Global Optimization on Complex Systems

Author:
David PÉREZ GONZÁLEZ

Supervisor:
Dr. Javier HERNÁNDEZ ROJAS

A final degree project submitted in fulfillment of the requirements for the degree of Grado de Física

in the

Department of Physics

July 3, 2018

Ach, die Physik! Die ist ja für die Physiker viel zu schwer! (Oh, physics! That's just too difficult for the physicists!) (¡Ay, la física! ¡Es demasiado difícil para dejársela a los físicos!)

David Hilbert.

UNIVERSITY OF LA LAGUNA

Abstract

Faculty of Sciences

Department of Physics

Grado de Física

Global Optimization on Complex Systems

by David PÉREZ GONZÁLEZ

Since years, optimization methods have been implemented in several fields to enhance the chances of getting solutions of problems where the time for solving them is practically the main obstacle. Their applications have been tested a plenty of times in different branches of science, from basics electronics to molecular biology, or even for calculating crystal structures in chemistry.

In this work, we are seeking for getting to know better the working mode of these methods. For that, we are focusing on the global optimization of Lennard-Jones clusters.

Our main goal is to explain the algorithms used for optimizing the potential that describes these clusters with its implementation in Python. With that, we will measure the efficiency of our programs testing some special clusters deeply and we will compare them checking which one is more efficient.

Durante años, los métodos de optimización global han sido implementados en diversos campos para mejorar las oportunidades de conseguir soluciones a problemas donde el tiempo para resolverlos es su mayor obstáculo. Sus aplicaciones han sido probadas con rigor en muchas ramas de la ciencia, desde electrónica básica hasta biología molecular, o incluso en el cálculo de estructuras cristalinas para diversos compuestos químicos.

En este trabajo de fin de grado, buscamos la manera de conocer mejor cómo se trabaja con estos métodos. Para ello, nos centramos en la optimización global de agregados de Lennard-Jones.

Nuestro principal objetivo es dar a conocer los algoritmos que hemos utilizado para la optimización de este potencial que describe a los agregados y luego, su implantanción en el lenguaje de Python. Con ello, podremos medir la eficiencia de nuestros programas analizando algunos agregados especiales con un estudio más exhaustivo y observar cuál es el más eficiente.

Contents

Chapter 1

Introduction

Global Optimization is all about finding the most effective conditions to reach our goal that is to determine the optimal value of a given function from all possible solutions. However, we also have to mention the concept of Local Optimization that, at the end, is the same but restricted to a set of possible solutions that are near by the same domain.

These definitions fit quite well with our global understanding of mathematics, but we cannot let Fermat, Euler, Lagrange, Newton, Leibniz and many others scientists out. Optimization is like it is nowadays, thanks to their development of a theory from extreme problems. Because of that, this discipline became one of the most useful fields in several branches of science and even other fields such as economics or engineering.

There are several examples of the utility of optimization, but since our objective is to focus on a specific one, we will only shortly mention a few of them. From the well-known traveling salesman-type problem, the design of microprocessor circuitry, the flow in a pipe network, the notorious protein folding problem [11] until all the new GPS apps that are really famous today, we can see the application of optimization at the bottom of their algorithm.

To express this concept mathematically:

Let be $f(x)$ a function where $x \in \mathbb{R}$.

We set a domain $D \in \mathbb{R}$ where if $x_0 \in D \Rightarrow |x - x_0| \leq \delta$

When $1 \gg \delta > 0$ *and* $f(x_0) \leq f(x) \Rightarrow x_0$ is a local minimum.

Once we can extend δ to the whole space we are working with, or at least that it is big enough for our purposes, in other words that x_0 is not only as good as any nearby points (local minimum), but also as every feasible point; then, we can define x_0 as 'global minimum', though we must know that it is not definitive. [1]

Otherwise, we know that for getting the minimum, if we have the form of the function, it is enough with equalling the derivative of the function to zero, mathematically, through the Fermat's theorem:

"Assuming $f(x)$ as the function before, if $f(x)$ is differentiable at the region near by x_0 and we know x_0 is a local minimum, it implies that $f'(x) = 0$" [2]

Then, we can get the conditions for that. If the form of the function is still not clear or we cannot get the conditions for that in a proper way, we can still use it for computing.

Moreover, once we want to compute a program for calculating the global minimum, it turns out that most of the times the program is getting a local minimum instead. That is because of the constraints we are giving to the domain D. Thus, the main task is trying to figure out how these parameters or restrictions should be tuned for efficiency.

In this work, our target is to focus on one of the most interesting optimization problems and explaining the methods used for solving it. Within materials science, finding the structure that minimizes the (free or not) energy of a system given is a hot topic still. In our case, the ground-state structure of a nanocluster of atoms interacting is the problem to solve. [3]

For this problem, we need to figure out which simple model can describe the system we are interested in. We know there are forces between atoms that depend on the position of them and the structural optimization we want to calculate means finding the positions of these atoms and for that, we have to take into account the potential energy of the structure we are looking for. The chosen model is the classical Lennard-Jones (LJ) pair potential, a well-known potential in physics that will be described in the chapter 2. [4]

"Why is it important to locate the global minimum?" Once we achieve the global minimum, the positions of the atoms is likely the one that a real cluster would form although we always have to check if the final distribution is the one obtained by an experiment with temperature equal to 0 and whether it is physically reasonable or not. [5]

The traditional approach to compute the global minimum is to perform several calculations of the potential from an initial distribution of the atoms. This first structure can be taken from either a specific one or random one. Then, we can use a special algorithm for calculating the local minimum. If we do this a lot of times or we choose the initial configuration in a proper way, we can find the global minimum.

However, as N (number of atoms) increases and consequently the size of the cluster, the time-consuming increases as well because the number of local minimum grows exponentially which reduces the likelihood of finding the global minimum, as we can observe in the table 1.1 that shows the known (or estimated) number of minima for different Lennard-Jones clusters. For this reason, some researchers are using heuristic methods based on an analogy to some natural process. [3]

N	2	4	7	10	13	15	19	33
Minima	1	1	4	57	366	~ 10700	$\sim 2 \cdot 10^6$	$\sim 4 \cdot 10^{14}$

TABLE 1.1: Number of minima found in some Lennard-Jones clusters [6].

One of these natural processes used for the analogy is the simulated annealing (SA) [3], a method that we will explain in the chapter 3. Also some traditional Monte Carlo methods can be used.

Nevertheless, some of these methods as the one we mentioned before (SA) are not able to find some more complex structures in bigger clusters or they take too much time. It is caused by the different and particular types of interatomic interactions. Thus, some investigators have proposed either modifying the methods mentioned before or using new ones that depend on another kind of strategy. [3]

On one hand, from the first group, we have a very common and popular method for calculating cluster structures, the basin-hopping [7]. We will take a closer look at it in the chapter 3.

On the other hand, from the second group, we have a very innovative and recently developed method not only for simple or theoretical cluster structures, but also for more complex and specific ones, the genetic algorithm [5]. In the chapter 3 we are explaining the theoretical frame this method is based on.

Finally, in the appendix B and C, we are also including the development of a program written in Python using the techniques mentioned in the respective chapters. Thanks to that, we will analyze the way the parameters (constraints) behave and the iterations needed from our programs to find the global minimum. Then, we are comparing the efficiency or even the succeed of both methods in the chapter 4. Apart from that, we are comparing some examples and interesting cases using our program with an official database [7] in the same chapter 4. With that, we can use the LJ potential as a test-bed for the global optimization algorithms we are developing to get to know if we can use them for other purposes.

1.1 Resumen en Español

La optimización global es un problema matemático que consiste en encontrar el valor óptimo de una función, que puede estar sujeta a ligaduras. Es un tema muy activo estos últimos años por sus enormes aplicaciones prácticas en diversos campos, desde problemas típicos del vendedor que quiere realizar el recorrido más corto posible, el diseño de circuitos electrónicos para encajarlos en carcasas especiales o reducir el calor que producen, hasta el plegamiento de proteínas y un largo etcétera que ocupa una extensa bibliografía.

La forma de obtener este valor óptimo puede ser muy variada, utilizando algoritmos de todo tipo. En este trabajo, sin embargo, nos centraremos en los tres métodos más eficientes y contrastados que existen en la bibliografía: a) Algoritmo Genético (“Algorithm Genetic”), b) Enfriamiento Simulado (“Simulated annealing”) y c) Salto entre mínimos (“Basin-Hopping”). Estos dos últimos los estudiaremos con gran detalle analizando los resultados encontrados en los agregados de Lennard-Jones y comprobando su utilidad. Por otro lado, el denominado Algoritmo Genético será materia de estudio de forma bibliográfica para su descripción y comprensión.

Chapter 2

Lennard-Jones Potential

In this chapter we are discussing shortly about the Lennard-Jones potential for computing the potential energy surface of a cluster knowing the positions of the atoms we are working with.

2.1 Presenting the Lennard-Jones potential

This potential is useful when we can take into account only the interaction caused by the electrostatic force induced between the dipole atoms, a consequence of fluctuations in the charge distribution. This model is named van der Waals solids. [8]

The formula for our potential between only two atoms with r the distance between them is:

$$V_{LJ}(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{2.1}$$

where ϵ is the depth of the potential well and σ is the finite distance at which the interparticle potential is zero. This potential function is plotted in figure 2.1.

"Why are the terms of the potential as they are?" The second term of the potential, with the exponent 6, can be derived from a second order perturbation theory of the induced dipole induced dipole interaction energy.

If we let the potential only with that term, the atoms would tend to come closer and closer to each other. Thus, to fix this in the LJ potential, they included the first term in a way that energy will increase rapidly when r is getting smaller and smaller.

By choosing these exponents, we can manipulate the potential easily in a way that, for example, in the case we have only two atoms, as we can see in the figure 2.1, getting the minimum of the function is straight forward (for the derivation, see the appendix A):

$V = -\epsilon$ and $r_{min} = \sqrt[6]{2}\sigma$

Once we want to take into account a longer amount of atoms, we use a Lennard-Jones 12-6 pairwise potential:

$$V_{ij}(r) = 4\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] \tag{2.2}$$

And the final potential would be:

$$V(r) = \sum_{i=2}^{N} \sum_{j<i}^{N} V_{ij}(r) \tag{2.3}$$

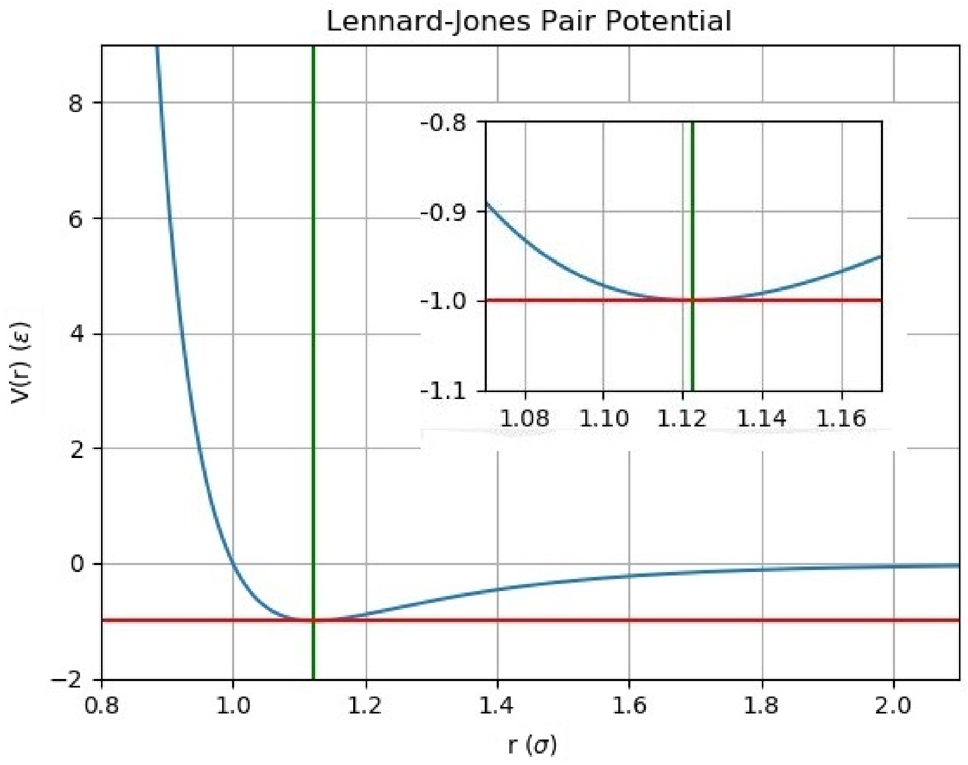

FIGURE 2.1: LJ pair potential with $\epsilon = 1$ and $\sigma = 1$.

Lennard-Jones potential provides us clusters with numerous features. They are not only an useful testing ground for global optimization algorithm (it is easy to program and some clusters, such as LJ_{38} and LJ_{75}, can discriminate between methods that are likely to be useful and those that are not), but also for analysis of structure, dynamics and thermodynamics in terms of the underlying potential energy surface. Otherwise, taking into account its simplicity, it is really remarkable that many of the global minimum structures have been observed experimentally for clusters of atoms and molecules from real elements. [9]

For getting the global minimum, we can also use the derivative, since for the programming methods we are using it is going to be really useful in a way that we can get the final solution easier and it will be more accurate. There are even some of them that are asking for it explicitly. For that, we are following the same steps as for the derivative with only two atoms.

In the following line we are presenting the derivative of the LJ potential for a dimer:

$$V'(r) = 24\epsilon \sum_{j\neq i}^{N} \left(\frac{\sigma^6}{r_{ij}^7} - \frac{2\sigma^{12}}{r_{ij}^{13}} \right) \tag{2.4}$$

The lower bound obtained for N = 2 coincides with the one for N = 3 and N = 4 that actually follows an equation: -N(N - 1)/2 (assuming that all pairs are at their equilibrium separation. However, from N = 5 onwards it is impossible to follow this rule and the energy is becoming larger and larger. [3]

2.2 Resumen en Español

En este capítulo discutimos acerca de la obtención del potencial de Lennard-Jones y su manera de computar la superficie de energía potencial de los agregados con su mismo nombre, a través de la posición que ocupan sus átomos en el espacio. También, describimos qué información podemos obtener acerca del mismo según los agregados que estudiemos. Añadimos su derivada puesto que muchos de los algoritmos de cálculo suelen requerir la misma para encontrar el propio mínimo local (un punto estacionario de la superficie de energía potencial).

Chapter 3

Global Optimization Methods

In this chapter, we are describing the three principle methods for global optimization. In the first section, we are briefly introducing in a bibliographical way one of the most innovative method, genetic algorithm (GA) [5]. In the second place, we will have a closer approach to one of the first global optimization methods, simulated annealing (SA) [3], because we are using it for finding the global minimum of the LJ potential given for calculating the most stable structure for a given number of atoms. Examples from that are presented in the next chapters. Finally, a third method is considered in this report, basin-hopping [7], a global optimization method whose results have been proved several times successfully. For this last method, we are also calculating the same as for the simulated annealing [3]. Thus, we are able to compare both methods with the examples mentioned before.

3.1 Genetic Algorithm

In this section, we are explaining the basic concepts of genetic algorithm [5] within its theoretical framework taking into account that it is an important method nowadays, but we are not applying it to our system.

We have a global optimization method that uses the principles of genetics, the natural evolution. For that, it is taking some operators that have the same function that their names in nature have. This analogy can be used with the parameters of the problem, too. Since we can compare ’genes’ with variables, the compounds of genes (’alleles’) with the individual value of the variables and even a chain of genes (’chromosomes’) with a string that is a trial solution of the problem. This comparison can be observed in the figure 3.1. [5]

Assume an initial population generated randomly that corresponds to the starting set of possible solutions of the function we want to optimize with genetic algorithm (GA) [5]. Then, all the strings are relaxed into the nearest local minimum by a minimization routine. With this new configuration, we are analyzing this first attempt giving to every string a parameter, called fitness, that is generating a measurement of the quality of this string in comparison with the function being optimized. This parameter is working with a chosen function to be more restricted with the solution or not. For example, if we are interested in minimizing a function, a low fitness will tell us that this string is good. [5]

Once all the strings are defined by a value of fitness, then they are compared to be chosen for being mating. Here, the fitness is important because it is giving more likelihood to be chosen to the ones that have a good fitness. This selection is done by mainly one way. This technique is named 'roulette wheel' and it consists on comparing the fitness with a random number given to each string. Since the fitness is taking values only from 0 to 1, this random value will be in the same interval as well. [5]

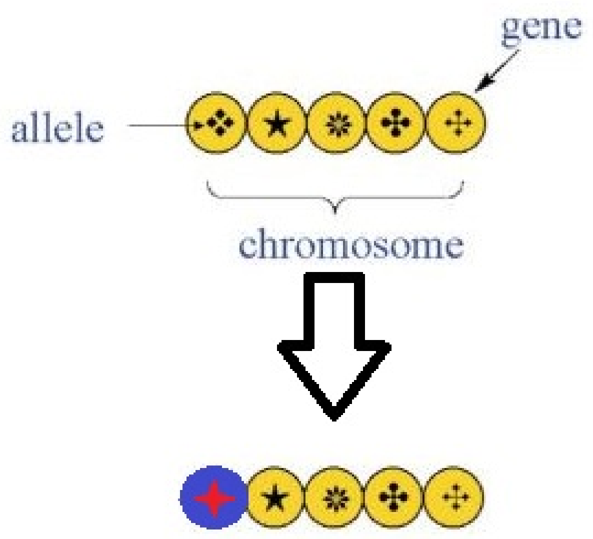

FIGURE 3.1: Schematic representation of the mutation operation of one individual where a single variable is modified. [5]

Since the strings are chosen, they are modified by some different operators such as 'mutation'. This one is modifying the values inside of the string, exchanging them or removing some of them to take new ones, as we can see in the figure 3.1 where we are removing one 'gene' for a new one. Then, the function is again relaxed and evaluated. [5]

After the new evaluation, where a new fitness value is given, is done, we will proceed with the 'natural selection'. In this step, the strings whose fitness value is not good enough are removed from our population and new ones are taken. [5]

Finally, we check if the new set of solutions converges to the one we are looking for. If yes, we keep this new set of solutions as the final set of solutions. If not, we start again the loop assigning a new fitness value. [5]

3.2 Simulated Annealing

Some researchers use some natural processes as an analogy for programming an algorithm of optimization. How this method is built is always compared with the process that has the same name in the metallurgy industry, but its theoretical basis approach is deeply rooted in statistical mechanics. Otherwise, there are some problems with this method if you want to use it for finding a global free energy minimum that changes with temperature because then, the system may become trapped inside of a well. [10]

This method imitates the dynamic process in which a system (generally metals) is heated until all their parts are malleable and then it is shaped in the way one wants meanwhile it is being cooled down. We will apply something similar to calculate the global minimum in the Lennard-Jones potential. For example, in the figure 3.2, we can observe how a random potential looks like and then, our trial (the ball) is being modified randomly as well for getting the global minimum ($x*$).

3.2.1 Method itself

We will take advantage of the fact that we have programmed this method in Python already (see Appendix B for a more detail version). Thanks to that we can explain the method itself by describing the code we programmed following the instructions of the supervisor. In the following, we will explain the implementation of this method.

First of all, we need to generate a random distribution of atoms in a tridimensional space. Keeping this thought in mind, we are generating a sphere (with radius $r = 5\sigma$) from where the 'atoms' that are chosen are located all in places where the distance to the nearest ones is at least the equilibrium separation ($r_{min} = \sqrt[6]{2}$).

After that, we are using the distances calculated before for obtaining the LJ potential (taking into account that $\epsilon = 1$ and $\sigma = 1$ for simplicity). For that, depending on how many atoms we have, there will be more or less terms.

Once we have the initial value for the potential from the initial distribution of atoms, we define the parameters we need for modifying the position or not.

These parameters are Δ and T. On one hand, the Δ value is used for changing the position of the atoms a certain quantity. On the other hand, the T (temperature) value is following the analogy mentioned at the beginning because its main purpose is to determine the likelihood for choosing a new value for the potential that, in principle, is not lower. We will explain that in more detail later.

After that, we set a condition for the loop we want to run based on the experience of our supervisor. This condition is the difference between the new value of the potential and the one we had before. This difference has to be less than 10^{-3} to stop the loop. Inside this loop we are choosing one atom randomly and changing its position following the next formula:

$$x_{new} = x_{old} + 2\Delta(\xi - 0.5) \tag{3.1}$$

where ξ is a random number distributed uniformly between 0 to 1.

We are doing that with all three axes and then recalculating the distances between the atoms this position affected. Here, we are also including a condition for the variation of the position to not let them leave a box of size 10 x 10 x 10. This condition is only made to avoid the possible case that the atoms are all going away from each other.

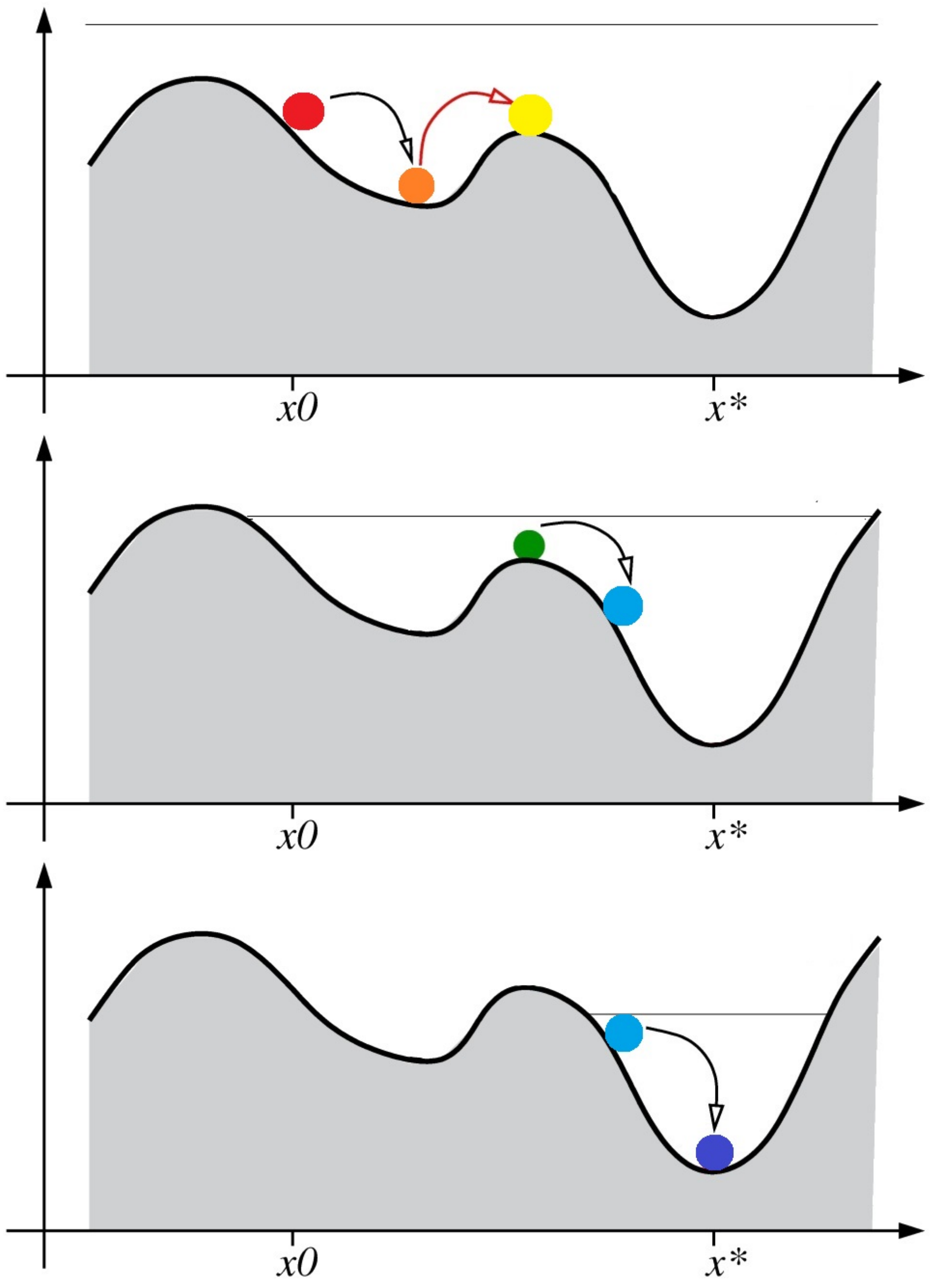

FIGURE 3.2: The horizontal line marks the maximum variation our trial can take depending on its temperature (from high T to low T: red>orange>yellow>green>blue). [18]

Hence, we recalculate the potential as well and here, it follows one of the most important steps in the program and also one of the characteristics of the Simulated Annealing method [3], the application of the Metropolis criterion to accept or reject new positions.

In the following, we will explain this criterion. If the new potential value is lower than the one we had before, the old one is exchanged by this new one immediately. Nevertheless, if this does not hold, then the new position of the atoms and, in consequence, the change in the potential, are accepted according to a Boltzmann probability that obeys the next expression:

$$exp((-(V_{new} - V_{old}))/k_B T)$$

where, T is the 'temperature', V is the LJ potential and k_B is the Boltzmann constant. The Boltzmann constant is set to one and, in consequence, the 'temperature' has reduced units.

Since we know the new value for the potential will not be larger than the old one, we know the value of the exponential will be always positive and one as maximum. Thus, we are taking a random number between 0 to 1 and if this value is lower than the one obtained for the exponential, we are accepting the new value of the potential and exchanging the position of the atoms. If not, we set the old values again.

Here, the 'temperature' is becoming more important because if the T value is high, then the likelihood of accepting the new state is also higher and once the 'temperature' is falling, the probability of changing the position of the atoms is practically nothing. Moreover, the T value is decreasing following a negative exponential function and only every 50 iterations in the loop.

Additionally, we set a condition for changing the Δ value. If after 100 iterations, we are accepting more new values than we reject, we change the Δ value multiplying it by 1.05 in a way that then, the changes are bigger and we can find a better value. If it is in the other way around, we change the Δ value multiplying it by 0.95 in a way that then, the changes are smaller and we can go more straightforward to the global minimum.

Once the loop is done, the minimization routine takes place to quench the structure found to the global minimum. This step is also telling us how far the trajectory is going down because the routine has its own condition. The fully trajectory is then, the sum of the iterations from the method itself and the routine, as we will show later. It is noticed that this method has problem obtaining a local minimum instead of the global one as we will compare in the next chapters.

Moreover, we are plotting the final solution to get to know if the structure obtained does have physical sense or not.

Furthermore, by plotting the final positions of the atoms, we can compare them with the ones that are plotted in some database [7], as we will do in the chapter 4 in more detailed.

3.3 Basin-Hopping

This method arises from the necessity of generating a more complex algorithm for getting more complex structures. It was developed to be useful for functions that consist of many minima separated by large barriers. Moreover, it is based on stochastic algorithm and precisely for that reason, although there is no way to determine if a true global minimum is found, we can check if from a plenty of random starting points, the function always converges to the same point. It is also an iterative method whose steps will be explained later, but they are mainly three [11]:

1^o) Random perturbation of the set of solution we have generated.

2^o) Use of routine minimization for getting the local minimum.

3^o) Accepting or rejecting the new set of solution based on the Metropolis criterion.

At the end, the potential energy for every point in the catchment basin of each local minimum becomes the energy of that minimum in a way we can divide the surface of our function into different basins and then, comparing them to get the global minimum. A graphical way to observe that is given in the figures 3.3 and 3.4. From a mathematical point of view:

$$\tilde{V}(\mathbf{X}) = min\{V(\mathbf{X})\} \tag{3.2}$$

where we are carrying out an energy minimization from different points ($\mathbf{X}$) producing a landscape consisting of plateaux at the energies of the local minimum. This transformation does not affect the relative energies at those minima.[10] Actually, "aside from removing all the transition state regions from the surface, the catchment basin transformation also accelerates the dynamics, because the system can pass between basins all along their boundaries. Atoms can even pass through each other without encountering prohibitive energy barriers."[11]

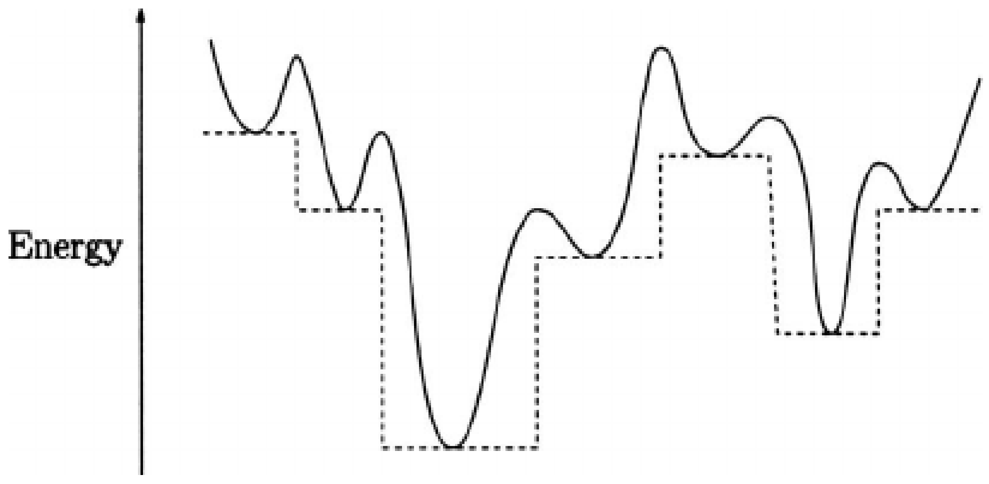

FIGURE 3.3: The effect of the basin transformation on an one-dimensional potential function. The solid and dashed black lines corresponds to the original and transformed potentials, V and $\tilde{V}$, respectively. [7]

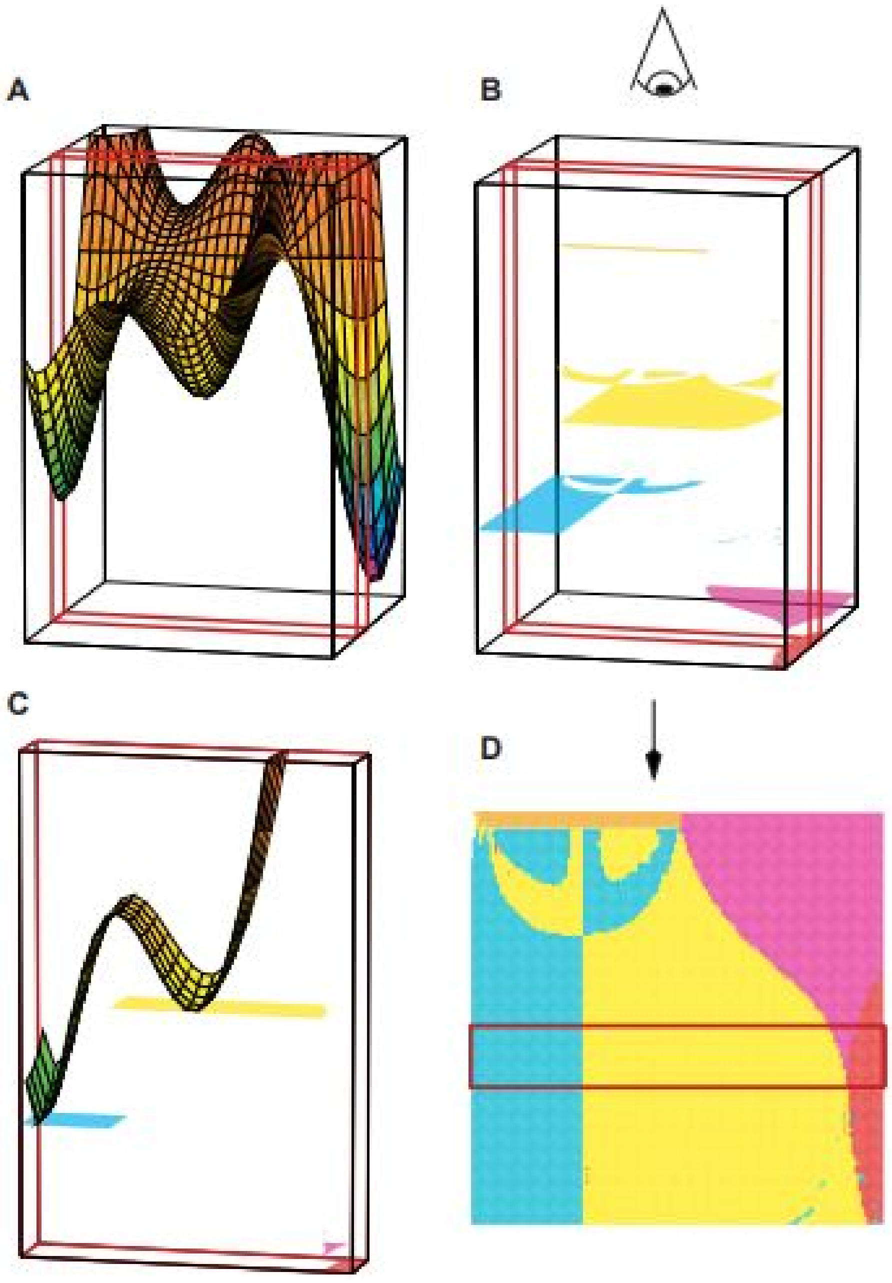

FIGURE 3.4: The effect of the basin transformation on a two-dimensional potential function. (A) Original surface. (B) Transformed surface. Each local minimum of $V(\mathbf{X})$ corresponds to a plateau or catchment basin for $\tilde{V}(\mathbf{X})$. The surfaces are colored consistently according to the energy. (C) Cut through the combined $\tilde{V}(\mathbf{X})$ and $V(\mathbf{X})$ surfaces for the red boxed region shown in all the other panels. (D) View of the transformed surface from above. [11]

3.3.1 Method itself

Since we have also programmed this method in Python (see Appendix C for checking how to implement this method in Python), we can explain the method itself by describing the code we programmed following the instructions of the supervisor.

The beginning of the basin-hopping algorithm [7] is pretty much the same as the one for the simulated annealing algorithm [3]. First, we generate a random distribution of atoms in a tridimensional space. For that, we create a sphere (with radius $r = 5\sigma$) from where the 'atoms' that are chosen are located all in places where the distance to the nearest ones is at least the equilibrium separation ($r_{min} = \sqrt[6]{2}$).

Now, we directly call the routine minimization to quench the random distribution chosen at the beginning obtaining a local minimum from the LJ potential (taking into account that $\epsilon = 1$ and $\sigma = 1$ for simplicity) already.

Again, the number of terms is depending on the number of atoms we have.

Now, we have to define the parameters we need for modifying the positions of the atoms. As before, these parameters are Δ and T with the same meaning as the one explained for the simulated annealing algorithm [3], except for the 'temperature' that now stays constant and, in this case, it does not refer to the real system temperature, it is only a parameter.

This time we set an input to determine the number of iterations we want the program to run instead of a condition.

Inside this loop we are choosing all the atoms to change their position following the exactly same formula as before (see formula 3.1). Here, we are not making the same condition as before for not letting them leave a box since we are using the routine of minimization directly.

Thus, we recalculate the potential using the routine of minimization again and then, we compare this new potential with the one obtained before applying the Metropolis criterion once again.

If the new potential value is lower than the one we had before, the old one is exchanged by this new one immediately. Nevertheless, if this does not hold, then the new position of the atoms and, in consequence, the change in the potential are accepted according to a Boltzmann probability that obeys the next expression:

$$exp((-(V_{new} - V_{old}))/k_B T)$$

where, T is the 'temperature', V is the LJ potential and k_B is the Boltzmann constant. The Boltzmann constant is set to one and, in consequence, the 'temperature' has reduced units.

Since we know the new value for the potential will not be larger than the old one, we know the value of the exponential will be always positive and maximum one. Thus, we are taking a random number from zero to one and if this value is lower than the one obtained for the exponential, we are accepting the new value of the potential and exchanging the position of the atoms. If not, we set the old values again.

Here, the 'temperature' is playing an important role because once we fix it, we can jump from one local minimum to another in an easier way.

However, we set the Δ value to 0.5 but it is modifiable. If after 100 iterations, we are accepting more new values than we are rejecting, we change the Δ value multiplying it by 1.05 in a way that then, the changes are bigger and we can find a better value. If it is in the other way around, we change the Δ value multiplying it by 0.95 in a way that then, the changes are smaller and we can go more straightforward to the global minimum.

Once the loop is accomplished, the distribution we get should be our global minimum, even though we know may not be. For getting to know that, we can compare the output value of our potential to the ones that are in a reliable database [7].

Finally, we are plotting the final solution to get to know if the obtained structure does have physical sense or not.

3.4 Resumen en Español

En este capítulo, describimos los tres métodos principales para la optimización global que habíamos comentado en la introducción. En la primera sección, introducimos de manera bibliográfica el que sería el método más innovador en cuanto que es el más reciente de los aquí estudiados, algoritmo genético ("Algorithm Genetic"). En segundo lugar, tenemos un acercamiento, más en detalle, de uno de los primeros métodos de optimización global, enfriamiento simulado ("Simulated Annealing"), donde programamos el propio método en Python y comprobamos su utilidad con diversos agregados de Lennard-Jones que serán presentados en los capítulos siguientes. Finalmente, el tercer y último método que hemos tenido en cuenta es el de salto entre mínimos ("Basin-Hopping"), un método de optimización global cuyos resultados han sido probados numerosas veces de forma satisfactoria y que nos permitirá comparar los ejemplos mencionados anteriormente.

Chapter 4

Simulation

In this chapter, we are describing the results obtained from the optimization of the LJ clusters to check if our programs are doing well or not. For that, up to 55 clusters from the LJ potential are tested. Actually, the main objective of our work is detailed here because if we are able to code a program based on the methods explained, it means we know the methods far enough down and therefore, we can also get to know the way the results from databases [7] are accomplished. Apart from that, in the second part, we are getting deeper in the explanation of why some clusters are special and we are trying to explain it based on our results and comparing them with the references we have.

4.1 General Analysis

Here, we are seeking for the global minimum of the first 55 LJ clusters and we are also including a comparison of the methods and their features to get to know the advantages from one to the other.

In the previous chapter, meanwhile we were explaining the method itself, we were introducing some parts of the code for supporting the explanation. Now, we are describing the parts that affect the efficiency in greater detail and the rest of it can be checked in the appendixes B and C.

The choice of the routine of minimization is directly related to the efficiency of the program, but not with its succeed finding the global minimum since this is depending on how complex the potential energy surface (PES) is. However, some other factors such as the initial temperature or the initial value of delta are also important. Of course, the size of our program and its simplicity are some very significant factors for the run-time as well. We needed to change several times the way we wanted to calculate the new positions and the new distances, coming up at the end with a solution consisting in recalculating only the ones that are affected by the change. Nevertheless, changing this in our algorithm took some thinking caused by the way the loops in Python work.

We have noticed that depending on the problem we are facing, we need a different kind of routine of minimization. Not only for that, but also for selecting the best values for the initial temperature and the delta, we have chosen the clusters LJ_{13} and LJ_{38} as trials. This choice was made in order to detect how the time per iteration in the algorithms behaves with a big enough but very stable cluster (LJ_{13}) and a very difficult one (LJ_{38}). Thanks to them, we realized which parameters are the ones that make our algorithm as fast as possible.

Before that, there was a very relevant part from our program that had to be done. The definition of the potential and its partial derivatives from the Cartesian coordinates we are using (X, Y, Z) for every pair interaction needed to be introduced in our algorithm in a way it can work with them. Although these partial derivatives are explicitly asked by the routine of minimization we are using, there were some that did not need them, but then they were not that accurate finding the local minimum.

The used routines of minimization were some of the ones given by the library of Python, 'scipy'. This library has a package called 'optimize' where we can find them. The chosen ones are: 'CG' (Conjugate Gradient), 'BFGS' (Broyden-Fletcher-Goldfarb-Shanno), 'Newton-CG', 'TNC' (Truncated Newton) and 'L-BFGS-B' (Limited 'BFGS' Bound) [13].

In every different routine, we set a configuration with different initial temperatures (0.5, 0.8, 1, 3, 5, 10) and delta (0.4, 0.5, 0.6). Nevertheless, the difference produced by the change of these initial values was almost unappreciated except for the temperature in the basin-hopping [7] (it is fixed) where 0.8 was the best. In consequence, for the simulated annealing [3] we chose the ones recommended by the professor, 1 for the temperature and 0.5 for the delta value and for the basin-hopping [7], we only changed the temperature to 0.8.

With respect to the different routines, it was deduced from the trials taken that the best ones for our purposes were 'BFGS' for the simulated annealing algorithm [3] and 'L-BFGS-B' for the basin-hopping algorithm [7].

This difference in the efficiency of the chosen routines of minimization made us wondering why this is happening. Having a look to some references, we figured out that the reason why it is like this is the way how these routines work.

On one hand, the 'BFGS' works calculating some matrices at each step. This requires a lot of space from the disk of our computer, although the final result will be more accurate even though our initial distribution is not good at all. [14]

On the other hand, 'L-BFGS-B' is an approximation to 'BFGS', which requires a lot less memory. 'L-BFGS-B' computes and stores an approximation to the matrices mentioned before, what means we need less disk space. [15]

At the end, each step of 'L-BFGS-B' is an attempt at approximating what the corresponding step of 'BFGS' would do. However, a single step of 'L-BFGS-B' takes a lot less space and time than a single step of 'BFGS'. Consequently, we can do many more steps of 'L-BFGS-B' within a particular time bound than 'BFGS'. Therefore, we might find that 'L-BFGS-B' converges faster, because it can do so many more iterations within a given amount of time than 'BFGS' can. Nevertheless, if the initial distribution of the solution for our function is not good enough, this approximation done by the 'L-BFGS-B' routine is actually making it harder to find the final solution.

That is why the best routine of minimization for the simulated annealing algorithm [3] is 'BFGS', where we do not have a good enough initial distribution for calculating the global minimum as we need for 'L-BFGS-B', when it is supposed to be faster.

Hence, for the basin-hopping algorithm [7], it turns out that since we are using the routine of minimization directly from the beginning, the initial distributions running afterwards are so much faster to calculate with 'L-BFGS-B'.

Once we have decided all the parameters we are taking and the algorithms are ready for running, we are testing them with the LJ clusters.

4.1.1 Examples

First, we are testing the simulated annealing algorithm [3]. For that, we are running the program one by one selecting the number of atoms we want to introduce in our cluster. Then, with five trajectories we are calculating the average of iterations the program is doing before the routine of minimization is called, what means before the condition we set (the difference between two energies calculated in a row has to be less than 10^{-3}) is reached. After that, we include these iterations in our work, as we can see in the figure 4.1. There, we can also check that the maximum number of iterations taken for one trajectory is 76 for the case of the LJ_{26}. The problem, as we can observe, is that the number of trajectories chosen is not enough for getting the global minimum for big clusters (from LJ_{26} on was not possible with only five trajectories in any case) and for some not that big ones either, such as LJ_{18}, LJ_{21}, LJ_{24} and LJ_{25}. However, we keep this in order to be able to compare this method with the basin-hopping [7] one, where we are setting the same number of trajectories. The number of trajectories that reach the global minimum per cluster is showed in the table 4.1 (only until LJ_{26}).

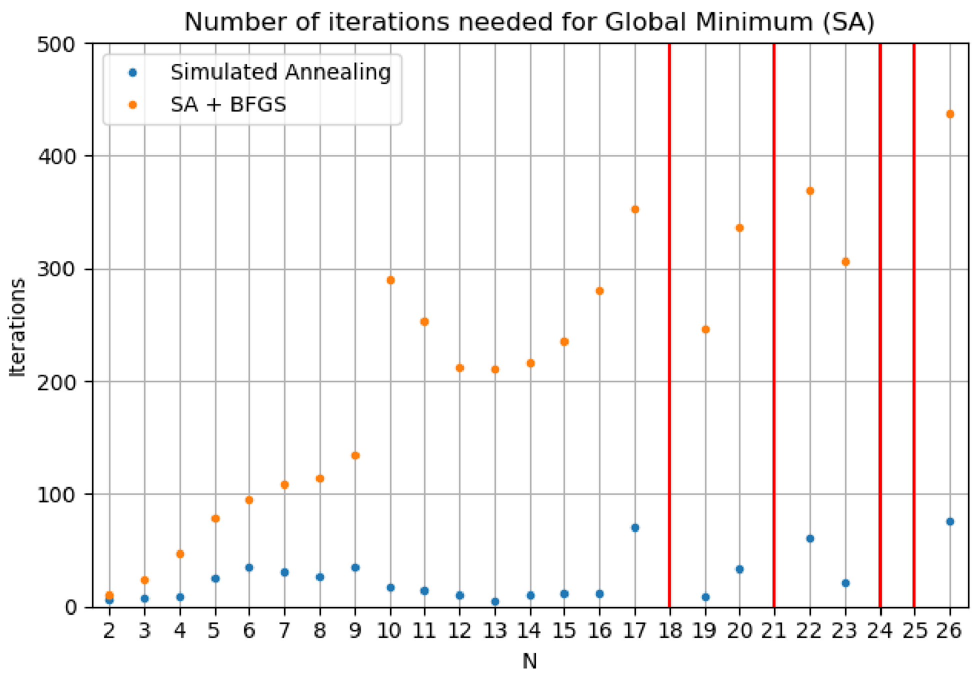

FIGURE 4.1: Number of iterations to obtain the global minima as a function of number of atoms. The red lines determine the global minima not obtained.

N	Succeed	N	Succeed	N	Succeed	N	Succeed	N	Succeed
2	5	7	1	12	2	17	1	22	1
3	5	8	5	13	4	18	0	23	1
4	5	9	3	14	1	19	2	24	0
5	5	10	1	15	2	20	1	25	0
6	1	11	2	16	1	21	0	26	1

TABLE 4.1: Number of trajectories (out of 5) that succeed finding the global minimum.

Actually, in these clusters where the global minimum was not found is observed that the tendency is to have a peak of iterations that maybe is making the algorithm unable to obtain the global minimum. In between the stable clusters found, the relative stability, which gives us the number of iterations, first decreases before increasing again as the next icosahedron is approached. [16] The main reason why this is happening is the structure how the clusters are packaged (how complex the PES is). The red lines showed in the figure 4.1 are directly those clusters whose relative stability decrease a lot because they have too many overlayers on the smaller Mackay icosahedron (it is a special kind of icosahedral packing that exposes only 111 close-packed type faces [9]). With 13 'atoms' we can see perfectly how the algorithm is able to get it really fast because it is a complete Mackay icosahedra [16] and then, the iterations are increasing a lot until the next stable structure is reached with 19 'atoms' because it is a double icosahedron, which can be considered as a 13-atom icosahedron with a particularly favourable overlayer. [16]

Otherwise, the number of iterations used for computing the global minimum are not comparable with the one from basin-hopping [7] since we are rating different steps. Here, we are setting the number of iterations before the routine is called and then, the number of iterations the routine needs to obtain the global minimum. However, in the basin-hopping [7] we are counting the number of times we need to call the routine of minimization for an initial distribution of atoms. Furthermore, the number of taken iterations here is determined by the own routine of minimization instead of being chosen by us as it happens in the basin-hopping algorithm [7].

About how temperature and delta are changing their values, we have to remark that, in the most of the cases the final temperature (set in 0.05) was never approached, but the delta value was all the time oscillating around 0.5.

Now, it is time for testing the basin-hopping algorithm [7]. For that, we are running the program one by one selecting the number of atoms we want to introduce in our cluster. Then, with five trajectories we are calculating the average of iterations the program is taking for approaching the global minimum that, at the end, it is the number of times we are calling the routine of minimization for calculating the local minimum. The iterations needed are plotted in the figures 4.2 and 4.3, where we have distinguished between small clusters and big clusters because of the difference in the number of iterations.

In this case, it is pretty clear that we are obtaining all the global minima without any exception. There are some whose number of iterations is really high, but still possible and reasonable. Moreover, in all the trajectories, the global minimum was found. Since we were using the LJ_{38} cluster as a trial for our algorithm, we noticed we needed almost 10000 iterations for getting it. Hence, for the rest of the clusters we use 10000 iterations per trajectory as a safe number in a way that, for sure, we are getting the global minimum since they are easier to obtain.

The LJ_{13} cluster is again showing the same behaviour as in the simulated annealing algorithm [3], but now we can also see the same in the LJ_{55} cluster for the same reason, both are complete Mackay icosahedra [16]. This can be checked in the formula where the total number of atoms in a complete Mackay icosahedron is given:

$\frac{1}{3}\,(10n^3 + 15n^2 + 11n + 3), \quad n = 1, 2, 3, \ldots,$

which produces the sequence 13, 55, 147, 309, etc. [9]

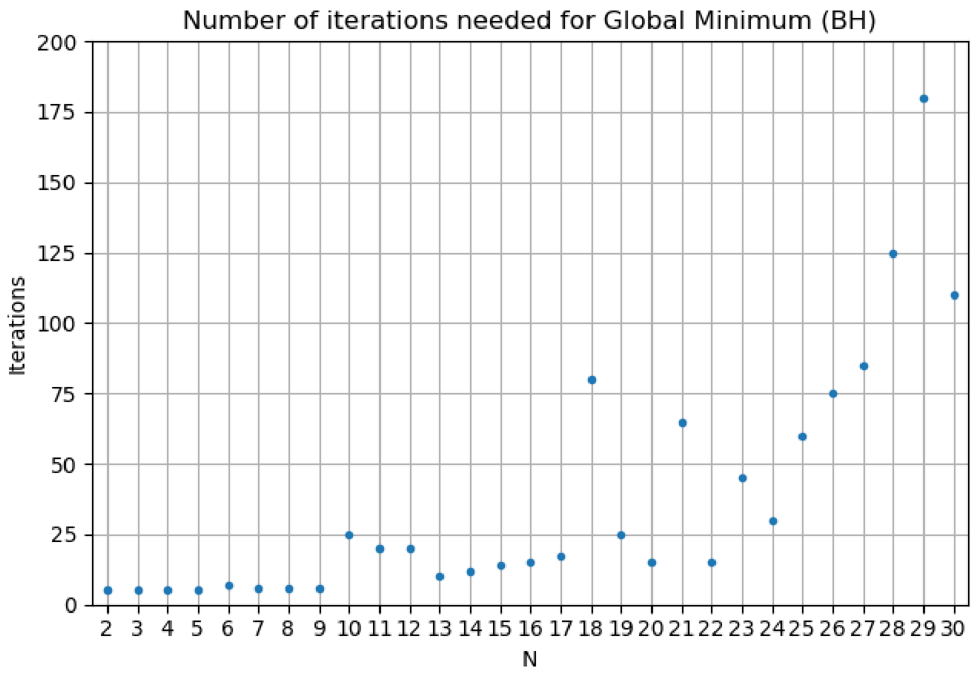

FIGURE 4.2: Number of iterations to obtain the global minima as a function of number of atoms.

As we can see in the figures 4.2 and 4.3, in between the stable clusters found from 2 to 55 'atoms', the relative stability, which gives us the number of iterations, first decreases before increasing again as the next icosahedron is approached again, as it happened wit SA. [16]

Almost all the remaining clusters are incomplete Mackay icosahedra where the stability of their global minimum or the iterations needed for achieving it is depending on the amount of overlayers from smaller Mackay icosahedra they are formed with [16]. The only exception in this group is the one that actually was more difficult to obtain, the LJ_{38}.

Otherwise, even though we were doing some analysis about the structure and consequently, about the potential energy surface, we have to admit that these numbers are qualitative since we have not accomplished any statistical work with a huge number of trajectories.

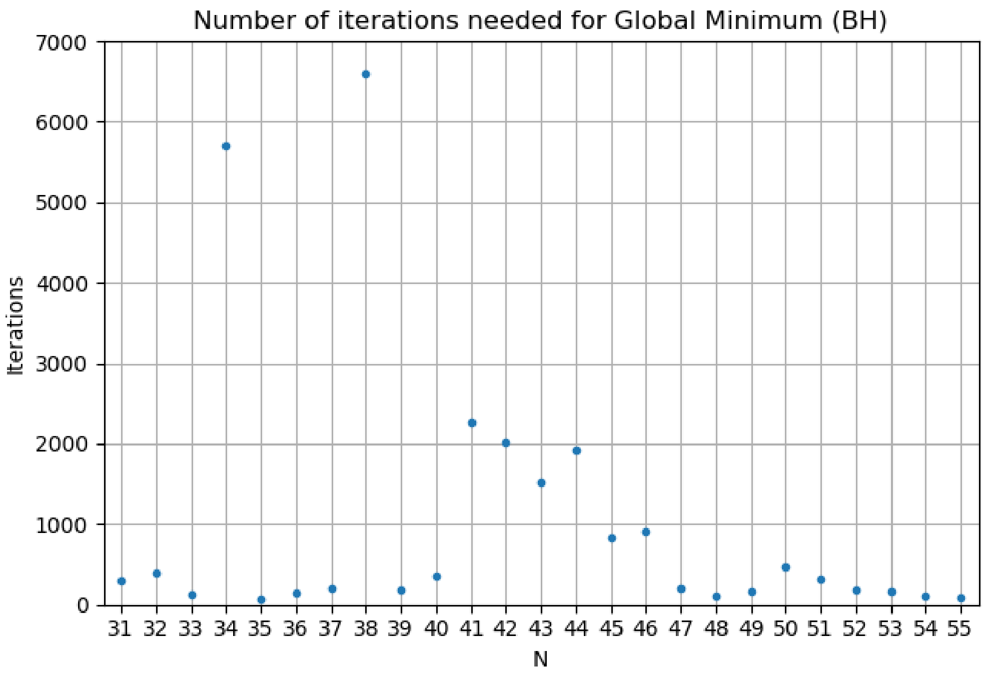

FIGURE 4.3: Number of iterations to obtain the global minima as a function of number of atoms.

4.2 Special Cases

As special cases, we are taking the LJ_{13} and the LJ_{38} clusters. The first one is chosen based on its special stability compared to the other sizes that is giving it the name of 'magic number' (also LJ_{55} is called like this) caused by its structure of a complete Mackay icosahedra. The other one is selected because, as we mentioned before, almost all the clusters have an incomplete Mackay icosahedra structure and this one is the only one from the clusters we are studying that does not have it. The global minimum at N = 38 is an fcc (face-centred-cubic) truncated octahedron. [16]

The distribution that is obtained, once our algorithms (in this case, both) find the global minimum, is plotted in the figure 4.4. Since there are not that many atoms, it is easy to see that the structure obtained is a complete Mackay icosahedra. It is also shown the one obtained from database in the figure 4.5 confirming our programs are doing well and calculating the correct set of positions for this cluster.

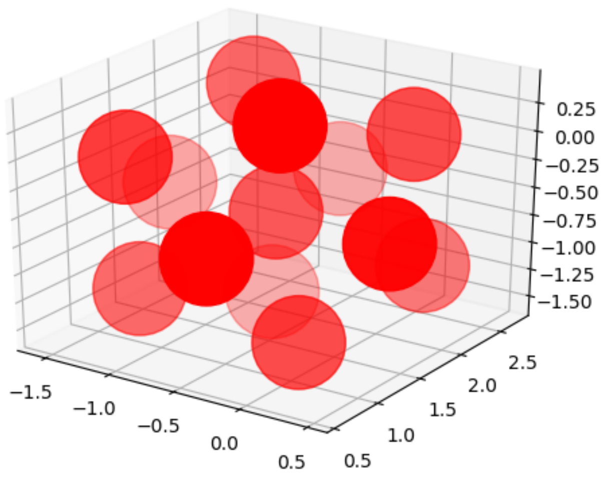

FIGURE 4.4: Global Minimum for LJ_{13}. A complete Mackay icosahedron plotted with Matplotlib [17].

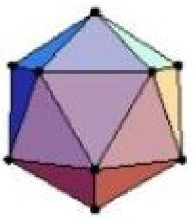

FIGURE 4.5: Cluster obtained by database for N = 13. [16]

Moreover, for the LJ_{38} the second-lowest minimum competes directly with the global one based upon icosahedral packing. Both are showed in the figure 4.6. Thus, the global minimum is relatively difficult to locate because it lies at the bottom of a narrow side potential energy funnel [9]. Both distributions obtained in our algorithm (in this case, only in the basin-hooping [7] one) are plotted in the figures 4.7 and 4.8. The geometry of both clusters is drawn to effectively check that it coincides with the ones in the figure 4.6.

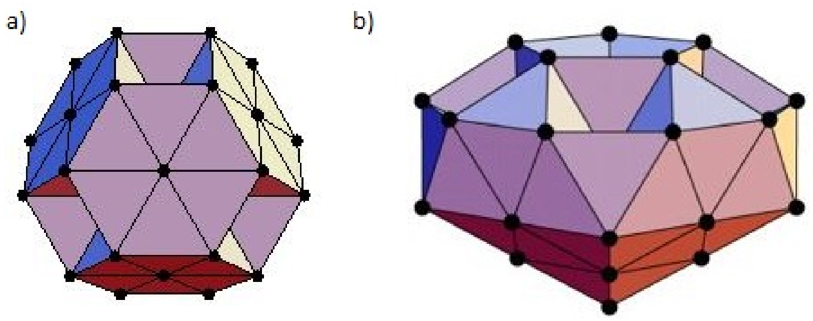

FIGURE 4.6: Clusters obtained by database for N = 38. a) Global minimum (Octahedron) and b) Second-lowest minimum (icosahedron packed) ([16]

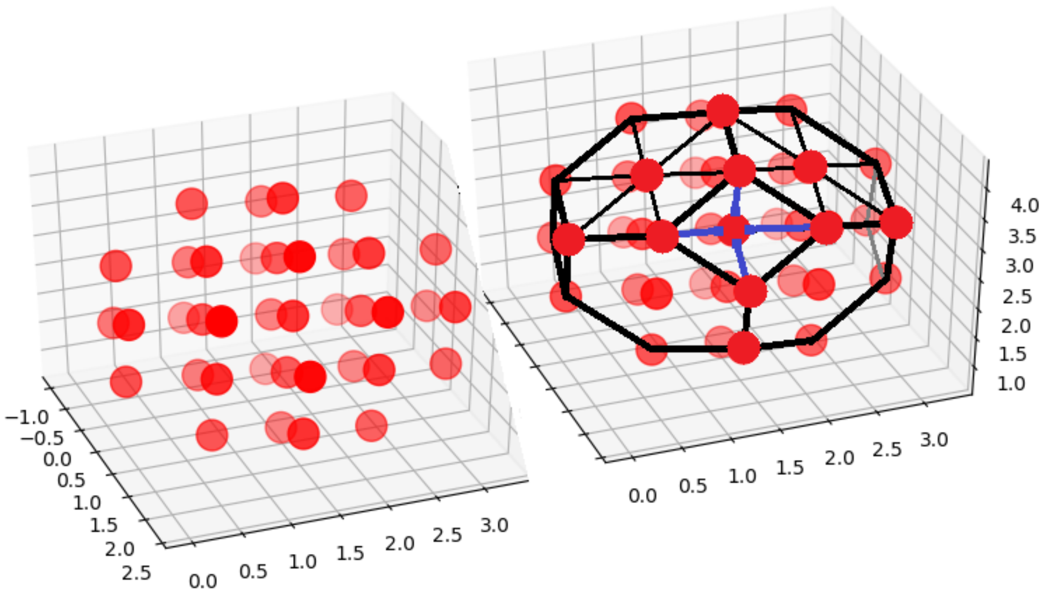

FIGURE 4.7: Global Minimum for LJ_{38} based upon nonicosahedral packing, it is a Octahedron, plotted with Matplotlib [17].

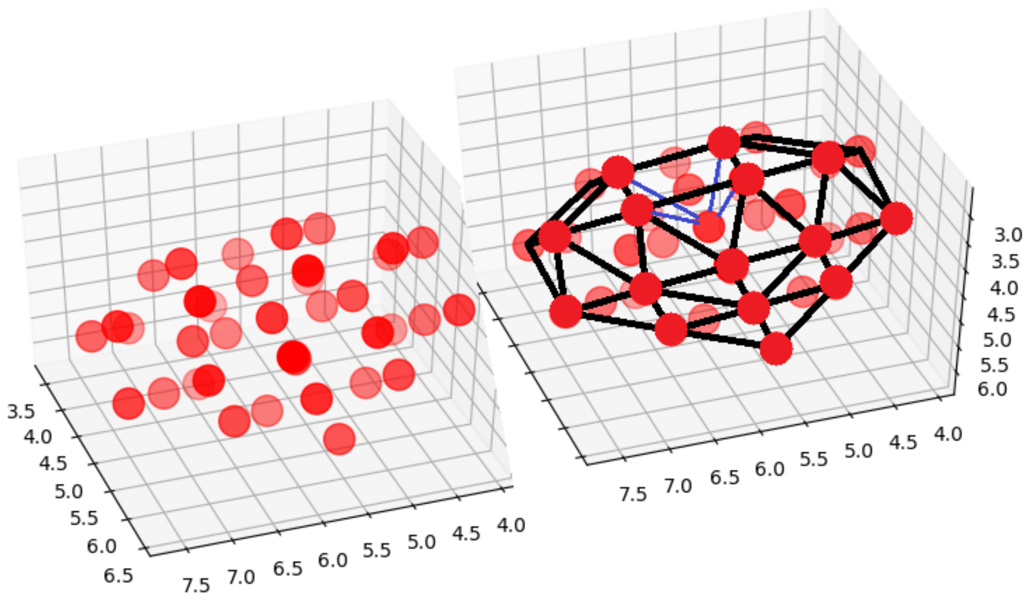

FIGURE 4.8: Second-lowest minimum for LJ_{38} based upon icosahedral packing plotted with Matplotlib [17].

4.3 Resumen en Español

En este capítulo, describimos los resultados obtenidos de la optimización realizada con los agregados de Lennard-Jones con nuestros programas para comprobar si los hemos realizado de manera correcta. Para ello, la idea es analizar los agregados que contienen hasta 55 átomos. De hecho, el principal objetivo de nuestro trabajo de fin de grado es detallado en este capítulo puesto que con estos programas se ha comprobado que hemos sido capaces de verificar los modelos teóricos con solvencia a través del código ideado, hallando los resultados ya conocidos. Esto significa que los hemos entendido lo suficiente para llegar a comprender cómo han sido calculados los mínimos globales de la base de datos de Cambridge. Aparte de esto, en una segunda sección, indagaremos en la explicación de por qué algunos de los agregados estudiados son especiales, teniendo en cuenta nuestros resultados y comparándolos con los que se obtienen en otros estudios.

Chapter 5

Conclusions

First of all, we have to admit that the use of the Lennard-Jones potential is making easier to check the efficiency of our methods. However, as an objective for future reports we should try to figure out a new potential that is able to describe the system we study in a better way or maybe, we can still use the LJ potential but adding other terms that let us be more accurate. For example, the addition of harmonic spring terms between adjacent atoms in the sequence can be useful for describing complex polymer [12]:

$$V_{ij}(r) = \sum_{j<i}^{N} 4\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] + \frac{k}{2} \sum_{i}^{N-1} (r_{i,i+1} - r_0)^2 \tag{5.1}$$

Also, a Lennard-Jones potential including zero-point energies can vary these results since its contribution to the potential energy of Van der Waals atomic clusters may be large. [9]

Apart from that, we were wondering why we did not find the global minimum on big clusters with the simulated annealing algorithm [3]. We could have run the program with more trajectories to check if the global minimum can be found at least, but the time-consuming was too high for making that work worthy. Nevertheless, we figured out that this behaviour is normal since even in some reference some researchers comment that this method is basically for small and intermediate clusters sizes. [3]

Therefore, either we still have to optimize the algorithm, which is a possible future direction, for example focusing on alternative cooling schedules, or we need to find other kind of condition for stopping the program, forcing it to be more accurate before entering the routine of minimization.

About the basin-hopping method [7], an important fact that affects the algorithm is that if we set an initial temperature that is wrong, we can maybe miss the global minimum. Then, maybe a possible new way to enhance this method can be modifying it after a certain amount of steps is reached as we do with the δ value.

In general, we have implemented two methods for performing structural optimization of a variety amount of atoms. We used Python as our programming language and it turned out to be such an useful tool for this purpose. Since we do not have to recompile the code every time it is modified, it is very fast and easy to change when it is necessary.

It is also remarkable to add that the time or iterations needed by our basin hopping [7] was approximately the same as for the one found in the Python library [13].

Another way to optimize the algorithms could be trying to run some parts of them in parallel. However, for that, we would have needed to compute our programs in a different language, such as Fortran or C, since for Python, this is not possible.

At the end, the main purpose of this work have been reached with both methods, understanding how they were built and the infinite possibilities they have for future works. Even for the genetic algorithm [5], we have got to know a lot about it and we can use that knowledge as a basis for future works as well.

For example, the simulated annealing method [3] can be used for different applications such as providing a model that generates valid Sudoku boards or a model that solves 'nonograms' (picture logic puzzles). [18]

Furthermore, the genetic algorithm (GA) method [5] cannot be used only for simple or theoretical cluster structures, but also for more complex and specific ones. From model Morse clusters to fullerenes, ionic clusters, water cluster, metal clusters and bimetallic 'nanoalloy' clusters, GA has proved its efficiency. In other fields such as chemistry, notable applications have been also found like, for example, "the prediction of protein secondary and tertiary structure, simulation of protein folding and structural studies of RNA and DNA, the design and docking of drug molecules, quantitative structure–activity relationships (QSAR), pharmacophore mapping and receptor modelling and combinatorial library design; the prediction of crystal structures and the solution of crystal structures from single crystal, powder and thin film diffraction data; the determination of molecular (including biomolecular) structure from NMR spectroscopy; and the control and optimisation of chemical processes". [5]

About the basin-hopping method [7] that we have computed, the main reason why there are not thousands of references about its applications is because it is a technique that was developed for a very precise objective, to exploit the features that must be present in an energy landscape for efficient relaxation to the global minimum [7]. Nevertheless, for its purpose is one of the best because it can even be developed to get new methods more and more efficient, such as minima hopping that it is a non-thermodynamic and reduce the rate of return to already visited minima. [19]

Nowadays, the frame of this research and its implementation in other fields is a really hot topic because it turned out that can be a very useful tool to get to know how to focus some others problems.

Moreover, there are more global optimization algorithms that need to be tested and maybe some that we do not even know since most papers focus on just one method. There are too many different possible approaches in even one single method and some features are common to a plenty of them, but all the methods need always to be improved to get more and more reliable results.

5.1 Resumen en Español

En las conclusiones, detallamos la manera en la que el potencial utilizado puede ser mejorado en vista a incrementar la posibilidad de encontrar estructuras de uso más realista. También, incluimos la explicación del porqué de la necesidad de utilizar diferentes rutinas de minimización en función del método que estemos empleando.

Por otro lado, intentamos comprender la superioridad del método del salto entre mínimos ("Basin-Hopping") respecto al enfriamiento simulado ("Simulated Annealing"), remarcando que, al fin y al cabo, el objetivo planteado como principal para este trabajo de fin de grado ha sido alcanzado, que no ha sido otro que la satisfactoria consecución de ambos métodos entendiendo su funcionamiento y dando a conocer su metodología. Incluso para el método de algoritmo genético ("Genetic Algorithm") se han sentado las bases para un posible estudio en el futuro entrando en más detalles e incluso llegando a programar el mismo.

Así, se comentan las posibilidades de los métodos aquí estudiados y se indica que más investigación es necesaria para poder tratar problemas cada vez más complejos de optimizar.

Appendix A

Derivation of Lennard-Jones Pair Potential

In this Appendix we are explaining the process for the derivation of the LJ pair potential, getting the minimum value of the potential from the value of the distance needed.

First, assume our potential as follows:

$$V_{LJ}(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{A.1}$$

The expression of the derivative of our potential is easy to get following the principles rules of derivation:

$$\frac{dV}{dr} = 4\epsilon \left[-12\left(\frac{\sigma^{12}}{r^{13}}\right) + 6\left(\frac{\sigma^{6}}{r^{7}}\right) \right] \tag{A.2}$$

$$\frac{dV}{dr} = 24\epsilon \left[\left(\frac{\sigma^{6}}{r^{7}}\right) - 2\left(\frac{\sigma^{12}}{r^{13}}\right) \right] \tag{A.3}$$

Once we have rearranged the terms, we equal to zero the potential to get the condition for the minimum:

$$\frac{dV}{dr} = 0 \Rightarrow 24\epsilon \left[\left(\frac{\sigma^{6}}{r^{7}}\right) - 2\left(\frac{\sigma^{12}}{r^{13}}\right) \right] = 0 \tag{A.4}$$

$$\left(\frac{\sigma^{6}}{r^{7}}\right) - 2\left(\frac{\sigma^{12}}{r^{13}}\right) = 0 \tag{A.5}$$

$$\frac{\sigma^{6}}{r^{7}} = 2\frac{\sigma^{12}}{r^{13}} \Rightarrow r^{6} = 2\sigma^{6} \tag{A.6}$$

After all this process, we obtain the minimum distance for the minimum of our potential:

$$r_{min} = \sqrt[6]{2}\sigma \tag{A.7}$$

Now, we sustitute this value into the potential to get to know which value of the potential is the minimum:

$$V_{min} = 4\epsilon \left[\left(\frac{\sigma}{\sqrt[6]{2}\sigma} \right)^{12} - \left(\frac{\sigma}{\sqrt[6]{2}\sigma} \right)^{6} \right] \tag{A.8}$$

$$V_{min} = 4\epsilon \left(\frac{1}{4} - \frac{1}{2} \right) \tag{A.9}$$

Finally, we are getting:

$$V_{min} = -\epsilon \tag{A.10}$$

Appendix B

Simulated Annealing Code

```
from numpy import *
from pylab import *
import random as rd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.optimize import *
#We ask for the number of atoms you want to organize,
N = int(input('Introduce the number of atoms you want:'))
# the iterations needed
ite = int(input('Introduce the number of parallel iterations you want:'))
# and the size of the box the atoms are located in
lim = int(input('Introduce the size of the box you want:'))
#Creating a sphere of radius 5*sigma and putting some spots around randomly in a way that
# they represent atoms in a sphere
r = 5.
x = array([])
y = array([])
z = array([])
a_range = arange(0.,2*pi,0.3)
b_range = arange(0.,pi,0.3)
for theta in a_range:
    for phe in b_range:
        x = append(x, 5 + r*cos(theta)*sin(phe))
        y = append(y, 5 + r*sin(theta)*sin(phe))
        z = append(z, 5 + r*cos(phe))
for i_ite in range(ite):
#We select the points now. For that we have chosen a bidimensional array, we set ones to not
# have problems. We choose them randomly with the condition of not being closer than (2^(1/6))
    position = ones((3,N))
    position_i = ones((3,N))
#To determine the len of the array we are taking the atoms from
    s = len(x)
#This number is for getting to know the number of colums we need later
    suma = 0
    for i in range(N):
        a = rd.randrange(0,s) #the random number for choosing the 'atom'
#Each row is an axe
        position[0,i] = x[a]
        position[1,i] = y[a]
        position[2,i] = z[a]
        suma = suma + i
#In this step, we are calculating the distances in all three axes we need between all atoms
#We set ones again to avoid problems. We set 5 row including 3 for the
```

```
# three dimensions and 2 for the labels and the colums come from the number of atoms we chose
    r_distancexyz = ones((5,suma))
#We set this value for taking the distance we want in the loop
    it = 0
    longitu = len(position[0,:])
    j = -1
#We go through all the atoms we have on-by-one to check the distances between them all
    while j < N:
        if j == -1:
            it = 0
        else:
            pass
        j = j + 1
#Once we are in one atom, we go through the next ones to calculate the distances between them
#We are doing that only for the atoms whose distance has not been calculated
        for k in arange(j + 1, longitu):
            r_distancexyz[0,it] = abs(position[0,j]-position[0,k])
            r_distancexyz[1,it] = abs(position[1,j]-position[1,k])
            r_distancexyz[2,it] = abs(position[2,j]-position[2,k])
            r_distancexyz[3,it] = int(j)
            r_distancexyz[4,it] = int(k)
#We are checking the distance in 3D between atoms to be less than sixth root
# of two to avoid the repulsion forces
            test_distance = sqrt(r_distancexyz[0,it]**2+r_distancexyz[1,it]**2+
                                 r_distancexyz[2,it]**2)
#Just in the case we want to calculate the equilibrium distance (2 atoms)
            if N == 2:
                pass
            elif test_distance < (2**(1/6)):
#Since this happens, we are taking a new configuration
                a = rd.randrange(0,s)
                position[0,k] = x[a]
                position[1,k] = y[a]
                position[2,k] = z[a]
#We make the loop running again, setting this value to -1
                j = -1
            else:
                pass
            it = it + 1
#Once all the atoms satisfy the condition, we create the array for the distances
    r_distance = array([])
#Now, we take all the distances in the three axes and calculate the distance in 3D
    for p in arange(len(r_distancexyz[0,:])):
        r_distance = append(r_distance, sqrt(((r_distancexyz[0,p])**2) +
                                             ((r_distancexyz[1,p])**2) +
                                             ((r_distancexyz[2,p])**2)))
#Define the potential with these r_distance
    V_f = 0.
    for d in arange(len(r_distance)):
        V_i = 4*((r_distance[d]**(-12)) - (r_distance[d]**(-6)))
        V_f = V_f + V_i
#We select the initial temperature a parameter for Boltzmann statistics
    T = 1
    T_i = T
```

```
#We jail the atoms in a box of limxlimxlim. We create a virtual copy to modify.
#We select a default position for iterations and the distances in three axes
# with labels included
    for p_j_j in arange(len(position_i[0,:])):
                    position_i[0,p_j_j] = position[0,p_j_j]
                    position_i[1,p_j_j] = position[1,p_j_j]
                    position_i[2,p_j_j] = position[2,p_j_j]
    r_distancexyz_i = r_distancexyz
#We define delta as the value for changing positions, we are also planning
# to modify it depending on the numbers of changing we are accepting
    delta = 0.5
#We define a value to check if we are accepting or not new values for the
# potential and then, changing the delta
    accept = 0
    reject = 0
#For changing the T and delta
    iteracc = 0
    factor = 1
    factor2 = 1
#For comparing the energies we are accepting to stop the loop
    V_f_i = 0.
    V_f_ant = 1.
#Running the loop with the condition we want. In this case, the difference between
# energies before and after we change it is supposed to be less than 0.001.
    while abs(V_f_ant-V_f_i) > 0.001:
        iteracc = iteracc + 1
#We select an atom randomly to change its position
        a_xyz = rd.randrange(0,len(position_i[0,:]))
#We change the position of this atom
#Choosing a random value for this purpose in the three axes
        scribble1 = rd.random()
        scribble2 = rd.random()
        scribble3 = rd.random()
#Conditions for jailing, keeping the atoms inside of the box
        if ((position_i[0,a_xyz]) + 2 * delta * (scribble1 - 0.5)) > lim:
            position_i[0,a_xyz] =  ((position_i[0,a_xyz]) - 2 * delta *
                      (scribble1 - 0.5))
        elif ((position_i[0,a_xyz]) + 2 * delta * (scribble1 - 0.5)) < 0:
            position_i[0,a_xyz] =  ((position_i[0,a_xyz]) - 2 * delta *
                      (scribble1 - 0.5))
        else:
            position_i[0,a_xyz] = ((position_i[0,a_xyz]) + 2 * delta *
                      (scribble1 - 0.5))
        if ((position_i[1,a_xyz]) + 2 * delta * (scribble2 - 0.5)) > lim:
            position_i[1,a_xyz] = ((position_i[1,a_xyz]) - 2 * delta *
                      (scribble2 - 0.5))
        elif ((position_i[1,a_xyz]) + 2 * delta * (scribble2 - 0.5)) < 0:
            position_i[1,a_xyz] = ((position_i[1,a_xyz]) - 2 * delta *
                      (scribble2 - 0.5))
        else:
            position_i[1,a_xyz] = ((position_i[1,a_xyz]) + 2 * delta *
                      (scribble2 - 0.5))
        if ((position_i[2,a_xyz]) + 2 * delta * (scribble3 - 0.5)) > lim:
            position_i[2,a_xyz] = ((position_i[2,a_xyz]) - 2 * delta *
                      (scribble3 - 0.5))
        elif ((position_i[2,a_xyz]) + 2 * delta * (scribble3 - 0.5)) < 0:
            position_i[2,a_xyz] = ((position_i[2,a_xyz]) - 2 * delta *
                      (scribble3 - 0.5))
        else:
            position_i[2,a_xyz] = ((position_i[2,a_xyz]) + 2 * delta *
                      (scribble3 - 0.5))
#Now, we are using the labels we included before. Thanks to them we know the distances
# that have to be modified and in consequence we are taking them to recalculate them
        for m in arange(len(r_distancexyz_i[4,:])):
            if r_distancexyz_i[3,m] == a_xyz:
                r_distancexyz_i[0,m] = abs(position_i[0,a_xyz]-
                               position_i[0,int(r_distancexyz[4,m])])
                r_distancexyz_i[1,m] = abs(position_i[1,a_xyz]-
                               position_i[1,int(r_distancexyz[4,m])])
                r_distancexyz_i[2,m] = abs(position_i[2,a_xyz]-
                               position_i[2,int(r_distancexyz[4,m])])
            elif r_distancexyz_i[4,m] == a_xyz:
                r_distancexyz_i[0,m] = abs(position_i[0,a_xyz]-
                               position_i[0,int(r_distancexyz[3,m])])
                r_distancexyz_i[1,m] = abs(position_i[1,a_xyz]-
                               position_i[1,int(r_distancexyz[3,m])])
                r_distancexyz_i[2,m] = abs(position_i[2,a_xyz]-
                               position_i[2,int(r_distancexyz[3,m])])
            else:
                pass
#We put them to one dimensional array
        r_distance_i = array([])
        for p_i in arange(len(r_distancexyz_i[0,:])):
            r_distance_i = append(r_distance_i,
                                  sqrt(((r_distancexyz_i[0,p_i])**2) +
                                       ((r_distancexyz_i[1,p_i])**2) +
                                       ((r_distancexyz_i[2,p_i])**2)))
#We set a default value for energy to calculate the new one
        V_f_i = 0.
        for d_i in arange(len(r_distance_i)):
            V_i_i = 4*((r_distance_i[d_i]**(-12))-(r_distance_i[d_i]**(-6)))
            V_f_i = V_f_i + V_i_i
#Let's compare the energy obtained and see if we change it or not
#If the potential is lower, then yes both energy and position are replaced
        if V_f_i < V_f:
            V_f_ant = V_f
            V_f = V_f_i
            for p_i_j in arange(len(position_i[0,:])):
                    position[0,p_i_j] = position_i[0,p_i_j]
                    position[1,p_i_j] = position_i[1,p_i_j]
                    position[2,p_i_j] = position_i[2,p_i_j]
            accept = accept + 1
#If not, we are using Boltzmann statistic for choosing the new one or not
        else:
            a_i = rd.random()
            if a_i < exp((-(V_f_i - V_f)) / T):
                V_f_ant = V_f
                V_f = V_f_i
                for p_i in arange(len(position_i[0,:])):
                    position[0,p_i] = position_i[0,p_i]
                    position[1,p_i] = position_i[1,p_i]
                    position[2,p_i] = position_i[2,p_i]
                accept = accept + 1
#If we are not choosing the new energy, then the positions are the ones we had before again
            else:
                V_f = V_f
                for p_i_i in arange(len(position_i[0,:])):
                    position_i[0,p_i_i] = position[0,p_i_i]
                    position_i[1,p_i_i] = position[1,p_i_i]
                    position_i[2,p_i_i] = position[2,p_i_i]
                reject = reject + 1
#We are changing the temperature following an exponential function
# every 500 steps until it is 0.05
            ex = 0
        if iteracc == (50*factor):
            ex = ex + 0.1
            T = exp(log(T)-ex)
            if T < 0.05:
                T = 0.05
            else:
                T = T
            factor = factor + 1
        else:
            T = T
#We are checking the acceptance of the potential every 1000 steps
        if iteracc == (100*factor2):
            if accept < reject:
                delta = delta * 0.95
            else:
                delta = delta * 1.05
            factor2 = factor2 + 1
        else:
            delta = delta
#Once the loop is ended, we are using a minimization routine for getting the minima
#Because of the way to work of this routine, we cannot use 2D arrays anymore
# and that is why we are appending them all to a single row
    position_i_serie = array([])
    position_i_serie = append(position_i_serie, position_i[0,:])
    position_i_serie = append(position_i_serie, position_i[1,:])
    position_i_serie = append(position_i_serie, position_i[2,:])
#For letting the routine which function we want to minimize, we have to define it first.
#Furthermore, in this routine is possible to define both the function and the derivative (jac)
# at the same time. However, one has to define them with the variables one wants to obtain
#On this case, they are the position_i_serie. For the function, we need the final value after
# inserting the positions. Nevertheless, the jac has to return an array with the partial
# derivatives of every single variable we are taking.
    def funcionDderiv(position_i_serie):
        V_f = 0.
        F_f_x = array([])
        F_f_y = array([])
        F_f_z = array([])
        longi = len(position_i[0,:])
#It is the same loop that one defined before so far, but including the partial derivatives
# and since it needs to be calculated between one atom with the rest of them, we need to
# include a loop for the ones we were avoiding before to not calculate the same twice.
        for j3 in arange(longi):
            F_i_i_x = 0.
            F_i_i_y = 0.
            F_i_i_z = 0.
            for k3 in arange(j3 + 1, longi):
                V_i=4*((((abs(position_i_serie[j3]-position_i_serie[k3]))**2)
                +((abs(position_i_serie[j3+longi]-position_i_serie[k3+longi]))**2)+
                ((abs(position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi]))**2))**(-6)
                -((((abs(position_i_serie[j3]-position_i_serie[k3]))**2)
                +((abs(position_i_serie[j3+longi]-position_i_serie[k3+longi]))**2)+
                ((abs(position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi]))**2)))**(-3))
                F_i_x=24*((position_i_serie[j3]-position_i_serie[k3]))*(((-2)*
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-7)))+
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-4)))
                F_i_y=24*((position_i_serie[j3+longi]-position_i_serie[k3+longi]))*(((-2)*
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-7)))+
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-4)))
                F_i_z=24*((position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi]))*(((-2)*
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-7)))+
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-4)))
                F_i_i_x = F_i_i_x + F_i_x
                F_i_i_y = F_i_i_y + F_i_y
                F_i_i_z = F_i_i_z + F_i_z
                V_f = V_f + V_i
#Here, we are taking the ones we were avoiding only for the partial derivatives
            for k3 in arange(j3):
                F_i_x=24*((position_i_serie[j3]-position_i_serie[k3]))*(((-2)*
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-7)))+
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-4)))
                F_i_y=24*((position_i_serie[j3+longi]-position_i_serie[k3+longi]))*(((-2)*
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-7)))+
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-4)))
                F_i_z=24*((position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi]))*(((-2)*
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-7)))+
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-4)))
                F_i_i_x = F_i_i_x + F_i_x
                F_i_i_y = F_i_i_y + F_i_y
                F_i_i_z = F_i_i_z + F_i_z
#We are summing both and then appending them
            F_f_x = append(F_f_x, F_i_i_x)
            F_f_y = append(F_f_y, F_i_i_y)
            F_f_z = append(F_f_z, F_i_i_z)
#We put them all together in a way there is a correspondence
# between position (variable) and partial derivative
        F_f = array([])
        F_f = append(F_f, F_f_x)
        F_f = append(F_f, F_f_y)
        F_f = append(F_f, F_f_z)
        return V_f, F_f
#We print the number of iterations needed, T and delta for getting the minimum
    print(iteracc)
    print(T)
    print(delta)
#Now, we are calling the minimization routine based on BFGS
    res= minimize(funcionDderiv,position_i_serie,method='BFGS',jac=True,options={'disp':True})
#Now, we are plotting in 3D the final result
    Xf = res.x[0:N]
    Yf = res.x[N:2*N]
    Zf = res.x[2*N:3*N]
    fig = plt.figure((2*i_ite) + 1)
    ax = plt.axes(projection='3d')
    ax.scatter(Xf, Yf, Zf, c='r', marker='o')
    plt.show()
```

Appendix C

Basin-Hopping Code

```
from numpy import *
from pylab import *
import random as rd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.optimize import *
#We ask for the number of atoms you want to organize,
N = int(input('Introduce the number of atoms you want:'))
# the iterations in paralell we want to execute,
ite = int(input('Introduce the number of paralell iterations you want:'))
# the number of iterations in the loop we want to run
iteraciones = int(input('Introduce the number of iterations you want:'))
#Creating a sphere of radius 5*sigma and putting some spots around randomly in a way
# that they represent atoms in a sphere
r = 5.
x = array([])
y = array([])
z = array([])
a_range = arange(0.,2*pi,0.3)
b_range = arange(0.,pi,0.3)
for theta in a_range:
    for phe in b_range:
        x = append(x, 5 + r*cos(theta)*sin(phe))
        y = append(y, 5 + r*sin(theta)*sin(phe))
        z = append(z, 5 + r*cos(phe))
for i_ite in range(ite):
#We select the points now. For that we have chosen a bidimensional array, we set ones to not
# have problems. We choose them randomly with the condition of not being closer than (2^(1/6))
    position = ones((3,N))
    position_i = ones((3,N))
#To determine the len of the array we are taking the atoms from
    s = len(x)
#This number is for getting to know the number of colums we need later
    suma = 0
    for i in range(N):
        a = rd.randrange(0,s) #the random number for choosing the 'atom'
        position[0,i] = x[a]
        position[1,i] = y[a]
        position[2,i] = z[a]
        suma = suma + i
#In this step, we are calculating the distances in all three axes we need between all atoms
#We set ones again to avoid problems. We set 5 row including 3 for the three dimensions and
# 2 for the labels and the colums come from the number of atoms we chose
```

```
    r_distancexyz = ones((5,suma))
#We set this value for taking the distance we want in the loop
    it = 0
    longitu = len(position[0,:])
    j = -1
#We go through all the atoms we have one-by-one to check the distances
    between them all
    while j < N:
        if j == -1:
            it = 0
        else:
            pass
        j = j + 1
#Once we are in one atom, we go through the next ones to calculate the
    distances between them
#We are doing that only for the atoms whose distance has not been
    calculated
        for k in arange(j + 1, longitu):
            r_distancexyz[0,it] = abs(position[0,j]-position[0,k])
            r_distancexyz[1,it] = abs(position[1,j]-position[1,k])
            r_distancexyz[2,it] = abs(position[2,j]-position[2,k])
            r_distancexyz[3,it] = int(j)
            r_distancexyz[4,it] = int(k)
#We are checking the distance in 3D between atoms to be less than sixth
    root
# of two to avoid the repulsion forces
            test_distance = sqrt(r_distancexyz[0,it]**2+r_distancexyz[1,it
                ]**2+
                                    r_distancexyz[2,it]**2)
#Just in the case we want to calculate the equilibrium distance (2 atoms)
            if N == 2:
                pass
            elif test_distance < (2**(1/6)):
#Since this happens, we are taking a new configuration
                a = rd.randrange(0,s)
                position[0,k] = x[a]
                position[1,k] = y[a]
                position[2,k] = z[a]
#We make the loop running again, setting this value to -1
                j = -1
            else:
                pass
            it = it + 1
#We need to use the routine minimization from the beginning, we set a 1D
    array for positions
    position_serie = array([])
    position_serie = append(position_serie, position[0,:])
    position_serie = append(position_serie, position[1,:])
    position_serie = append(position_serie, position[2,:])
    position_i_serie = position_serie
#We create a virtual copy to modify. We select a default position for
    iterations and the
# distances in three axes with labels included. For letting know the
    routine which function we
# want to minimize, we have to define it first. Furthermore, in this
    routine is possible to
# define both the function and the derivative (jac) at the same time.
    However, one has to
# define them with the variables one wants to obtain. In this case, they
    are the
# position_i_serie. For the function, we need the final value after
    inserting the positions.
```

```
#Nevertheless, the jac has to return an array with the partial derivatives of every single
# variable we are taking.
    def funcionDderiv(position_i_serie):
        V_f = 0.
        F_f_x = array([])
        F_f_y = array([])
        F_f_z = array([])
        longi = len(position[0,:])
#It is a loop for calculating the potential, but including the partial derivatives and since
# it needs to be calculated between one atom with the rest of them, we need to include a loop
# for the ones we were avoiding before.
        for j3 in arange(longi):
            F_i_i_x = 0.
            F_i_i_y = 0.
            F_i_i_z = 0.
            for k3 in arange(j3 + 1, longi):
                V_i = 4*((((abs(position_i_serie[j3]-position_i_serie[k3]))**2)+
                ((abs(position_i_serie[j3+longi]-position_i_serie[k3+longi]))**2)+
                ((abs(position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi]))**2))**(-6)-
                ((((abs(position_i_serie[j3]-position_i_serie[k3]))**2)+
                ((abs(position_i_serie[j3+longi]-position_i_serie[k3+longi]))**2)+
                ((abs(position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi]))**2)))**(-3))
                F_i_x=24*((position_i_serie[j3]-position_i_serie[k3]))*(((-2)*
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-7)))+
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-4)))
                F_i_y=24*((position_i_serie[j3+longi]-position_i_serie[k3+longi]))*(((-2)*
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-7)))+
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-4)))
                F_i_z=24*((position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi]))*(((-2)*
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-7)))+
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-4)))
                F_i_i_x = F_i_i_x + F_i_x
                F_i_i_y = F_i_i_y + F_i_y
                F_i_i_z = F_i_i_z + F_i_z
                V_f = V_f + V_i
#Here, we are taking the ones we were avoiding only for the partial derivatives
            for k3 in arange(j3):
                F_i_x=24*((position_i_serie[j3]-position_i_serie[k3]))*(((-2)*
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-7)))+
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-4)))
                F_i_y=24*((position_i_serie[j3+longi]-position_i_serie[k3+longi]))*(((-2)*
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-7)))+
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-4)))
                F_i_z=24*((position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi]))*(((-2)*
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-7)))+
                (((position_i_serie[j3]-position_i_serie[k3])**2+
                (position_i_serie[j3+longi]-position_i_serie[k3+longi])**2+
                (position_i_serie[j3+2*longi]-position_i_serie[k3+2*longi])**2)**(-4)))
                F_i_i_x = F_i_i_x + F_i_x
                F_i_i_y = F_i_i_y + F_i_y
                F_i_i_z = F_i_i_z + F_i_z
            #We are summing both and then appending them
            F_f_x = append(F_f_x, F_i_i_x)
            F_f_y = append(F_f_y, F_i_i_y)
            F_f_z = append(F_f_z, F_i_i_z)
        #We put them all together in a way there is a correspondence
        # between position (variable) and partial derivative
        F_f = array([])
        F_f = append(F_f, F_f_x)
        F_f = append(F_f, F_f_y)
        F_f = append(F_f, F_f_z)
        return V_f, F_f
    result = minimize(funcionDderiv, position_serie, method='L-BFGS-B', jac = True)
```

```
#If we want to modify the accuracy of the routine: options={'ftol':1e-6, '
    gtol':1e-6}
    V_f = result.fun
    for p in arange(len(position[0,:])):
        position[0,p] = result.x[p]
        position[1,p] = result.x[p+len(position[0,:])]
        position[2,p] = result.x[p+2*len(position[0,:])]
        position_i[0,p] = result.x[p]
        position_i[1,p] = result.x[p+len(position[0,:])]
        position_i[2,p] = result.x[p+2*len(position[0,:])]
    print(V_f)
#We select a temperature as a parameter for Boltzmann statistics
    T = 0.8
#We define delta as the value for changing positions, we are also planning
# to modify it depending on the numbers of changing we are accepting
    delta = 0.5
#We define a value to check if we are accepting or not changing
    accept = 0
    reject = 0
#For changing the T and delta
    iteracc = 0
    factor = 1
    factor2 = 1
    ex = 0
#Running the loop with the condition we want.
    for iteracion in arange(iteraciones):
        iteracc = iteracc + 1
#We select all atoms to change their position
        for a_xyz in arange(len(position_i[0,:])):
#We change the position of these atoms
#Choosing a random value for this purpose in the three axes
            scribble1 = rd.random()
            scribble2 = rd.random()
            scribble3 = rd.random()
            position_i[0,a_xyz] = ((position_i[0,a_xyz]) + 2 * delta * (
                scribble1 - 0.5))
            position_i[1,a_xyz] = ((position_i[1,a_xyz]) + 2 * delta * (
                scribble2 - 0.5))
            position_i[2,a_xyz] = ((position_i[2,a_xyz]) + 2 * delta * (
                scribble3 - 0.5))
#Minimization Routine (Root finding)
#After that, we are using a minimization routine for getting the minima
    again
        position_i_serie = array([])
        position_i_serie = append(position_i_serie, position_i[0,:])
        position_i_serie = append(position_i_serie, position_i[1,:])
        position_i_serie = append(position_i_serie, position_i[2,:])
        res = minimize(funcionDderiv, position_i_serie, method='L-BFGS-B',
            jac = True)
#If we want to modify the accuracy of the routine: options={'ftol':1e-6, '
    gtol':1e-4}
        V_f_i = res.fun
#If energy is lower, then yes both energy and position are replaced
        if V_f_i < V_f:
            V_f_ant = V_f
            V_f = V_f_i
            for p_i in arange(len(position_i[0,:])):
                position_i[0,p_i] = res.x[p_i]
                position_i[1,p_i] = res.x[p_i+len(position_i[0,:])]
                position_i[2,p_i] = res.x[p_i+2*len(position_i[0,:])]
                position[0,p_i] = res.x[p_i]
                position[1,p_i] = res.x[p_i+len(position_i[0,:])]
                position[2,p_i] = res.x[p_i+2*len(position_i[0,:])]
            accept = accept + 1
            print(V_f)
            print(delta)
#If not, hence we are using a Boltzmann statistic for whether choose or not the new value
        else:
            a_i = rd.random()
            if V_f == V_f_i:
                for p_i_j_j in arange(len(position_i[0,:])):
                    position_i[0,p_i_j_j] = position[0,p_i_j_j]
                    position_i[1,p_i_j_j] = position[1,p_i_j_j]
                    position_i[2,p_i_j_j] = position[2,p_i_j_j]
                reject = reject + 1
                print(V_f)
                print(delta)
                print('Iguales')
                print(V_f_i)
            elif a_i < exp((-(V_f_i - V_f)) / T):
                V_f_ant = V_f
                V_f = V_f_i
                for p_i_j in arange(len(position_i[0,:])):
                    position_i[0,p_i_j] = res.x[p_i_j]
                    position_i[1,p_i_j] = res.x[p_i_j+len(position_i[0,:])]
                    position_i[2,p_i_j] = res.x[p_i_j+2*len(position_i[0,:])]
                    position[0,p_i_j] = res.x[p_i_j]
                    position[1,p_i_j] = res.x[p_i_j+len(position_i[0,:])]
                    position[2,p_i_j] = res.x[p_i_j+2*len(position_i[0,:])]
                accept = accept + 1
                print(V_f)
                print(delta)
#If we are not choosing the new energy, then the positions are the ones we had before again
            else:
                print(V_f_i)
                for p_i_j in arange(len(position_i[0,:])):
                    position_i[0,p_i_j] = position[0,p_i_j]
                    position_i[1,p_i_j] = position[1,p_i_j]
                    position_i[2,p_i_j] = position[2,p_i_j]
                reject = reject + 1
                print(V_f)
                print(delta)
        if iteracc == (100*factor2):
            if accept < reject:
                delta = delta * 1.05
            else:
                delta = delta * 0.95
            factor2 = factor2 + 1
        else:
            delta = delta
        print(iteracc)
#Once the loop is done we collect the data of the atoms and representing them
    X = position[0,:]
    Y = position[1,:]
    Z = position[2,:]
    fig = plt.figure(2*i_ite + 1)
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(X, Y, Z, c='r', s = 250, marker='o')
    plt.show()
```

Bibliography

[1] Jan Brinkhuis and Vladimir Tikhomirov, Optimization: Insights and Applications, Princeton Series in Applied Mathematics (2005)

[2] Malinowska, Agnieszka B. and Torres, Delfim F. M., The diamond-alpha Riemann integral and mean value theorems on time scales, Dynam. Systems Appl. (2008) https://arxiv.org/abs/0804.4420

[3] L.T. Wille, Simulated annealing and the topology of the potential energy surface of Lennard-Jones clusters, Computational Materials Science, 17, 551 (2000)

[4] J.E. Jones, On the determination of molecular fields. II. From the equation of state of a gas, Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, 106, 738, "463-477" (1924) doi:10.1098/rspa.1924.0082

[5] Roy L. Johnston, Evolving better nanoparticles: Genetic algorithms for optimising cluster geometries, School of Chemistry, University of Birmingham, Edgbaston, Birmingham, (2003)

[6] Schelstraete, Sigurd and Verschelde, Henri, Finding Minimum-Energy Configurations of Lennard-Jones Clusters Using an Effective Potential, The Journal of Physical Chemistry A, 101, 3, "310-315" (1997) doi:10.1021/jp9621181

[7] Wales, David J. and Doye, Jonathan P. K., 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, 101, 28, "5111-5116" (1997) doi:10.1021/jp970984n

[8] Kaarle Ritvanen, Python Scripts for Structural Optimization of Small Molecules, Helsinki University of Technology, (2004)

[9] Wales, D., Energy Landscapes: Applications to Clusters, Biomolecules and Glasses, Cambridge Molecular Science (2004) doi:10.1017/CBO9780511721724

[10] Wales, D., Energy Landscapes: Applications to Clusters, Biomolecules and Glasses, Cambridge Molecular Science, "333-335" (2004) doi:10.1017/CBO9780511721724.007

[11] David J. Wales and Harold A. Scheraga, Global Optimization of Clusters, Crystals, and Biomolecules, Science's Compass, 285, (1999)

[12] Wales, D., Energy Landscapes: Applications to Clusters, Biomolecules and Glasses, Cambridge Molecular Science, "241-282" (2004) doi:10.1017/CBO9780511721724.006

[13] Oliphant, Jones and Peterson, https://scipy.org/ (2018)

[14] Adrian S. Lewis and Michael L. Overton, Nonsmooth Optimization Via BFGS, SIAM J. Optimiz., "1-35" (2009) http://www.optimization-online.org/DB-FILE/2008/12/2172.pdf

[15] C. Zhu, R. H. Byrd, P. Lu and J. Nocedal, L-BFGS-B Fortran Subroutines for Large Scale Bound Constrained Optimization, Nothrwestern University (1994)

[16] Wales, D., Energy Landscapes: Applications to Clusters, Biomolecules and Glasses, Cambridge Molecular Science, "434-529" (2004) doi:10.1017/CBO9780511721724.009

[17] John D. Hunter, https://matplotlib.org/ (2018)

[18] Caparrini F. S., Local Search Algorithms in NetLogo (2018) http://www.cs.us.es/ fsancho/?e=132

[19] Goedecker S., Minima hopping: An efficient search method for the global minimum of the potential energy surface of complex molecular systems, The Journal of Chemical Physics, 120, 9911 (2004) doi:10.1063/1.1724816