Nature Inspired Optimization

'Continuous Function Optimisation'

Matthew Flint - 1247903 - mxf203@bham.ac.uk

February 26, 2015

Contents

1   Introduction

This assignment aims to assess how you implement and experimentally compare different nature-inspired algorithms. It aims at giving you some first hands-on experience of applying a nature-inspired algorithm in a given context and drawing conclusions about some operators from a set of experimental results. We consider different mutation operators in the context of continuous function optimisation.
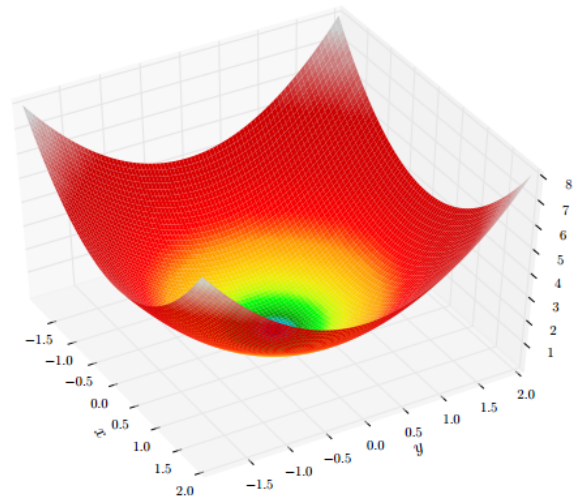
---

2   Task

Consider the following continuous benchmark function SPHERE:

$$SPHERE(x) = \sum_{i=1}^{n} x_i^2 \text{ with } -\infty < x < \infty$$

The SPHERE function has a unique global minimum in:
$$SPHERE(x_1, \ldots, x_n) = SPHERE(0, \ldots, 0) = 0$$

and can be visualised for n = 2 as shown above. It is a suitable benchmark to investigate if and how an algorithm starting from some arbitrary point in the search space is able to converge to a local optimum, which is an important property for nature-inspired algorithms.

In the lecture you have learned about two mutation operators in the context of continuous function optimisation, uniform and non-uniform mutation. Here, we consider these and two additional operators, Gaussian mutation with and without 1/5-rule. You can find explanations and pseudo code for all four operators in the appendix of this assignment. Your task is to implement and compare the performance of these four operators on the SPHERE function.

---

3   Detailed Instructions

The assignment requires you to implement the above four operators (Algorithms 2, 3, 4 and 5 in the appendix) within a simple (1+1) evolution strategy only using mutation (see Algorithm 1 in the appendix) on the SPHERE function in a programming language of your choice (e. g., Java, C, C++, Matlab). You will then run a set of experiments as detailed below to assess the performance of the mutation operators. Additionally, you are required to describe and discuss your implementation and your experimental findings in a report. The report should in particular discuss the following aspects (amongst any other points which you might consider important):

- How did you implement the algorithm and the fitness function (e. g., programming language, design decisions, . . .)? What data structures, algorithms or libraries did you use, e. g., for a random number generator? What other software/scripts did you use to facilitate your experiments/analysis?

- How does the performance of the different operators compare? How do parameters of the operators/the fitness function influence the performance?

- Try to explain why the operators behave the way you observe in your experiments. What properties, strengths or weaknesses can you observe? Can you name reasons for them?

As a minimum you need to run experiments using the following parameter settings. You are free to test additional settings if you feel these will help you with the discussion. Make sure to include these in your report as well. Note that uniform mutation does not have any parameters.

- A 10-dimensional SPHERE function (i. e., n = 10) with search space restricted to $[-100, 100]^{10}$
- Non-uniform mutation with parameter $b \in \{0.05, 1.0, 10.0, 20.0\}$.
- Gaussian mutation without 1/5-rule and parameter $\sigma \in \{0.05, 0.5, 1.0, 10.0\}$
- Gaussian mutation with 1/5-rule and initial uniformly distributed over [1, 100].

You need to perform 30 independent runs for each of the above settings. Each run should have an upper limit of 5,000 iterations. For each of the runs you need to record the fitness value for each iteration of the run. Visualise and present your data as follows:

- Plot the average function value over time (i. e., iteration number on the x axis; average function value over the 30 trails on the y axis). Use logscale for the y axes to improve visibility. You should have such a graph for each of the following scenarios:
  - a figure comparing different parameters for non-uniform mutation
  - a figure comparing different parameters for Gaussian mutation without 1/5 rule
  - a figure comparing uniform mutation, Gaussian mutation with 1/5 rule and the best (i. e., best fitness value after 5,000 iterations) parameter setting for non-uniform and Gaussian mutation without 1/5 rule
- Create a table stating the average fitness value and its standard deviation (over the 30 runs) after 5,000 iterations for each of the considered settings. You can determine these values with a statistics or spreadsheet software of your choice (e. g., R, Matlab, Microsoft Excel, LibreOffice).

---

4 Implementation

I chose to use JAVA as the programming language that I used to implement the algorithms, mainly because it is the language which I have most experience in. This enabled me to quickly convert the pseudo code functions and mathematical equations provided in the assignment JAVA for testing on my machine.

4.1 (1+1) evolution strategy algorithm

My implementation allows the user to first choose which mutation algorithm they wish to use via console input out of the four implemented. It then creates the initial search pool for the run, $x = (x_1, \ldots, x_n)$ by picking x_i uniformly at random from $[-100, 100]$. It then calls a function, iterate(), which loops T times (in my testing this was 5000) calling the selected mutation algorithm for each iteration and storing that iterations fitness calculation into an array. Finally, it outputs the array of the fitness for each iteration to a file. This whole process is then repeated 30 times for the 30 runs of the program we are required to perform and all of the results for each iteration of each run is stored in the output file (with spaces separating each result).

This has the effect of the program running and exporting the data for all 30 runs of 5000 iterations in one go which saves a lot of time and allows the data to be exported straight into a spreadsheet application (Excel).

4.2 Fitness Function

$SPHERE(x) = \sum_{i=1}^{n} x_i^2$ with $-\infty < x < \infty$ I implemented the fitness function in a dedicated method inside my code.

It has a parameter X which is an array of doubles, which the function calculates the sum of all of the elements squared.

```java
private double calculateFitness(double[] x) {
    double sum = 0;
    int count = n - 1;
```

```
    while (count >= 0) {
        sum = sum + (x[count] * x[count]);
        count--;
    }
    return sum;
}
```

I have checked this function against the graph provided for $X[n]$ where $n = 2$ which proved successful. Therefore I am confident in my implementation of this method.

4.3 Fitness Checking

I also implemented the method to check the fitness to see if the offspring is better than the parent in its own function. This method first calculates the fitness of the parent & offspring and then compares them to see which is better. If the offspring is better then the function returns boolean True and if not it returns False.

```
private boolean fitnessCheck(double[] offspring) {
    // Calculate the fitness of both offspring and X
    double offspringFitness = calculateFitness(offspring);
    double parentFitness = calculateFitness(X);

    // Check which is the better fitness
    if (offspringFitness < parentFitness) {
        return true;
    } else {
        return false;
    }
}
```

4.4 Initial search point

My function for creating the initial search pool uses Java's Math.random() to generate uniform random numbers to be used as the initial values for the array X. I have included a range on the random values produced to that it satisfies $[-100 < X_i < 100]$.

```
private void createInitialSearchPool() {
    // Populate the X array with random numbers
    for (int i = 0; i < n; i++) {
        double rand = searchSpaceMin + Math.random() * (searchSpaceMax);
        X[i] = rand;
    }
    System.out.print("Initial value of 'X' set to [ ");
    for (int i = 0; i < n; i++) {
        System.out.print(X[i] + " ");
    }
    System.out.println("]");
}
```

4.5 Output File

My function to write the output results to a file uses Java's FileReader & FileWriter functions. It first tries to see if the file already exists and if so it reads it into an array of lines. These lines are then used as prefixes to the current iterations data so that the eventual file contains all 5000 iteration's fitness calculations for each of the 30 runs (results separated by spaces). This data can then be easily parsed into a spreadsheet application.

4.6 Uniform Mutation Algorithm

My implementation of the Uniform Mutation Algorithm uses randomly generated numbers to pick an index of the X array and then replace it with a new value which is also randomly generated (within the bounds of $[-100 < x < 100]$).

```java
public static double[] UniformMutation(double[] x) {
    double[] xNew = x.clone();

    // pick a random index to mutate
    int max = x.length - 1;
    int min = 0;
    Random rand = new Random();
    int index = rand.nextInt((max - min) + 1) + min;

    // Generate a new random number for that index
    xNew[index] = -10 + Math.random() * (10 - -10);

    return xNew;
}
```

4.7 Non-Uniform Mutation Algorithm

My implementation of a non-uniform mutation algorithm chooses a random index of X and then mutates this value by adding or subtracting a random value. The mutation strength, i. e., the size of this change, decreases over time.

This implements the algorithm: $\triangle(t, y) = y \cdot 1 - r^{1 - \frac{t}{T}})^b$ which decreases the size of the mutation change as the number of iterations, t, increases. In my code this is also a separate function called NonUniformMutationDelta().

```java
public static double[] NonUniformMutation(double[] x,
        int currentIterationNumber, int numberOfIterations, double b) {
    double[] xNew = x.clone();

    // pick a random index to mutate
    int max = x.length - 1;
    int min = 0;
    Random rand = new Random();
    int index = rand.nextInt((max - min) + 1) + min;
    double newValue;

    // randomly select a mutation
    if (rand.nextDouble() >= 0.5) {
        double nonUniformMutationDelta = NonUniformMutationDelta(
                currentIterationNumber, (100 - xNew[index]), b,
                numberOfIterations);
        newValue = xNew[index] + nonUniformMutationDelta;
    } else {
        double nonUniformMutationDelta = NonUniformMutationDelta(
                currentIterationNumber, (xNew[index] + 100), b,
                numberOfIterations);
        newValue = xNew[index] - nonUniformMutationDelta;
    }
    xNew[index] = newValue;

    return xNew;
}
```

4.8 Gaussian Mutation

Gaussian mutation adds a normally-distributed random value to each component of the search point. The size of this value is controlled by a parameter σ, which is usually called step size. I used the apache distribution library to generate a normal distribution with standard deviation of 1 and then applied the distribution * step size to each element in the X array.

```java
public static double[] GaussianMutation(double[] x, double stepSize) {
    double[] xNew = x.clone();

    NormalDistribution dist = new NormalDistribution(0, 1);
```

```
    for (int i = 0; i < xNew.length; i++) {
        double t = dist.sample();
        xNew[i] = xNew[i] + (stepSize * t);
    }

    return xNew;
}
```

4.9   (1+1) evolution strategy with 1 5-rule

This evolutionary algorithm changes the step size based on how well the algorithm is doing according to the fitness function. It keeps a track of the number of successful & unsuccessful iterations and adjusts the step size accordingly (If this is larger than 1 5 we double σ , if it is smaller σ is halved).

This algorithm uses the Gaussuan mutation as described in the previous section and the changes are implemented in the $(1+1)$ evolution function. Essentially, if the success rate of the evolutions after i iterations is > 0.2 then the step size is doubled, else if it is < 0.2 then it is halved. This has the effect of broadening & narrowing possible mutation space for each iteration based on whether the algorithm is a long was off the local optimum.

4.10   Software

I have used the Eclipse IDE to program the assignment in JAVA, Microsoft Excel has been used as a spreadsheet application to process the data and produces graphs of the results. I have also uses LaTeXto write & produce the report.

---

5   Experimental Results

For all of my testing and the results presented I used a 10-dimensional SPHERE function (i.e. $n = 10$) with the search space restricted to $[-100, 100]^{10}$.

5.1   Non-Uniform Mutation with parameter $b \in \{0.05, 1.0, 10.0, 20.0\}$
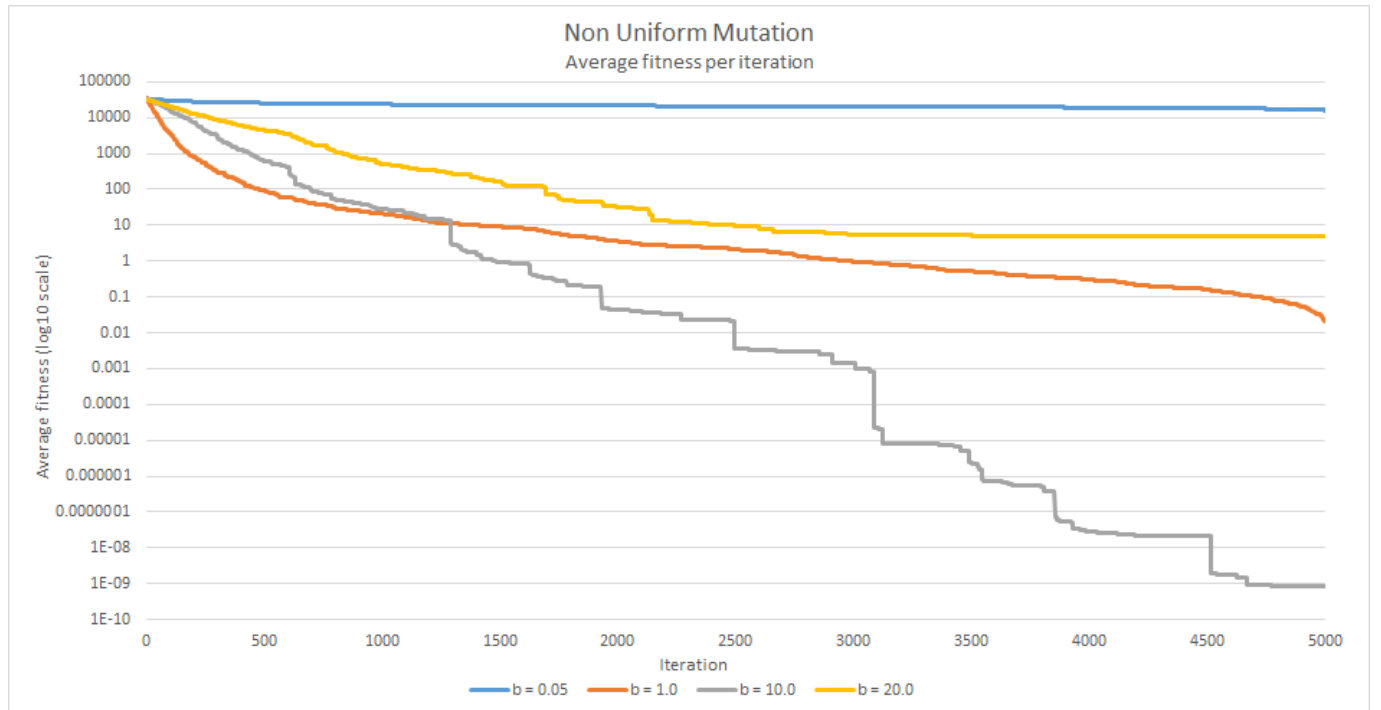
As we can see from the graph, for a 10-dimensional sphere function over 5000 iterations the Non-Uniform mutation algorithm did not succeed in finding an optimal solution. The best fitness would have been 0 but the algorithm does not manage to reach that.

The different parameters for b did yield separate and very different results (as shown in the graph). A value of $b = 0.05$ produces a very shallow curve with very little reduction in the average fitness value over the 5000 iterations. This proved to be the worst out of all of the parameter values that I tried. This could be due to the way in which the $\Delta(t, y)$ works as the whole second half of the equations is to the exponent b. In this case $b = 0.05$ which causes the whole equation to become very small, which is then multiplied to the value y. This causes a very small overall change in the values of X which could mean that it does not have sufficient time (in the 5000) iterations to deviate very much from its original values.

A value of $b = 1.0$ produces a rapid initial change (over the first few hundred iterations) which then steadily continues along a constant gradient. The change in the average fitness is greater than with $b = 0.05$ and it does get closer to the best average fitness value (with the lowest reached being 0.02. I believe this is again due to the $\Delta(t, y)$ function as the exponent in this case is 1 so it is effectively irrelevant. This means that the mutation is directly dependent upon the random number generated for r and the iteration out of the total iterations. This is why it starts to level off after a while, because as t approaches T the value of r approaches 1 so $\Delta(t, y)$ starts to approach $y \times 1$. This has the effect of not introducing as much variation into each iteration as it would have done at the start of the run.

A value of $b = 10.0$ produces a constantly declining line which has large jumps in some sections followed by a mini plateau for a while. This value turns out to be the best in my testing as it produced the lowest fitness values out of all of the runs (though it did not achieve the optimal solution). I believe this to be happening because in the $\Delta(t, y)$ equation, b is the exponent, so with this being 10 it is possible to jump large distances between values for x_i. However it is not too large so that the value it produces for x_i is either above or below the range $[-100, 100]$ and so would be truncated at the maximum/minimum values. This allows it to gradually step down from its local value.

A value of $b = 20.0$ produced a very high line on the graph. Is is a worse solution than both $b = 1.0$ & $b = 10.0$ however it is better than $b = 0.05$. I believe this is due to the $\Delta(t, y)$ equation, b is the exponent, so with this being 20 it is possible to jump large distances between values for x_i. In the majority of cases this might push it over the threshold range $[-100, 100]$ which would cause it to be set to the maximum/minimum value. As there is a 50% chance that either of these cases can happen, and as the value is hitting max/min thresholds often the line tends to stagnate and not vary as much as it could do. This causes it to not seek as many solutions and limits its reach for the best fitness value.

From all of this testing I believe that their is a trend that can be seen from the data.
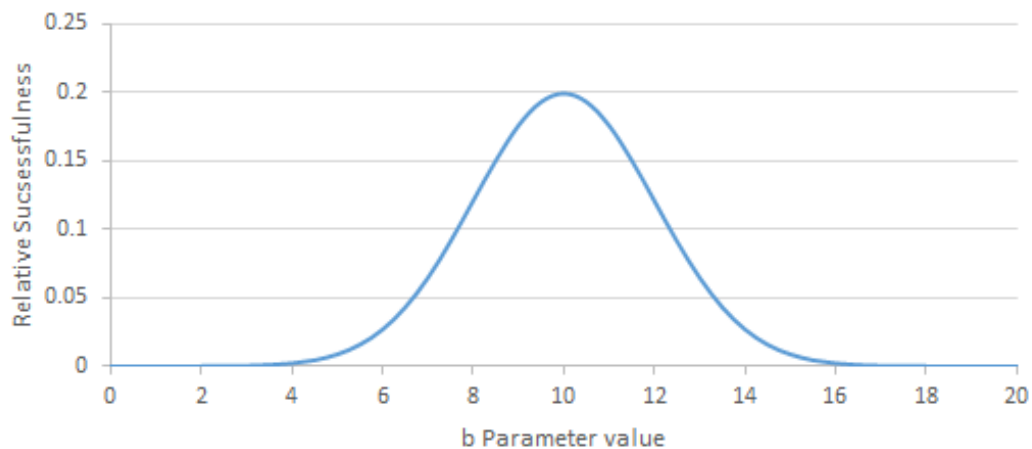
This graph shows the relative successfulness of the algorithm against its b parameter. I would expect that a b value around 10 to produce the best results and as you go either side of this you start to produce a worse solution.

5.2 Gaussian mutation without 1/5-rule and parameter $\sigma \in \{0.05, 0.5, 1.0, 10.0\}$
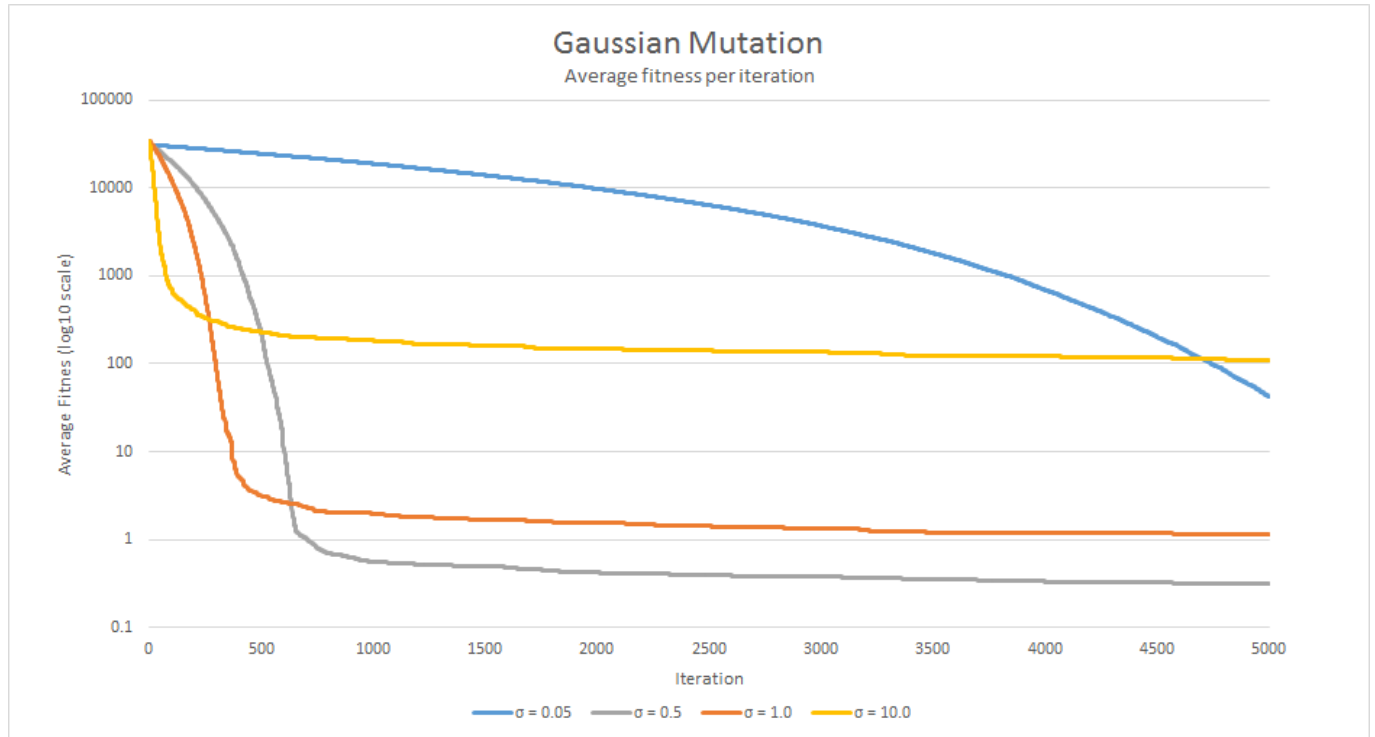
As we can see from the graph, for a 10-dimensional sphere function over 5000 iterations the Gaussian mutation algorithm did not succeed in finding an optimal solution. The best fitness would have been 0 but the algorithm does not manage to reach that.

The different parameters for σ did yield separate and very different results (as shown in the graph). A value of $\sigma = 0.05$ produces a very shallow curve (comparably) with a gradual reduction in the average fitness as the iterations continue. This is likely to be because σ is very small so the equation $x_i = x_i + \sigma \times m_i$ will have a very small quantity added to it, so it will not vary very much.

A value of $\sigma = 0.5$ produces better results, a very steep drop over the first few hundred iterations followed by a plateau of very little change. This produced the best result as it achieved the lowest average fitness value.

A value of $\sigma = 1.0$ produces similar results to $\sigma = 0.5$, a very steep drop over the first few hundred iterations followed by a plateau of very little change. However it did not find a fitness value quite as low and was about $3\times$ worse.

Finally, a value of $\sigma = 10.0$ produces a very steep initial search over just a couple of hundred iterations. However it soon levels out and then there is not much change for the remainder of the 5000 iterations.

5.3 Comparison of Uniform Mutation, Gaussian mutation with 1/5 rule and the best parameter setting for non-uniform and Gaussian mutation without 1/5 rule.
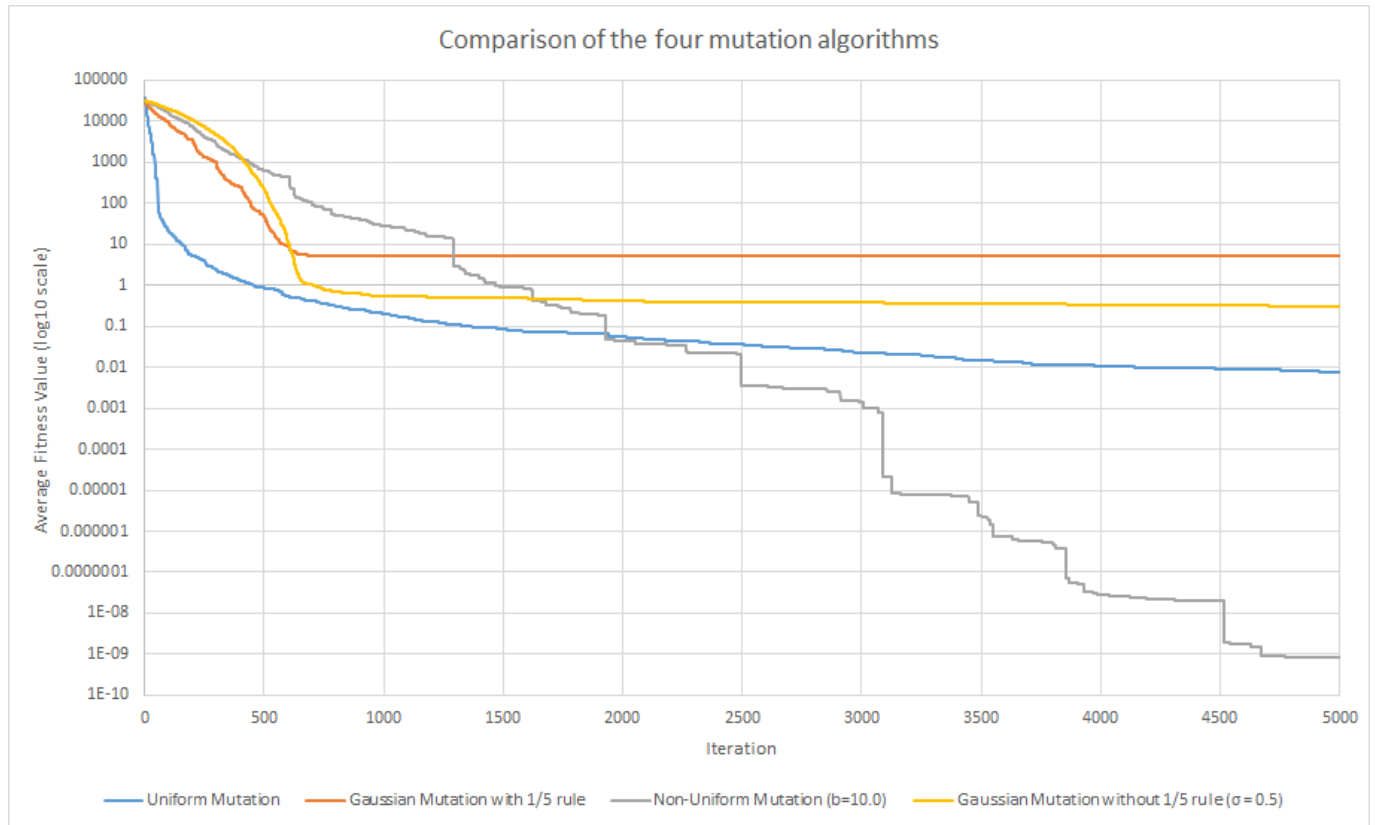
This comparison graph of all of the four different mutation algorithms (choosing the best parameters in the case of Non-Uniform and Gaussian mutation without 1/5 rule) shows clearly that one of the algorithms far surpasses the others in terms of finding the best fitness value. That is the Non-Uniform mutation algorithms with a parameter of $b = 10.0$.

We can also see how the others compare to each other, with Uniform Mutation coming in 2nd (though far behind 1st place). 3rd going to Gaussian Mutation without 1/5 rule and a parameter of $\sigma = 0.5$. And finally Gaussian Mutation with 1/5 rule coming in last.

---

6 References

1. Arkin, R. C. (1998).
   Behavior-based robotics. MIT press, p. 1, 15, 20, 24, 212-213

2. Mark W. Spong and Masayuki Fujita, 2004.
   Control In Robotics. IEEE Control Systems Society.
   Available: ieeecss.org/sites/ieeecss.org... [Online]
   (Accessed 11/11/2014)

3. Stuart J. Russell and Peter Norvig. 2003.
   Artificial Intelligence: A Modern Approach (2 ed.). Pearson Education.

4. John Kelleher,
   Robot Control Architectures, [DT211/1 Applied Computing, DIT School of Computing].
   Available: www.comp.dit.ie/jkelleher... [Online]
   (Accessed 11/11/2014)

5. Margaret E. Jefferies, Wai-Kiang Yeap,
   Robotics and Cognitive Approaches to Spatial Mapping,

Volume 38 of Springer Tracts in Advanced Robotics, ISSN 1610-7438. p 175.

6. S. Thrun, D. Fox, W. Burgard, and F. Dellaert,
   Robust monte carlo localization for mobile robots. Artificial Intelligence 128,1-2 (2001), 99141.
   Available: citeseer.nj.nec.com/th... [Online]
   (Accessed 20/11/2014)

7. S. Thrun, D. Fox, W. Burgard,
   Markov localisation for mobile robots in dynamic environments. Journal of Artificial Intelligence Research 11
   (1999), 391-427.
   Available: citeseer.nj.nec.com/... [Online]
   (Accessed 20/11/2014)

8. D. Fox,
   Markov Localisation (1999).
   Available at: cs.cmu.edu/afs/cs/... [Online]
   (Accessed 25/11/2014)

9. J. J. Leonard and H. F. Durrant-Whyte,
   Directed Sonar Sensing for Mobile Robot Navigation, Kluwer Academic, Boston, 1992.

10. B. Schiele and J. L. Crowley,
    A comparison of position estimation techniques using occupancy grids, in Proceedings of IEEE Conference on
    Robotics and Automation (ICRA), vol. 2, pp. 1628 to 1634, 1994.

11. S. Thrun, D. Fox, W. Burgard, and F. Dellaert,
    Monte carlo localization for mobile robots. Institute of Computer Science III, University of Bonn, D-53117 Bonn.
    Available at: www.cc.gatech.edu/... [Online]
    (Accessed 25/11/2014)

12. ROS.org,
    AMCL Package Summary,
    Available: wiki.ros.org/amcl [Online]
    (Accessed 21/10/2014)

13. D. Fox et al,
    "Particle Filters for Robot Localisation",
    Available: citeseerx.ist.psu.edu/viewdoc... [Online]
    (Accessed 24/11/2014)

14. R. R. Murphy.
    Introduction to AI Robotics. The MIT Press, Cambridge, Massachusetts, 2000.

15. L. De Silva and H. Ekanayake,
    Behavior-based Robotics And The Reactive Paradigm - A Survey. University of Colombo School of Computing.
    Proceedings of International Workshop on Data Mining and Artificial Intelligence (DMAI 08).
    Available: www.cs.utah.edu/ãlnds/p... [Online]
    (Accessed 27/11/2014)

16. D. Fox,
    Adapting the Sample Size in Particle Filters Through KLD-Sampling . Department of Computer Science, University of Washington.
    Available: www.cs.washington.edu/ro... [Online]
    (Accessed 27/11/2014)

17. D.Fox,
    KLD-Sampling: Adaptive Particle Filters, Department of Computer Science, University of Washington.
    Available: papers.nips.cc/paper... [Online]
    (Accessed 27/11/2014).

18. D. Pinto,
    The Kullbach-Liebler Distance, 2007.
    Available: www.cs.buap.mx/... [Online]
    (Accessed 27/11/2014)

19. Jean-Claude Latombe,
    'Robot Motion Planning: Edition en anglais'. The Springer International Series in Engineering and Computer
    Science, Springer Science and Business Media, 1991 p.6, 22

20. D.V. Lu, D. Herschberger, W.D. Smart,
    Layered Costmaps for Contest-Sensitive Navigation, Washington University in St. Louis, Department of Computer Science and Engineering.
    Available: www.cse.wustl.edu... [Online]
    (Accessed 02/12/2014)

21. Zaman, S.; Slany, W.; Steinbauer, G.,
    "ROS-based mapping, localization and autonomous navigation" Electronics, Communications and Photonics Conference (SIECPC), 2011 Saudi International , vol., no., pp.1,5, 24-26 April 2011 doi: 10.1109/SIECPC.2011.5876943.
    Available: ieeexplore.ieee.org... [Online]
    (Accessed 02/12/2014)

22. Pioneer P3-DX,
    Adept Mobile Robots,
    Available: mobilerobots.com/... [Online]
    (Accessed 02/12/2014)

23. Scanning Range Finger (sokuiki sensor),
    hokuyo-apt.jpt,
    Available: mhttps://www.hokuyo-aut.jp... [Online]
    (Accessed 02/12/2014)

24. M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, A. Ng,
    ROS: an open-source Robot Operating System Computer Science Department, University of Southern California.
    Available: ai.stanford.edu.. . [Online]
    (Accessed 02/12/2014)

25. What is ROS,
    ROS.org,
    Available: wiki.ros.org/ROS/... [Online]
    (Accessed 02/12/2014)

26. Navigation Package Summary,
    ROS.org,
    Available: wiki.ros.org/navigation [Online]
    (Accessed 02/12/2014)

27. Marder-Eppstein, E., Berger, E., Foote, T., Gerkey, B. and Konolige, K. (2010).
    The Office Marathon: Robust navigation in an indoor office environment. 2010 IEEE International Conference on Robotics and Automation.
    Available: wiki.ros.org/Papers... [Online]
    (Accessed 29/12/2014)

28. Bellotto, N. and Huosheng Hu, (2009).
    Multisensor-Based Human Detection and Tracking for Mobile Service Robots. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 39(1), pp.167-181.
    Available: webpages.lincoln.ac.uk/... [Online]
    (Accessed 29/12/2014)

29. Beccera, M. (2012)
    "Sistema De Seguimento de Personaspara un Robot Novil de Servicio". Universidad Nacional Autonoma De Mexico.
    Available: https://dl.dropboxuserco... [Online]
    (Accessed 29/12/2014)