Helsinki University of Technology
Tfy-3.393 Special assignment in computational physics

Python Scripts for Structural Optimization of Small Molecules

Kaarle Ritvanen
55057S
January 28, 2004

Contents

1 Introduction **2**

2 Calculating Potential Energy **4**
2.1 Lennard–Jones Potential 4
2.2 Density Functional Theory 5

3 Minimization of Multivariable Functions **7**
3.1 BFGS Method 7
3.2 Pulay Mixing 8

4 Python Implementation of Structure Optimizer **10**
4.1 Python Language 10
4.2 Program Architecture 10
4.2.1 Module `Atoms` 10
4.2.2 Module `Calculators` 11
4.2.3 Module `Minimizers` 12
4.3 Remarks on Implementation 12
4.4 Example Script 13

5 Results **14**
5.1 Optimal Structures for Lennard–Jones Clusters 14
5.2 Comparison of Minimization Algorithms 16
5.2.1 Minimization Using Pulay Mixing 18
5.3 Methane Molecule 19

6 Conclusions **21**

A Derivations for Lennard–Jones Potential **23**
A.1 Minimum Energy for Two Atoms 23
A.2 Forces for Multiple Atoms 23

B Technical Details on Implementation **25**
B.1 Line Minimization Algorithm 25
B.2 Interface to the GSL Minimization Routines 26
B.2.1 Source Code of Module `Minimizers.GSL` 26

C Optimal Structure Plots for Lennard–Jones Clusters **35**

Chapter 1

Introduction

There are forces between atoms in a molecule. The forces depend on the position of the atoms, and the atoms tend to place themselves so that the forces vanish. Structural optimization means finding such positions for the atoms, provided that there is a way to calculate the potential energy of the molecule and the forces affecting the atoms.

There are several ways to calculate the potential energy between the atoms. For so called van der Waals solids it is convenient to use Lennard–Jones potential to approximate the interaction between the atoms [1]. There are also more general methods for evaluating the energy and the forces, for instance those based on Density Functional Theory (DFT) [2]. These methods are discussed briefly in Chapter 2.

Also the minimization of the energy function may be done in several ways. Given a method for computing the energy and the forces (negative gradient of energy) at arbitrary points, the problem of structural optimization can be seen as a special case of more general problem: minimization of a scalar function depending on several variables. If we have N atoms in a molecule, we have to find a minimum for function

$$E(\vec{\boldsymbol{r}}_1, \ldots, \vec{\boldsymbol{r}}_N) = E(x_1, y_1, z_1, \ldots, x_N, y_N, z_N) \tag{1.1}$$

where $r_1, \ldots, r_N$ denote the positions of the atoms. Function E depends on $3N$ variables. We will assume that the forces affecting the atoms

$$\vec{\boldsymbol{F}}_i(\vec{\boldsymbol{r}}_1, \ldots, \vec{\boldsymbol{r}}_N) = -\frac{\partial E}{\partial \vec{\boldsymbol{r}}_i} \qquad i = 1, \ldots, N \tag{1.2}$$

may also be calculated at arbitrary points because good algorithms for multidimensional minimization make use of this information. It is convenient to treat the forces as one $3N$-dimensional vector function

$$\boldsymbol{F} = \begin{pmatrix} \vec{\boldsymbol{F}}_1 \\ \vdots \\ \vec{\boldsymbol{F}}_N \end{pmatrix} = -\nabla E(\vec{\boldsymbol{r}}_1, \ldots, \vec{\boldsymbol{r}}_N) \tag{1.3}$$

instead of N different 3-dimensional functions because the minimization algorithms make no difference between the coordinates. These algorithms are discussed in Chapter 3.

It is worth noting that the energy and force computations and multivariable function minimization may be performed independently of each other. The purpose of this special assignment is to create a framework for doing structural optimizations, no matter which two methods are used. This framework and its interfaces are described in Chapter 4.

Chapter 2

Calculating Potential Energy

In this chapter we discuss briefly two methods for computing the potential energy of a molecule when the positions of the atoms are given. These methods are Lennard–Jones potential and Density Functional Theory.

2.1 Lennard–Jones Potential

Lennard–Jones is useful when modeling so called van der Waals solids. That means solids where the only significant attractive interaction is the electrostatic force induced between the dipole atoms, possibly as a result of fluctuations in the charge distribution. [1]

The formula for Lennard–Jones potential between two atoms, the distance of which is r, is:

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

This potential function has been plotted in Figure 2.1.

The term with exponent 6 represents the attractive van der Waals interaction between the atoms. The proportionality to r^{-6} can be derived from quantum mechanics [3].

When the atoms get so close to each other that their charge distributions begin to overlap, the Pauli exclusion principle kicks in. The principle states that two electrons occupying the same spatial region cannot have the same quantum numbers. The principle is satisfied when some electrons are promoted to higher-energy states thus changing their quantum numbers. This naturally leads to higher potential energy when the atoms are getting closer. [1]

The term proportional to r^{-12} represents the phenomenon caused by the exclusion principle. The exponent is large enough to provide a rapid increase in energy as r gets smaller. 12 is a nice choice because it makes the algebraic manipulation of $U(r)$ easy. For example, the distance for minimum energy $r_{min} = \sqrt[6]{2}\sigma$, and the minimum energy $U(r_{min}) = -\varepsilon$ (see Appendix A for details).

If we assume that there is no other forces between the atoms of a molecule than the one caused by pairwise Lennard–Jones potential, the total potential

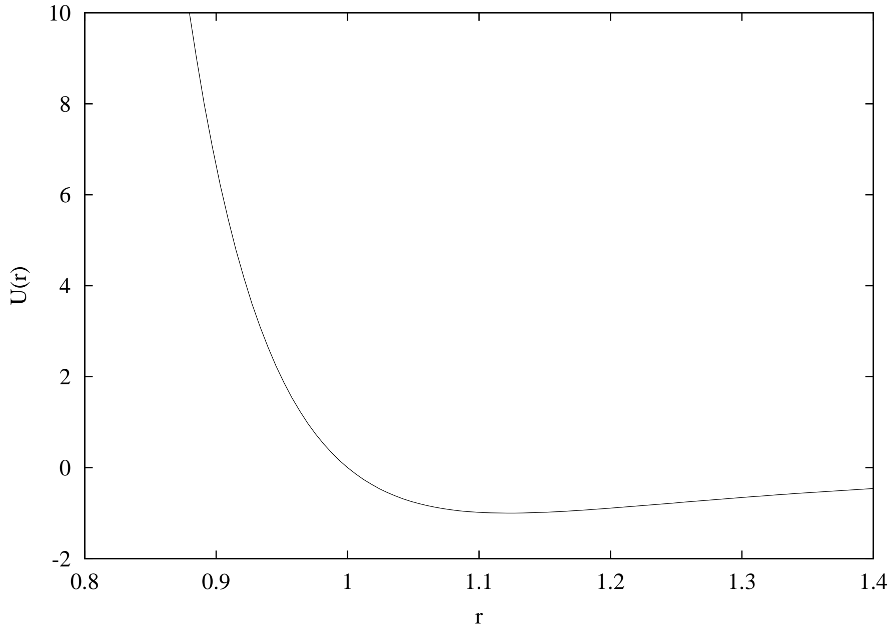

Figure 2.1: Lennard–Jones potential with $\varepsilon = 1$ and $\sigma = 1$.

energy of the molecule is the sum of such potentials:

$$E(\vec{\boldsymbol{r}}_1, \ldots, \vec{\boldsymbol{r}}_N) = \sum_{i=2}^{N} \sum_{j<i} U(|\vec{\boldsymbol{r}}_i - \vec{\boldsymbol{r}}_j|) \tag{2.2}$$

The forces $\vec{\boldsymbol{F}}_i, i = 1, \ldots, N$ caused by this kind of potential are (see Appendix A for derivation):

$$\vec{\boldsymbol{F}}_i(\vec{\boldsymbol{r}}_1, \ldots, \vec{\boldsymbol{r}}_N) = 24\varepsilon \sum_{j \neq i} \left(\frac{\sigma^6}{|\vec{\boldsymbol{r}}_j - \vec{\boldsymbol{r}}_i|^8} - \frac{2\sigma^{12}}{|\vec{\boldsymbol{r}}_j - \vec{\boldsymbol{r}}_i|^{14}} \right) (\vec{\boldsymbol{r}}_j - \vec{\boldsymbol{r}}_i) \tag{2.3}$$

2.2 Density Functional Theory

A stationary system of N electrons is described by a wavefunction $\Psi(\vec{\boldsymbol{r}}_1, \cdots, \vec{\boldsymbol{r}}_N)$ satisfying the stationary Schrödinger equation

$$\hat{\boldsymbol{H}}\Psi = \left(-\frac{\hbar^2}{2m_e} \nabla^2 + V(\vec{\boldsymbol{r}}_1, \cdots, \vec{\boldsymbol{r}}_N) \right) \Psi = E\Psi \tag{2.4}$$

The ground state energy is the lowest eigenvalue E_{min} of this eigenvalue problem, so in theory, E_{min} should be the minimum value of $\langle \Psi | \hat{\boldsymbol{H}} \Psi \rangle$ over all possible normalized wavefunctions (for which $\langle \Psi | \Psi \rangle = 1$). Unfortunately, there are an infinite number of possible wavefunctions, so this is not very efficient approach for finding the ground state energy.

Density Functional Theory suggest a more efficient method for solving this problem. Charge density $n(\boldsymbol{r})$ is presented with single-particle states ψ_i as

follows:

$$n(\boldsymbol{r}) = 2\sum_i |\psi_i(\boldsymbol{r})|^2 \tag{2.5}$$

One could start by making an initial guess of $n(\boldsymbol{r})$ and use that to solve the orthonormal states Ψ_i from the Kohn–Sham equation

$$\left(-\frac{\hbar^2}{2m_e}\nabla^2 + V_{ion}(\boldsymbol{r}) + V_H(\boldsymbol{r}) + V_{XC}(\boldsymbol{r})\right)\psi_i = \epsilon_i\psi_i \tag{2.6}$$

where

$$V_H(\boldsymbol{r}) = \int_{\mathbb{R}^{3N}} \frac{n(\boldsymbol{r'})}{|\boldsymbol{r}-\boldsymbol{r'}|} d\boldsymbol{r'} \tag{2.7}$$

$$V_{XC}(\boldsymbol{r}) = \frac{\delta E_{XC}[n(\boldsymbol{r})]}{\delta n(\boldsymbol{r})} \tag{2.8}$$

Then Equation (2.5) can be used to construct a better approximation for $n(\boldsymbol{r})$. This process is repeated until self-consistency has been achieved. The total energy can be calculated from the self-consistent $n(\boldsymbol{r})$ and corresponding eigenvalues ϵ_i. [4]

A program package named MIKA[1] has been being developed by Laboratory of Physics at Helsinki University of Technology and CSC (Finnish IT Center for Science) to perform this kind of calculations.

[1] `http://www.csc.fi/physics/mika/`

Chapter 3

Minimization of Multivariable Functions

This chapter discusses two methods for multivariable function minimization. First we take a look at Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm which is a quasi-Newton method. After that a method called Pulay mixing is described briefly.

3.1 BFGS Method

Let us consider an N-variable function $f(\boldsymbol{x})$. If the function is smooth enough it can be approximated near point $\boldsymbol{x}_i$ as

$$f(\boldsymbol{x}) = f(\boldsymbol{x}_i)+(\boldsymbol{x}-\boldsymbol{x}_i)\cdot\nabla f(\boldsymbol{x}_i)+\frac{1}{2}(\boldsymbol{x}-\boldsymbol{x}_i)\cdot\boldsymbol{A}(\boldsymbol{x}_i)\cdot(\boldsymbol{x}-\boldsymbol{x}_i)+\mathcal{O}(|\boldsymbol{x}-\boldsymbol{x}_i|^3) \quad (3.1)$$

where

$$\boldsymbol{A}(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_N \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_N^2} \end{pmatrix} \quad (3.2)$$

is the Hessian matrix of $f(\boldsymbol{x})$. Differentiating Equation (3.1) and ignoring the third and higher order terms yields

$$\nabla f(\boldsymbol{x}) = \nabla f(\boldsymbol{x}_i) + \boldsymbol{A}(\boldsymbol{x}_i)\cdot(\boldsymbol{x}-\boldsymbol{x}_i) \quad (3.3)$$

Suppose that f has a minimum at $\boldsymbol{x}$. Then $\nabla f(\boldsymbol{x}) = 0$, and solving for $\boldsymbol{x}$ gives

$$\boldsymbol{x} = \boldsymbol{x}_i - \boldsymbol{A}(\boldsymbol{x}_i)^{-1}\cdot\nabla f(\boldsymbol{x}_i) \quad (3.4)$$

The idea of quasi-Newton methods is to construct a sequence of matrices $\boldsymbol{H}_i$ having the following property [5]:

$$\lim_{i\to\infty} \boldsymbol{H}_i = \boldsymbol{A}(\boldsymbol{x})^{-1} \quad (3.5)$$

If we replace $\boldsymbol{A}(\boldsymbol{x}_i)^{-1}$ in Equation (3.4) with $\boldsymbol{H}_i$, we get a formula for computing an approximation $\boldsymbol{x}_{i+1}$ for the minimum point:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \boldsymbol{H}_i\cdot\nabla f(\boldsymbol{x}_i) \quad (3.6)$$

Now the remaining question concerns how to construct a sequence of matrices $\boldsymbol{H}_i$ that converges rapidly to the inverse Hessian matrix of f. BFGS algorithm suggests using the following updating formula [5]:

$$\begin{aligned}\boldsymbol{H}_{i+1} =& \boldsymbol{H}_i + \frac{(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i) \otimes (\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)}{(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i) \cdot (\nabla f(\boldsymbol{x}_{i+1}) - \nabla f(\boldsymbol{x}_i))} - \\ & \frac{[\boldsymbol{H}_i \cdot (\nabla f(\boldsymbol{x}_{i+1}) - \nabla f(\boldsymbol{x}_i))] \otimes [\boldsymbol{H}_i \cdot (\nabla f(\boldsymbol{x}_{i+1}) - \nabla f(\boldsymbol{x}_i))]}{(\nabla f(\boldsymbol{x}_{i+1}) - \nabla f(\boldsymbol{x}_i)) \cdot \boldsymbol{H}_i \cdot (\nabla f(\boldsymbol{x}_{i+1}) - \nabla f(\boldsymbol{x}_i))} + \\ & [(\nabla f(\boldsymbol{x}_{i+1}) - \nabla f(\boldsymbol{x}_i)) \cdot \boldsymbol{H}_i \cdot (\nabla f(\boldsymbol{x}_{i+1}) - \nabla f(\boldsymbol{x}_i))] \boldsymbol{u}_{i+1} \otimes \boldsymbol{u}_{i+1}\end{aligned} \tag{3.7}$$

where

$$\begin{aligned}\boldsymbol{u}_{i+1} =& \frac{\boldsymbol{x}_{i+1} - \boldsymbol{x}_i}{(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i) \cdot (\nabla f(\boldsymbol{x}_{i+1}) - \nabla f(\boldsymbol{x}_i))} - \\ & \frac{\boldsymbol{H}_i \cdot (\nabla f(\boldsymbol{x}_{i+1}) - \nabla f(\boldsymbol{x}_i))}{(\nabla f(\boldsymbol{x}_{i+1}) - \nabla f(\boldsymbol{x}_i)) \cdot \boldsymbol{H}_i \cdot (\nabla f(\boldsymbol{x}_{i+1}) - \nabla f(\boldsymbol{x}_i))}\end{aligned} \tag{3.8}$$

It can be shown that if f is a quadratic form, the matrix sequence constructed using Equation (3.7) reaches the inverse Hessian of f by a finite number of steps [6].

It is important to notice that in order to $\boldsymbol{x}_{i+1} - \boldsymbol{x}_i$ to be a descent direction, the following condition must be met:

$$(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i) \cdot \nabla f(\boldsymbol{x}_i) < 0 \tag{3.9}$$

Equation (3.6) implies

$$(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i) \cdot \nabla f(\boldsymbol{x}_i) = -(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i) \cdot \boldsymbol{H}_i^{-1} \cdot (\boldsymbol{x}_{i+1} - \boldsymbol{x}_i) \tag{3.10}$$

so it follows that $\boldsymbol{H}_i^{-1}$ (and thus $\boldsymbol{H}_i$) has to be positive definite. Therefore it is important to make sure that the initial value $\boldsymbol{H}_0$ satisfies this property. Also roundoff error may cause some $\boldsymbol{H}_i$ not to be positive definite, so it is important to take care of this possibility when implementing BFGS. [5]

3.2 Pulay Mixing

Pulay mixing was originally intended to be used with Hartree–Fock calculations but is nowadays also used in calculations of Density Functional Theory [7]. Now we try to use this method in order to optimize molecular structures.

Suppose we know few values of the gradient of function $f(\boldsymbol{x})$ to be minimized nearby the actual minimum. We denote these points as $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$. The idea behind this method is to assume that $\nabla f(\boldsymbol{x})$ is almost linear near the minimum point:

$$\nabla f(\sum_{i=1}^{n} \alpha_i \boldsymbol{x}_i) \approx \sum_{i=1}^{n} \alpha_i \nabla f(\boldsymbol{x}_i) \tag{3.11}$$

The gradient should become almost zero if we select each α_i as follows [8]:

$$\alpha_i = \frac{\sum_{j=1}^{n} \boldsymbol{A}_{ji}^{-1}}{\sum_{j=1}^{n} \sum_{k=1}^{n} \boldsymbol{A}_{jk}^{-1}} \tag{3.12}$$

where

$$\boldsymbol{A}_{ij} = \nabla f(\boldsymbol{x}_i) \cdot \nabla f(\boldsymbol{x}_j) \tag{3.13}$$

We could start by a set of points located nearby the minimum of $f(\boldsymbol{x})$ and make a guess for the minimum point by computing the linear combination of them where coefficients are given by Equation (3.12). However, we cannot continue the iteration by adding the new point to the set and then computing the linear combination again. All further points would just be linear combinations of the points belonging to the original set, and it is very unlikely that the minimum point really is a linear combination of them. It is necessary to add at least one linearly independent point to the set after each iteration.

Chapter 4

Python Implementation of Structure Optimizer

4.1 Python Language

Python is an interpreted and object-oriented programming language. It has such features as classes, exceptions and modules. Also some very high level data types like lists and maps are embedded to the language itself. It is also possible to write extensions to Python programs in C and C++ (see Appendix B).

Most Python related sources and binaries published by the Python Software Foundation[1] are distributed under a GNU General Public License compatible license. However, the license is even more relaxed than the GPL.

4.2 Program Architecture

The implemented structure optimizer consists of three modules:

Atoms data structures containing the essential information about the atoms

Calculators calculators for computing the potential energy and the forces exerted to the atoms

Minimizers multivariable function minimization algorithms

By using these modules it is quite straightforward to create scripts that optimize the structures of some particular molecules. Such script just defines the initial positions of the atoms, minimization algorithm and the method for computing the potential energy and the forces. Of course, these methods and minimization algorithms may require some parameters specific to them to be set. An example of using this program are given in Section 4.4

4.2.1 Module `Atoms`

This module contains two classes, namely `Atom` and `Atoms`. Class `Atom` represents a single atom and it contains information about its position and type (which element it is). It has the following methods:

[1] `http://www.python.org/`

getType returns the atom type

getPosition returns the position of the atom

setPosition sets the position to a new value

getForce returns the force affecting to the atom

setForce sets the force to a new value

invalidateForce invalidates the previously calculated force

When `getForce` is called first time, the forces affecting to all the atoms are calculated. When the position of any of the atoms belonging to the same molecule is changed by calling `setPosition`, the forces are recalculated when `getForce` is called next time. Method `invalidateForce` has the same effect but it does not change the position of the atom.

Class `Atoms` represents a set of atoms, i.e. a molecule, the structure of which is to be optimized. Its methods are:

addAtom adds an atom to the set

writeGXYZ writes the positions of the atoms to a file in GXYZ format

getSize returns the size of the cell

getOrigin returns the location of the origin

calculate calculates the potential energy and the forces

getEnergy returns the potential energy

setEnergy sets the potential energy

invalidate invalidates the energy and forces calculated previously

optimizeStructure moves the atoms to the optimum positions

Calling `getEnergy` causes the potential energy and the forces to be recalculated if they are not valid. Calling method `invalidate` or `addAtom` leads to invalidation of those values.

4.2.2 Module `Calculators`

This module provides the means to compute the potential energy of a set of atoms. Every class in this module has at least the following methods:

setAtoms used once to set the reference to the atom set (the energy of which is to be calculated)

setOptions sets some parameters (specific to the method)

execute calculates the potential energy and the forces, and sets the results to the atom data structures

As all the calculator classes have similar interface, they can be used exactly in the same way after the method specific options have been set.

4.2.3 Module `Minimizers`

Classes in this module implement the minimization algorithms. As was the case in module `Calculators`, the classes of this module also have a similar interface when compared to each other.

Minimizers are initialized with the initial position $\boldsymbol{x}_0$. All minimizers have method named `getNewX` which is called several times after that. This method takes two arguments, namely the function value and its gradient at the same position. At first call, these arguments contain the value and gradient evaluated at the initial position. Each call to `getNewX` returns a new suggestion for the minimum position. On next call to `getNewX`, the values to be given as arguments are evaluated at the position returned by the previous call to it, like this:

$$\boldsymbol{x}_{i+1} = \texttt{getNewX}(f(\boldsymbol{x}_i), \nabla f(\boldsymbol{x}_i)) \tag{4.1}$$

4.3 Remarks on Implementation

An energy calculator for Lennard–Jones was implemented. It just uses Equations (2.2) and (2.3) to compute the potential energy and the forces exerted to the atoms. Also a Python interface to the `rspace` program[2] was written. It generates an appropriate input file, runs the program and finally parses the output and updates the atom data structures according to them.

Both minimization methods described in Chapter 3, namely BFGS and Pulay mixing, were implemented. However, the BFGS implementation does not directly use Equation (3.6) to compute the next approximation for the minimum. Instead, the next BFGS step will be the minimum point of the function along the line between $\boldsymbol{x}_i$ and $\boldsymbol{x}_i - \boldsymbol{H}_i \cdot \nabla f(\boldsymbol{x}_i)$. This is more effective because if we took the full step, we might get too far from the minimum for the quadratic approximation to be valid [5]. The line minimization algorithm used between BFGS steps is described in detail in Appendix B.

The Pulay mixing implementation maintains a fixed-size history of the points it has calculated by using Equation (3.12) and the gradient values at these points. These values are used to generate further guesses for the minimum point with Equation (3.12). As pointed out in Section 3.2, it is necessary to include also linearly independent points to the input set. Therefore for every other call the Pulay mixing implementation returns a point calculated by subtracting the gradient from the point returned on the previous call. This point is then temporarily added to the input set for the next evaluation of Equation (3.12).

In addition to these two methods, an interface to the GNU Scientific Library[3] minimization functions was implemented. This was accomplished by using the Python C API. The details are presented in Appendix B. The library provides the steepest descent algorithm, vector BFGS and the conjugate gradient method (the Fletcher–Reeves and Polak–Ribiere versions).

---

[2] `rspace` is a part of MIKA program package

[3] `http://www.gnu.org/software/gsl/`

4.4 Example Script

This section presents an example of a Python script that utilizes this structural minimization framework. It uses BFGS as the minimization algorithm and `rspace` program to compute the energy and the forces.

```
#!/usr/bin/python2.2

from Atoms import *
from Calculators.RSpace import RSpace
from Minimizers.BFGS import BFGS

# create new calculator instance and set its options
calc = RSpace()
calc.setOptions(base_dir='"/home/ker/MIKA"',
                states=4,
                boundary_conditions='PERIODIC',
                occ='4*2',
                grid=(20, 20, 20),
                scf_treshold=0.0001,
                lattice_constant=1.8904,
                distance_unit=1.8904)

# create new atom set and initialize atom positions
atoms = Atoms(calc,
              (3.0, 3.0, 3.0),     # cell size
              (0.5, 0.5, 0.5),     # origin
              Atom('C', (0.0, 0.0, 0.0)),
              Atom('H', (1.0, 1.0, 1.0)),
              Atom('H', (-1.0, -1.0, 1.0)),
              Atom('H', (1.0, -1.0, -1.0)),
              Atom('H', (-1.0, 1.0, -1.0)))

# perform optimization
atoms.optimizeStructure(BFGS, maxiter=100, threshold=1e-6)

# print the atom positions
for atom in atoms:
    print atom

# write the positions to a file
atoms.writeGXYZ('CH4.gxyz')
```

More examples can be found in Chapter 5.

Chapter 5

Results

5.1 Optimal Structures for Lennard–Jones Clusters

The program was used to find the optimal structures for small Lennard–Jones clusters. The following Python script was used to find such structures for every N-atom Lennard–Jones cluster, where $4 \leq N \leq 21$:

```
#!/usr/bin/python2.2

from Atoms import *
from Calculators.LennardJones import LennardJones
from Minimizers.BFGS import BFGS
import random

def addAtom(atoms):
    atoms.addAtom(Atom('C',
                       (random.uniform(-1.0, 1.0),
                        random.uniform(-1.0, 1.0),
                        random.uniform(-1.0, 1.0))))

calc = LennardJones()
calc.setOptions(epsilon=1.0, sigma=1.0)

atoms = Atoms(calc, (2.5, 2.5, 2.5), (0.5, 0.5, 0.5))
for N in range(1, 22):
    addAtom(atoms)
    if N > 3:
        atoms.optimizeStructure(BFGS)
        atoms.writeGXYZ('gxyz/LJ%d.gxyz' % N)
        print 'cluster N = %d complete' % N
```

At first this script calculates an optimal structure for a four-atom cluster. It continues by storing the previous result, adding a new atom to a random position and then calling method `optimizeStructure` to obtain the equilibrium

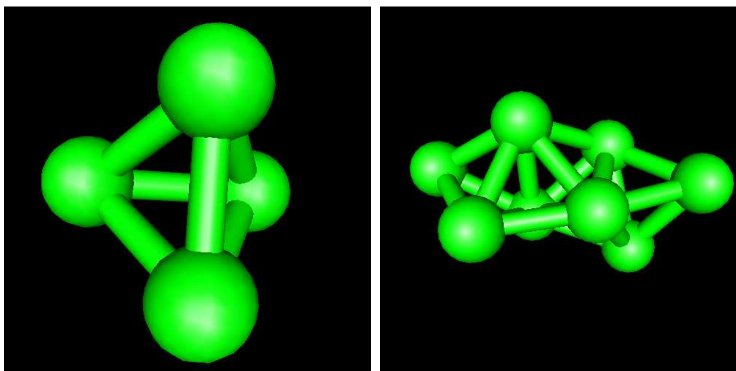

Figure 5.1: 4-atom and 8-atom Lennard–Jones clusters

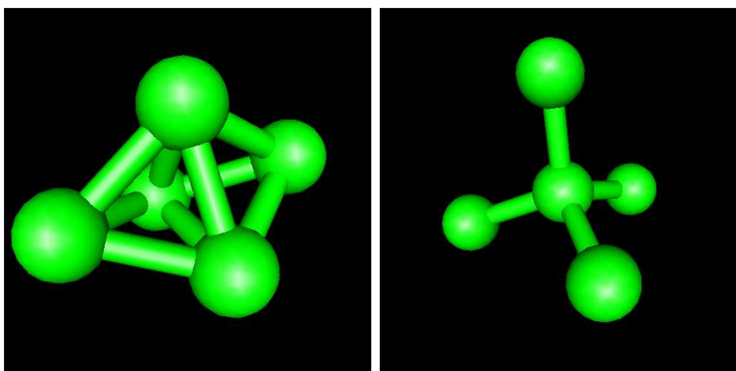

Figure 5.2: Two different 5-atom clusters

structure for the bigger cluster. This step is repeated until the 21-atom cluster has also been handled.

The optimal structures were plotted by using a program named gOpenMol[1]. For instance, Figure 5.1 shows equilibrium positions for four-atom and eight-atom clusters. The optimal structures for all N-atom clusters ($4 \leq N \leq 21$) calculated by this script are shown in Appendix C.

It is important to note that the minimum point found by some multidimensional minimization algorithm may not be the global minimum point for that function. Thus the optimal structure found by this program depends on the initial positions of the atoms. For example, Figure 5.2 shows two possible structures for a five-atom cluster, the both of which are in equilibrium. The structure shown on the right side may be obtained if the atoms are initially already located at CH_4-like positions.

As the algorithms used in connection with this assignment do not necessarily find the global minimum, it is very unlikely that one could find any exceptional optimal structures by using them. For example, it has been discovered that the global minimum energies of small gold clusters are achieved by planar structures although some local minima can be found for purely three-dimensional

[1] `http://www.csc.fi/gopenmol/`

structures [9]. Therefore a local minimization algorithm would very probably converge to such non-global minimum point.

Nevertheless, there are such methods that find the global minimum of a multivariable function [10]. For instance, the Simulated Annealing method and its variants have been used to solve this kind of problems [11]. However, these methods are beyond the scope of this assignment.

5.2 Comparison of Minimization Algorithms

The multivariable minimization algorithms provided by the GNU Scientific Library (Conjugate gradient/Fletcher–Reeves, Conjugate Gradient/Polak–Ribiere and Vector BGFS) were compared with my own BFGS implementation which also uses parabolic line minimization. The following script was used to carry out these tests:

```
#!/usr/bin/python2.2

from Atoms import *
from Calculators.LennardJones import LennardJones
from Minimizers import *

def calculator():
    calc = LennardJones()
    calc.setOptions(epsilon=1.0, sigma=1.0)
    return calc

algorithms = [(BFGS.BFGS, None),
              (GSL.GSL, 'conjugate_fr'),
              (GSL.GSL, 'conjugate_pr'),
              (GSL.GSL, 'vector_bfgs')]

for minClass, algName in algorithms:
    print 'Algorithm: %s/%s' % (str(minClass), str(algName))
    molecules = [Atoms(calculator(), (1.0, 1.0, 1.0), (0.5, 0.5, 0.5),
                       Atom('C', (-0.5, 0.0, 0.0)),
                       Atom('C', (0.5, 0.0, 0.0))),
                 Atoms(calculator(), (3.0, 3.0, 3.0), (0.5, 0.5, 0.5),
                       Atom('C', (0.0, 0.0, 0.0)),
                       Atom('C', (0.75, 0.7, 0.7)),
                       Atom('C', (-0.7, -0.7, 0.7)),
                       Atom('C', (0.7, -0.7, -0.7)),
                       Atom('C', (-0.7, 0.7, -0.7))),
                Atoms(calculator(), (3.0, 3.0, 3.0), (0.5, 0.5, 0.5),
                       Atom('C', (0.0, 0.0, 0.0)),
                       Atom('C', (-1.0, 0.0, 0.0)),
                       Atom('C', (0.0, 1.0, 0.0)),
                       Atom('C', (2.0, 0.0, -0.5)),
                       Atom('C', (-1.0, -0.1, -2.0))),
                 Atoms(calculator(), (3.0, 3.0, 3.0), (0.5, 0.5, 0.5),
                       Atom('C', (0.0, 1.0, 1.0)),
```

Algorithm	2 atoms	5 atoms (ord.)	5 atoms	10 atoms
BFGS/Parabolic	144	178	224	642
ConjGr/FR	27	1287	2095	6208
ConjGr/PR	27	568	936	2026
Vector BFGS	27	313	639	1422

Table 5.1: Numbers of iteration steps

Algorithm	2 atoms	5 atoms (ord.)	5 atoms	10 atoms
BFGS/Parabolic	−1.00	−9.10	−9.10	−27.2
ConjGr/FR	−1.00	−8.20	−9.10	−26.5
ConjGr/PR	−1.00	−8.20	−9.10	−26.5
Vector BFGS	−1.00	−8.20	−9.10	−26.4

Table 5.2: Minimization results

```
                          Atom('C', (0.0, 0.1, 0.2)),
                          Atom('C', (0.2, 1.2, 2.0)),
                          Atom('C', (1.0, 1.1, 0.2)),
                          Atom('C', (-1.0, 0.1, 1.0)),
                          Atom('C', (1.0, 1.5, -0.0)),
                          Atom('C', (-1.0, 1.0, 1.0)),
                          Atom('C', (0.5, 1.0, 0.0)),
                          Atom('C', (2.0, 0.1, -0.5)),
                          Atom('C', (-1.0, -0.1, -2.0)))]
for molecule in molecules:
    print 'Molecule: ' + str(molecule)
    molecule.optimizeStructure(minClass, maxiter=10000,
                               threshold=1e-4, algorithm=algName)
```

As it can be seen from the above listing, there are four test cases. Every one of them is a Lennard–Jones cluster. The first one is a simple two-atom cluster. The second and third tests are five-atom clusters. In the second case, the atoms are initially almost in a CH_4-like order. The last test is a ten-atom cluster. The stopping criterion is that the root-mean-square of the forces must be less than 10^{-4}.

Table 5.1 shows the number of iteration steps required until the forces were small enough. It seems that also the GSL routines use some line minimization algorithm since they all yielded the same number of steps for the two-atom case. For that case, the GSL methods needed less steps than my BFGS implementation. However, it outperformed the GSL methods when the number of atoms was increased. One reason for that may be the fact that GSL provided no reasonable way to limit the range of the variables (atom coordinates) as my BFGS implementation does. The minimum energy values calculated by the algorithms are presented in Table 5.2.

5.2.1 Minimization Using Pulay Mixing

The plain Pulay mixing method did not turn out to be very workable method for structural optimization. It did not converge at all unless the initial positions of the atoms were already very close to the minimum. Therefore there is no point using it because in general case, the minimum is not known and the atoms may be in arbitrary position in the beginning.

However, the convergence of the Pulay mixing method was really fast if the initial positions were located nearby the actual minimum. It was even faster than the convergence of BFGS algorithm. Then I came up with the idea that Pulay mixing could be used after the required precision had been achieved by some other algorithm. A composite minimizer class `Minimizers.Pulay.BFGSPulay` was written to examine this possibility.

The following script demonstrates the efficiency of Pulay mixing near the minimum:

```
#!/usr/bin/python2.2

from Atoms import *
from Calculators.LennardJones import LennardJones
from Minimizers import *

calc = LennardJones()
calc.setOptions(epsilon=1.0, sigma=1.0)

def getAtoms():
    return Atoms(calc,
                 (3.0, 3.0, 3.0),
                 (0.5, 0.5, 0.5),
                 Atom('C', (0.0, 0.0, 0.0)),
                 Atom('C', (0.75, 0.7, 0.7)),
                 Atom('C', (-0.7, -0.7 - 0.0, 0.7)),
                 Atom('C', (0.7, -0.7, -0.7)),
                 Atom('C', (-0.7, 0.7, -0.7)),
                 Atom('C', (0, 0, 0.2)))

def printAtoms(atoms):
    for atom in atoms:
        print atom

for i in range(4, 13):
    t = 10 ** -i

    print 'BFGS, t = %.0e' % t
    atoms = getAtoms()
    atoms.optimizeStructure(BFGS.BFGS, threshold=t)
    printAtoms(atoms)

    print 'Pulay/BFGS, t = %.0e' % t
    atoms = getAtoms()
    atoms.optimizeStructure(Pulay.BFGSPulay, threshold=t)
```

RMS Limit	BFGS	Pulay/BFGS
10^{-4}	347	347
10^{-5}	358	353
10^{-6}	366	363
10^{-7}	376	365
10^{-8}	383	367
10^{-9}	392	369
10^{-10}	410	373
10^{-11}	420	373
10^{-12}	428	379

Table 5.3: Required iterations for certain precisions

```
    printAtoms(atoms)
```

The composite minimizer uses BFGS with parabolic line minimization until the root-mean-square of the forces decreases below 10^{-4}. Table 5.3 shows how many iterations are needed in order to reduce the root-mean-square of the forces below certain limit (in this particular problem). Based on these results, it seems that doing the last steps using Pulay mixing slightly reduces the amount of steps.

5.3 Methane Molecule

The Python interface to `MIKA/rspace` was tested by performing a structural optimization to a methane molecule (CH_4). The carbon atom was placed to the origin. The four hydrogen atoms were placed $0.7\sqrt{3}\,\text{Å} \approx 1.21\,\text{Å}$ away from the carbon atom such that they form vertices for a tetrahedron. The size of the cubic unit cell was set to 4 Å. The following Python script was used:

```
#!/usr/bin/python2.2

from Atoms import *
from Calculators import *
from Minimizers import *

x = 0.7

calc = RSpace.RSpace()

atoms = Atoms(calc,
              (4.0, 4.0, 4.0),
              (0.5, 0.5, 0.5),
              Atom('C', (0.0, 0.0, 0.0)),
              Atom('H', (x + 0.1, x, x)),
              Atom('H', (-x, -x, x)),
              Atom('H', (x, -x, -x)),
```

```
              Atom('H', (-x, x, -x)))

calc.setOptions(base_dir='"/home/tto/MIKA"',
                states=8,
                boundary_conditions='PERIODIC',
                occ='4*2 4*0',
                grid=(20, 20, 20),
                scf_treshold=0.0001,
                lattice_constant=1.8904,
                distance_unit=1.8904,
                finite_temperature='.false.',
                mixing_method='GRPULAY')

atoms.optimizeStructure(BFGS.BFGS, threshold=1e-4)

for atom in atoms:
    print atom
```

The root-mean-square of the forces decreased under 10^{-4} hartrees per Bohr radius after 36 steps. All the hydrogen atoms settled to distance of 1.07 Å from the carbon atom, the total energy being -217.7 eV.

Chapter 6

Conclusions

We implemented a framework for performing structural optimization of small molecules. Python programming language was used, and it turned out to be quite suitable tool for this purpose. As the code does not need to be recompiled every time it is modified, it is very fast and easy to change when necessary. When used with `MIKA/rspace`, most of the CPU time is used to compute the potential energy and the forces exerted to the atoms according to the technique described in Section 2.2. Those computations are performed by compiled FORTRAN code, so the overhead of Python script interpretation is not significant.

In addition to the `MIKA/rspace` interface, a Lennard–Jones energy and force calculator was implemented.

Two multidimensional minimization algorithms were implemented, namely BFGS with parabolic line minimization and Pulay mixing. The latter did not work very well by itself, but when combined with BFGS, it turned out to be quite efficient.

It is worth noting that using Cartesian coordinates is not very efficient in structural optimizations. Consider a two-atom molecule. If we use Cartesian coordinates, we have six degrees of freedom. However, the potential energy essentially depends on only one degree of freedom: the distance between the atoms. It might be better to use so called generalized natural internal coordinates [12].

Bibliography

[1] S. Elliott. *The Physics and Chemistry of Solids*. John Wiley & Sons, Chichester, 1998.

[2] R. G. Parr, W. Yang. *Density-Functional Theory of Atoms and Molecules*. Monographs on Chemistry. Clarendon Press, Oxford, 1989.

[3] C. Kittel. *Introduction to Solid State Physics*. 7th edition. John Wiley & Sons, New York, 1996.

[4] M. Heiskanen. *Multigrid Methods for Electronic Structure Calculations*. Master's Thesis. Department of Engineering Physics and Mathematics, Helsinki University of Technology, 1999.

[5] W. H. Press et al. *Numerical Recipes in FORTRAN 77: The Art of Scientific Computing*. 2nd edition. Cambridge University Press, Cambridge, 1992.

[6] E. Polak. *Computational Methods in Optimization*. Academic Press, New York, 1971.

[7] D. R. Bowler, M. J. Gillian. *An Efficient and Robust Technique for Achieving Self Consistency in Electronic Structure Calculations*. Chemical Physics Letters, Vol. 325, pp. 473–476. 2000.

[8] G. Kresse, J. Furthmüller. *Efficient Iterative Schemes for* ab initio *Total Energy Calculations Using a Plane Wave Basis Set*. Physical Review B, Vol. 54, No. 16, pp. 11169–11186. 1996.

[9] H. Häkkinen et al. *On the Electronic and Atomic Structures of Small* Au_N^- *(*$N = 4$–14*) Clusters: A Photoelectron Spectroscopy and Density-Functional Study*. Journal of Physical Chemistry A, Vol. 107, pp. 6168–6175. 2003.

[10] J. Lee et al. *Unbiased Global Optimization of Lennard–Jones Clusters for* $N \leq 201$ *Using the Conformational Space Annealing Method*. Physical Review Letters, Vol. 91, No. 8, p. 080201. 2003.

[11] S. Kirkpatrick et al. *Optimisation by Simulated Annealing*. Science, Vol. 220, pp. 671–680. 1983.

[12] M. von Arnim, R. Ahlrichs. *Geometry Optimization in Generalized Natural Internal Coordinates*. Journal of Chemical Physics, Vol. 111, pp. 9183–9190. 1999.

Appendix A

Derivations for Lennard–Jones Potential

A.1 Minimum Energy for Two Atoms

The formula for Lennard–Jones potential was given by Equation (2.1). If it is differentiated with respect to r, we get

$$\frac{dU(r)}{dr} = 24\varepsilon \left(\frac{\sigma^6}{r^7} - \frac{2\sigma^{12}}{r^{13}} \right) \tag{A.1}$$

For r_{min} this must be zero, so

$$24\varepsilon \left(\frac{\sigma^6}{r_{min}^7} - \frac{2\sigma^{12}}{r_{min}^{13}} \right) = 0 \tag{A.2}$$

Solving for r_{min} gives

$$\begin{aligned} \frac{\sigma^6}{r_{min}^7} &= \frac{2\sigma^{12}}{r_{min}^{13}} \\ r_{min}^6 &= 2\sigma^6 \\ r_{min} &= \sqrt[6]{2}\sigma \end{aligned} \tag{A.3}$$

If we substitute this into Equation (2.1), we get

$$U(r_{min}) = 4\varepsilon \left(\frac{1}{4} - \frac{1}{2} \right) = -\varepsilon \tag{A.4}$$

A.2 Forces for Multiple Atoms

The total potential energy for multiple atoms was given by Equation (2.2). Now we derive Equation (2.3) from it. We denote the jth component of vector $\vec{\boldsymbol{r}}_i$ by

r_{ij}. Differentiating Equation (2.2) with respect to r_{kl} we get

$$\begin{aligned}\frac{\partial E}{\partial r_{kl}} &= \frac{\partial}{\partial r_{kl}} \sum_{i=2}^{N} \sum_{j<i} U(|\vec{\boldsymbol{r}}_i - \vec{\boldsymbol{r}}_j|) \\ &= \frac{\partial}{\partial r_{kl}} \sum_{j=1}^{k-1} U(|\vec{\boldsymbol{r}}_k - \vec{\boldsymbol{r}}_j|) + \frac{\partial}{\partial r_{kl}} \sum_{i=k+1}^{N} U(|\vec{\boldsymbol{r}}_i - \vec{\boldsymbol{r}}_k|) \\ &= \frac{\partial}{\partial r_{kl}} \sum_{j\neq k} U(|\vec{\boldsymbol{r}}_k - \vec{\boldsymbol{r}}_j|) \\ &= \sum_{j\neq k} \frac{\partial}{\partial r_{kl}} U(|\vec{\boldsymbol{r}}_k - \vec{\boldsymbol{r}}_j|)\end{aligned} \tag{A.5}$$

Let us define

$$\rho_{kj} = |\vec{\boldsymbol{r}}_k - \vec{\boldsymbol{r}}_j| = \sqrt{\sum_{m=1}^{3} (r_{km} - r_{jm})^2} \tag{A.6}$$

Now

$$\frac{\partial E}{\partial r_{kl}} = \sum_{j\neq k} \frac{\partial U(\rho_{kj})}{\partial r_{kl}} = \sum_{j\neq k} \frac{\partial U(\rho_{kj})}{\partial(\rho_{kj}^2)} \frac{\partial(\rho_{kj}^2)}{\partial r_{kl}} \tag{A.7}$$

These partial derivatives can be calculated separately from Equations (2.1) and (A.6):

$$\begin{aligned}\frac{\partial U(\rho_{kj})}{\partial(\rho_{kj}^2)} &= 4\varepsilon \frac{\partial}{\partial(\rho_{kj}^2)} \left(\frac{\sigma^{12}}{(\rho_{kj}^2)^6} - \frac{\sigma^6}{(\rho_{kj}^2)^3} \right) \\ &= 12\varepsilon \left(\frac{\sigma^6}{\rho_{kj}^8} - \frac{2\sigma^{12}}{\rho_{kj}^{14}} \right)\end{aligned} \tag{A.8}$$

$$\begin{aligned}\frac{\partial(\rho_{kj}^2)}{\partial r_{kl}} &= \frac{\partial}{\partial r_{kl}} \sum_{m=1}^{3} (r_{km} - r_{jm})^2 \\ &= \frac{\partial}{\partial r_{kl}} (r_{kl} - r_{jl})^2 \\ &= 2(r_{kl} - r_{jl})\end{aligned} \tag{A.9}$$

Combining these and Equation (A.7) yields

$$\frac{\partial E}{\partial r_{kl}} = 24\varepsilon \sum_{j\neq k} \left(\frac{\sigma^6}{\rho_{kj}^8} - \frac{2\sigma^{12}}{\rho_{kj}^{14}} \right) (r_{kl} - r_{jl}) \tag{A.10}$$

Therefore Equation (1.2) gives

$$\begin{aligned}\vec{\boldsymbol{F}}_k = -\frac{\partial E}{\partial \vec{\boldsymbol{r}}_k} = -\begin{pmatrix} \frac{\partial E}{\partial r_{k1}} \\ \frac{\partial E}{\partial r_{k2}} \\ \frac{\partial E}{\partial r_{k3}} \end{pmatrix} &= 24\varepsilon \sum_{j\neq k} \left(\frac{\sigma^6}{\rho_{kj}^8} - \frac{2\sigma^{12}}{\rho_{kj}^{14}} \right) \begin{pmatrix} r_{j1} - r_{k1} \\ r_{j2} - r_{k2} \\ r_{j3} - r_{k3} \end{pmatrix} \\ &= 24\varepsilon \sum_{j\neq k} \left(\frac{\sigma^6}{\rho_{kj}^8} - \frac{2\sigma^{12}}{\rho_{kj}^{14}} \right) (\vec{\boldsymbol{r}}_j - \vec{\boldsymbol{r}}_k)\end{aligned} \tag{A.11}$$

which is essentially the same as Equation (2.3).

Appendix B

Technical Details on Implementation

B.1 Line Minimization Algorithm

As stated in Section 4.3, a one-dimensional minimization is performed between each BFGS step. Method `getNewX` of the BFGS minimizer class just keeps returning points along the line until the difference of function values evaluated at nearby located points is considered small enough. Then a new BFGS step is taken so that the minimum along the line is considered to be the point $\boldsymbol{x}_{i+1}$ in Equations (3.7) and (3.8).

The minimization algorithm itself is a slightly simplified version of the well-known Brent's algorithm. The algorithm takes as input two points, between which the minimum is supposed to be. In this case, we are trying to find the minimum of

$$g_i(t) = f(\boldsymbol{x}_i - t\boldsymbol{H}_i \cdot \nabla f(\boldsymbol{x}_i)) \qquad t \in [0, 1] \tag{B.1}$$

so these points are initialized to 0 and 1. The algorithm also maintains information about a third point which is located somewhere between the two end-points. Initially this central point is located halfway between the end-points. The algorithm tries then to find a concave parabola that passes through these three points. Depending on those points, we have two possibilities:

1. *Such parabola exists.* In this case the minimum point of that parabola is calculated, after which either of the end-points is discarded so that the minimum point of the parabola becomes the new central point which is located between the old central point and the other end-point. The iteration is continued with these values.

2. *No concave parabola passes through these points.* In this case the algorithm determines which end-point is further away from the central point. The corresponding interval is divided according to the golden section ratio (≈ 0.318966) and the function is evaluated at the dividing point. Now the algorithm is aware of function values at four distinct points. If the minimum of these values is found at either of the end-points, the other endpoint is discarded. Otherwise it is found at either of the points between

them and in that case the end-point that is not adjacent to the minimum point is discarded. The iteration is continues using the remaining three values.

Finally, after having calculated the minimum point t_{min} of $g_i(t)$ precisely enough, we select

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - t_{min}\boldsymbol{H}_i \cdot \nabla f(\boldsymbol{x}_i) \tag{B.2}$$

to be the next BFGS step.

B.2 Interface to the GSL Minimization Routines

The Python C API was used in order to make it possible to use the minimization routines provided by the GNU Scientific Library within this structural optimization framework. Although the Python C API itself was quite complicated, it was not the worst problem on creating this interface.

The approach used in the `Minimizers` module differs somehow from the one used in the GSL. In our `Minimizers` module every minimizer class has a method named `getNewX` which is called to acquire the next suggestion for the minimum point where the function and its gradient are evaluated during the next iteration step. In GSL, we provide the minimizer with C functions that evaluate the function and its gradient in the point given to it as an argument. However, although GSL provides a function[1] which is said to perform only one iteration per call, it usually calls the evaluation functions several times during one "iteration".

As every evaluation of the function to be minimized can consume a significant amount of time, I consider it better to have explicit control over how many times the function is actually evaluated. That is why the approach used in the GSL was not adopted to this system.

The interface works by forking a new process on creating a new minimizer instance. This new process then calls the GSL iteration function in a tight loop. The trick is to provide GSL with special evaluation functions that suspend the process and wait until `getNewX` method is called. `getNewX` then returns the point given to the evaluation function. When `getNewX` is called next time, the value of the function or its gradient is passed to the process performing the minimization. The other value is also cached so that if GSL happens to ask the values of both the function and its gradient, they are not calculated twice.

B.2.1 Source Code of Module `Minimizers.GSL`

Here is the complete listing of the commented source code of the extension module `Minimizers.GSL`. It may be used as a reference when implementing new C extensions to Python.

```
#include <Python.h>
#include <errno.h>
#include <gsl/gsl_multimin.h>
#include <gsl/gsl_vector.h>
#include <string.h>
```

[1] The name of this function is `gsl_multimin_fdfminimizer_iterate`.

```
#include <structmember.h>
#include <sys/socket.h>
#include <unistd.h>

/*
 * Data structure maintaining the internal state of the minimizer object.
 */
typedef struct {
  PyObject_HEAD      /* used by the Python interpreter */
  int variables;     /* variable count */
  int sock;          /* communication socket */
  int new;           /* first call to check_values? */
  double *x;         /* current posistion */
  double val;        /* cached function value at the current position */
  double *grad;      /* cached gradient value at the current position */
} GSL;

/*
 * Destructor for minimizer objects.
 */
void GSL_dealloc(GSL *self) {
  close(self->sock);
  self->ob_type->tp_free((PyObject *) self);
}

/*
 * GSL vector -> double array
 */
void vec_get(const gsl_vector *vec, double *arr, int size) {
  int i;
  for (i = 0; i < size; i++)
    arr[i] = gsl_vector_get(vec, i);
}

/*
 * Double array -> GSL vector
 */
void vec_set(gsl_vector *vec, double *arr, int size) {
  int i;
  for (i = 0; i < size; i++)
    gsl_vector_set(vec, i, arr[i]);
}

/*
 * Update the function and gradient values to cache by sending
 * the position to the socket and receiving the values from it.
 * Terminate the process if an EOF is read.
 */
void update_values(const gsl_vector *x, GSL *obj) {
  int i;
```

```
  /* store position */
  vec_get(x, obj->x, obj->variables);

  /* write position */
  if (!obj->new)
    write(obj->sock, obj->x, obj->variables * sizeof(double));

  /* read function value */
  i = read(obj->sock, &obj->val, sizeof(double));

  /* terminate if EOF */
  if (i == 0) {
    close(obj->sock);
    exit(0);
  }

  /* read gradient value */
  for (i = 0; i < obj->variables; i++)
    read(obj->sock, obj->grad + i, sizeof(double));
}

/*
 * Check if the cache already contains the correct values.
 * If not, call update_values.
 */
void check_values(const gsl_vector *x, GSL *obj) {
  int i;
  if (obj->new) {
    update_values(x, obj);
    obj->new = 0;
  }
  else for (i = 0; i < obj->variables; i++)
    if (gsl_vector_get(x, i) != obj->x[i]) {
      update_values(x, obj);
      break;
    }
}

/*
 * Returns the value of the function at x.
 */
double func(const gsl_vector *x, void *params) {
  GSL *obj = (GSL *) params;
  check_values(x, obj);
  return obj->val;
}

/*
 * Stores the value of the gradient of the function at x
```

```
 * to GSL vector g.
 */
void grad(const gsl_vector *x, void *params, gsl_vector *g) {
  int i;
  GSL *obj = (GSL *) params;
  check_values(x, obj);
  vec_set(g, obj->grad, obj->variables);
}

/*
 * Stores the value of the function to f and the value of its
 * gradient to g.
 */
void fgrad(const gsl_vector *x, void *params, double *f, gsl_vector *g) {
  *f = func(x, params);
  grad(x, params, g);
}

/*
 * This is called by the newly forked minimizer process.
 */
void minimizer_loop(GSL *obj, gsl_multimin_fdfminimizer *minimizer,
                    gsl_vector *initial_x) {
  gsl_multimin_function_fdf function;

  /* set up the state data structure */
  obj->new = 1;
  obj->x = (double *) malloc(sizeof(double) * obj->variables);
  obj->grad = (double *) malloc(sizeof(double) * obj->variables);

  /* set up the GSL function data structure */
  function.f = &func;
  function.df = &grad;
  function.fdf = &fgrad;
  function.n = obj->variables;
  function.params = obj;

  /* set up the GSL minimizer */
  gsl_multimin_fdfminimizer_set(minimizer, &function, initial_x, 1.0, 0.0);

  /* perform an adequate number of iterations... */
  while (1) gsl_multimin_fdfminimizer_iterate(minimizer);
}

/* Algorithms provided by the GSL */
#define ALG_NUM 4
const char *alglist[ALG_NUM] = {"conjugate_fr", "conjugate_pr",
                                "vector_bfgs", "steepest_descent"};

/*
```

```
 * Constructor for minimizer objects.
 */
int GSL_init(GSL *self, PyObject *args, PyObject *kwds) {
  static char *kwlist[] = {"x", "max", "algorithm", NULL};
  PyObject *x, *max;
  char *algorithm;
  int ok, i;
  int socks[2];
  const gsl_multimin_fdfminimizer_type *mintype;
  gsl_multimin_fdfminimizer *minimizer;
  gsl_vector *initial_x;

  /* make sure that the arguments remain in the memory
     during this function call */
  Py_INCREF(args);

  /* parse the arguments */
  ok = PyArg_ParseTupleAndKeywords(args, kwds, "OOs", kwlist,
                                   &x, &max, &algorithm);

  if (ok) {
    ok = 0;

    /* determine the algorithm to be used */
    for (i = 0; i < ALG_NUM; i++)
      if (!strcmp(algorithm, alglist[i])) {
        ok = 1;
        switch (i) {
        case 0:
          mintype = gsl_multimin_fdfminimizer_conjugate_fr;
          break;
        case 1:
          mintype = gsl_multimin_fdfminimizer_conjugate_pr;
          break;
        case 2:
          mintype = gsl_multimin_fdfminimizer_vector_bfgs;
          break;
        case 3:
          mintype = gsl_multimin_fdfminimizer_steepest_descent;
        }
      }

    if (ok) {
      /* convert lists to tuples */
      if (PyList_Check(x))
        x = PyList_AsTuple(x);

      /* determine variable count */
      self->variables = PyTuple_Size(x);
```

```
      /* create UNIX socket pair */
      if (socketpair(AF_LOCAL, SOCK_STREAM, 0, socks)) {
        ok = 0;
        PyErr_SetString(PyExc_OSError, strerror(errno));
      }
      else {
        /* fork new process */
        i = fork();
        switch (i) {
        case -1:
          ok = 0;
          PyErr_SetString(PyExc_OSError, "Process creation failed");
          break;
        case 0:
          /* new process: set up certain things and call minimizer_loop */
          close(0);
          close(1);
          close(socks[1]);
          self->sock = socks[0];
          minimizer = gsl_multimin_fdfminimizer_alloc(mintype, self->variables);
          initial_x = gsl_vector_alloc(self->variables);
          for (i = 0; i < self->variables; i++)
            gsl_vector_set(initial_x, i, PyFloat_AsDouble(PyTuple_GetItem(x, i)));
          minimizer_loop(self, minimizer, initial_x);
        }
        /* parent process */
        close(socks[0]);
        self->sock = socks[1];
      }
    }
    else PyErr_SetString(PyExc_ValueError, "Algorithm not found");
  }

  /* the arguments are no longer used (as Python objects) */
  Py_DECREF(args);

  return ok ? 0 : -1;
}

PyObject *GSL_getNewX(GSL *self, PyObject *args) {
  int i;
  PyObject *x, *grad;
  double val;
  Py_INCREF(args);

  /* parse the arguments and throw an exception if an error occurs */
  if (!PyArg_ParseTuple(args, "dO", &val, &grad)) {
    Py_DECREF(args);
    return NULL;
  }
```

```
  /* convert lists to tuples */
  if (PyList_Check(grad))
    grad = PyList_AsTuple(grad);

  /* check tuple size */
  if (PyTuple_Size(grad) != self->variables) {
    Py_DECREF(args);
    PyErr_SetString(PyExc_ValueError, "Gradient length is not correct");
    return NULL;
  }

  /* write the function value to the socket */
  if (write(self->sock, &val, sizeof(double)) == -1) {
    Py_DECREF(args);
    PyErr_SetString(PyExc_OSError, strerror(errno));
    return NULL;
  }
  /* write the gradient value */
  for (i = 0; i < self->variables; i++) {
    val = PyFloat_AsDouble(PyTuple_GetItem(grad, i));
    if (write(self->sock, &val, sizeof(double)) == -1) {
      Py_DECREF(args);
      PyErr_SetString(PyExc_OSError, strerror(errno));
      return NULL;
    }
  }

  Py_DECREF(args);

  /* create a new list to which the result is stored */
  x = PyList_New(0);
  if (x == NULL) return NULL;

  /* read the coordinates from the socket and append them to
     the result list */
  for (i = 0; i < self->variables; i++) {
    if (read(self->sock, &val, sizeof(double)) == -1) {
      PyErr_SetString(PyExc_OSError, strerror(errno));
      return NULL;
    }
    if (PyList_Append(x, Py_BuildValue("d", val)))
      return NULL;
  }

  return x;
}

/*
 * Method list for the minimizer class.
```

```
 */
static PyMethodDef GSL_methods[] = {
  {"getNewX", (PyCFunction) GSL_getNewX, METH_VARARGS, ""},
  {NULL}
};

/*
 * Properties for the minimizer class.
 */
static PyTypeObject GSLType = {
    PyObject_HEAD_INIT(NULL)
    0,                         /* ob_size */
    "Minimizers.GSL.GSL",      /* tp_name */
    sizeof(GSL),               /* tp_basicsize */
    0,                         /* tp_itemsize */
    (destructor) GSL_dealloc,  /* tp_dealloc */
    0,                         /* tp_print */
    0,                         /* tp_getattr */
    0,                         /* tp_setattr */
    0,                         /* tp_compare */
    0,                         /* tp_repr */
    0,                         /* tp_as_number */
    0,                         /* tp_as_sequence */
    0,                         /* tp_as_mapping */
    0,                         /* tp_hash */
    0,                         /* tp_call */
    0,                         /* tp_str */
    0,                         /* tp_getattro */
    0,                         /* tp_setattro */
    0,                         /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT,        /* tp_flags */
    "GSL Minimizer",           /* tp_doc */
    0,                         /* tp_traverse */
    0,                         /* tp_clear */
    0,                         /* tp_richcompare */
    0,                         /* tp_weaklistoffset */
    0,                         /* tp_iter */
    0,                         /* tp_iternext */
    GSL_methods,               /* tp_methods */
    0,                         /* tp_members */
    0,                         /* tp_getset */
    0,                         /* tp_base */
    0,                         /* tp_dict */
    0,                         /* tp_descr_get */
    0,                         /* tp_descr_set */
    0,                         /* tp_dictoffset */
    (initproc) GSL_init,       /* tp_init */
    0,                         /* tp_alloc */
    0,                         /* tp_new */
};
```

```
/*
 * Module method list (empty).
 */
static PyMethodDef module_methods[] = {
  {NULL}
};

/*
 * Module initialization function.
 */
void initGSL() {
  PyObject *m;

  /* use default allocator */
  GSLType.tp_new = PyType_GenericNew;

  if (PyType_Ready(&GSLType) < 0) return;

  /* initialize the module */
  m = Py_InitModule3("Minimizers.GSL", module_methods,
                     "GSL Minimizer module");
  if (m == NULL) return;

  /* initialize the GSL class */
  Py_INCREF(&GSLType);
  PyModule_AddObject(m, "GSL", (PyObject *) &GSLType);
}
```

Appendix C

Optimal Structure Plots for Lennard–Jones Clusters

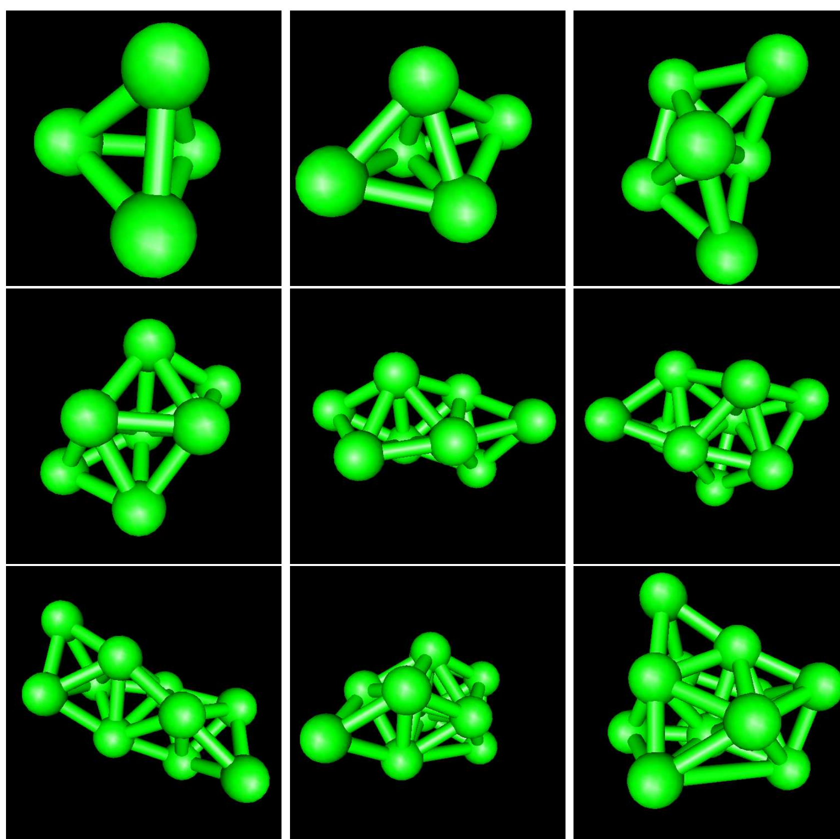

Figure C.1: Equilibrium structures (cluster size 4–12)

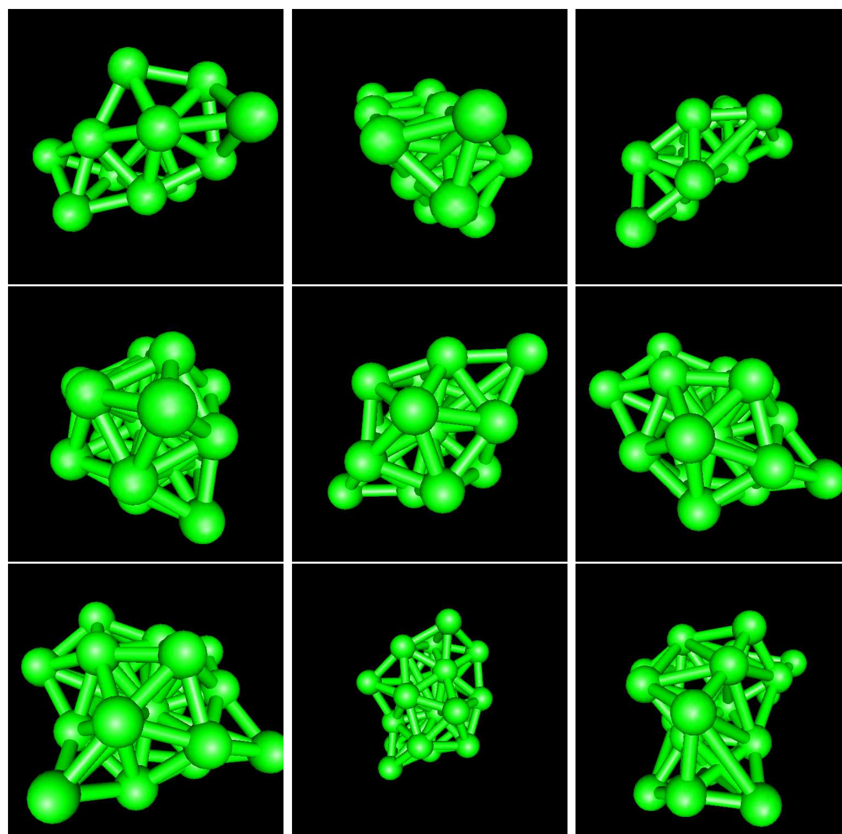

Figure C.2: Equilibrium structures (cluster size 13–21)