

Department of Mathematics & Computer Science

Numerical optimization methods applied to molecular geometries

2WH40 Bachelor Final project

M.C. Koot 1589164

14-07-2023

Supervisors:
Dr. Björn Baumeier

Contents

1	Introduction to the Project	4
2	Groundstate vs Excited State	5
2.1	Defining the Potential Energy V	5
2.2	Solving Schrödinger's Equation	5
2.3	Defining the Wavefunction	6
2.4	Quantum Mechanics of Interacting Electrons	7
3	Optimization methods	8
3.1	Used variables and notation	8
3.2	Broyden-Fletcher-Goldfarb-Shanno algorithm	8
3.2.1	Convergence criteria:	10
3.2.2	Algorithm described:	10
3.3	Conjugate Gradients	11
3.4	SLSQP	13
3.4.1	Using Least Squares to find Lagrange multipliers	13
3.4.2	Merit function	14
3.4.3	Convergence criteria	14
3.5	Trust-Constraint	15
3.5.1	Updating Lagrange Multipliers	16
3.5.2	Merit Function	16
3.5.3	Algorithm	17
4	Code Overview	18
4.1	Directory Structure	18
4.2	XYZ Files	18
4.3	Calculating Gradients	19
4.4	Running xtp_tools	19
4.5	Extracting Energies from .orb Files	20
4.6	Updating Files	20
4.7	Implementing scipy.optimize.minimize	20
4.8	Output and Trajectory File	21
5	Planarization of NH3	23
6	Analysis of methods in practice	24
6.1	H2O results	25
6.2	CO2 results	27
6.3	Non-convergence problems	28
7	Conclusion	30
8	Appendix	32
8.1	Wolfe's condition	32
8.2	Sherman-Morrison-Woodbury formula	33
8.3	Code	33

1 Introduction to the Project

"The world is not made of molecules, the world is made of stories" - Muriel Rukeyser

For the sake of this project the world is made of molecules, the project focuses on optimizing the structures of individual molecules by minimizing their total energy in the ground state. Code has been developed over the past four months to facilitate this optimization process. The initial step of creating this code involved moving the atoms of an arbitrary molecule a distance δ in the x , y , and z directions.

By performing this movement for every atom of the molecule at distances δ and $-\delta$, numerical derivatives can be computed for each atom using central difference (or any other first-order numerical method). The numerical derivative for atom i in direction x is given by $(dx)_i = (E_{\text{atom};i}(x + \delta) - E_{\text{atom};i}(x - \delta)) / 2\delta$.

These numerical derivatives in the x , y , and z directions are then combined into a vector to obtain the energy gradients for each atom in the molecule. These gradients are particularly useful as they closely relate to the forces acting upon each individual atom, denoted by $F_{\text{on atom}} = -\nabla E = (dx, dy, dz)^T$. Minimizing these forces helps to stabilize the molecule, making it valuable for identifying stable molecular structures.

The minimization process is handled by `scipy.optimize.minimize` [3], which is the optimizer of choice for this project. This calculator takes in the energy, energy gradient of a molecule-structure and the initial structure itself, and performs calculations to obtain a lower energy value by making slight adjustments to the atom configuration. The way these atoms are adjusted is done using a numerical method. The numerical method that will be discussed here are: BFGS, CG, SLSQP, and trust-constraint. The chosen method determines the approach used for shifting the molecules during the optimization process. As the methods are iterative, multiple iterations are required by the calculator to achieve an optimized molecule.

Furthermore, this document will explore the code implementation for performing these optimizations, as well as discuss some of the optimization methods employed. The performance of these methods will also be evaluated.

Finally, we will attempt to address several questions, such as whether the methods yield an optimal structure, if any of the numerical methods are superior for optimizing molecular geometries, and whether there are more reliable or less reliable methods. Additionally, we will explore the possibility of optimizing excited state energy and provide insights on the feasibility of this type of optimization.

2 Groundstate vs Excited State

To illustrate the distinction between ground-state energies and excited state energies, we will explore a particle-in-a-box-model in this section. This model provides a straightforward framework for calculations and offers potential values for the particle's energy and wavefunction. It is important to note that this example simplifies the energy calculations compared to the complexities involved in working with molecules.

2.1 Defining the Potential Energy V

For this example, we adopt the approach described in [7], where the potential energy of the particle within the box is defined as $V(x) = 0$. Outside the box, i.e., for $x < 0$ or $x > L$, $V(x) = \infty$, ensuring that the particle does not reside at the walls or outside the box. It is worth noting that the particle's movement is restricted to the x-axis.

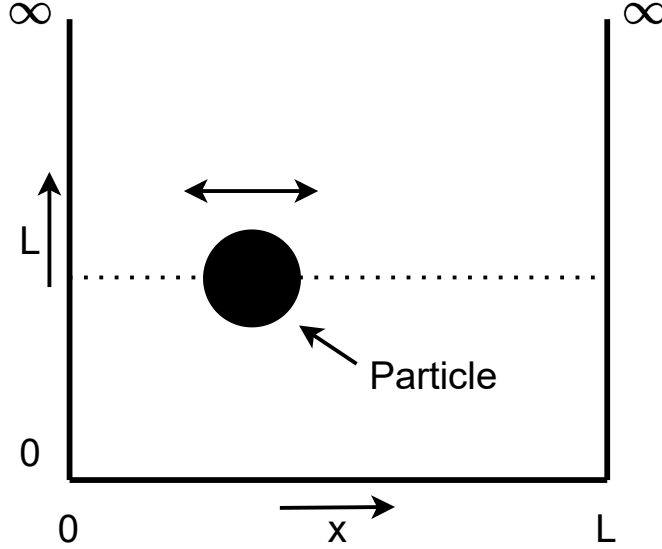


Figure 1: Figure illustrating a particle in a box as described above

Based on this scenario, we will explore the potential energy states of the particle and examine the corresponding wavefunction denoted as ψ and the squared magnitude, ψ^2 . Analyzing ψ^2 allows us to better understand the probability distribution and likely locations of the particle.

2.2 Solving Schrödinger's Equation

The time-independent Schrödinger's equation for a particle with mass m , moving in one direction with energy E , is given by:

$$-\frac{\hbar^2}{2m} \frac{d^2\psi(x)}{dx^2} + V(x)\psi(x) = E[\psi(x)] \quad (1)$$

where:

- \hbar represents the reduced Planck constant, defined as $\hbar = \frac{h}{2\pi}$, with h being the Planck constant.
- m is the mass of the particle.
- $\psi(x)$ denotes the stationary, time-independent wavefunction.
- $V(x)$ represents the potential energy as a function of position.
- E corresponds to the energy of the particle, which is a real number.

In our model where $V(x) = 0$, the equation simplifies to:

$$-\frac{\hbar^2}{2m} \frac{d^2\psi(x)}{dx^2} = E[\psi(x)] \quad (2)$$

The general solution to this equation is given by:

$$\psi(x) = A \sin(kx) + B \cos(kx) \quad (3)$$

where A , B , and k are constants.

2.3 Defining the Wavefunction

To apply boundary conditions to the system, we use the fact that finding the particle at $x = 0$ or $x = L$ is impossible due to the infinite potential energy at those points. This constraint leads to the following condition:

$$A \sin(k \cdot 0) + B \cos(k \cdot 0) = B = 0 \quad (4)$$

Hence, in our case B needs to be 0 to fulfill the boundary conditions which gives $\psi(x) = A \sin(kx)$

To determine the value of k , we differentiate the wavefunction with respect to x :

$$\frac{d\psi}{dx} = kA \cos(kx) \quad (5)$$

$$\frac{d^2\psi}{dx^2} = -k^2 A \sin(kx) \quad (6)$$

By substituting the wavefunction ψ into Equation 6, we obtain:

$$\frac{d^2\psi}{dx^2} = -k^2 \psi \quad (7)$$

Comparing this result with the Schrödinger equation, we find:

$$k = \left(\frac{8\pi^2 m E}{h^2} \right)^{1/2} \quad (8)$$

Substituting the value of k back into our wavefunction, we have:

$$\psi(x) = A \sin \left(\left(\frac{8\pi^2 m E}{h^2} \right)^{1/2} x \right) \quad (9)$$

Next, we determine the value of A by applying the normalization condition. Since the probability of finding the particle inside the box is 1, the integral of ψ^2 over the box must equal 1:

$$\int_0^L \psi^2(x) dx = 1 \iff A^2 \int_0^L \sin^2 \left(\frac{n\pi x}{L} \right) dx = 1 \quad (10)$$

The solution to this integral can be found in an integral table, yielding:

$$A = \sqrt{\frac{2}{L}} \quad (11)$$

Thus, the normalized wavefunction becomes:

$$\psi(x) = \sqrt{\frac{2}{L}} \sin \left(\frac{n\pi}{L} x \right), \text{ where } 0 < x < L \quad (12)$$

Note that this the wavefunction is different for different values of n .

The allowed energies can be determined by solving for E in Equation 2 which gives:

$$E_n = \frac{n^2 h^2}{8mL^2} \quad (13)$$

These energies are always greater than 0, given that $n \in \mathbb{N} \setminus \{0\}$ and h is a constant with $h > 0$. It is important to note that we assume $m, L \in \mathbb{R}$.

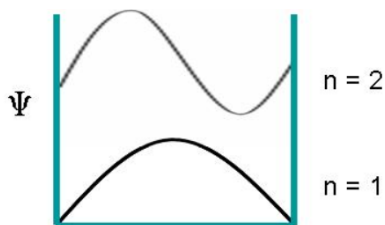


Figure 2: Wave functions at $n = 1$ and the $n = 2$ energy levels.

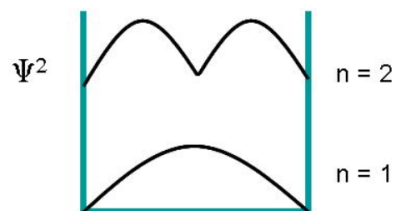


Figure 3: Probability of finding the particle at position x for energy levels $n = 1$ and $n = 2$.

Both these pictures are taken from [7].

Now, this is relevant since I will be using xtp tools to calculate the ground-state energies, corresponding to $n = 1$. Additionally, xtp tools will be used to compute singlet energies, which represent higher energy state (we will take $n = 2$ in section 5). Note that not all higher energy states are singlet states.

2.4 Quantum Mechanics of Interacting Electrons

The particle-in-a-box system discussed above introduces the concept of discrete energies and ground and excited states. However, when applying Schrödinger's equation to molecules, several challenges arise:

The wavefunction becomes a function of the positions of all the electronic and nuclear coordinates. The potential is formed by Coulomb interactions among all charged particles. Due to these complexities, it is difficult to find analytical solutions for steric effects*, and even numerical solutions are challenging to obtain. To address these challenges, approximate theories have been developed. However, these approximate methods come with limitations and are not universally applicable. One such theory is the GW-BSE method, which combines the GW approximation and the Bethe-Salpeter equation to calculate electronic properties and excitations in molecules.

By employing the GW-BSE method, we can obtain insights into the electronic structure, electronic properties, and excited states of molecules. This computational approach allows for a more accurate description of the behavior of electrons and their interactions within a molecular system.

Overall, the GW-BSE method serves as a valuable tool in understanding and predicting the electronic properties and excitations in molecular geometries. It enables researchers to study complex systems and obtain information that may not be accessible through analytical or simpler computational methods. These methods will be used for computing the energy of the molecules, which will be done using xtp tools of votca [1] and for a paper about this subject we cite: [10].

*steric effects: non-bonding interactions acting on the molecule affecting the shape

3 Optimization methods

This chapter discusses four numerical methods utilized by `scipy.optimize.minimize`, hereafter referred to as `scipy.minimize`. These methods are employed to iteratively search for an improved molecule structure. Specifically, they operate by taking the total energy and some form of gradient calculation as input at each step.

3.1 Used variables and notation

In the following sections, we explore the numerical methods from an analytical perspective. To establish a common framework, we introduce the following variables:

- \mathcal{H}_k^{-1} : This represents the approximation of the Hessian at iteration k in the subsequent sections. It is important to note that \mathcal{H}_k^{-1} is not used to denote the analytical Hessian, as the analytical Hessian cannot be computed for our specific problem.
- Generally, the methods seek a direction along which a more optimal solution can be found, denoted by \mathbf{d} .

We will clarify the notation used throughout the discussion:

- $\|\cdot\|$ denotes the ℓ^2 norm. In the three-dimensional case, it is defined as follows: $\|(x_1, x_2, x_3)\| = \sqrt{x_1^2 + x_2^2 + x_3^2}$
- $\|\cdot\|_\infty$ represents the infinity norm. In the three-dimensional case, it is defined as: $\|(x_1, x_2, x_3)\|_\infty = \max(|x_1|, |x_2|, |x_3|)$
- (\mathbf{x}, \mathbf{y}) denotes the dot product of vectors \mathbf{x} and \mathbf{y} .
- Boldface math symbols, such as \mathbf{x} , indicate vectors.
- In the upcoming sections, the subscript k is used to denote a variable or vector at the k -th iteration of the algorithm.

3.2 Broyden-Fletcher-Goldfarb-Shanno algorithm

To explain how the Broyden-Fletcher-Goldfarb-Shanno algorithm works (from now on referred to as BFGS), we will closely follow the explanations provided in Nocedal and Wright’s book on numerical optimization [8].

We start by describing the quadratic model of the objective function evaluated at \mathbf{x}_k , where $f(\mathbf{x}_k)$ denotes our energy calculated at iteration k , and $\nabla f(\mathbf{x}_k)$ is our gradient at the same iteration which are calculated using xtp tools of votca [1], f and ∇f will also be used in the coming sections where these will mean the same. We will start by stating the objective function, which is the function we are going to try to minimize as a sub-problem to minimizing $f(\mathbf{x}_k)$:

$$m_k(\mathbf{d}_k) = f(\mathbf{x}_k) + \nabla f(\mathbf{x}_k)\mathbf{d}_k + \frac{1}{2}\mathbf{d}_k^T \mathcal{H}_k^{-1} \mathbf{d}_k \quad (14)$$

Here, \mathcal{H}_k^{-1} is an $n \times n$ positive semi-definite (PSD) symmetric matrix (The Hessian does satisfy these assumptions) that will be updated at every iteration. Now, the minimizer \mathbf{d}_k can be written down as:

$$\mathbf{d}_k = -\mathcal{H}_k^{-1} \nabla f(\mathbf{x}_k) \quad (15)$$

Here, $-\mathcal{H}_k^{-1}$ is guessed (initially) by `scipy.minimize`, and $\nabla f(\mathbf{x}_k)$ is computed using numerical derivatives by my code (as explained in Section 4.3). The new iterate \mathbf{x}_{k+1} is updated in the following way:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k \quad (16)$$

Here, the step-length α_k has to satisfy Wolfe’s condition.

Now, guessing this \mathcal{H}_k^{-1} at every iteration, there is an algorithm that does this in the following manner:

$$\mathcal{H}_{k+1}^{-1} \alpha_k \mathbf{d}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k) \quad (17)$$

To simplify notation, we introduce the following variables:

$$\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k = \alpha_k \mathbf{d}_k \text{ and } \mathbf{y}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k) \quad (18)$$

Using this notation, we end up with the following equation, which is called the secant equation:

$$\mathcal{H}_{k+1}^{-1} \mathbf{s}_k = \mathbf{y}_k \quad (19)$$

Given \mathbf{s}_k and \mathbf{y}_k using Equation (19), we have that the PSD symmetric matrix \mathcal{H}_{k+1}^{-1} maps \mathbf{s}_k into \mathbf{y}_k , which is only possible if:

$$\mathbf{s}_k^T \mathbf{y}_k > 0 \quad (20)$$

This so-called curvature condition turns out to hold for convex and non-convex functions.

It turns out that when Equation (20) holds, Equation (19) always has infinite solutions for \mathcal{H}_{k+1}^{-1} . To determine a unique solution for \mathcal{H}_{k+1}^{-1} , we impose another condition, which is that \mathcal{H}_{k+1}^{-1} has to be in some manner closest to \mathcal{H}_k . Using this approach, we arrive at the following optimization problem:

$$\begin{aligned} \min_{\mathcal{H}_{k+1}^{-1}} & \|\mathcal{H}_{k+1}^{-1} - \mathcal{H}_k^{-1}\| \\ \text{Subject to: } & \mathcal{H}_{k+1}^{-1} = (\mathcal{H}_{k+1}^{-1})^T, \mathcal{H}_{k+1}^{-1} \mathbf{s}_k = \mathbf{y}_k \end{aligned}$$

This problem turns out to have the unique solution:

$$\mathcal{H}_{k+1}^{-1} = (\mathcal{I} - \rho_k \mathbf{y}_k \mathbf{s}_k^T) \mathcal{H}_k^{-1} (\mathcal{I} - \rho_k \mathbf{s}_k \mathbf{y}_k^T) + \rho_k \mathbf{y}_k \mathbf{y}_k^T \quad (21)$$

Here,

$$\rho_k = \frac{1}{\mathbf{y}_k^T \mathbf{s}_k} \quad (22)$$

This is called the DFP update formula.

For the implementation of the method, a few more pieces have to come together. One of these is finding the search direction (aforementioned \mathbf{d}_k) using matrix-vector multiplications. This can be done using the Sherman-Morrison-Woodbury formula as described in subsection 8.2. We derive the following expression for the updating of the inverse Hessian approximation $H_k = \mathcal{H}_k^{-1}$:

$$H_{k+1}^{-1} = H_k^{-1} - \frac{H_k^{-1} \mathbf{y}_k \mathbf{y}_k^T H_k^{-1}}{\mathbf{y}_k^T H_k^{-1} \mathbf{y}_k} + \frac{\mathbf{s}_k \mathbf{s}_k^T}{\mathbf{y}_k^T \mathbf{s}_k} \quad (23)$$

We note one important thing here that the last two terms of Equation (23) are rank-one matrices, so that H_k undergoes a rank-two modification. This is a very nice concept of quasi-Newton method updating: Rather than recalculating the Hessian (or inverse Hessian) at every iteration, the algorithm combines information about the current Hessian with the new information from the objective function to get a new approximation of the updated Hessian.

Algorithm 1 Algorithm BFGS

```
1:  $k \leftarrow 0$ 
2: While  $\|\nabla f(\mathbf{x}_k)\| > \text{tol}$ 
3:   Compute search direction:  $\mathbf{d}_k = -\mathcal{H}^{-1}f(\mathbf{x}_k)$ 
4:   Set  $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$ , where  $\alpha_k$  is computed from a line-search procedure
5:     which has to satisfy Wolfe's condition as mentioned in subsection 8.1.
6:   Define  $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$  and  $\mathbf{y}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$ , compute  $\mathcal{H}_k^{-1}$ 
7:   Compute  $\mathcal{H}_{k+1}$  using (24)
8:    $k \leftarrow k + 1$ 
```

Figure 4: BFGS algorithm as given in [8]

The actual BFGS statement is found when making a change in argument that led to Equation (21). Instead of imposing conditions on \mathcal{H}_k^{-1} , we impose condition on the inverse H_k^{-1} . In this case, the updated H_{k+1}^{-1} still needs to satisfy the secant equation:

$$H_{k+1}^{-1} \mathbf{y}_k = \mathbf{s}_k \quad (24)$$

Now using the similar argument that we want to find a unique solution H_{k+1} closest to H_k , which leads to the BFGS Hessian inverse updating formula:

$$H_{k+1}^{-1} = (\mathcal{I} - \rho_k \mathbf{s}_k \mathbf{y}_k^T) H_k^{-1} (\mathcal{I} - \rho_k \mathbf{y}_k \mathbf{s}_k^T) + \rho_k \mathbf{s}_k \mathbf{s}_k^T \quad (25)$$

Starting the method:

Now the last issue is making an initial guess for H_0 . Now for this, there is no easy solution. Sometimes an educated guess can be made based on \mathbf{x}_0 . When this does not seem feasible, the identity matrix (or a multiple of the identity matrix) can (and will) be chosen.

3.2.1 Convergence criteria:

The convergence criteria employed by `scipy.minimize` are convergence criteria on the gradient and on the step size. The first is that:

$$\|\nabla f(\mathbf{x}_k)\| > \text{tol}$$

And for the step size, we have the criteria of Wolfe's condition as mentioned in subsection 8.1. Unsuccessful convergence, as in Table 1, is mostly due to Wolfe's condition coming up with no possible step size (which might also indicate our solution being optimal in some manner).

3.2.2 Algorithm described:

To make the algorithm more clear, we will describe the algorithm shortly below:

Given a starting point \mathbf{x}_0 , initial Hessian approximation \mathcal{H}_0^{-1} , and convergence tolerance `tol`, we have: Once this algorithm converged, we have the minimum point $\mathbf{x}_{\min} = \mathbf{x}_k$.

This is a brief description of how the BFGS method works. The actual implementation in `scipy.minimize` might contain additional steps or modifications to improve performance or handle specific scenarios.

3.3 Conjugate Gradients

This explanation of the conjugate gradients method (referred to as CG) closely follows Shewchuk (1994) [9] for the first part and Hestenes (1952) [5] for the second part. In general, CG is a method used to solve equations of the form:

$$A\mathbf{x} = \mathbf{b}, \quad (26)$$

where A represents the Hessian matrix and \mathbf{b} denotes the forces acting on the atoms (our gradient).

Assuming that A is symmetric and positive semi-definite, we can minimize the energy function $f(\mathbf{x})$ of a molecule by finding the solution to $A\mathbf{x} = \mathbf{b}$. To explain this, we employ the quadratic form:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T \mathbf{x} + c. \quad (27)$$

To minimize $f(\mathbf{x})$, setting $\nabla f(\mathbf{x})$ to zero is useful. By substituting this into (27), we obtain:

$$\nabla f(\mathbf{x}) = \frac{1}{2}A^T \mathbf{x} + \frac{1}{2}A\mathbf{x} - \mathbf{b}. \quad (28)$$

Simplifying this equation based on the assumption of symmetry in A , we have:

$$\nabla f(\mathbf{x}) = A\mathbf{x} - \mathbf{b}. \quad (29)$$

Setting $\nabla f(\mathbf{x})$ to zero yields the equation $A\mathbf{x} = \mathbf{b}$.

In the context of our specific problem, we assume that A is an approximation of the inverse Hessian denoted by \mathcal{H}_k^{-1} . The inverse Hessian satisfies the aforementioned assumptions, making it suitable for our purposes. In practice, this comparison also holds.

Now, we will introduce and explain the formulas involved in using the CG method:

$$\mathbf{d}_0 = \mathbf{r}_0 = \mathbf{b} - \mathcal{H}_k^{-1}\mathbf{x}_0, \quad (30)$$

$$\alpha_k = \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{d}_k^T \mathcal{H}_k^{-1} \mathbf{d}_k}, \quad (31)$$

$$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathcal{H}_k^{-1} \mathbf{d}_k, \quad (32)$$

$$\beta_{k+1} = \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k}, \quad (33)$$

$$\mathbf{d}_{k+1} = \mathbf{r}_{k+1} + \beta_{k+1} \mathbf{d}_k, \quad (34)$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k. \quad (35)$$

For efficient convergence, we solve the sub-problem $\mathcal{H}_k^{-1}\mathbf{x} = \mathbf{b}'$, where \mathcal{H}_k^{-1} is an $n \times n$ matrix, and $\mathbf{b}'_k = -\nabla f(\mathbf{x}_k) - \mathcal{H}_k^{-1}\mathbf{x}_0$. We solve this sub-problem first to obtain orthogonal \mathbf{d}_k and \mathbf{r}_k for $k > 0$, which leads to faster and usually better convergence compared to our initial problem. We solve the problem by following these two steps:

Initial step:

We take an estimate \mathbf{x}_0 of our solution \mathbf{x} and compute the residual \mathbf{r}_0 and directions \mathbf{d}_0 using (30).

General steps:

After determining the estimate \mathbf{x}_k with $k \geq 1$, the residual \mathbf{r}_k , and the direction \mathbf{d}_k , we can compute solutions to the modified problem, \mathbf{r}_{k+1} and \mathbf{d}_{k+1} , using equations (31) through (35).

This provides a solution \mathbf{x} to the problem $\mathcal{H}_k^{-1}\mathbf{x} = \mathbf{b}'_i$ as follows:

$$\mathbf{x} = \sum_{i=0}^{k-1} \frac{(\mathbf{d}_i, \mathbf{b}'_i)}{(\mathcal{H}_i^{-1} \mathbf{d}_i, \mathbf{d}_i)} \cdot \mathbf{d}_i \quad (36)$$

Although using this formula might be practical for smaller systems, it is preferred to use equations (31) through (35) since storing the \mathbf{d}_i values may not be computationally costly.

In [4], there is a helpful diagram illustrating the CG algorithm, which describes these steps iteratively (see Figure 5).

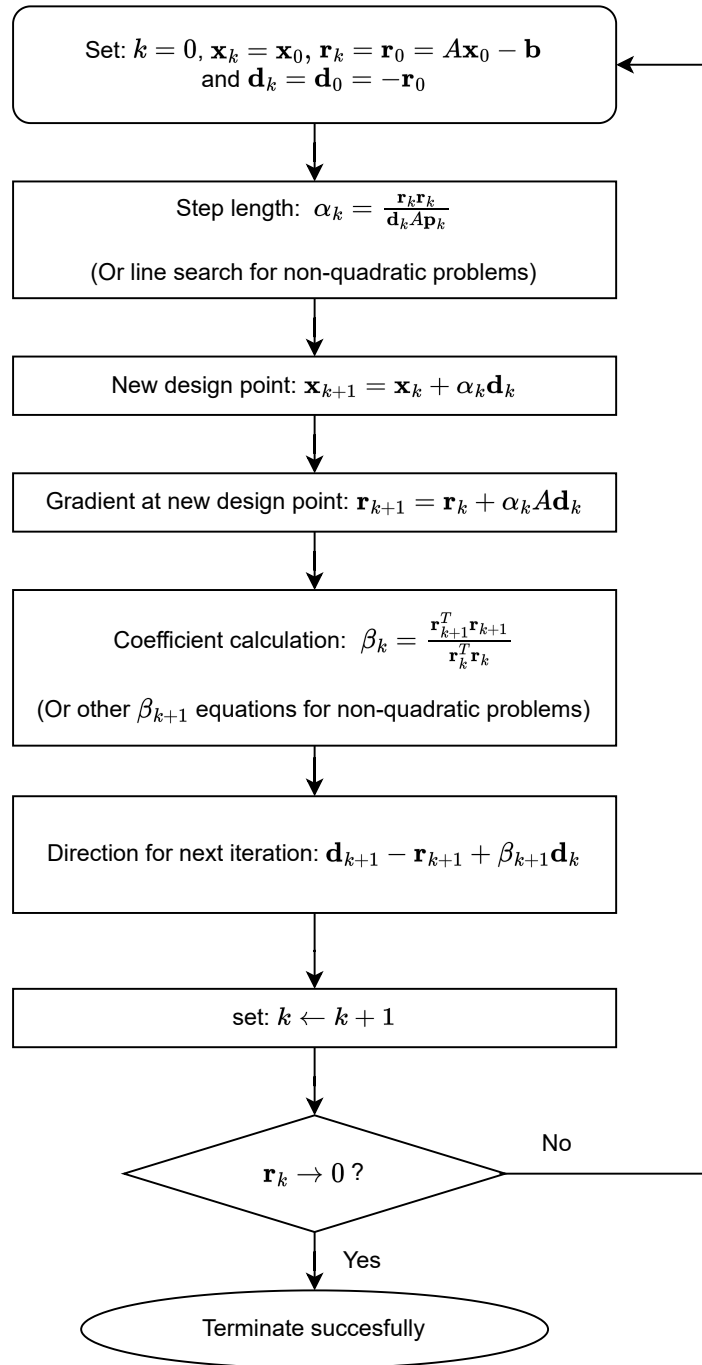


Figure 5: Diagram illustrating how CG works (Figure recreated from [4]).

3.4 SLSQP

We will refer to the works of [4] and [2] for this section on SLSQP.

Sequential Least Squares Programming (SLSQP) is a method used to solve multi-objective optimization problems (MOPs) of the form:

$$\min_{\mathbf{x} \in \mathbb{R}^n} \mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_K(\mathbf{x}) \end{bmatrix} \text{ subject to } \mathbf{h}(\mathbf{x}) = \begin{bmatrix} h_1(\mathbf{x}) \\ h_2(\mathbf{x}) \\ \vdots \\ h_N(\mathbf{x}) \end{bmatrix} = \mathbf{0} \text{ and } \mathbf{g}(\mathbf{x}) = \begin{bmatrix} g_1(\mathbf{x}) \\ g_2(\mathbf{x}) \\ \dots \\ g_P(\mathbf{x}) \end{bmatrix} \leq \mathbf{0} \quad (37)$$

In our specific case, the second constraint $g(\mathbf{x}) \leq 0$ is unnecessary. Thus, we take $h(\mathbf{x}) = g(\mathbf{x}) = \nabla f(\mathbf{x})$. This allows us to transform the problem into:

$$\min_{\mathbf{x} \in \mathbb{R}^n} f(\mathbf{x}), \text{ subject to: } \nabla f(\mathbf{x}) = \mathbf{0} \quad (38)$$

Similar to the BFGS and CG methods, we use an objective function derived from the standard Lagrangian:

$$L(\mathbf{x}_k, \lambda_k) = f(\mathbf{x}_k) + \lambda_k^T \nabla f(\mathbf{x}_k) \quad (39)$$

Here, λ denotes the Lagrange multipliers for equality constraints.

The algorithm applies Newton's method to the Karahn-Kuhn-Tucker (KKT) conditions to find the next optimal point. This involves solving the system:

$$\begin{cases} \left(\frac{\partial^2 L_k}{\partial \mathbf{x}^2} \right)_k \Delta \mathbf{x} + \left(\frac{\partial \nabla f(\mathbf{x})}{\partial \mathbf{x}} \right)_k^T \lambda_{k+1} = - \left(\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right)_k \\ \left(\frac{\partial \nabla f(\mathbf{x})}{\partial \mathbf{x}} \right)_k \mathbf{d}_k = - \nabla f(\mathbf{x}) \end{cases} \quad (40)$$

Here, $\mathbf{d}_k = \mathbf{x} - \mathbf{x}_k$ and λ_{k+1} are the unknowns. Solving (40)

is equivalent to solving the problem:

$$\begin{aligned} & \text{Minimize } \frac{1}{2} \mathbf{d}_k^T \left(\frac{\partial^2 L(x, \lambda_k)}{\partial \mathbf{x}^2} \right)_k \mathbf{d}_k + \left(\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right)_k^T \mathbf{d}_k \\ & \text{Subject to } \left(\frac{\partial \nabla f(\mathbf{x})}{\partial \mathbf{x}} \right)_k \mathbf{d}_k = - \nabla f(\mathbf{x}_k) \end{aligned} \quad (41)$$

Solving (41) provides a direction \mathbf{d}_k that can be used for a line search to achieve an improvement in the objective function.

3.4.1 Using Least Squares to find Lagrange multipliers

The Lagrange multipliers are necessary for local convergence analysis and Hessian approximation through BFGS. We consider the problem as formulated in (38), with the Lagrangian:

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda^T \nabla f(\mathbf{x}) \quad (42)$$

Assuming we are near the optimal solution at \mathbf{x}_k , if the gradients $\frac{\partial f(\mathbf{x}_k)}{\partial \mathbf{x}}$ and $\frac{\partial \nabla f(\mathbf{x}_k)}{\partial \mathbf{x}}$ have been obtained, then $\frac{\partial L(\mathbf{x}_k, \lambda)}{\partial \mathbf{x}}$ should be close to 0 because:

$$\frac{\partial L(\mathbf{x}_k, \lambda_k)}{\partial \mathbf{x}} = \frac{\partial f(\mathbf{x}_k)}{\partial \mathbf{x}} + \lambda_k^T \frac{\partial \nabla f(\mathbf{x}_k)}{\partial \mathbf{x}} \approx 0 \quad (43)$$

However, an λ_k that satisfies $\frac{\partial L(\mathbf{x}_k, \lambda_k)}{\partial \mathbf{x}} = 0$ may not exist. To find an λ_k that minimizes the derivative of L as closely as possible, we use Least Squares. This leads to the following expression for λ_k :

$$\begin{aligned}
\lambda_k &= - \left[\sum_{i=1}^n \left(\frac{\partial \nabla f(\mathbf{x}_k)}{\partial \mathbf{x}_i} \right)^2 \right]^{-1} \sum_{i=1}^n \left(\frac{\partial f(\mathbf{x}_k)}{\partial \mathbf{x}_i} \cdot \frac{\partial \nabla f(\mathbf{x}_k)}{\partial \mathbf{x}_i} \right) \\
&= - \left[\left(\frac{\partial \nabla f(\mathbf{x}_k)}{\partial \mathbf{x}} \right)^T \left(\frac{\partial \nabla f(\mathbf{x}_k)}{\partial \mathbf{x}} \right) \right]^{-1} \left(\frac{\partial \nabla f(\mathbf{x}_k)}{\partial \mathbf{x}} \right)^T \left(\frac{\partial f(\mathbf{x}_k)}{\partial \mathbf{x}} \right)
\end{aligned} \tag{44}$$

This concludes the process of finding the Lagrange multipliers λ_k .

3.4.2 Merit function

To determine whether the line search applied to (41) actually leads to an improvement, we use a merit function. According to [4], the line search can be seen as the merit function of the line search. In our case, the merit function is given by:

$$\phi(\mathbf{x}; \rho) = f(\mathbf{x}) + \rho (\|\nabla f(\mathbf{x})\| + \|g^+(\mathbf{x})\|) \text{ with } \rho > 0 \tag{45}$$

The term $\|g^+(\mathbf{x})\|$ represents the norm of the vector that contains all the values for which the constraint $g(\mathbf{x}) \leq 0$ is violated. Since this constraint is not important in our case, we will disregard this term.

3.4.3 Convergence criteria

The convergence criteria for the SLSQP method are as follows:

$$\|\nabla f(\mathbf{x}_k)\| < \mathbf{tol} \tag{46}$$

If the norm of the gradient is sufficiently small, we can conclude successful convergence. Another convergence criterion is based on the maximum absolute value of the constraint violation being smaller than a user-defined tolerance, which can be expressed as:

$$\|\nabla f(\mathbf{x}_k)\|_{\infty} < \mathbf{tol} \tag{47}$$

This criterion is more stringent than equation (46), so in our particular case, we can omit this constraint.

Another convergence criterion is based on the change in the objective function being significantly small, indicating successful convergence:

$$|f(\mathbf{x}_{k+1}) - f(\mathbf{x}_k)| < \mathbf{tol} \tag{48}$$

These are the convergence criteria used in the SLSQP method. Which are mentioned in [4] and [2].

3.5 Trust-Constraint

This method is a combination of methods, but in our case, we use the equality constrained part of this method. It is an implementation of the Byrd-Omojokun Trust-Region SQP method described in [6]. We will now discuss this method as described in [6]. The method is used to optimize equality constrained problems of the form:

$$\min_{\mathbf{x} \in \mathbb{R}^m} \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_K(\mathbf{x}) \end{bmatrix} = \min_{\mathbf{x} \in \mathbb{R}^m} f(\mathbf{x}) \text{ s.t. } \nabla f(\mathbf{x}) = \begin{bmatrix} \nabla f_1(\mathbf{x}) \\ \nabla f_2(\mathbf{x}) \\ \vdots \\ \nabla f_N(\mathbf{x}) \end{bmatrix} = \mathbf{0} \quad (49)$$

Notes: If second derivatives of f and h are not provided, BFGS or l -BFGS approximations are provided by the code.

We will first introduce some notation:

$$A(x) = [\nabla f_1(x), \nabla f_2(x), \dots, \nabla f_M(x)] \quad (50)$$

And note that the Lagrangian problem of (49) is: $L(x, \lambda) = f(x) - \lambda^T f(x)$, where λ is the vector of Lagrange multipliers.

Now we start explaining the algorithm. The algorithm starts by trying to solve for \mathbf{d} using \mathbf{x}_k , Δ_k (the trust radius), and λ_k :

$$\min_{\mathbf{d} \in \mathbb{R}^m} \mathbf{d}^T g_k + \frac{1}{2} \mathbf{d}^T \nabla^2 L(\mathbf{x}_k, \lambda_k) \mathbf{d} \quad (51)$$

$$\text{Subject to: } A_k^T \mathbf{d} + f(\mathbf{x}_k) = 0, \quad (52)$$

$$\|\mathbf{d}\| \leq \Delta_k \quad (53)$$

But our constraint (53) may prevent the algorithm from finding a proper solution to (52). Hence, to use the algorithm as described above, we need a relaxation parameter $\xi \in (0, 1)$ which we use to find \mathbf{v}_k , the vertical subproblem:

$$\min_{\mathbf{v} \in \mathbb{R}^m} \|A_k^T \mathbf{v} + \nabla f(\mathbf{x}_k)\| \quad (54)$$

$$\text{Subject to: } \|\mathbf{v}\| \leq \xi \Delta_k, \quad (55)$$

Although the above problem has a number of solutions, it can be shown that a solution \mathbf{v}_k can

be expressed as a linear combination of the columns of A_k , which allows us to decouple this sub-problem from the next one.

Using the \mathbf{v}_k we found above, we modify (52) as follows:

$$A_k^T \mathbf{d}_k = A_k^T \mathbf{v}_k \quad (56)$$

This modification ensures that the feasible region for \mathbf{d}_k is not empty, with $\mathbf{d}_k = \mathbf{v}_k$ as a valid solution.

Having formulated our algorithm, we proceed to determine \mathbf{d}_k using the following steps. This algorithm utilizes a complementary matrix Z_k to the matrix A_k such that $A_k^T Z_k = 0$. We define $\mathbf{d}_k = \mathbf{v}_k + Z_k \mathbf{u}_k$, where \mathbf{u}_k at iteration k is calculated by solving the following optimization problem:

$$\min_{\mathbf{u} \in \mathbb{R}^{M-m}} (g_k \nabla_x^2 L_k \mathbf{v}_k)^T Z_k \mathbf{u} + \frac{1}{2} \mathbf{u}^T Z_k^T \nabla_x^2 L_k Z_k \mathbf{u} \quad (57)$$

$$\text{Subject to: } \|Z_k \mathbf{u}\| \leq \sqrt{\Delta_k^2 - \|\mathbf{v}_k\|^2} \quad (58)$$

We then set

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{d}_k \quad (59)$$

3.5.1 Updating Lagrange Multipliers

At each iteration, we also update the Lagrange multipliers in an iterative way using the following equation:

$$(A_{k+1}^T A_{k+1}) \lambda_{k+1} = A_{k+1}^T \nabla f(\mathbf{x}_{k+1}) \quad (60)$$

This linear system is then solved using CG or sparse Cholesky. The accuracy of the first-order KKT measure $\|g_k A_k \lambda_k\|_\infty$ needs to be good enough. When CG is used, the residuals of (60) become smaller than $10^{-2} \cdot \max\{\|\nabla f(\mathbf{x})\|_\infty, \|g_k A_k \lambda_k\|_\infty\}$.

3.5.2 Merit Function

To determine whether a step \mathbf{d}_k makes a sufficient amount of progress, we use a merit function. The merit function used by the algorithm in [6] is:

$$\phi(d, \mu) = d^T \nabla f(\mathbf{x}_k) + \frac{1}{2} d^T \nabla_k^2 L(\mathbf{x}_k) d + \mu \|A_k^T d + \nabla f(\mathbf{x}_k)\| \quad (61)$$

Where $\mu > 0$ is called the penalty parameter.

Since we still need to choose how our penalty parameter μ is chosen we will set up a framework for this. This process start by setting up an alternative merit function in which we use replace f by the model objective as in (51) and linearize the constraints given in (61) this gives:

$$\hat{\phi}(\mathbf{d}_k, \mu) = \mathbf{d}_k^T g(\mathbf{x}_k) + \frac{1}{2} \mathbf{d}_k^T W_k \mathbf{d}_k + \mu \|A_k^T \mathbf{d}_k + \nabla f(\mathbf{x}_k)\| \quad (62)$$

Where W_k denotes $\nabla^2 L(\mathbf{x}_k, \lambda_k)$ or atleast an l -BFGS approximation to it. Now we define p_red as:

$$\begin{aligned} \text{p_red} &= \hat{\phi}(0, \mu) - \hat{\phi}(\mathbf{d}_k, \mu) \\ &= -\mathbf{d}_k^T g(\mathbf{x}_k) - \frac{1}{2} \mathbf{d}_k^T W_k \mathbf{d}_k + \mu (\|\nabla f(\mathbf{x}_k)\| - \|A_k^T \mathbf{d}_k + \nabla f(\mathbf{x}_k)\|) \end{aligned} \quad (63)$$

We now compute an trial value for μ which we call μ^+ :

$$\mu^+ = \max \left\{ \mathbf{u}_k, 0.1 + \frac{\mathbf{d}_k^T g(\mathbf{x}_k) + \frac{1}{2} \mathbf{d}_k^T W_k \mathbf{d}_k}{\|\nabla f(\mathbf{x}_k)\| - \|A_k^T \mathbf{d}_k + \nabla f(\mathbf{x}_k)\|} \right\} \quad (64)$$

The equation (64) cannot be used in general as the penalty parameter (which is the term of (63): $(\|\nabla f(\mathbf{x}_k)\| - \|A_k^T \mathbf{d}_k + \nabla f(\mathbf{x}_k)\|)$) can become positive when $\mathbf{v}_k = 0$ which implies that $\nabla f(\mathbf{x}_k) = 0$ in this case (64) cannot be used as p_red becomes positive from the decrease that \mathbf{d}_k makes in the horizontal subproblem (57)-(57).

Before accepting this as \mathbf{u}_{k+1} we check whether the actual reduction in merit function is sufficient using a_red :

$$\begin{aligned} \text{a_red} &= \phi(\mathbf{x}_k, \mu^+) - \phi(\mathbf{x}_k + \mathbf{d}_k, \mu^+) \\ &= f(\mathbf{x}_k) + f(\mathbf{x}_k + \mathbf{d}_k) + \mu^+ (\|\nabla f(\mathbf{x}_k)\| - \|\nabla f(\mathbf{x}_k + \mathbf{d}_k)\|) \end{aligned} \quad (65)$$

If $\text{a_red} \geq \eta \text{ p_red}$, then \mathbf{d}_k is accepted and \mathbf{u}_{k+1} is set to μ^+ . Otherwise some second-order procedure could be used, these are mentioned in (4.7) of [6].

Algorithm 2 Algorithm of trust-constr in our case (Byrd-Omojokun algorithm)

```
1:  $k \leftarrow 0$ 
2: Loop
3:   Compute  $f_k, \nabla f(\mathbf{x}_k), g_k, A_k$  and  $Z_k$ 
4:   Compute multipliers  $\lambda_k$ 
5:   if  $\|g_k - A_k \lambda_k\|_\infty < \text{tol}$  and  $\|\nabla f(\mathbf{x}_k)\|_\infty < \text{tol}$  , then stop
6:   Compute  $\mathbf{v}_k$  by solving the vertical sub-problem: (54)-(55)
7:   Compute  $\nabla^2 L_k(\mathbf{x}_k, \lambda_k)$  or update the  $l$ -BFGS approximation
8:   Compute  $\mathbf{u}_k$  by solving the horizontal sub-problem: (57)-(58)
9:   Set  $\mathbf{d}_k = \mathbf{v}_k + Z_k \mathbf{u}_k$ 
10:  Compute the actual reduction in the merit function  $a\_red$ 
11:  and the predicted reduction  $p\_red$ 
12:  if  $\frac{a\_red}{p\_red} > \eta$ 
13:    Then set  $\mathbf{x}_k = \mathbf{x}_k + \mathbf{d}_k$ ,  $\Delta_{k+1} \geq \Delta_k$ 
14:    Else set  $\mathbf{x}_{k+1} = \mathbf{x}_k$ ,  $\Delta_{k+1} \leq \|\mathbf{d}_k\|$ 
15:   $k \leftarrow k + 1$ 
```

Figure 6: General algorithm of trust constraint as given in [6]

Algorithm 3 Trust-region update algorithm

```
1: if  $\frac{a\_red}{p\_red} \leq 0.9$ 
2:   then  $\Delta_{k+1} = \max\{10\|\mathbf{d}_k\|, \Delta_k\}$ 
3: else if  $\frac{a\_red}{p\_red} \leq 0.3$ 
4:   then  $\Delta_{k+1} = \max\{2\|\mathbf{d}_k\|, \Delta_k\}$ 
5: Else
6:    $\Delta_{k+1} = \Delta_k$ 
```

Figure 7: a Trust region update formula from [6]

3.5.3 Algorithm

Constants $\text{tol} > 0$ and $\mu \in (0, 1)$ are given and choose \mathbf{x}_0 and $\Delta_0 > 0$ The algorithm above is a quick description of what we have talked about in this section.

Lastly we will discuss an effective algorithm for updating the trust region Δ : This algorithm is able to drastically improve convergence rates with more often than not little harm. Hence implementing algorithm is likely a good idea. Many more trust-region updates are possible but we refrain from talking about more of these algorithms here.

4 Code Overview

In this section, I will provide an overview of the work done for this project, focusing on the code and implementations.

4.1 Directory Structure

Understanding the file structure is crucial for comprehending the code. It facilitates easy access to the generated data. The following diagram illustrates the file structure:

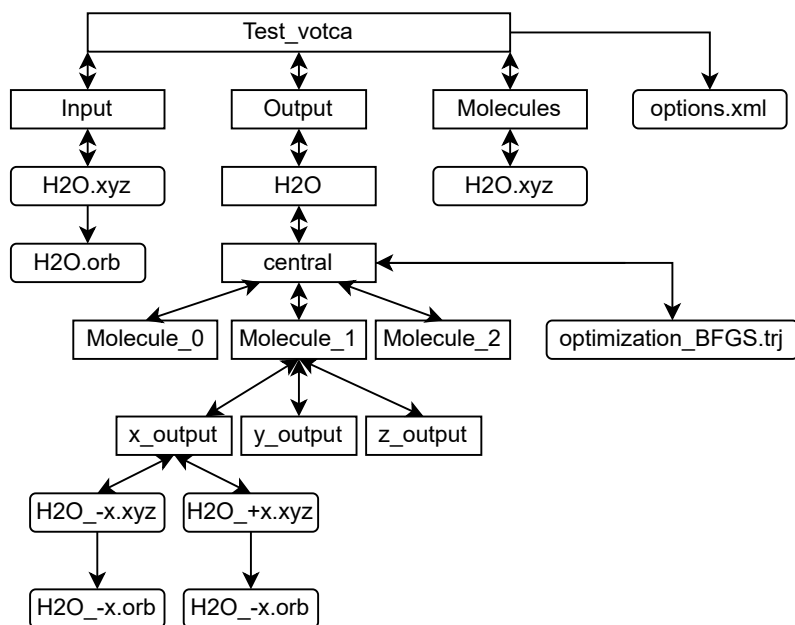


Figure 8: File structure diagram

In the diagram, square brackets represent directories, and rounded brackets represent files. The files generated by other files are denoted by a single arrow. For example, the files `x_output`, `y_output`, and `z_output` are generated inside the directories `molecule_0`, `molecule_1`, and `molecule_2`. Additionally, within each of the sub-directories `x_output`, `y_output`, and `z_output`, we have generated the $\pm x$, $\pm y$, and $\pm z$ files (assuming the method is central).

4.2 XYZ Files

Another essential aspect is the structure of an `.xyz` file, which follows this general format:

```

3
Water molecule
O 0.00000 0.00000 0.12957
H 0.00000 0.83199 -0.51828
H 0.00000 -0.83199 -0.51828
  
```

Figure 9: Example of an `.xyz` file structure

The first line denotes the number of atoms. The second line is a name given to the molecule. The subsequent lines provide the atom positions in Cartesian coordinates. Therefore, `.xyz` files can describe various types of molecules.

To generate the $\pm x$, $\pm y$, and $\pm z$ files, $\pm\delta$ is added to either the x , y , or z direction. For example, adding δ in the x -direction generates the $+x$ file, while adding $-\delta$ generates the $-x$ file.

4.3 Calculating Gradients

To calculate the gradients of the atoms, `xtp_tools` is used on the $\pm x$, $\pm y$, and $\pm z$ files to obtain the final energies for these configurations. The gradient calculation is performed as follows:

$$\nabla f = \begin{pmatrix} \frac{E_{\text{molecule}}(x+\delta) - E_{\text{molecule}}(x-\delta)}{2\delta} \\ \frac{E_{\text{molecule}}(y+\delta) - E_{\text{molecule}}(y-\delta)}{2\delta} \\ \frac{E_{\text{molecule}}(z+\delta) - E_{\text{molecule}}(z-\delta)}{2\delta} \end{pmatrix} = \begin{pmatrix} dx \\ dy \\ dz \end{pmatrix} \quad (66)$$

Now the gradients have been computed for a single iteration.

4.4 Running `xtp_tools`

To execute `xtp_tools`, a system call is made to the `xtp_tools` executable. Here is an example of the code:

```
command = ["xtp_tools -e dftgwbse -t {} -o options.xml -c job_name={}".format(num_threads, file)]
process = subprocess.Popen(command, cwd="test_votca", shell=True)
```

Figure 10: Example of a system call to `xtp_tools`

The command line arguments in more detail are as follows:

- `xtp_tools`: Specifies the use of `xtp_tools`.
- `-e dftgwbse`: Specifies the use of the `dftgwbse` calculator.
- `-t num_threads`: Specifies the number of threads used for calculations. Increasing this number can improve performance for large molecules.
- `-o options.xml`: Specifies the options file used by `xtp_tools`. Further details on the options will be discussed later.
- `-c job_name=file`: Specifies the file path for the desired input file. Note that the `.orb` file associated with the `xtp_tools` run is also located in the same directory as this file.

Once these calculations are executed and all the $\pm x$, $\pm y$, and $\pm z$ files are generated, we are ready to extract the final energies from the `.orb` files.

4.5 Extracting Energies from .orb Files

The .orb files are compressed HDF5 databases. To access them, they are loaded into an HDF5 database. The following code snippet demonstrates how to extract the `qm_energy`, which represents the total final energy calculated:

```
with h5py.File(file_path, "r") as f:
    group = f['QMdata']
    qm_energy_dict[i][key] = group.attrs['qm_energy'][0]
```

Figure 11: Code snippet to create the energy dictionary

The `qm_energy_dict` is the final piece of the puzzle for computing the gradients, as mentioned in Equation 66. The corresponding final energy is assigned to this key. For example, considering `H2O_+z.orb`, the extension is removed, and the final `+z` letters are used as the key. When calculating the gradients, the energy for the `+z` case can be retrieved using the `+z` key.

4.6 Updating Files

During the gradient calculation iterations, instead of saving all `H2O_ \pm direction.xyz` files, they are overwritten with the new ones required for the current iteration's gradient calculations. The same process is followed for the `H2O.orb` file generated in the "input" directory at each iteration. The same applies to all the $\pm x$, $\pm y$, and $\pm z$ files in each of the molecule directories.

4.7 Implementing `scipy.optimize.minimize`

Now the implementation of the `scipy.minimize` function will be explained:

```
result = scipy.optimize.minimize(energy, x0, jac=gradients, method=num_method, tol=tolerance)
```

Figure 12: call to `scipy.minimize`

The three main inputs that change in each iteration are `energy`, `x0`, and `gradients`. The value of `x0` is determined by the numerical method used, which will be discussed further in Section 3. The two aspects that we actively influence in each calculation are the energy and gradient calculations, as discussed above. However, there is a challenge: we have been working with .xyz files, but the `scipy.minimize` function only accepts arrays as input. To address this, we perform the following conversion:

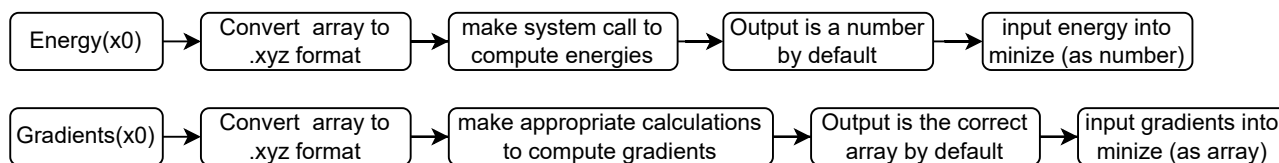


Figure 13: Example of energy and gradient calls

Essentially, we convert an .xyz file into an array of atom positions. In the array form, we lose some information, such as the number of atoms (although we will assume this for simplicity) and the molecule name. However, this information can easily be retrieved by referring to the input file corresponding to the calculation.

4.8 Output and Trajectory File

Another crucial aspect to consider is the output format. First we look at the terminal's output. The initial part of the output, structured by the code, appears as follows:

```
Molecule Name: H2O
Numerical method used: BFGS
Number of threads used: 6
Tolerance of the method used for termination constraints: 1e-07
BSE singlet energy added to total energy in each iteration: False

Starting optimization:
2023-06-14 21:15:44 - Iteration 0
Total energy: -76.36335878152408
Gradients: [[ 1.57029945e-09  1.27897692e-10  1.52547273e-01]
 [ 7.10542736e-12  1.09516851e-01 -7.62736504e-02]
 [-7.10542736e-12 -1.09516851e-01 -7.62736523e-02]]
Atom positions: [[ 0.      0.      0.12957]
 [ 0.      0.83199 -0.51828]
 [ 0.     -0.83199 -0.51828]]
```

Figure 14: First part of generated output

The important settings used are mentioned at the beginning of the output. The optimization process starts afterward. At each iteration, the time, total energy, gradients, and atom positions are displayed. If no gradients are calculated for a particular iteration (which can happen), or if the atom positions are not updated, it will be mentioned accordingly.

After the optimization process is completed, the following output is generated by `scipy.minimize`:

```
Total energy: -76.37736973661914
Message: Desired error not necessarily achieved due to precision loss.
Success: False
Status: 2
  Fun: -76.37736973661914
    x: [-2.502e-09  2.348e-09  8.621e-02  4.312e-09  7.649e-01
        -4.966e-01 -1.614e-09 -7.649e-01 -4.966e-01]
  Nit: 9
  Jac: [ 7.105e-12 -3.197e-10  3.800e-03 -7.105e-11  4.321e-03
        -1.900e-03  4.263e-11 -4.321e-03 -1.900e-03]
Hess_inv: [[ 1.000e+00  1.443e-12 ... -1.455e-07  4.223e-08]
 [ 1.443e-12  1.000e+00 ...  3.772e-07 -9.478e-08]
 ...
 [-1.455e-07  3.772e-07 ...  6.269e-01 -3.628e-02]
 [ 4.223e-08 -9.478e-08 ... -3.628e-02  8.445e-01]]
Nfev: 50
Njev: 41
```

Figure 15: Final part of output

Most of the output is self-explanatory, but there are a few details that require further explanation:

- **Success:** This indicates whether success was achieved, and the determination of success is method-specific.
- **Fun:** This is the best value achieved by the objective function.
- **Jac:** This is the estimate of the Jacobian, which is not used in our calculations.
- **Hess_inv:** This is the estimate of the inverse Hessian matrix.
- **Nfev:** This indicates the number of times the objective function was evaluated.
- **Njev:** This represents the number of iterations where the gradients were calculated. These iterations typically take longer than the **Nfev** iterations.

Another important part of the output is the trajectory file, which contains all the iterations of the .xyz file used during the optimization process. Therefore, the trajectory file is essentially a collection of Figure 9 files combined together.

5 Planarization of NH₃

In this section, we will explore an interesting phenomenon known as the planarization of NH₃ (ammonia). This phenomenon occurs when singlet excited state energies are incorporated into the total energies used for both gradient calculations and energy calculations. After implementing these singlet energies, we initiate the optimization process, and the planarization of NH₃ becomes evident quite quickly:

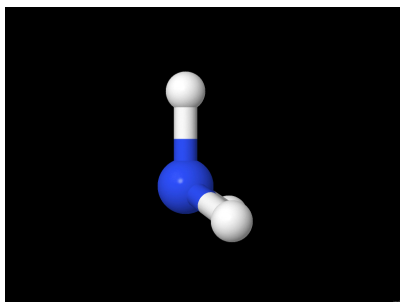


Figure 16: Regular NH₃ structure

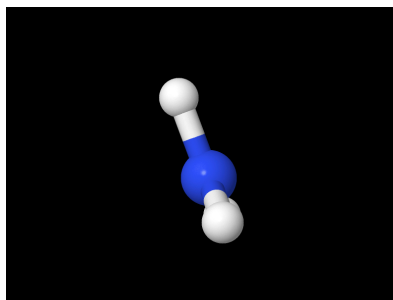


Figure 17: Optimized NH₃ structure with excited state energies

As observed, the NH₃ molecule appears to flatten in the optimized structure. This phenomenon is referred to as the planarization of NH₃/ammonia. It is an intriguing behavior, and we will now delve into explaining it.

To comprehend the planarization, we start by discussing the ground state structure of NH₃, where we will also visualize the lone pair of electrons:

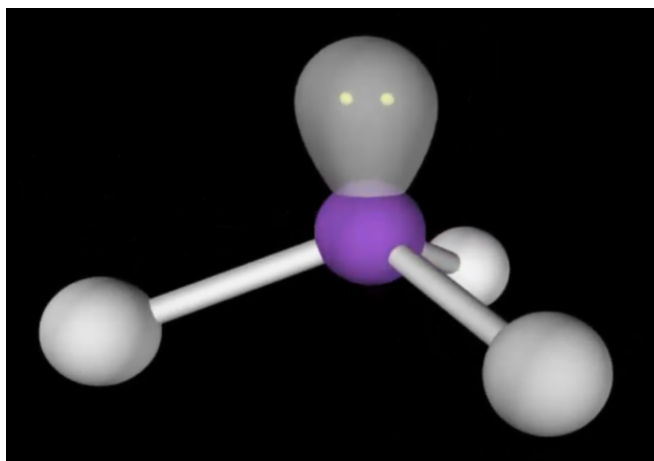


Figure 18: Ground state structure of NH₃

In an ideal scenario, the hydrogen atoms would prefer to be as far away from each other as possible. However, in the above structure, the hydrogen atoms are closer to each other than they ideally should be. This is attributed to the repulsion exerted by the lone pair of electrons, which pushes the hydrogen atoms away. At a certain point, the force exerted by the lone pair pushing the hydrogen atoms is counterbalanced by the natural repulsion between the hydrogen atoms, resulting in the structure depicted in Figure 18.

Now, when we consider singlet states of the molecule, by definition, all electrons are paired in bonds. Therefore, there is no force impeding the hydrogen atoms from moving as far away from each other as the bonds allow. This leads to the planar structure shown in Figure 16.

The occurrence of this planarization is an indication that my code optimizes structures in a correct manner.

6 Analysis of methods in practice

For the following analysis, we will use data obtained from calculations performed by my code to evaluate the advantages and disadvantages of different methods. We will begin by presenting a general table for comparison:

Molecule	method	Succes	nfev	njev	final energy (Hartree)	bondlength (angstrom)	angle (degrees)
CO2	BFGS	FALSE	22	11	-188,350686	1.37	180.0
CO2	CG	FALSE	38	26	-188,02072	1.74	180.0
CO2	SLSQP	TRUE	13	2	-118,350692*	1.37/1.773	180.0
CO2	trust-constr	TRUE	29	29	-188,427181	1.27	180.0
H2O	BFGS	FALSE	50	41	-76,377369	0.96	105.4
H2O	CG	FALSE	41	29	-76,376698	0.96	109.3
H2O	SLSQP	TRUE	17	5	-76,377382	0.96	105.1
H2O	trust-constr	TRUE	25	25	-76,375628	0.96	108.6
CO	BFGS	FALSE	47	35	-113,230708*	1.11	180.0
CO	CG	FALSE	127	115	-113,230708*	6.4	180.0
CO	SLSQP	TRUE	128	14	-113,230708*	9.7	180.0
CO	trust-constr	TRUE	26	26	-113,230708*	2.83	180.0
NH3	BFGS	FALSE	31	19	-56,511478	1.02	109.4
NH3	CG	FALSE	53	42	-56,513123	1.01	109.5
NH3	SLSQP	TRUE	49	22	-56,513123*	1.01	109.5
NH3	trust-constr	TRUE	31	31	-56,511216	1.0	109.5

Table 1: Table of final energy of methods. * denotes problem with convergence

Some important notes about this table:

The tested molecules are CO, CO2, NH3, and H2O. Final energies were computed using xtp-tools and optimized using `scipy.minimize` with the BFGS, CG, SLSQP, and trust-constraint methods. Each method provides a success indication and a status indication, which are method-specific parameters. nfev (in our case) denotes an iteration where the energy was recomputed, while njev (in our case) denotes an iteration where the gradient was recomputed for the modified molecule. On average, energy calculations take around 20 seconds, while gradient calculations take around 3 minutes.

The lowest final energies for each molecule are highlighted in bold in the table.

The bond lengths and angles for the molecules were measured as depicted in the following diagram:

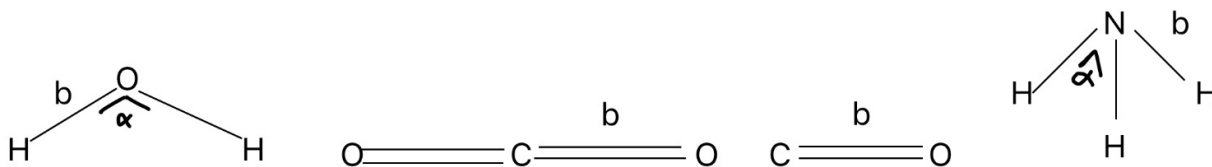


Figure 19: Figure showing how bond lengths and angles were measured

It is worth mentioning that the initial .xyz structures used were pre-optimized structures of H2O, CO, CO2, and NH3, with bond lengths increased by 10

Now, we will discuss some measures for evaluating the performance of the methods. One possible measure could be the number of times the lowest energy structure was found. However, no definitive conclusion can be drawn from this measure since SLSQP, CG, and trust-constraint methods each find the optimal structure once (excluding * calculations).

Another measure could be the average computation time. To assess this, we computed the average time for nfev iterations and njev computations. This provides an approximate computation time per method per molecule. The following table shows the average computation times, excluding * calculations:

BFGS avg	CG avg	SLSQP avg	trust-constr avg
5893	7993	2140	4480

Table 2: Average computation times sorted per method

From the table, it can be seen that SLSQP has the lowest average computation time. However, it should be noted that SLSQP has only one valid calculation for this average time computation. The second-best method in terms of average computation time is trust-constraint. Based on these performance measures, trust-constraint would be the recommended method. However, it is important to note that no definitive conclusions can be drawn based on this dataset.

Another measure that could be considered is the success rate provided by `scipy.minimize`. However, since the definition of success and convergence varies for each method, it is not used as a performance measure in this analysis.

6.1 H2O results

This section presents an analysis of the energy progress for each method used in order to gain a better understanding of their performance. We begin by examining a graph depicting the energy progress:

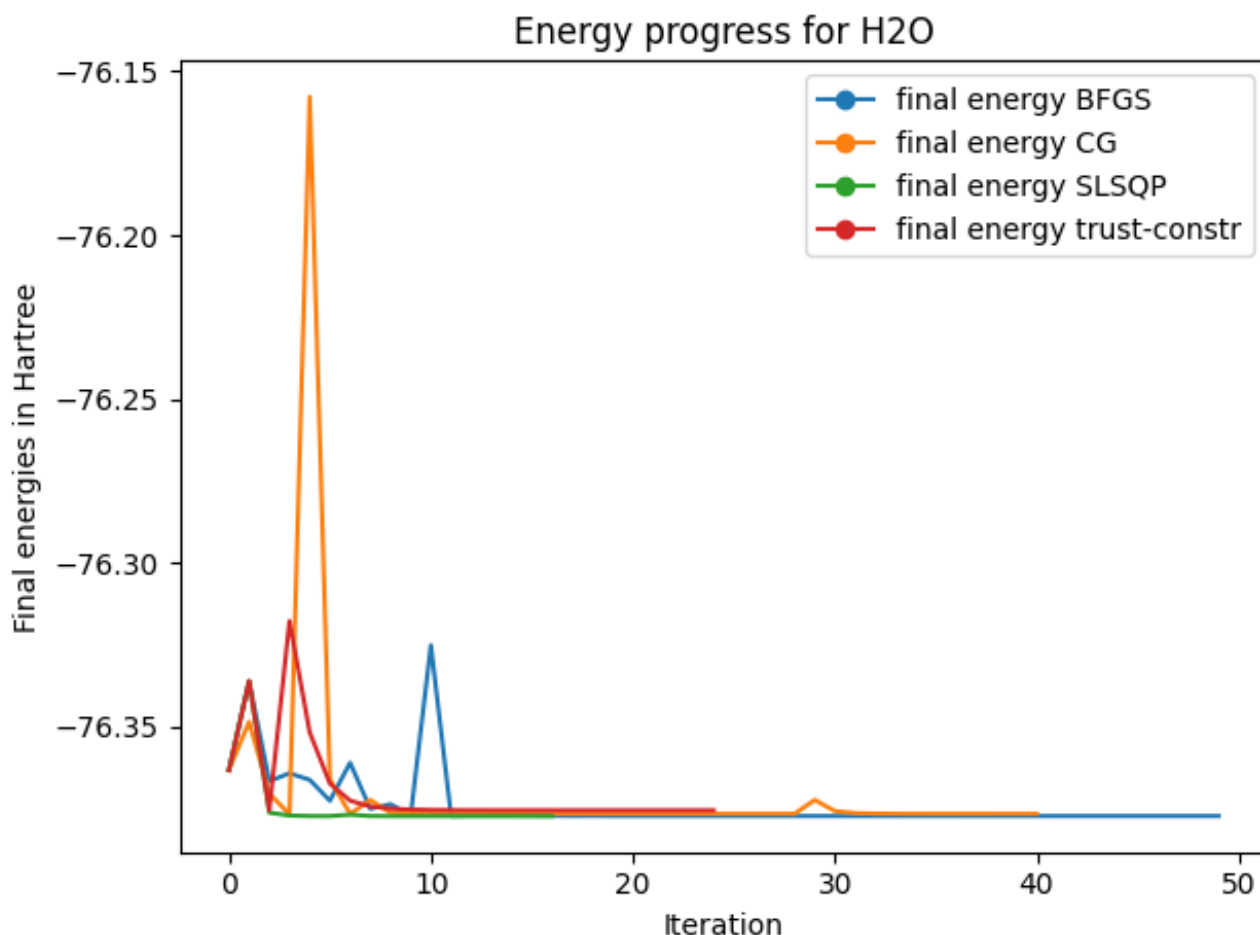


Figure 20: Energy for different iterations sorted by each method

As observed, all energies converge to a value close to -76.38, indicating that all methods reach a similar region. BFGS demonstrates a more aggressive behavior during the initial iterations, while CG shows fewer spikes in this particular case. SLSQP converges quickly with a low number of iterations, leading to the lowest final energy. Trust-constraint shows a few aggressive spikes initially but stabilizes and achieves successful convergence.

To provide further clarity on the energy progress, we include a graph showing the progression of the gradient

norms for each method:

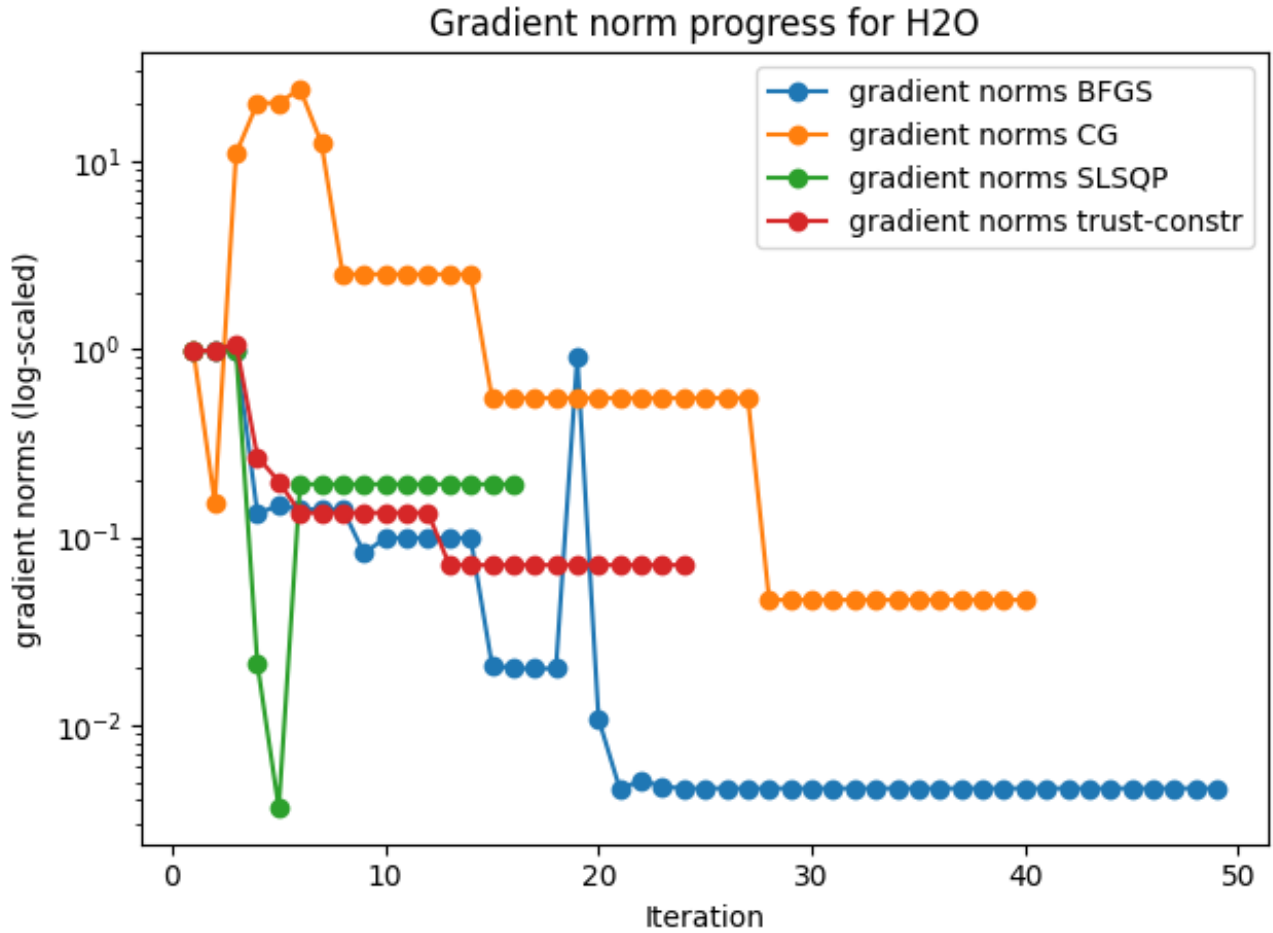


Figure 21: Gradient norms (log-scaled) for different iterations sorted by each method

Due to the presence of numerous outliers in the dataset, we have opted to utilize a logarithmic scale for the graph. This approach offers the advantage of visualizing outliers, although it can slightly complicate interpretation.

When comparing different methods, it is evident that the norms for CG consistently exceed those of any other method. However, this consistent disparity in norms does not appear to have a significant impact on the overall energy progress, although large norm changes do result in spikes in the energy for CG. For BFGS this does not seem to hold, the substantial spike just before iteration 20, does not seem to translate into a spike in the energy.

SLSQP consistently exhibits small norms, likely contributing to the rapid convergence. The correlation between small norms and an optimal structure suggests that these norms play a crucial role. In contrast, the norms for trust-constraint exhibit larger values, hovering in between 1 and 0.1. Interestingly, significant differences in norms do not appear to significantly influence the progression of energy.

6.2 CO2 results

Similar to the H2O results, we will discuss the CO2 results by examining the progress of the energy at each iteration for each method:

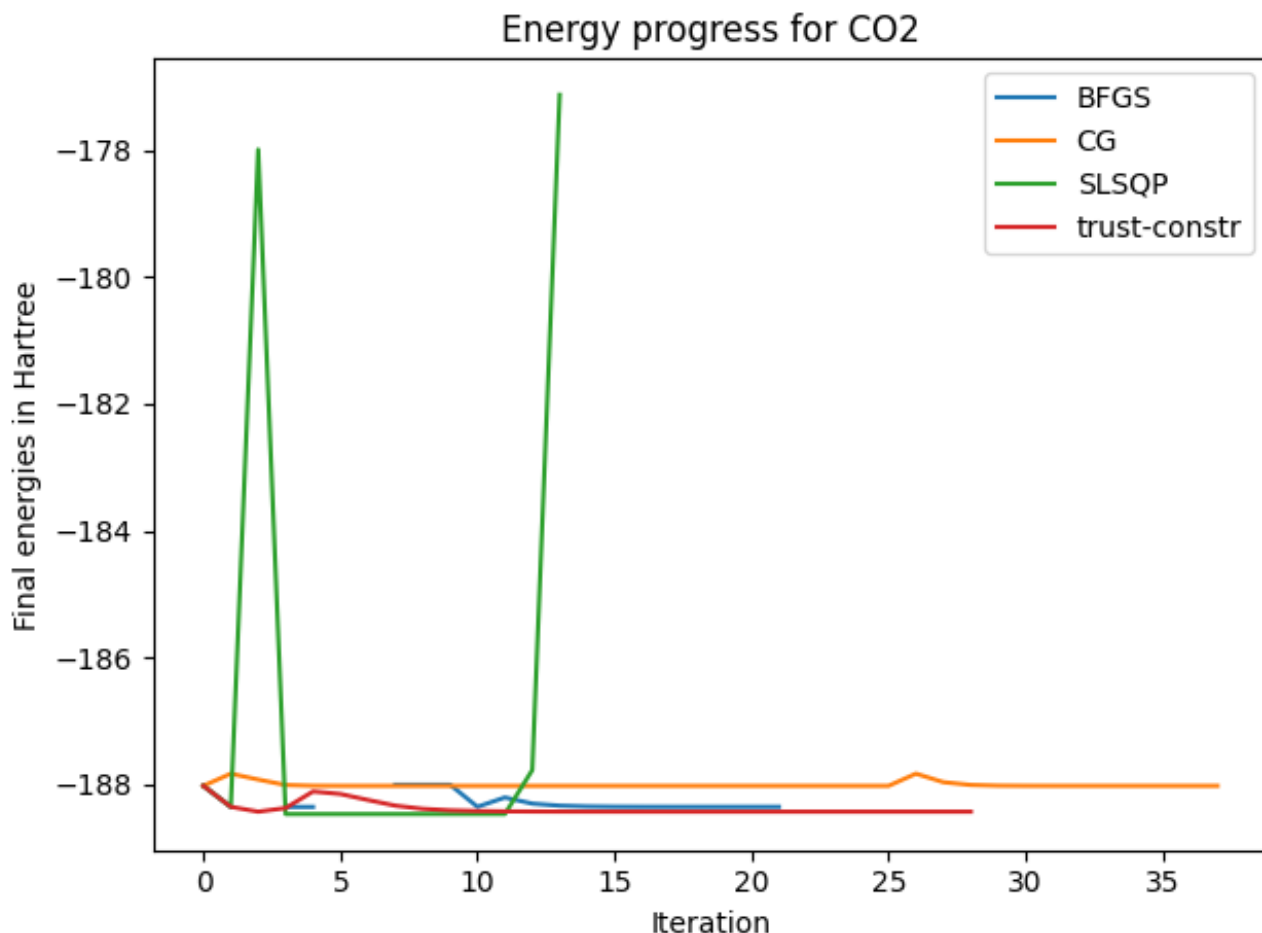


Figure 22: Energy for different iterations sorted by each method

In the analysis of CO2, we observe larger variations in energy compared to the analysis of H2O. Several notable observations can be made. Firstly, in the CG method, there is a consistent late spike in energy. Additionally, the graph for SLSQP appears to diverge towards the end. However, it is important to note that this divergence is a result of non-convergence issues, which will be discussed further in the upcoming section.

It is worth mentioning that the trust-constraint method demonstrates the most stable progression of energy, with a minimal number of spikes. This observation is consistent with what we observed in Figure 20.

Furthermore, it is important to acknowledge the presence of missing values in our analysis. These missing values are a result of extreme outliers (deviating by a factor of +10), which were intentionally removed to provide a clearer representation of the progression of the individual methods. At the end of the SLSQP-line, we also removed some even more extreme outliers (+20 in magnitude).

in the CO2 analysis, the graph below provides further insights:

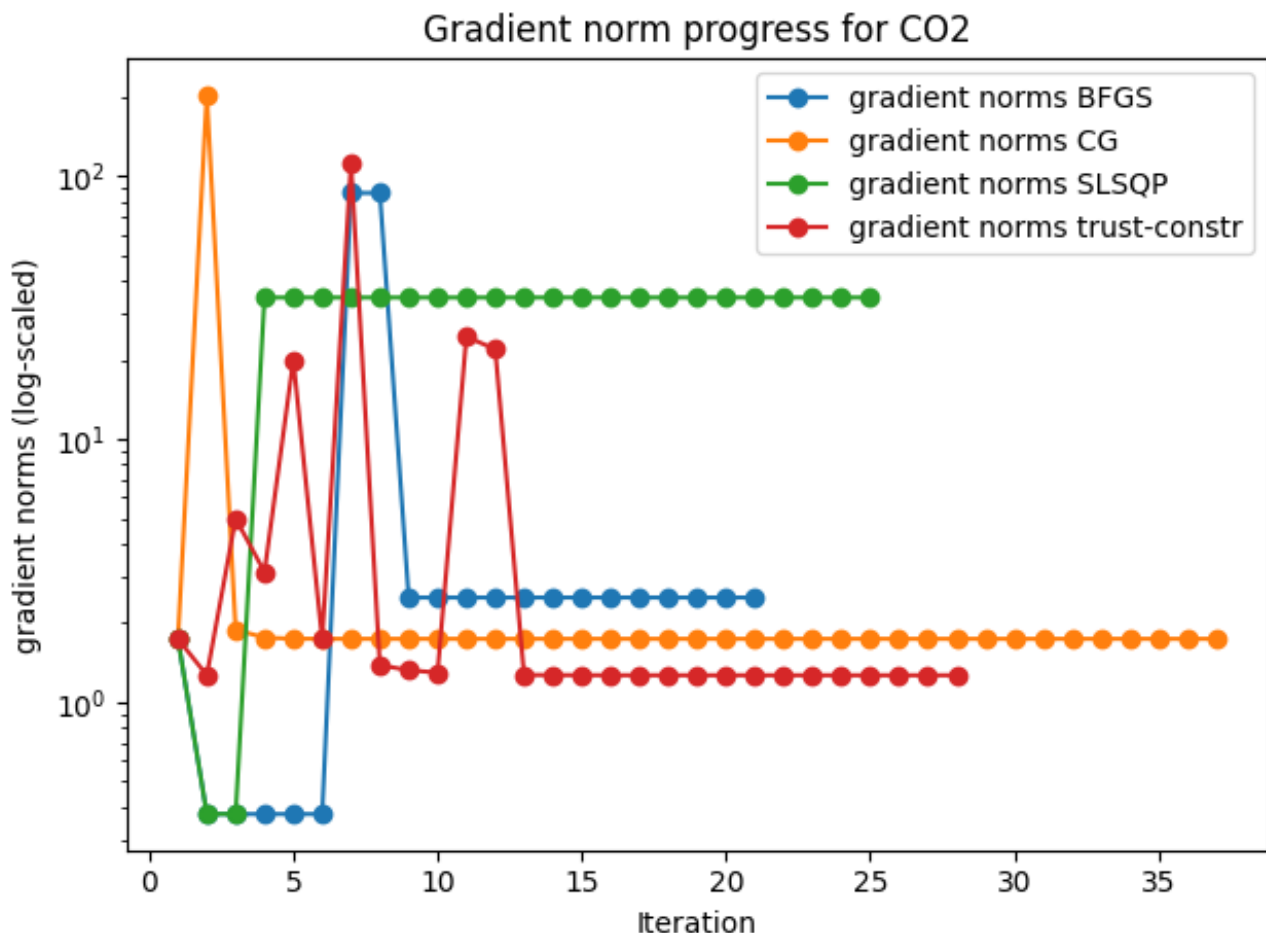


Figure 23: Gradient norms (log-scaled) for different iterations sorted by each method

First of all we note that there are far more outliers in the CO2 norms, ranging from 18 to 200. This suggests that gradient norms may not be the best indicator for analyzing energy progress. However, it is worth noting that lower gradients correspond to lower final energies, indicating a potential correlation between gradient norms and energy progress in the final iterations. Which might suggest that gradient norm is an indicator of minimal energy (which is also what we expect since $-\nabla E = F$).

Furthermore, it is evident from the graph that gradient norms do not approach zero. This behavior may be attributed to the precision of the gradient computations. Additionally, the displayed energies for the SLSQP methods are inaccurate, as will be explained in the following section.

6.3 Non-convergence problems

In this section, we will address a problem in the code and propose a possible solution. As discussed in subsection 4.5, the code currently uses xtp_tools to calculate energies and generate .orb files containing the final energy values. However, the code does not check whether xtp_tools successfully converges and generates a new .orb file. Consequently, if xtp_tools fails to converge, the last .orb file will not be replaced, and the code will not optimize such structures.

To address this issue, a mechanism should be implemented to check for successful convergence of xtp_tools. If non-convergence is detected, certain input parameters could be adjusted to potentially solve the problem. The following parameters are possible candidates for modification:

- Basisset and Auxbasisset in the options.xml file used as input for the xtp_tools calculations.
- Recalculating the gradient with a different delta to generate a new structure closer to convergence, thus increasing the chances of successful convergence by xtp_tools.

- Modifying the way `scipy.minimize` selects new structures during the optimization process.

These adjustments could potentially address the non-convergence problem and allow for optimization of structures that previously encountered convergence issues.

7 Conclusion

This section will be dedicated to answering the questions posed in the introduction as best as possible. First of all, are there numerical methods which seem to perform better than others? This question can be answered in a variety of ways. Firstly, note that there is a lack of sufficient data to draw a conclusive comparison between different methods, considering that almost half of the recorded data is invalid. One method, SLSQP, appears to be less reliable based on the collected data. Out of the four molecules tested with SLSQP, it only converged once. However, due to the small size of the data set, this result cannot rule out SLSQP from being an effective method. All other methods exhibit similar reliability based on this data.

The excited state optimization of NH₃ proceeded flawlessly for all methods. However, it is important to note the presence of code issues that are discussed in subsection 6.3. It is not possible to conclude that the success of NH₃ calculations implies the effectiveness of these calculations for all types of molecules.

Lastly, we will discuss the identification of optimal structures. All methods, except BFGS, were able to find optimal structures. Nonetheless, determining whether these "optimal" structures are close to the truly optimized structures is challenging.

References

- [1] Björn Baumeier, Nicolas Renaud, Wouter Scharpach, Vivek Sundaram, Onur Çaylak, Gianluca Tirimbò, Christoph Junghans, Joshua Brown, Felipe Zapata Ruiz, Jens Wehner, Ruben Gerritsen, Marvin Bernhardt, and David Rosenberger. VOTCA, 2021.
- [2] Paul T. Boggs and Jon W. Tolle. Sequential Quadratic Programming. *Acta Numerica*, 4:1–51, 1995.
- [3] Kevin Carlberg. Optimization in Python. 2019.
- [4] HE DAQUAN. Robust Aerodynamic Optimization through Conjugate Gradient Method with Taguchi’s Theory. Technical report, 2015.
- [5] Magnus R Hestenes and Eduard Stiefel. Methods of Conjugate Gradients for Solving Linear Systems 1. Technical Report 6, 1952.
- [6] Marucha Lalee, Jorge Nocedal, Todd Plantenga, and Siam J Optim. ON THE IMPLEMENTATION OF AN ALGORITHM FOR LARGE-SCALE EQUALITY CONSTRAINED OPTIMIZATION *. Technical Report 3, 1998.
- [7] Libretexts. Particle in a 1-Dimensional box.
- [8] Jorge Nocedal and Stephen J Wright. Numerical Optimization. 2006.
- [9] Jonathan Richard Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. 1994.
- [10] Jens Wehner, Lothar Brombacher, Joshua Brown, Christoph Junghans, Onur Çaylak, Yuriy Khalak, Pranav Madhikar, Gianluca Tirimbò, and Björn Baumeier. Electronic Excitations in Complex Molecular Environments: Many-Body Green’s Functions Theory in VOTCA-XTP. *Journal of Chemical Theory and Computation*, 14(12):6253–6268, 12 2018.

8 Appendix

8.1 Wolfe's condition

Wolfe's condition is primarily used to determine whether a line-search method provided a sufficient amount of objective function decrease. It can be measured using the following inequality:

$$f(\mathbf{x}_k + \alpha_k \mathbf{d}_k) \leq f(\mathbf{x}_k) + c_1 \alpha_k \nabla f(\mathbf{x}_k)^T \mathbf{d}_k \quad (67)$$

where c_1 is a constant in the range $(0, 1)$. This inequality ensures that the reduction in the objective function f is proportional to the step length α_k and the directional derivative $\nabla f(\mathbf{x}_k) \mathbf{d}_k$. The right-hand side of Equation 67 can be denoted by $l(\alpha_k)$, which represents a linear function. The slope of this linear function is $c_1 \nabla f(\mathbf{x}_k)^T \mathbf{d}_k$. We consider $\alpha_k > 0$ to be an appropriate value when Equation 67 is satisfied. In practice, a small value of c_1 , such as 10^{-4} , is commonly used. The default value in the `scipy.minimize` implementation is 10^{-4} (we used the default values in our calculations).

However, the condition Equation 67 alone is not sufficient to ensure that the algorithm makes significant progress. To address this, a second condition called the curvature condition is introduced:

$$\nabla f(\mathbf{x}_k + \alpha_k \mathbf{d}_k)^T \mathbf{d}_k \geq c_2 \nabla f(\mathbf{x}_k)^T \mathbf{d}_k \quad (68)$$

Here, c_2 is a constant chosen such that $c_1 < c_2 < 1$. The left-hand side of Equation 68 can be denoted as $\phi'(\alpha)$, representing the derivative of the left-hand side of Equation 67 (denoted as $\phi(\alpha)$). The curvature condition ensures that the slope of ϕ at α_k is greater than $c_2 \phi'(0)$.

The figure below illustrates the step lengths that satisfy Wolfe's condition:

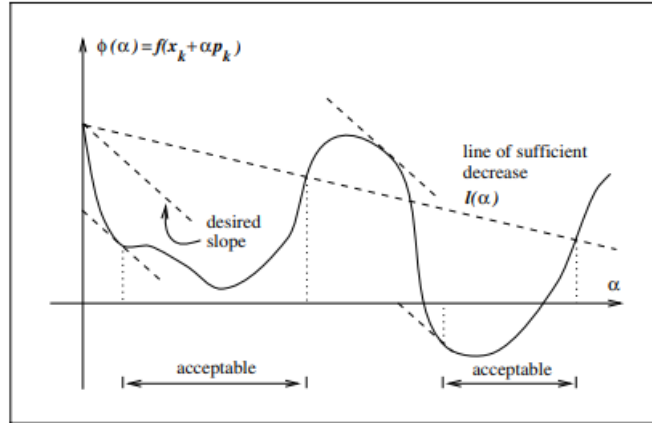


Figure 24: Step-lengths satisfying Wolfe's condition (picture from Nocedal and Wright, Numerical Optimization, 2006)

This visual representation provides a clearer understanding of what Wolfe's condition considers acceptable values for α_k . A strongly negative slope of $\phi'(\alpha_k)$ indicates that larger step sizes (α_k) can lead to improvement. On the other hand, when $\phi'(\alpha_k)$ approaches zero, it suggests that there is little room for improvement in the current search direction, and the line search may be terminated. The constant value c_2 typically has default values of 0.1 and 0.9, depending on the method used to determine the search direction. In summary, Wolfe's condition ensures the selection of an appropriate step size α_k for a corresponding line search with direction \mathbf{d}_k , requiring the satisfaction of the following two inequalities:

$$\begin{aligned} f(\mathbf{x}_k + \alpha_k \mathbf{d}_k) &\leq f(\mathbf{x}_k) + c_1 \alpha_k \nabla f(\mathbf{x}_k)^T \mathbf{d}_k \\ \nabla f(\mathbf{x}_k + \alpha_k \mathbf{d}_k)^T \mathbf{d}_k &\geq c_2 \nabla f(\mathbf{x}_k)^T \mathbf{d}_k \end{aligned} \quad (69)$$

8.2 Sherman–Morrison–Woodbury formula

The Sherman–Morrison–Woodbury formula, described by Bindel and Demmel in "Matrix computations" (2009), provides a solution to the equation $A + \mathbf{u}\mathbf{v}^T$ when a factorization of A already exists.

The formula is derived using block Gaussian elimination. To compute $(A + \mathbf{u}\mathbf{v}^T)\mathbf{x}$, we typically first compute $\xi = \mathbf{v}^T\mathbf{x}$, and then $(A + \mathbf{u}\mathbf{v}^T)\mathbf{x} = A\mathbf{x} + \mathbf{u}\xi$. Therefore, to solve the equation $(A + \mathbf{u}\mathbf{v}^T)\mathbf{x} = \mathbf{b}$, we can express it as an extended system:

$$\begin{pmatrix} A & \mathbf{u} \\ \mathbf{v}^T & -1 \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \xi \end{pmatrix} = \begin{pmatrix} \mathbf{b} \\ 0 \end{pmatrix} \quad (70)$$

By factorizing A , we obtain:

$$\begin{pmatrix} A & \mathbf{u} \\ \mathbf{v}^T & -1 \end{pmatrix} = \begin{pmatrix} \mathcal{I} & 0 \\ \mathbf{v}^T A^{-1} & 1 \end{pmatrix} \begin{pmatrix} A & \mathbf{u} \\ 0 & -1 - \mathbf{v}^T A^{-1} \mathbf{u} \end{pmatrix} \quad (71)$$

We apply forward substitution with the block lower triangular factor:

$$\begin{aligned} \mathbf{y} &= \mathbf{b} \\ \eta &= -\mathbf{v}^T A^{-1} \mathbf{y} \end{aligned} \quad (72)$$

Then, we apply backward substitution with the block upper triangular factor:

$$\begin{aligned} \xi &= (-1 - \mathbf{v}^T A^{-1} \mathbf{u})^{-1} \eta \\ \mathbf{x} &= A^{-1}(\mathbf{y} - \mathbf{u}\xi) \end{aligned}$$

Combining all the algebraic steps, we find:

$$\mathbf{x} = \left[A^{-1} - \frac{A^{-1} \mathbf{u} \mathbf{v}^T A^{-1}}{1 + \mathbf{v}^T A^{-1} \mathbf{u}} \right] \mathbf{b} \quad (73)$$

This formula is known as the Sherman–Morrison formula.

The generalization of this formula to the rank- n case is called the Sherman–Morrison–Woodbury formula and is described as follows:

$$(A + UV^T)^{-1} = A^{-1} - A^{-1}U(\mathcal{I} + V^T A^{-1}U)^{-1}V^T A^{-1} \quad (74)$$

8.3 Code

```
import os
import h5py
import numpy as np
import subprocess
import xml.etree.ElementTree as ET
from time import time
from typing import List, Tuple
import scipy
import openpyxl
import argparse
import datetime

# create a timer decorator in order to log execution times:
def timer_func(func):
    def wrap_func(*args, **kwargs):
        t1 = time()
        result = func(*args, **kwargs)
        t2 = time()
        print(f'Function {func.__name__!r} executed in {(t2-t1):.4f}s')
```



```

        return result
    return wrap_func

class XYZFile:
    """
    This XYZFile class is used in combination with XYZFileManager
    in order to read and modify .xyz sub-folder files (possibly other filetypes)
    and output the read/modified files into an possibly different sub folder.

    This XYZFile class deals with reading and modifying the files.
    """
    def __init__(self, file_path: str):
        self.file_path = file_path
        self.atom_list = []

        with open(file_path) as f:
            self.num_atoms = int(f.readline().strip())
            self.comment = f.readline().strip()
            for line in f:
                atom_data = line.split()
                self.atom_list.append((atom_data[0], float(atom_data[1]), float(atom_data[2]),
                                       float(atom_data[3])))

    def modify_coordinates(self, direction: str, delta: float, atom_number: int):
        """
        This function is able change the individual atom positions within the molecule in which
        we change the atoms position
        in a 'x', 'y' or 'z' direction.
        In order to change this position we must first choose a atom in which we want to change
        the x,y or z directions
        this is done using the atom_number parameter in which we choose an integer 0,1,..., n-1
        where the molecule has n atoms.
        this numbering is done by the .xyz file structure hence the first atom we find in this .
        xyz file is atom 0.
        """
        for i, atom in enumerate(self.atom_list):
            if i == atom_number:
                if direction == 'x':
                    self.atom_list[i] = (atom[0], atom[1]+delta, atom[2], atom[3])
                elif direction == 'y':
                    self.atom_list[i] = (atom[0], atom[1], atom[2]+delta, atom[3])
                elif direction == 'z':
                    self.atom_list[i] = (atom[0], atom[1], atom[2], atom[3]+delta)

    def write_xyz_file(self, output_path: str, name_suffix=None):
        """
        This function is used to write a (modified) file into a new file
        with a new position and a possibly new name (suffix) in the later created folder
        structure.
        """
        if name_suffix is not None:
            file_name, file_ext = os.path.splitext(os.path.basename(output_path))
            output_path = os.path.join(os.path.dirname(output_path), f"{file_name}_{name_suffix}{file_ext}")

        with open(output_path, 'w') as f:
            f.write(str(self.num_atoms) + '\n')
            f.write(self.comment + '\n')
            for atom in self.atom_list:
                f.write('{} {:.12.6f} {:.12.6f} {:.12.6f}\n'.format(atom[0], atom[1], atom[2], atom[3]))

class XYZFileManager:
    """
    The XYZFileManager class is used to link sub(-sub) folders to each other at which point we
    can
    output possibly modified .xyz files (also possibly other types with some modifications) into
    the output_file path.

    Which is done in order to then run possible other calculations which in our case means
    running xtp tools using
    system calls.
    """
    def __init__(self, input_dir: str, output_dir: str):
        self.input_dir = input_dir

```

```

self.output_dir = output_dir

def get_xyz_files(self):
    """
    This function is able to get all .xyz files in a specified folder it also finds possible
    .xyz files in sub(-sub) folders.
    """
    xyz_files = []
    for root, dirs, files in os.walk(self.input_dir):
        for file in files:
            if file.endswith('.xyz'):
                xyz_files.append(os.path.join(root, file))
    return xyz_files

def modify_xyz_file(self, direction: str, delta: float, atom_number: int, input_file,
                    name_suffix=None):
    """
    This function uses the .modify_coordinates and .write_xyz_file in order to modify and
    write the
    changed .xyz file into a new position which is specified by the output_path parameter
    during the initialization of the
    XYZFileManager class.

    For information about the input of the parameters: direction, delta, atom_number,
    name_suffix go to
    the XYZFile class and look at .modify_coordinates and .write_xyz_file
    """
    self.input_file = input_file

    file = XYZFile(self.input_file)
    file.modify_coordinates(direction, delta, atom_number)
    output_path = os.path.join(self.output_dir, os.path.relpath(self.input_file, self.
                                                                    input_dir))
    file.write_xyz_file(output_path, name_suffix=name_suffix)

class ComputeGradients:
    """
    Compute an numerical gradient of an input molecule using a delta change in x,y and z
    direction
    """
    def __init__(self, input_file: str, output_dir: str, delta: float, method: str):
        self.input_file = input_file
        self.output_dir = output_dir
        self.input_dir = os.path.dirname(self.input_file)
        self.delta = delta
        self.method = method

        # forward, backward or central finite difference:
        if self.method in ['forward', 'backward', 'central']:
            pass
        else:
            raise ValueError('method is not valid')

        # find file name and number of atoms in order to generate folder structure:
        self.molecule_name = os.path.splitext(os.path.basename(self.input_file))[0]
        self.XYZ = XYZFile(self.input_file)
        self.num_atoms = self.XYZ.num_atoms

    def generate_calculation_files(self):
        """
        This function is able to calculate the energy gradients based on small changes in atom
        positions x,y or z
        of a molecule.
        """
        # We will start by generating the file structure in which we will place the files
        # which we want to generate in order to use xtp_tools.

        # an if statement is necessary since otherwise an error is raised that the directories
        # already exist
        if os.path.exists(self.output_dir+'/{}/{}'.format(self.molecule_name, self.method)):
            pass
        else:
            print("creating necessary subfolders")

            # change directory to the one which we want to add files
            os.chdir(self.output_dir)

```

```

# create the main directory for the molecule
os.makedirs("{}".format(self.molecule_name), exist_ok=True)
os.makedirs("{} / {}".format(self.molecule_name, self.method), exist_ok=True) # could be
                                         made for each delta by adding (+
                                         delta')!

output_path = "{} / {}".format(self.molecule_name, self.method)
# create subfolders:
for i in range(0, self.num_atoms):
    os.makedirs("{} / {} / molecule_{}".format(self.molecule_name, self.method, i),
                exist_ok=True)

    # create x_output, y_output and z_output directories for the molecule_i
    # directories
    for direction in ["x", "y", "z"]:
        os.makedirs("{} / {} / molecule_{} / {}".format(self.molecule_name, self.
                                                         method, i, direction),
                    exist_ok=True)

# -----

# generate /pm delta files in each direction for each molecule:
for i in range(0, self.num_atoms):
    manager0 = XYZFileManager(self.input_dir, self.output_dir + "{} / {} / molecule_{} / x_output
                              ".format(self.molecule_name, self.
                                         method, i))
    manager1 = XYZFileManager(self.input_dir, self.output_dir + "{} / {} / molecule_{} / y_output
                              ".format(self.molecule_name, self.
                                         method, i))
    manager2 = XYZFileManager(self.input_dir, self.output_dir + "{} / {} / molecule_{} / z_output
                              ".format(self.molecule_name, self.
                                         method, i))

    if self.method in ['backward', 'central']:
        manager0.modify_xyz_file('x', -self.delta, i, self.input_file, '-x')
        manager1.modify_xyz_file('y', -self.delta, i, self.input_file, '-y')
        manager2.modify_xyz_file('z', -self.delta, i, self.input_file, '-z')

    if self.method in ['forward', 'central']:
        manager0.modify_xyz_file('x', self.delta, i, self.input_file, '+x')
        manager1.modify_xyz_file('y', self.delta, i, self.input_file, '+y')
        manager2.modify_xyz_file('z', self.delta, i, self.input_file, '+z')

# make seperate case for 'backward' and 'forward' methods.
# I chose to put this original file in the x_output map arbitrarily
if self.method in ['backward', 'forward']:
    manager0.modify_xyz_file('x', 0, 0, self.input_file, 'og')

def get_gradients_xyz(self, input_file): #, input_file: str ,output_dir: str):
    """
    Should read .orb output files and be able to assign \pm direction properly to a variable
    which we can then use to calculate \
    nabla E

    """
    global num_threads
    self.input_file = input_file

    # find .xyz files for each molecule:
    new_xyz_files = []
    for i in range(self.num_atoms):
        manager = XYZFileManager(self.output_dir + "{} / {} / molecule_{}".format(self.
                                           molecule_name, self.method, i), "")
        new_xyz_files.append(manager.get_xyz_files())

    # removes '.xyz' extensions and 'test_votca/' prefix from new_xyz_files for input into
    # command/xtp_tools

    for i in range(self.num_atoms):
        j = 0
        for file in new_xyz_files[i]:
            file = os.path.splitext(file)[0]
            new_xyz_files[i][j] = file.removeprefix("test_votca/")
            j += 1

    # run xtp tools on all files on all files in new_xyz_files
    for i in range(self.num_atoms):
        for file in new_xyz_files[i]:
            command = ["xtp_tools -e dftgwbose -t {} -o options.xml -c job_name={}".format(
                num_threads, file)]

```

```

process = subprocess.Popen(command, cwd = "test_votca", shell = True, stdout=
                           subprocess.DEVNULL, stderr=
                           subprocess.DEVNULL)

process.wait()

# read out input file in order to gain information about number of atoms:
molecule_name = os.path.splitext(os.path.basename(self.input_file))[0]
XYZ = XYZFile(self.input_file)
num_atoms = XYZ.num_atoms

# find .orb files for each molecule:
output_file_paths = [[] for _ in range(self.num_atoms)]

for i in range(self.num_atoms):
    for dirpath, dirnames, filenames in os.walk("test_votca/output"+"/{}/-/{}/molecule_{}".
                                                format(self.molecule_name, self.
                                                        method, i)):
        for filename in filenames:
            if filename.endswith('.orb'):
                fullpath = os.path.join(dirpath, filename)
                output_file_paths[i].append(fullpath)

qm_energy_dict = {}
for i in range(self.num_atoms):
    qm_energy_dict[i] = {}
    for j, file_path in enumerate(output_file_paths[i]):
        key = file_path[-6:-4]
        with h5py.File(file_path, "r") as f:
            group = f['QMdata']
            if add_bse:
                bse_singlet = group['BSE_singlet']
                eigenvalues_data = bse_singlet['eigenvalues']
                eigenvalue = eigenvalues_data[()][0]
                qm_energy_dict[i][key] = group.attrs['qm_energy'][0] + eigenvalue[0]
            else:
                qm_energy_dict[i][key] = group.attrs['qm_energy'][0]

gradientE = []
px = []
py = []
pz = []
mx = []
my = []
mz = []

if self.method in ['forward', 'central']:
    # This if statement is necessary since otherwise i get an '+x' key error
    if '+x' in qm_energy_dict[1]:
        for i in range(self.num_atoms):
            px.append(qm_energy_dict[i]['+x'])
            py.append(qm_energy_dict[i]['+y'])
            pz.append(qm_energy_dict[i]['+z'])

if self.method in ['backward', 'central']:
    if '-x' in qm_energy_dict[1]:
        for i in range(self.num_atoms):
            mx.append(qm_energy_dict[i]['-x'])
            my.append(qm_energy_dict[i]['-y'])
            mz.append(qm_energy_dict[i]['-z'])

if self.method in ['backward', 'forward']:
    og = qm_energy_dict[3]['og']

gradients = np.zeros((0, 3))

if self.method in ['backward']:
    for i in range(self.num_atoms):
        gradx = (og-mx[i])/self.delta
        grady = (og-my[i])/self.delta
        gradz = (og-mz[i])/self.delta
        gradients = np.vstack((gradients, np.array([gradx, grady, gradz])))
if self.method in ['central']:
    for i in range(self.num_atoms):
        gradx = (px[i]-mx[i])/(2*self.delta)

```

```

        grady = (py[i]-my[i])/(2*self.delta)
        gradz = (pz[i]-mz[i])/(2*self.delta)
        gradients = np.vstack((gradients,np.array([gradx,grady,gradz])))

    if self.method in ['forward']:
        for i in range(self.num_atoms):
            gradx = (px[i]-og)/(self.delta)
            grady = (py[i]-og)/(self.delta)
            gradz = (pz[i]-og)/(self.delta)
            gradients = np.vstack((gradients,np.array([gradx,grady,gradz])))

    return gradients

def get_array_from_xyz(self, input_file):
    self.input_file = input_file
    self.array = np.zeros((0, 3))

    with open(input_file) as f:
        self.num_atoms = int(f.readline().strip())
        self.comment = f.readline().strip()
        for line in f:
            atom_data = line.split()
            self.array = np.vstack((self.array,[float(atom_data[1]), float(atom_data[2]),
                                                    float(atom_data[3])]))

    return self.array

def get_xyz_from_array(self, array, output_subfolder=None):
    """
    Takes as input an input file path (input_file), an array of coordinates which we want to
    place into the input_file and an
    location output_subfolder.

    In which the updated .xyz file is placed.
    """

    self.array = array
    self.output_subfolder = output_subfolder
    if self.output_subfolder == None:
        self.output_subfolder = "test_votca/input"
    # Extract the directory and filename from the input path
    input_directory = os.path.dirname(self.input_file)
    input_file_name = os.path.basename(self.input_file)

    # Read the existing XYZ file
    with open(self.input_file, 'r') as f:
        lines = f.readlines()

    # Update the atom coordinates
    num_atoms = int(lines[0])
    if num_atoms*3 != self.array.shape[0]:
        print("Number of atoms in the input file and array do not match!, {}, {}".format(
            num_atoms, self.array.shape[0]))

        return

    j = 0
    for i in range(num_atoms):
        lines[i + 2] = f"{lines[i + 2].split()[0]} {self.array[3*j]:.6f} {self.array[1+3*j]:.
        6f} {self.array[2+3*j]:.6f}\n"

        j += 1

    # Create the output subfolder if it doesn't exist
    output_directory = self.output_subfolder
    os.makedirs(output_directory, exist_ok=True)
    # Construct the output file path within the subfolder
    output_xyz_path = os.path.join(output_directory, input_file_name)

    # Write the modified XYZ file
    with open(output_xyz_path, 'w') as f:
        f.writelines(lines)

def get_gradients_array(self, array):
    """
    Takes an array as input and outputs gradients of the atoms which
    are represented by this array (note which atoms is dependent on the input_file)
    """

    self.array = array
    xyz_file = self.get_xyz_from_array(self.input_file)

```

```

        gradients = self.get_gradients_xyz(xyz_file)

    return gradients

def on_modified(self, event):
    if not event.is_directory and event.src_path == self.source_file:
        print(f"File '{self.source_file}' modified. Copying to '{self.destination_folder}'")
        shutil.copy2(self.source_file, self.destination_folder)

counter = 0
def compute_energy(self, array):
    """
    Computes the energy of an molecule with given array input
    """
    global add_bse
    global num_threads

    self.generate_calculation_files()

    self.array = array
    self.get_xyz_from_array(self.array)

    calculations_directory = "test_votca/input"

    # make distinction since energy is plainly calculated in the 'forward'/'backward' case
    if self.method == 'central':
        command = ["xtp_tools -e dftgwbse -t {} -o options.xml -c job_name=input/{}".format(
            num_threads, self.molecule_name)]
        process = subprocess.Popen(command, cwd = "test_votca", shell = True, stdout=
            subprocess.DEVNULL, stderr=
            subprocess.DEVNULL)

        process.wait()
        with h5py.File(calculations_directory+"/{}.orb".format(self.molecule_name).format(self.
            molecule_name, self.method, self.
            molecule_name), "r") as f:

            group = f['QMdata']
            if add_bse:
                bse_singlet = group['BSE_singlet']
                eigenvalues_data= bse_singlet['eigenvalues']
                eigenvalue = eigenvalues_data[()][0]
                tot_energy = group.attrs['qm_energy'][0] + eigenvalue[0]
                print(group.attrs['qm_energy'][0], eigenvalue[0])
            else:
                tot_energy = group.attrs['qm_energy'][0]

    return tot_energy

def compute_gradients(self, array):
    """
    Combines methods in class in order to get calculations.
    """
    self.array = self.get_gradients_xyz(self.input_file)
    return self.array

counter = 0
def energy(array):
    global molecule_name
    global method
    global delta
    global counter
    global num_method
    if add_bse:
        opt_file_path = "test_votca/output/{}/{} optimization_{}_BSE.trj".format(molecule_name,
            method, num_method)
    else:
        opt_file_path = "test_votca/output/{}/{} optimization_{}.trj".format(molecule_name,
            method, num_method)

    file_path = "test_votca/input/{}.xyz".format(molecule_name)
    if counter > 0:
        print("\n", flush = True)
    else:
        print("\n", flush= True)
        print("starting optimization :", flush = True)
        # empty trajectory file:
        command = ["rm {}".format(opt_file_path)]

```

```

        empty_trj_file = subprocess.Popen(command, shell=True, stdout=subprocess.DEVNULL, stderr=
                                         subprocess.DEVNULL)

        empty_trj_file.wait()
        # Get the current date and time
        current_datetime = datetime.datetime.now()
        # Format the date and time as a string
        formatted_datetime = current_datetime.strftime("%Y-%m-%d %H:%M:%S")
        print("{} - iteration {}".format(formatted_datetime, counter), flush = True)
        counter += 1

    energy = ComputeGradients("test_votca/input/{}.xyz".format(molecule_name),"test_votca/output"
                              ,delta,method)

    energy = energy.compute_energy(array)
    print("total energy : {}".format(energy), flush = True)

    # the next lines are used to write a trajectory for the optimization.
    with open(file_path, 'r') as file:
        xyz_content = file.read()
    with open(opt_file_path, 'a') as opt_file:
        opt_file.write(xyz_content)

    return energy

def gradients(array):
    global delta
    global method
    global molecule_name
    gradients = ComputeGradients("test_votca/input/{}.xyz".format(molecule_name),"test_votca/
                                output",delta ,method)

    gradients = gradients.compute_gradients(array)
    print("gradients : {}".format(gradients), flush = True)
    gradients = gradients.flatten()
    print("atom positions : {}".format(array.reshape(-1,3)), flush = True)
    return gradients

x0 = np.zeros((0, 3))
x0 = np.vstack((x0,[0,0,0.11779]))
x0 = np.vstack((x0,[0,0.75545,-0.47116]))
x0 = np.vstack((x0,[0,-0.75545,0.47116]))
x0 = x0.flatten()

# Create an argument parser
parser = argparse.ArgumentParser(description='Script description')

# Add arguments to the parser
parser.add_argument('molecule_name', type=str, help='Name of the molecule')
parser.add_argument('num_method', type=str, help='Name of the numerical method')
parser.add_argument('num_threads', type=int, help='Number of threads to use')
parser.add_argument('tolerance', type=float, help='accepted tolerances in methods')
parser.add_argument('-add_bse', action='store_true', help='Flag to add BSE singlet energy')

# Parse the command-line arguments
args = parser.parse_args()

# Access the arguments
molecule_name = args.molecule_name
num_method = args.num_method
num_threads = args.num_threads
tolerance = args.tolerance
add_bse = args.add_bse

print('Molecule Name:', molecule_name)
print('Numerical method used:', num_method)
print('Number of threads used:', num_threads)
print('Tolerance of the method used for termination constraints:', tolerance)
print('BSE singlet energy added to total energy in each iteration:', add_bse)

method = "central"
delta = 0.001
getarray = ComputeGradients("test_votca/molecules/{}.xyz".format(molecule_name),"test_votca/
                             output",delta ,method)
x0 = getarray.get_array_from_xyz("test_votca/molecules/{}.xyz".format(molecule_name))
x0 = x0.flatten()
result = scipy.optimize.minimize(energy, x0, jac=gradients, method=num_method, tol=tolerance)
# Print the result
print(result, flush = True)

```

```

# flush in the print statements makes it so that output is shown in terminal

input_dir = "test_votca/input"
output_dir = "test_votca/output"

# this function is able to change the options.xml file within python
def edit_xml(file_path: str, job_name= None, charge= None, spin= None, basisset= None,
             auxbasisset= None, functional= None, tasks=None,
             gwbse=None):

    tree = ET.parse(file_path)
    root = tree.getroot()

    if job_name:
        job_name_ph = root.find('dftgwbse/job_name')
        job_name_ph.text = job_name

    if charge:
        charge_ph = root.find('dftgwbse/dftpackage/charge')
        charge_ph = charge

    if spin:
        spin_ph = root.find('dftgwbse/dftpackage/')
        spin_ph = spin

    if basisset:
        basisset_ph = root.find('dftgwbse/dftpackage/basisset')
        basisset_ph = basisset

    if auxbasisset:
        auxbasisset_ph = root.find('dftgwbse/dftpackage/auxbasisset')
        auxbasisset_ph = auxbasisset

    if functional:
        functional_ph = root.find('dftgwbse/dftpackage/functional')
        functional_ph = functional

    if tasks:
        tasks_ph = root.find('dftgwbse/dftpackage/tasks')
        tasks_ph = tasks

    if gwbse:
        gwbse_ph = root.find('dftgwbse/dftpackage/gwbse')
        gwbse_ph = gwbse

    tree.write(file_path)

# edit_xml('test_votca/options.xml', 'molecules')
# this works and changes job_name to molecules

```

Abstract