is impractical in many circumstances, which has prevented widespread usage of these algorithms. There has been an attempt to formulate a class of algorithms which combine the best of both deterministic and stochastic algorithms.

The Scientific Computing Tools for Python (SciPy) library implements a hybrid Basin-hopping algorithm [5]. The default implementation of this algorithm iteratively samples new candidate solutions from a uniform distribution, performs a local optimization and uses the Metropolis criterion in an acceptance test in order to reach global minima of a problem. The implementation offers various flexibilities such as using one of the many different deterministic algorithms for local minimization and customizing the random perturbation function among others. In this work I used this method to minimize several 2D continuous functions of varying complexity and compared the accuracy, efficiency and robustness of Basin-hopping with other deterministic functions.

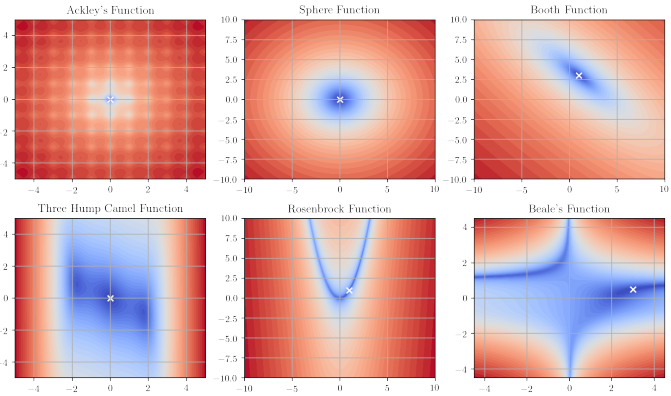

Fig. 1. Contour of various 2D continuous functions

II. Methods

A. Continuous Testing Functions for Optimization

I used an implementation of various continuous testing functions for optimization available on Github [3]. This module

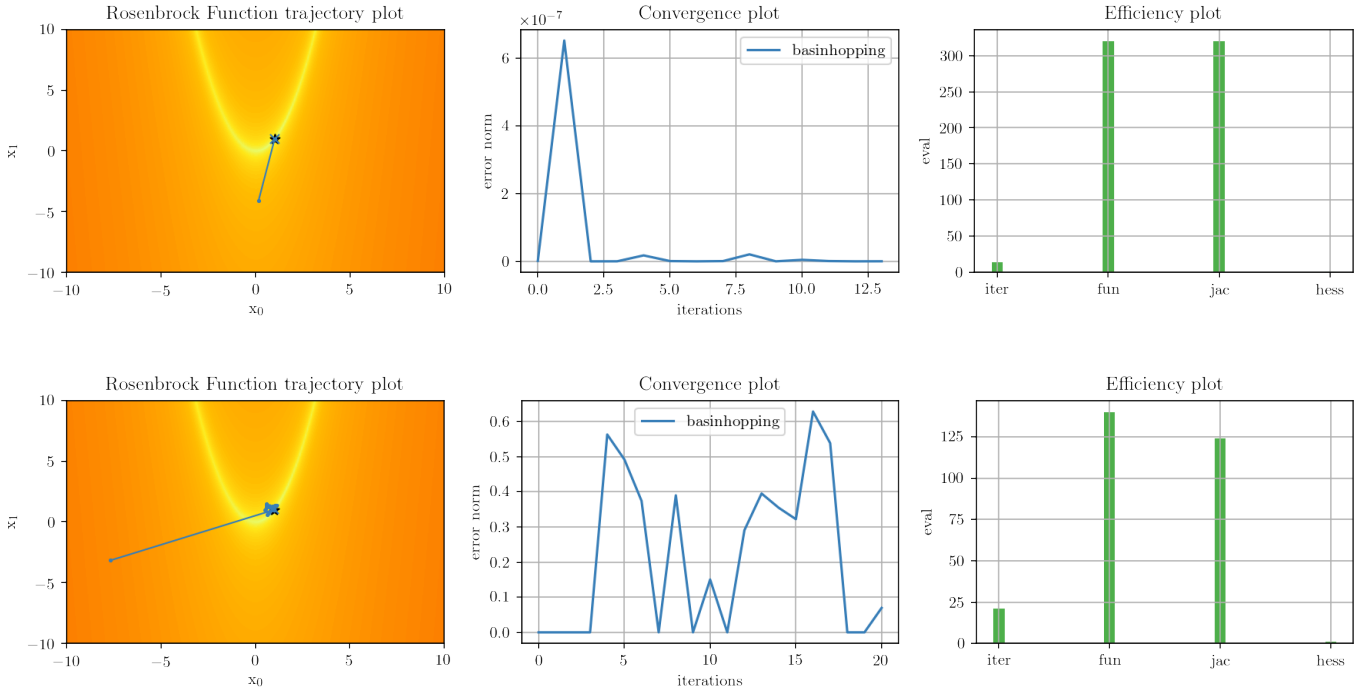

Fig. 2. Difference in the number of function evaluations when using `BFGS` or `dogleg` algorithm for local minimization in Basin-hopping.

contains functions in one, two and multi-dimension. Each test function is implemented as a class with methods for calculating the cost, gradient and Hessian. Additional information such as the global minimum value and the function's domain are also provided. I found that out of 40 2D functions, 14 have all these methods implemented.

Some of the functions that I used are:

- With many local minima: Ackley, Levy13
- Bowl-Shaped: Sphere
- Plate-Shaped: Booth, Matyas, McCormick
- Valley-Shaped: Three Hump Camel, Rosenbrock
- Steep Ridges/Drops: Absolute, Absolute Skewed, Easom, Beale, Goldstein-Price, Styblinski-Tang

B. Basin-hopping Algorithm

Basin-hopping is a stochastic algorithm which attempts to find the global minimum of a smooth scalar function of one or more variables [6]. It is presented in Alg. 1. The algorithm implemented in the SciPy library was described by Wales and Doye [5].

1) Heuristics:

- Random perturbation of the coordinates
- Local minimization
- Accept or reject the new coordinates based on the minimized function value

2) Notes on the SciPy Implementation:

- `LOCALSEARCH` uses the Broyden-Fletcher-Goldfarb-Shanno (`BFGS`) algorithm by default [12]. Using the `minimizer_kwargs` parameter of Basin-hopping, it is possible to apply any `scipy.optimize.minimize` local minimization method and specify the gradient/Hessian as required. It is also possible to use a custom *do nothing* function, along with a suitable `take_step` callback, which may implement simulated annealing.
- A `callback` routine is implemented for each minima found. It can be used for tracking the performance of the algorithm.
- `PURTURB` uniformly samples a new X_{i+1} from $[X_i - \texttt{stepsize}, X_i + \texttt{stepsize}]$ in each dimension. `stepsize` is adjusted automatically by the Basin-hopping algorithm, but can also be passed as a parameter. The `take_step` callback can replace the default step-taking routine.
- The acceptance test uses the Metropolis criterion of standard Monte Carlo algorithm. Steps are always accepted if $f(X_{i+1}) < f(X_i)$. Otherwise, they are accepted with

Algorithm 1 Basin-hopping $\min y = f(x)$ [6]

$i \leftarrow 0$
$X_i \leftarrow$ random initial point in variable space
$Y_i \leftarrow \texttt{LOCALSEARCH}(X_i)$
while `STOP` not satisfied **do**
  $X_{i+1} \leftarrow \texttt{PURTURB}(Y_i)$
  $Y_{i+1} \leftarrow \texttt{LOCALSEARCH}(X_{i+1})$
  if $f(Y_{i+1}) < f(Y_i)$ **then**
    $i \leftarrow i + 1$
  end if
end while

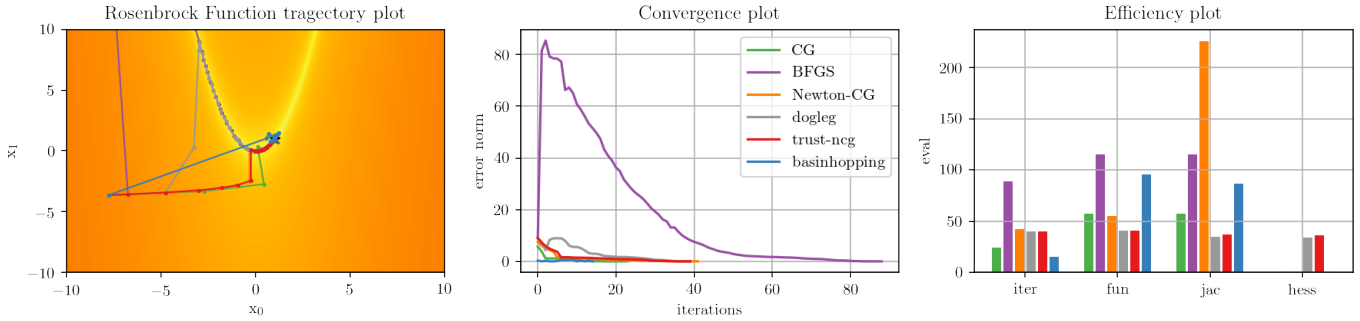

Fig. 3. Trajectory for different algorithms, convergence and efficiency (number of iterations, function (cost, gradient and Hessian) evaluations for the Rosenbrock function)

probability

$$exp(-\frac{f(X_{i+1}) - f(X_i)}{T})$$

Where T is the *temperature* used in the Metropolis criterion. If $T = 0$, then all steps that increase the function value are rejected. An `accept_test` callback is also implemented for defining custom acceptance test.

- *Other parameters:* An optional `interval` parameter is available which determines how often `stepsize` is updated. The `niter_success` parameter stops the run if global minimum candidate remains the same for this number of iterations.
- It is advised that both `T` and `stepsize` should be chosen based on the typical difference between the local minima of the function being optimized.

C. Experiments

I performed the following experiments to understand how the Basin-hopping algorithm works, and how it fares compared to the other deterministic algorithms implemented in SciPy.

1) Using different local optimization methods: For this experiment I looked at how Basin-hopping performs for different algorithm choices for the `LOCALSEARCH` routine.

2) Cross-validation for `T` & `stepsize`: For this experiment I tried to find the optimal `T` and `stepsize` parameters for minimizing the 2D *Rosenbrock* function. `T` was varied logarithmically between .01 and 100; and `stepsize` was varied linearly between 0.1 and 3.0. A grid search was performed for each pair of (`T`, `stepsize`). For each pair, the experiment was repeated 100 times with random initialization of the starting value. The mean number of function (cost, gradient and Hessian) evaluations and error norms were reported.

3) Comparing Accuracy & Efficiency: For this experiment, I looked at how Basin-hopping fares in terms of convergence and efficiency compared to several deterministic algorithms. The algorithms used for comparison were: conjugate gradient (`CG`), `BFGS`, `Newton-CG`, `dogleg` and trust-region Newton-CG (`trust-ncg`). These algorithms are implemented in the `scipy.optimize.minimize` routine. `CG`, `BFGS` and `Newton-CG` only require the gradient of the function. `dogleg` and `trust-ncg` require the Hessian, `dogleg` requires the Hessian to be positive semi-definite in addition. I used these methods to optimize different functions from the same starting point. The convergence (error norm vs iterations) and number of function evaluations were noted.

4) Comparing Robustness: For this experiment, I used all of the above methods to optimize different functions, each one a number of times (100) with random initializations. The mean and standard deviation of the average number of function (cost/gradient/Hessian) evaluations and error norm were noted.

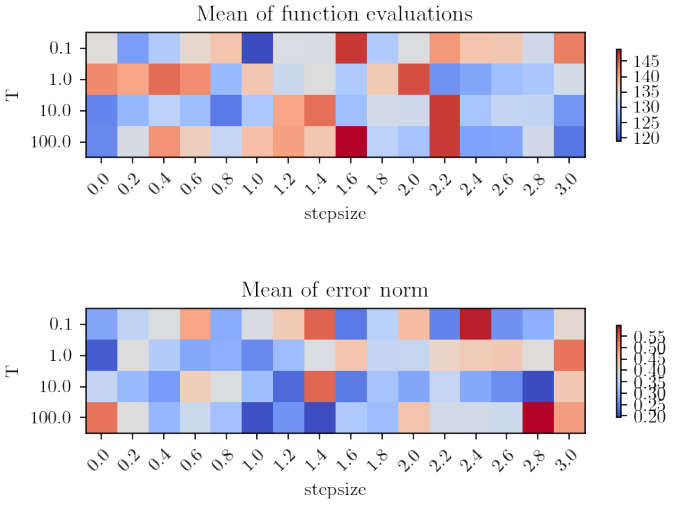

Fig. 4. Basin-hopping performance for different pairs of (`T`, `stepsize`)

III. Results

Some of the function contours used for performing the experiments are depicted in Fig. 1

A. Using different local optimization methods

Fig. 2 shows the efficiency (and convergence) of Basin-hopping using `BFGS` and `dogleg` for the `LOCALSEARCH` step. `dogleg` requires fewer function evaluations because of using the the Hessian for faster convergence. However, `BFGS` shows better accuracy at minimizing functions with many local minima.

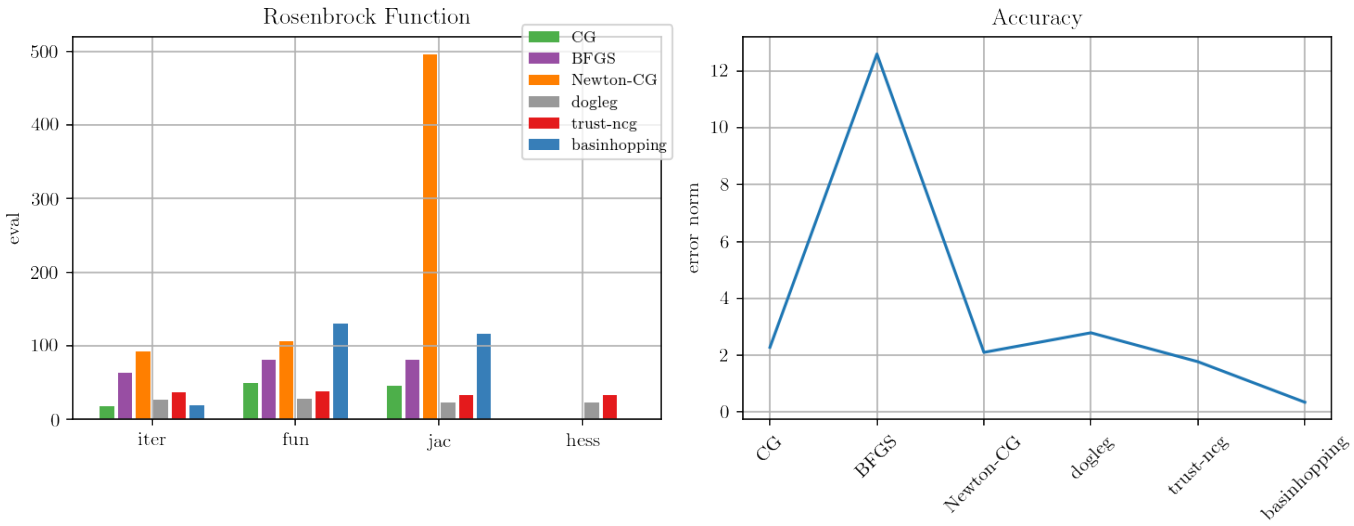

Fig. 5. Average function evaluation and error norm over 100 repetitions of each algorithm at minimizing the Rosenbrock function

B. Cross-validation for `T` & `stepsize`

Fig. 4 shows the performance of Basin-hopping for different pairs of (`T`, `stepsize`). It is difficult to choose a `T` from this plot, although lower `stepsize` seems to result in less error.

C. Comparing Accuracy & Efficiency

Fig. 3 shows the convergence for various algorithm at minimizing the Rosenbrock function. It shows that Basin-hopping with `dogleg` needs less than 100 function cost and gradient evaluations to converge. This is quite low compared to typical stochastic algorithms. It also requires the least stochastic iterations than any other method.

D. Comparing Robustness

Fig. 5 shows the mean number of iterations, function (cost, gradient and Hessian) evaluations and error norms over 100 repetitions of each algorithm at minimizing the Rosenbrock function. The numbers hover around 100 for Basin-hopping, which is feasible considering that it results in the lowest average error norm.

IV. Discussion

I tried to minimize 14 functions (some of them quite nasty) using the deterministic algorithms available from SciPy, and a hybrid algorithm (Basin-hopping). There are 26 more functions in the Github repository which do not implement the gradient or Hessian. All of the SciPy optimization routines implement a `callback` method at each iteration. I used this method to track the local minima and error norm.

Basin-hopping error is lower, but function evaluation is higher when using the gradient-based `BFGS` for local minimization compared to the Hessian-based `dogleg`. Compared to the deterministic algorithms, Basin-hopping with `dogleg` is more efficient but less accurate, and Basin-hopping with `BFGS` is the opposite. The former can beat deterministic algorithms at minimizing smooth functions, and the latter can do the same at minimizing functions with rough surfaces. It is easy to break the deterministic algorithms using a starting point outside of the suitable domain. For instance, `BFGS` blows up but Basin-hopping appears very reliable for minimizing the Beale function.

Following are the analyses of some common errors encountered when performing the experiments.

- "Desired error not necessarily achieved due to precision loss": occurs due to the inability to solve the minimization subproblem.
- "free variable 'gfk' referenced before assignment in enclosing scope": occurs in Newton-CG due to bad starting points.
- "array must not contain infs or NaNs": occurs in `trust-ncg` when the gradient calculation blow up. Typically caused by bad starting points outside the function domain.
- "A linalg error occurred, such as a non-psd Hessian": happens in `dogleg`.
- "A bad approximation caused failure to predict improvement": happens in `trust-ncg`.

A. Future Directions

I found the "Continuous Testing Functions for Optimisation" Github repository very useful. It saved a lot of time at the start of the project. However, not all the functions in this module have the gradient and Hessian implemented. It will be a good idea to contribute these implementations in this repository.

Further experimentation with the Basin-hopping options is necessary in order to find suitable `T` and `stepsize`. In addition, how the algorithm adjusts the `stepsize` using the `interval` parameter is also worth a look.

For this experiment I did not look at the timing or memory consumption, as all the SciPy routines were very fast at solving

the continuous functions. Even the robustness experiment with 14 functions, 6 different algorithms and 100 repetitions only took a few seconds even without parallelization. It will be worth looking at the timing and memory performance, especially for solving non-deterministic problems. There is a nice Python module consisting of such problems and stochastic algorithms called NiaPy [8].

Another important measure of performance is the optimization status returned by the SciPy `mimimize` routine. It tells us if the solution actually converged or not. In the robustness experiment, at times the algorithms did not converge but the error norm and function evaluation were still taken into account. The status should also be taken into account for better assessment of the robustness.

REFERENCES

[1] J. Nocedal and S. J. Wright, Numerical optimization, 2nd ed. New York: Springer, 2006.

[2] D. P. Kroese, T. Taimre, and Z. I. Botev, Handbook of Monte Carlo Methods. John Wiley & Sons, 2013.

[3] M. Herman, Simulated Annealing & the Metropolis Algorithm: A Parameter Search Method for Models of Arbitrary Complexity, p. 14.

[4] R. Christensen, W. Johnson, A. Branscum, and T. E. Hanson, Bayesian Ideas and Data Analysis: An Introduction for Scientists and Statisticians, 1 edition. Boca Raton, FL: CRC Press, 2010.

[5] D. J. Wales and J. P. K. Doye, Global Optimization by Basin-Hopping and the Lowest Energy Structures of Lennard-Jones Clusters Containing up to 110 Atoms, The Journal of Physical Chemistry A, vol. 101, no. 28, pp. 51115116, Jul. 1997. Available: https://pubs.acs.org/doi/10.1021/jp970984n

[6] B. Olson, I. Hashmi, K. Molloy, and A. Shehu, Basin Hopping as a General and Versatile Optimization Framework for the Characterization of Biological Macromolecules, Advances in Artificial Intelligence, 2012. [Online]. Available: https://www.hindawi.com/journals/aai/2012/674832/. [Accessed: 14-Dec-2018].

[7] L. Marris, Continuous Test Functions for Optimisation. Available: https://github.com/lukemarris/ctf. 2018.

[8] NiaOrg, NiaPy: Python micro framework for building nature-inspired algorithms. .

[9] scipy.optimize.basinhopping SciPy v0.18.1 Reference Guide. [Online]. Available: https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.optimize.basinhopping.html. [Accessed: 14-Dec-2018].

[10] scipy.optimize.minimize SciPy v0.18.1 Reference Guide. [Online]. Available: https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.optimize.minimize.html#scipy.optimize.minimize. [Accessed: 14-Dec-2018].

[11] W. K. Hastings, Monte Carlo sampling methods using Markov chains and their applications, Biometrika, vol. 57, no. 1, pp. 97109, Apr. 1970.

[12] R. Fletcher, Practical methods of optimization, 2nd ed. Chichester; New York: Wiley, 1987.