Comparative Analysis of the Bat Algorithm vs. Genetic Algorithm, Steepest Descent Method and Quasi-Newton-BFGS

Uriel David Tello Padilla[1]

[1] *Higher School of Physics and Mathematics, National Polytechnic Institute*
[2] *Bio-inspired algorithms*

Abstract

In this study, the results of the comparison of performances of the bat, genetic, steepest descent and quasi-Newton-BFGS algorithms are presented. The parameters considered for such comparison are: evaluations of the objective function and precision of the solution.

Keywords: bio-inspired algorithms, genetic algorithms.

1 Introduction

There are a lot of algorithms inspired by nature. The purpose of this work is to analyze the performance of the bat algorithm(**AM**) and compare it with a genetic algorithm(**AG**), the maximum slope descent method(**DPM**) and the Broyden-Fletcher-Goldfarb-Shanno method(**BFGS**). We'll start by describing the basic ideas, main steps, pseudocode, and implementation details of **AM**. Finally, the execution of each method is carried out for three test functions and the results are shown in a comparative table. All algorithms used in this work have been encoded in the programming language **Julia**. In order for the reader to understand how the implementations are carried out and how the algorithm actually works, the code for **AM** is provided in the appendix.

1.1 The bat algorithm(Yang 2020).

Bats, especially microbats, use echolocation to navigate and forage. These bats emit a series of short ultrasonic bursts in the frequency range from 25 kHz to about 150 kHz, and these short pulses typically last a few milliseconds. The loudness of such bursts and the rate of pulse emission vary during the hunt, especially when searching for prey. Increasing the frequency will reduce the wavelength of the ultrasound pulses, thereby increasing the resolution and accuracy of prey detection. These known echolocation features of bats can be simulated in **AM**, which was developed by Xin-She Yang in 2010 (Yang X.S. Cruz C. González J.R. Pelta 2010). **AM** uses frequency tuning as a random driving force, in combination with the use of pulse emission rate r and loudness A for the bat population.

In **AM**, there are n bats that form an iteratively evolving population. The location of each bat is denoted by $x_i (i = 1, 2, \dots, n)$, which can be considered as the solution vector to the optimization problem in question. Since bats fly in the search space, each bat is also associated with a velocity vector v_i.

A position vector is considered as a solution vector to an optimization problem in a search space of dimension D with D independent design variables,

$$\mathbf{x} = [x_1, x_2, x_3, \dots, x_D]. \tag{1}$$

During the iterations, each bat can vary its pulse emission rate r_i, its loudness A_i and its frequency

Computer Science (2022)

DOI: 666

Corresponding author
Author One

Edited by
David Tello

f_i. Frequency variations or tuning can be done by the equation

$$f_i = f_{\min} + \beta(f_{\max} - f_{\min}), \tag{2}$$

where $f_{\min}$ and $f_{\max}$ are the minimum and maximum ranges, respectively, of the frequency f_i for each bat i. Although the frequency variations are not actually uniform, we will use uniform variations for simplicity.
The frequency variations are then used to modify the speeds of the bats in the population so that

$$v_i^{t+1} = v_i^t + \left(x_i^t - x_*\right) f_i, \tag{3}$$

where $x*$ is the best solution obtained by the population in iteration t. Once a bat's velocity is updated, its position (or solution vector) x_i can be updated by

$$x_i^{t+1} = x_i^t + (\Delta t) v_i^{t+1}, \tag{4}$$

where Δt is the iteration or time increment. It's worth noting that all iterative algorithms update discretely, which means we can set Δt. Therefore, we can simply consider the vectors without physical units and then write the update equation as

$$x_i^{t+1} = x_i^t + v_i^{t+1}. \tag{5}$$

For local modification around the best solution, we can use

$$x_{\text{nuevo}} = x_{\text{anterior}} + \sigma \epsilon_t A^{(t)}, \tag{6}$$

where ϵ_t is a random number drawn from a normal distribution $N(0, 1)$ and σ is the standard deviation that acts as a scale factor. Here, $A^{(t)}$ is the average loudness at iteration t. For simplicity, we can use $\sigma = 0.1$ in our later implementation.

1.2 Pulse emission and volume

Loudness and pulse rate variations can be included in the **AM** to influence the way they are explored and exploited. In the real world, its variants are very complicated, depending on the species of bat. However, we use a monotone variation here. The pulse emission rate r_i can monotonically increase from a lower value r_i^0, while the volume can decrease from a higher value i (such as $A^0 = 1$) to a much lower value. So, we have

$$A_i^{t+1} = \alpha A_i^t, \tag{7}$$

$$r_i^{t+1} = r_i^{(0)}[1 - \exp(-\gamma t)], \tag{8}$$

where $0 < \alpha < 1$ and $\gamma > 0$ are constants.
Based on the formulas above, we can see that when t is large enough $(t \to \infty)$, we get $At \to 0$ and $r_i^t \to r_i^{(0)}$. For simplicity, we can use $\alpha = 0.97$ and $\gamma = 0.1$ in this simple implementation. Furthermore, we will use the same pulse emission frequency and volume for all bats, which greatly simplifies the implementations of the demo codes presented in this chapter.

1.3 Pseudocode and parameters

There are quite a few parameters in **AM**. Apart from the population size n and the frequency ranges f_{min} and $f_{\max}$, there are two parameters (α, β) and two initial values (A_0^0 and r_i^0). We will use the following values in our implementation: $\alpha = 0.97$, $\gamma = 0.1$, $f_{\min} = 0$, $f_{\max} = 2$ and $A_i^{(0)} = r_i^{(0)} = 1$

for all bats. For the population size n, $n = 10$ will be used in the implementation of the following demo.

Algorithm 1: The Bat Algorithm

```
1  function AM ($f, n$);
   Input   :Function, population size $n$, parameters $\alpha$ y $\gamma$
   Output :best overall position
2  Generate initial population $x_i$ and $v_i (i = 1, 2, ..., n)$;
3  Set the frequencies $f_i$, the pulse rate $r_i$ and the volume $A_i$;
4  Initialize $iter = 0$ (iteration counter);
5  while (iter < $t_{\max}$) do
6      Loop $r_i$ y $A_i$ ; Adjust frequencies; Generate new solutions using the equations (3) y (5)
7      if rand > $r_i$ then
8          Select a solution from the set of best solutions; Modify locally around the best
            selected solution;
9      end
10     Do a random fly to generate a new solution;
11     if rand > $A_i$ and $f(x_i) < f(x_j)$(to minimize) then
12         Accept the new solution;
13     end
14     Classify the current population of bats and find the best solution x*;
15 end
16 return the best solution
```

2 Comparison of methods

The Sphere, Rosenbrock and Ackley functions were considered for the comparison of the methods.

2.1 Technical Implementation Details of the Algorithms.

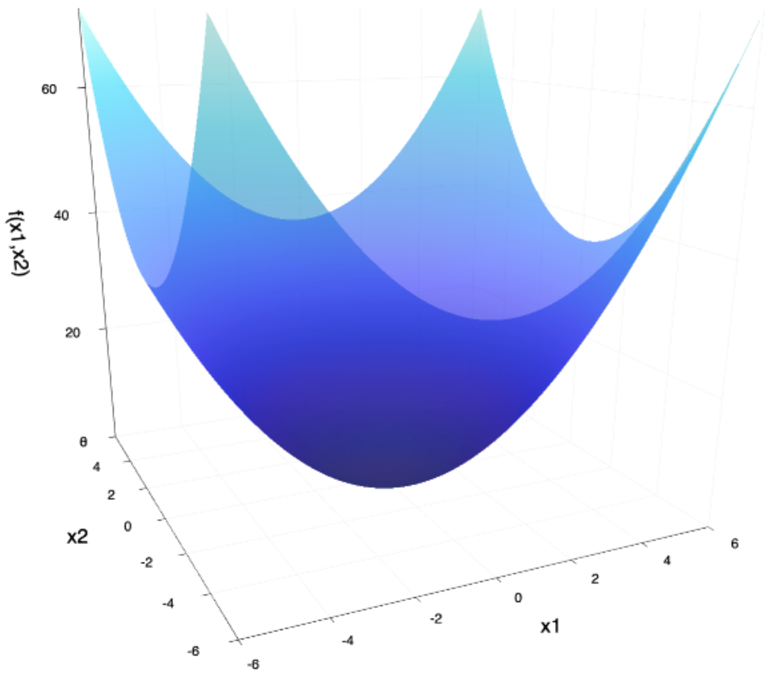
In order to make a fair comparison between all methods, the stop criteria of the (1) algorithm has been modified, that is, the stopping criterion that originally consisted of reaching a maximum number of iterations has been replaced by a tolerance for the value of the function as shown below on lines 18 to 20 of (2).

Algorithm 2: The Bat Algorithm(modified)

```
1  function AM ($f$, $n$);
   Input  :Function, population size $n$, parameters $\alpha$ and $\gamma$
   Output :Best overall position
2  Generate initial population $x_i$ and $v_i (i = 1, 2, ..., n)$;
3  Set the frequencies $f_i$, the pulse rate $r_i$ and the volume $A_i$;
4  Initialize $iter = 0$ (iteration counter);
5  while (true) do
6      Loop $r_i$ and $A_i$ ; Adjust frequencies; Generate new solutions using the equations (3) and
         (5)
7      if $rand > r_i$ then
8          Select a solution from the set of best solutions;
9          fmin = min(Fitness) ;
10         Modify locally around the best selected solution;
11     end
12     Do a random fly to generate a new solution;
13     fnew=F(Solutions)
14     if $rand > A_i$ and $f(x_i) < f(x_j)$(to minimize) then
15         Accept the new solution;
16     end
17     Classify the current population of bats and find the best solution x*;
18     if fmin <= Eps then
19         break
20     end
21 end
22 return the best solution
```

In this experiment we consider vectors of dimension $d = 2$, however the code of each of the methods is extensible to higher dimensions. For the **AM** and **AG** algorithms, a population of $n = 10$ was considered. The search space in the case of **AM** was considered as lower bound -1 and upper bound 1. Regarding the **AG** the interval $[-10, 10]$ was considered to create a population with a uniform distribution. For the **DPM** and **BFGS** methods, a multi-start process was implemented which generates ten random points around $[-10, 10]$, therefore the values corresponding to these methods are calculated by means of the average of the values obtained from the multi-start.

2.2 Sphere function



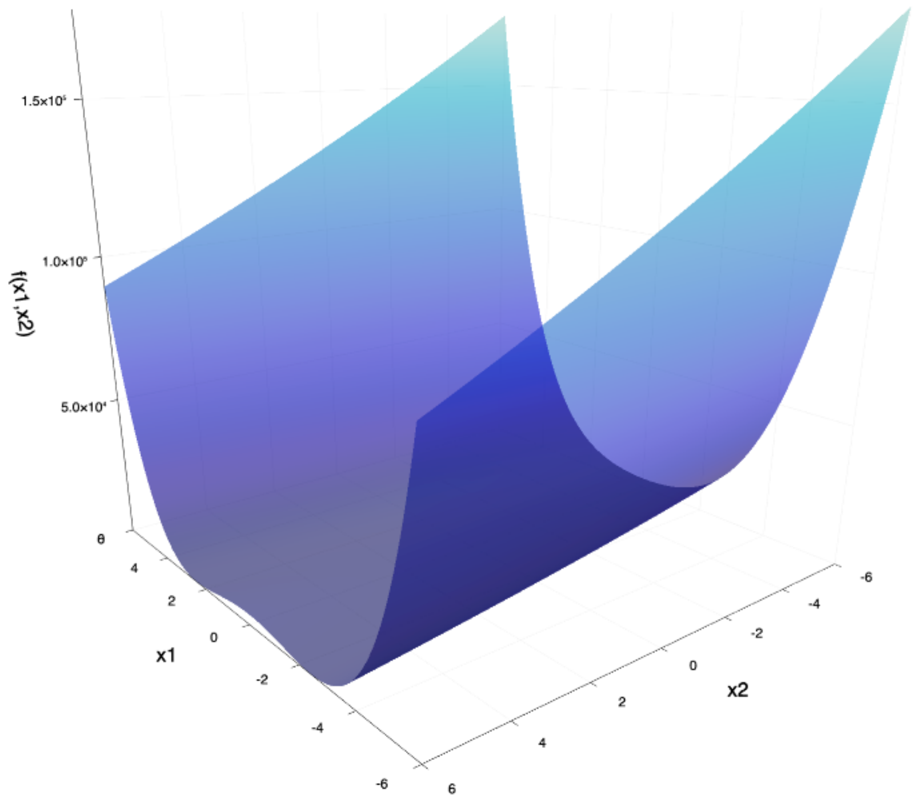
$$f(\mathbf{x}) = \sum_{i=1}^{d} x_i^2. \tag{9}$$

Figure 1. The Sphere function has d local minima except the global one. It is continuous, convex and unimodal. The plot shows its two-dimensional shape. The function is usually evaluated on the hypercube $x_i \in [-5.15, 5.12]$, for all $i = 1, \ldots, d$. **Global minimum**: $f(x) = 0$, at $x = (0, \ldots, 0)$.

Table 1. Comparative table showing the performance results of the bat algorithm, the genetic algorithm, the steepest descent method, and the BFGS quasi-Newton method for finding the minimum of the function (9).

Method/Algorithm	Objective Function Calls	Solution Precision	Runtime
AM	234	$(8.5939e^{-}5, 2.5939e^{-}5)$	0.191061 s
AG	48,233	$(-0.00012, 0.00018)$	0.726477 s
DPM	14.0	$(-4.955e^{-}8, -1.651e^{-}8)$	0.011141 s
BFGS	20.1	$(3.6642e^{-}8, 1.6834e^{-}8)$	0.015317 s

2.3 Rosenbrock function



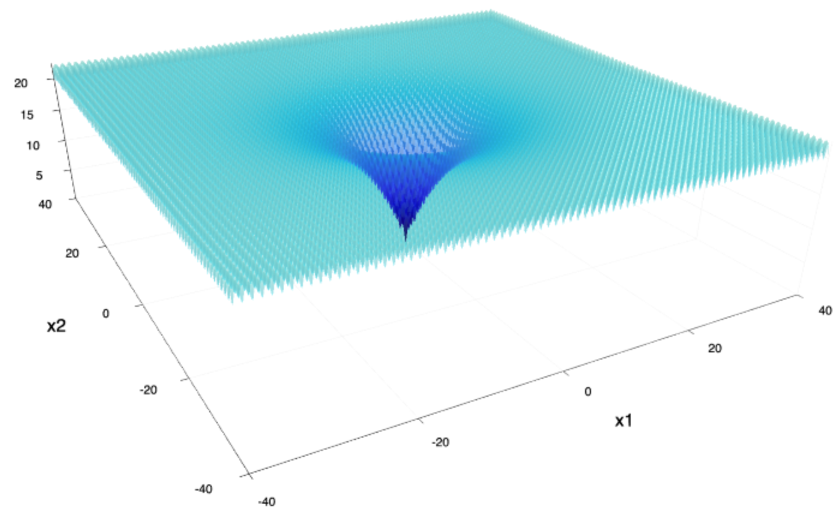
$$f(\mathbf{x}) = \sum_{i=1}^{d-1} \left[100\left(x_{i+1} - x_i^2\right)^2 + (x_i - 1)^2\right]. \tag{10}$$

Figure 2. The Rosenbrock function, also known as the Valley or Banana function. The function is usually evaluated on the hypercube $x_i \in [-5, 10]$, for all $i = 1, \ldots, d$, although it may be restricted to the hypercube $x_i \in [-2.048, 2.048]$, for all $i = 1, \ldots, d$. **Global minimum**: $f(x) = 0$, at $x = (1, \ldots, 1)$.

Table 2. Comparative table showing the performance results of the bat algorithm, genetic algorithm, maximum slope descent method, and BFGS quasi-Newton method for finding the minimum of the function (10).

Method/Algorithm	Objective Function Calls	Solution Precision	Runtime
AM	11	$(0.00905, -0.06643)$	0.182204 s
AG	3,515,251	$(0.9998, 0.9996)$	38.488292 s
DPM	8.1	$(3.74225, 5.21089)$	0.0175868 s
BFGS	33.0	$(-3.58000e29, 2.57790e22)$	0.1884089 s

2.4 Ackley function



$$f(\mathbf{x}) = -a \exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1). \quad (11)$$

Figure 3. The Ackley function is widely used to test optimization algorithms. In its two-dimensional form, as shown in the graphic above, it is characterized by a nearly flat outer region and a large hole in the center. The function poses the risk that optimization algorithms, particularly climbing algorithms, get stuck in one of their many local minima. The function is usually evaluated on the hypercube $x_i \in [-32, 768, 32, 768]$, for all $i = 1, \ldots, d$. **Global minimum**: $f(x) = 0$, at $x = (0, \ldots, 0)$.

Table 3. Comparative table showing the performance results of the bat algorithm, genetic algorithm, maximum slope descent method, and BFGS quasi-Newton method for finding the minimum of the function (11).

Method/Algorithm	Objective Function Calls	Solution Precision	Runtime
AM	457	$(-2.08598e^{-}8, -1.23800e^{-}9)$	0.204535s
AG	–	––	–
DPM	12.6	$(4.13902, 3.92473)$	0.0175133 s
BFGS	16.5	$(-3.49734e11, 79.11090)$	0.0172195 s

3 Conclusions

In this study it has been established that **AM** is potentially more efficient than **AG**. For **AG** you can see that you have too large a number of calls to the target function. On the other hand, the execution time is much longer than any of the other methods.

The accuracy of the solution obtained with **AM** is better than all other methods, however, the number of calls to the objective function is quite high compared to **DPM** and **BFGS**.

Relative to execution time, **AM** runs almost ten times longer than **DPM**and **BFGS**. It can be seen that the **DPM** in each case, is the method that has a better execution time and also makes fewer calls to the target function. Regarding the precision of the solution **DPM** has a good precision in the case of the function Sphere(9), but not for the Rosenbrock function(10) and Ackley's function(11).

While **AM**isn't the fastest or the fewest target function calls, it's clear that it's a highly competitive option.

An interesting and necessary follow-up to this work is to consider a wider range of existing algorithms that use much more demanding test functions in higher dimensions which will pose more challenges to the algorithms and thus potentially reveal the strengths of such comparisons. and weaknesses of all algorithms of interest.

Appendices

A Bat algorithm code.

```
using Distributions

 function AM(F::Function, population_size::Int64)
    """ ================================================= ====
    # INPUTS:
    #
    # Function objective function
    # size_of_population size of the population
    # ================================================= == """
    f_counter = 0 ; # counter of calls to the objective function
    # Parameter settings
    n = population_size;
    A = 1.0
    r = 1.0
    alpha = 0.97
    gamma = 0.1
    # Rango de frequencia
    freqMin = 0 ;
    freqMax = 2 ;
    # Tolerance
    eps = 0.0000001 ;
    iter = 0;       # iter counter
    # Variable dimensions
    d = 2 ;
    # Upper and lower bounds, respectively
    lB = -1
    uB = 1
    # Starting the arrays
    Freq = zeros(n,1) ; # Initial Frequency
```

```
v = zeros(n,d) ; # Initial Speeds
Lb = lB*ones(1,d) ;# Lower Bounds
Ub = uB*ones(1,d) ; # Upper bounds

# Starting population of n bats (solutions)
Sol = Array{Float64}(undef, n,d)
Fitness = Array{Float64}(undef, 1,n)
for i = 1: n
    Sol[i,:] = Lb + (Ub-Lb).*rand(1,d)
    Fitness[i] = sum(F(Sol[i,:]));
end
f_counter = f_counter + n ;
# Best bat(solution) in the initial population
fmin = findmin(Fitness, dims = 2)[1][1] # function value
index = findmin(Fitness, dims = 2)[2][1][2] # index
bestie = Sol[index,:]



# Start of iterations
while true
    # Variation of the parameters
    r = r*(1 - exp(-gamma*iter)) ;
    A = alpha*A ;
    # Create arrays to store values
    Freq = Array{Float64}(undef, n, 1) ;
    v = Array{Float64}(undef, n, d) ;
    S = Array{Float64}(undef, n, d) ;
    # Repeat over all bats
    for i = 1: n
        Freq[i] = freqMin + (freqMax - freqMin)*rand() ;
        v[i,:] = v[i,:] + (Sol[i,:] - bestie)*Freq[i] ;
        S[i,:] = Sol[i,:] + v[i,:] ;
        # Check the change condition
        if rand() > r
            S[i,:] = bestie' + 0.1*randn(1,d)*A;
        end
        # Evaluation of solutions
        Fnew = sum(F(S[i,:])) ;
        # If the solution improves
        if ( Fnew <= Fitness[i] ) & (rand() > A )
            Sol[i,:] = S[i,:] ;
            Fitness[i] = Fnew ;
        end
        # Update bestie in the population
        if Fnew <= fmin
          bestie = S[i,:] ;
          fmin = Fnew ;
        end
    end # for
iter = iter + 1 ;
f_counter = f_counter + 1 ;
if fmin <= eps # Stop criteria
```

```
        break
    end
    println("Iterations: ", iter," ,fmin: ", fmin) ;
    end # while
    return bestie, fmin, f_counter
end # function
```

References

Yang, X.-S. 2020. *Nature-Inspired Computation and Swarm Intelligence.* April. ISBN: 9780128226094.

Yang X.S. Cruz C. González J.R. Pelta, D. T. G. 2010. "A new metaheuristic bat-inspired algorithm." *Nature Inspired Cooperative Strategies for Optimization (NISCO 2010). In: Studies in Computational Intelligence* 284:65–74.