Eötvös Loránd Tudományegyetem
Informatikai Kar
Információs Rendszerek Tanszék

Developing software calculating quantum systems' wave functions

János Márk Szalai-Gindl
Egyetemi tanársegéd

Antal Száva
Programtervező Informatikus BSc

Budapest, 2017

Acknowledgements

I would like to thank Professor István Csabai and my supervisor, János Márk Szalai-Gindl for the constant help and dedicated support I received during the making of this thesis.

Contents

Chapter 1

Introduction

1.1 Scientific background

Tens of thousands of scientific observations are made each day, as mankind seeks to gather more and more information about the world we live in. In an attempt to perceive and process every part of it, state of the art technologies are used in certain cases. Consequently, understanding problems, carrying out measurements while searching for solutions may need the support of contemporary computational technologies. An example for that is determining the exact molecular interactions inside the human body in order to try getting closer to curing specific diseases. The topic of this thesis is software connected to a specific scientific method used in quantum chemistry. Being able to solve the underlying problem of this particular theory allows the thorough observation of real life examples.

In quantum mechanics, apart from classical characteristics, particles can be described by waves too. The specific location and momentum of particles, like the electrons in a molecule, cannot be determined simultaneously at a given time (Heisenberg's uncertainty principle).[1] Only the probability of possible locations can be given. Instead of the classical model of electrons revolving around the nucleus of a given atom, in this probabilistic representation electrons are described by wave functions. The most likely locations for electrons are described by the concept of atomic orbitals, spherical distributions around the nucleus of an atom. The various atomic orbitals can have different energy levels depending on the type of the atom, the distance of orbitals from the nucleus and its symmetry.[1]

To determine the energy levels of complex molecules (for example when considering proteins inside the human body) one needs to take into account all the nuclei and all the electrons. Electrons are not allocated to individual atoms anymore, instead they are shared by many atoms and can be described by molecular orbitals.[2]

A molecular orbital can be represented as the combination of atomic orbitals from each atom making up the molecule. [1] In theory, atomic and molecular orbitals can be calculated by solving the Schrödinger equation, which can be written as a large linear algebra problem, essentially finding eigenvectors and eigenvalues of the Hamiltonian matrix. [1] Unfortunately, except for very simple cases, no analytic solution is known and for large molecules like proteins even the numerical solution cannot be carried out by current technologies. For this reason, approximations are required.

The Hückel method uses the linear combinations of atomic orbitals in determining the energies of certain molecular orbitals inside of a molecule. Originally it only considered electrons with specific bonds located in pi orbitals. In 1963, Roald Hoffmann extended this theory by also taking into account the other outermost electrons of the molecular models, thus including all valence electrons. [3] With this extension, this is the so-called extended Hückel method, the subject of this thesis work. Considering the mathematics behind the theory, the overlappings of atomic orbitals inside a molecule are described by so-called overlap matrices. The overlap matrix and a Hamiltonian matrix (both are special kinds of complex matrices) are used during the calculations. [4]

To initialize the calculations, the matrices and the atomic orbitals can be relatively easily calculated from the 3D positions of the atoms and the time-consuming part of the program is the solution of the linear algebraic equations. The 3D positions can be obtained from certain model considerations, or more often from various physical measurements (X-ray crystallography or NMR) of the molecules. These structures are deposited into public databases, like Protein Data Bank (PDB) as will be described later. In a typical scenario, researchers take 3D structures from PDB and after some preparatory steps use this data as input for the extended Hückel calculation which outputs energy levels (eigenvalues) and molecular wave functions (eigenvectors).

1.2 Important file formats

The following file formats have a significant role in this project. The PDB and YAH formats are used as input file formats, whereas the Matrix Market is an optional, yet favorable output format supported by the application.

1.2.1 PDB format

A **Protein Data Bank (PDB)** formatted file describes molecules with a three-dimensional structure, using various keywords of standardized order. With this description method, protein and nucleic acid structures can be specified. Coordinates of atoms, connections between them, ligands and sidechain rotamers are characteristics that these files may contain and thus uniquely determine a molecule. The file format was named after a database of proteins, the **Protein Data Bank**.

Certain keywords (HEADER, TITLE and AUTHOR) are used for the description of main aspects of the file, namely the protein's name, the scientists having put together this structure into the file and further information.[7]

Each PDB file may contain several models for proteins. These models start with the **MODEL** keyword and continue with a serial number for the model. The **ENDMDL** word indicates the end of such a section.[8]

Coordinates may be described with two different keywords: **ATOM** or **HETATM**. The key to their difference is that ATOM records are used for standard chemicals whereas for those amino acids and nucleotides that are non-polymer, the HETATM keyword is to be included in the beginning of each line. These coordinate lists are ended with the **TER** keyword.[9]

One of the values describing the position of an atom is called occupancy. If its value is 1, then based on the X-ray crystallography the position of a specific atom is certain. Sometimes there were observations when it was situated at a given set of coordinates, but at other times was found in another position. In these cases the atom might appear more than once in a PDB file and then its occupancy value is less than 1, the sum of all the values being equal to 1. [10]

Connectivity among the atoms may be described using the **CONECT** keyword.[11]

```
HEADER    DNA                                           26-JAN-81    1BNA
TITLE     STRUCTURE OF A B-DNA DODECAMER. CONFORMATION AND DYNAMICS
...
AUTHOR    H.R.DREW,R.M.WING,T.TAKANO,C.BROKA,S.TANAKA,K.ITAKURA,
AUTHOR   2 R.E.DICKERSON
...
ATOM      1  O5' DC A   1      18.935  34.195  25.617  1.00 64.35      O
ATOM      2  C5' DC A   1      19.130  33.921  24.219  1.00 44.69      C
ATOM      3  C4' DC A   1      19.961  32.668  24.100  1.00 31.28      C
ATOM      4  O4' DC A   1      19.360  31.583  24.852  1.00 37.45      O
ATOM      5  C3' DC A   1      20.172  32.122  22.694  1.00 46.72      C
...
ATOM    487  C4  DG B  24      17.231  22.893  27.570  1.00 13.89      C
TER     488      DG B  24
HETATM  489  O   HOH A  25      19.736  30.706  18.656  1.00 51.86      O
...
HETATM  568  O   HOH B 104      18.692  31.584   4.596  1.00 72.98      O
MASTER  355    0    0    0    0    0    0    6  566    2    0    2
END
```

Listing 1.1: Example PDB file for the deoxyribonucleic acid

1.2.2 YAH format

The **YAH** format is a chemical format containing the description of molecule and is the input configurations for the **YAeHMOP** application. Although there are many possible ways to describe details of molecules in such files, in this thesis only certain cases of these possibilities will be discussed.

The files can be divided into two main parts: first, using the **Geometry** keyword and disclosing the total number of atoms, their coordinates are listed with the possibility of using different input formats. After details on the positions of the atoms, come the printing options specifying the types of calculations to be carried out. An extended version of the YAH format was used during the development process. Keywords in use are found in the Input section of the User Documentation.

```
1BNA
Molecular
Geometry
566
O 18.935 34.195 25.617
C 19.13 33.921 24.219
C 19.961 32.668 24.1
O 19.36 31.583 24.852
C 20.172 32.122 22.694
O 21.35 31.325 22.681
...
O 14.292 25.159 2.287
O 9.396 27.092 16.993
O 19.987 21.691 6.802
O 18.692 31.584 4.596
Charge
0

Print
Orbital Mapping
Overlap
Hamil
End_print

Dump Hamil
Dump Overlap
Dump Matrix Market
```

Listing 1.2: Example YAH file for the deoxyribonucleic acid

1.2.3 Matrix Market format

The **Matrix Market (MM)** file format is widespread for representing certain types of matrices. One of the main purposes of using it is to describe sparse matrices, matrices that have almost only zero elements. When carrying out computations with matrices, we are representing their elements using arrays or multidimensional arrays depending on the dimensions of the original matrices. This way, however, when dealing with a sparse matrix, it might happen that in most cases we are allocating memory to store zero elements. Thus, it can be concluded that this technique is not affordable in terms of memory management. Consequently, using the characteristics of sparse matrices, it serves as a better representational model to only store the coordinates of the elements that are non-zero and their respective values.

The previously described concept is the general idea giving the basis for the Matrix Market format as well. When representing a matrix in the MM format, the user needs to give the format of the matrix in the first line of the file: either as coordinate or as array. This decision depends on whether the matrix to be represented is sparse or dense; the coordinate format is to be used for sparse matrices.

The next property to be given after a non-breaking space delimiter is the matrix qualifier describing the type of the elements in the matrix. Alternatives include **Real** and **Complex**, names denoting the types of matrices represented **Integer** for those with only integer elements. Users can also choose a **Pattern** qualifier for cases when a pattern determines the nonzero elements. Then information regarding the symmetry of the matrix is disclosed: **general**, **symmetric and hermitian** or **skew-symmetric**.[12]

This is followed by the structured comments part, a section extending previous versions of this format. Useful for providing more information regarding the file, this section may also contain customized comments regarding the matrix.

The final part of a Matrix Market formatted file is the description of coordinates. The first line of this section refers to the dimensions followed by the number of nonzero elements in the matrix. Then come the nonzero elements in each line with the standard structure of rownumber, columnnumber, value of matrix element.

Example for the Matrix Market format[13]:

```
1     0       0       6       0
0   10.5      0       0       0
0     0     .015      0       0
0  250.5      0     −280    33.32
0     0       0       0      12
```

Listing 1.3: Example sparse matrix of 5 dimensions

```
%%MatrixMarket matrix coordinate real general
%=================================================================
%
% This ASCII file represents a sparse MxN matrix with L
% nonzeros in the following Matrix Market format:
%
% +----------------------------------------------+
% |%%MatrixMarket matrix coordinate real general |
% |%                                             |
% |% comments                                    |
% |%                                             |
% |    M  N  L                                   |
% |     I1  J1  A(I1, J1)                        |
% |     I2  J2  A(I2, J2)                        |
% |     I3  J3  A(I3, J3)                        |
% |         . . .                                |
% |     IL JL  A(IL, JL)                         |
% +----------------------------------------------+
%
% Indices are 1-based, i.e. A(1,1) is the first element.
%
%=================================================================
  5  5  8
    1      1    1.000e+00
    2      2    1.050e+01
    3      3    1.500e-02
    1      4    6.000e+00
    4      2    2.505e+02
    4      4   -2.800e+02
    4      5    3.332e+01
    5      5    1.200e+01
```

Listing 1.4: Market Matrix file for the example matrix

Chapter 2

User Documentation

2.0.1 Introduction

The program created during this thesis work is an extensive, high performance software made for supporting various chemical formats while carrying out extended Hückel calculations. Requiring preliminary knowledge in certain scientific fields, it was created for individuals aiming to investigate energy levels of molecules for research purposes. End users include experts, people of academia and scientists. These functionalities provided by this software will be used as part of a university project, the application was produced with the intention of assisting scientific activities. All observations made during the creation are documented herein to serve the same purpose.

The application may be used for the following purposes:

- convert between molecular formats by integrating,
- carry out extended Hückel method calculations,
- use a graphic user interface providing further data processing possibilities,
- use the sparse matrix output format for the results of the calculations.

2.0.2 System requirements

Certain parts of the software were based on the code of already existing open source applications. Specific examples for their usage are discussed later on in the relevant chapters, they come integrated in the application.

As far as the hardware requirements are concerned, although multi-core processors are not required for the application to run, it is highly recommended to use computers equipped with such CPUs to exploit resources and obtain a higher performance. The examples shown in this thesis were run on a system with 6 GB memory. Though there are no standard requirements for execution, the application highly depends on computer resources whilst running certain configuration options. Thus for robust inputs, less or more memory may result in change of run success due to runtime memory allocation.

The program was created and tested on an **Ubuntu 16.04 LTS** system (4.4.0-43-generic: Xenial).

2.0.3 Dependencies

Below is a list of dependencies that are needed for the run of the application and their form of installation:

- **CMake 2.6** or newer
- **C++ 14 compiler**, ideally chosen from the following list (with OpenMP support): `http://www.openmp.org/resources/openmp-compilers/`
- **Boost** (`http://www.boost.org/users/download/`)

  ```
  sudo apt-get install libboost-all-dev
  ```

- **OpenBlas** (contains LAPACK with optimized sources as default, further information at `http://www.openblas.net/`)

  ```
  sudo apt-get install libopenblas-dev
  ```

  Please note that other existing LAPACK and BLAS distributions on the system may conflict with the run of the application using OpenBlas.

- **OpenBabel** (see following section for installation guide)
- **Python 3.3 or newer** (found at `https://www.python.org/downloads/`)
- *(optional)* **Jupyter Notebook** (installation guide found at `https://jupyter.readthedocs.io/en/latest/install.html`)

2.0.4 Open Babel installation

Openbabel is a package of tools for operations with chemical formats. Though the Openbabel API is integrated into the application discussed in this thesis, the following installation configurations are required for it to function.

During the thesis work the 2.4.1 version was used, this package can be acquired by downloading it using the following link: `https://sourceforge.net/projects/openbabel/files/openbabel/2.4.1/openbabel-2.4.1.tar.gz/download`

As the package is compressed to use up less storage, users will need to have a tool that can carry out decompression for the .tar.gz gzip format. Ubuntu's standard tool, the Archive Manager (File Roller) may be used for these purposes. Another alternative is to open a terminal in the folder that contains the compressed package and run the following command:

```
tar -xvzf openbabel-2.4.1.tar.gz
```

After this, setting up the Openbabel software can be done by following the installation instructions contained in the Install file found in the Openbabel-2.4.1 folder that has just been decompressed. These instructions include creating a build directory as well as using cmake (CMake 2.4.8 or later required) and then the makefile to compile and install the package.

One important thing to notice is the possibility of specifying options before using cmake. An example for that is that users may determine the directory where OpenBabel will be installed. If not specified, the /usr/local/ directory will be the default. This, however, means that it might require superuser privileges for one to use the package, or as described later on in this document, to integrate the API of the software as a package. Thus the following command with the aforementioned option is advisable (users may choose a target that is in their home folder):

```
cmake -DCMAKE_INSTALL_PREFIX=/home/username/openbabel ..
```

```
make install
```

Having successfully installed the package, environmental variables may need to be set up in order for it to be used (as described in the Install file):
BABEL_LIBDIR - the location of Open Babel format plugins
BABEL_DATADIR - the location of the data files

For personal configuration the BABEL_LIBDIR environmental variable had to be set. Generally, the following command can be used to set it ($LIB_PATH stands for the absolute link of your lib folder inside your extracted openbabel folder):

```
export BABEL_LIBDIR=$BABEL_LIBDIR:$LIB_PATH
```

Finally, the following command was executed for the sample input of a folded titin molecule, to convert between formats. Keep in mind that superuser rights may be needed for the output file to be written, if no user specified install location was given.

```
sudo obabel Titin-folded-monomer.sdf -O Titin-folded-monomer
    .pdb
```

2.0.5 Installation process

Sources related to the application may be found on the CD attached to this document or by visiting the following Github open repository: `https://github.com/antalszava/HuckelCode`.

Upon having the correct dependencies installed, the set up process of the application is simple.

Before compilation of the modules, the following steps need to be taken: The CMakeLists.txt in the **huckelcode** folder needs to be changed so that instead of "usr/lib", the path to the LAPACK and BLAS packages is set in the following two lines:

```
set(BLAS_DIR /usr/lib/)
set(LAPACK_DIR /usr/lib/)
```

Similar to this, in the **input** folder, a minor change needs to be done in the CMakeLists.txt file. The system-specific path of the OpenBabel shared object needs to be set in the following line:

```
target_link_libraries(input_pdb /home/user/openbabel/lib/
    libopenbabel.so)
```

Listing 2.1: Example path of /home/user/openbabel/lib/ for the shared object

After this, each module can be compiled by navigating into their folders and executing the following commands:

```
cmake .
```

This command has created the Makefile to be used by executing the next simple command:

```
make
```

This way executable files were created. After this, copy the two executable files named **bind** and **input_pdb** into the **user_interface** folder. From here, through the terminal with the use of **HuckelCode.py** or with the **HuckelCode.ipynb** Jupyter Notebook, the application may be launched.

2.1 Input

The input for the application may be of two types. The aim is to create the YAH format for the calculation units to process the molecule and its content. If the molecule was not given in this particular format, then an input conversion will take place after which the calculation process commences. If the input is a format other than YAH (such as PDB or SDF), then two separate files are needed. These two files are:

1. A molecule file of the OpenBabel supported formats and
2. A configuration file containing the printing options in the correct format.

Files of certain molecules can be obtained for example from the website of the Protein Data Bank.

2.1.1 Configuration file

The **configuration file** contains the name of the molecule, followed by the list of the previously mentioned printing options and a "true" or "false" value with a "=" delimiter in the format presented below. This file is either created manually using a sample configuration file, or it can be generated using the Jupyter Notebook of the application.

Descriptions that are given in the YAeHMOP documentation with further configuration keywords for printing options:

- *"**distance matrix** - Print the distance matrix*
- ***overlap population** - Print the Mulliken overlap population matrix*
- ***reduced overlap population** - Print the Mulliken reduced overlap population matrix*

- ***charge matrix*** - *Print the charge matrix*
- ***wave functions*** - *Print the wave functions for the molecule*
- ***net charges*** - *Print the net charges on the atoms, as determined using Mulliken population analysis*
- ***overlap*** - *Print the overlap matrix*
- ***hamil*** - *Print the Hamiltonian matrix*
- ***electrostatic*** - *Print the electrostatic contribution to the total energy*
- ***levels*** - *Toggles the printing of energy levels at each k point in an extended calculation*
- ***fermi*** - *Print the Fermi energy (this is primarily useful in combination with the Walsh option, described below)*
- ***orbital energy*** - *Allows the energy of a particular orbital to be printed*
- ***orbital coeff*** - *Allows the coefficient of a particular atomic orbital in a given molecular orbital to be printed*
- ***orbital mapping*** - *Generates the scheme used to number the individual atomic orbitals in a calculation"*[6]
- **dump overlap** - Creates sparse formatted binary output for the Hamiltonian and overlap matrices
- **dump hamil** - Creates sparse formated binary output for the Hamiltonian and overlap matrices
- **dump sparse** - Creates sparse formated binary output for the Hamiltonian and overlap matrices
- **dump matrix market** - Creates sparse Matrix Market formatted binary output for the Hamiltonian and overlap matrices

```
molecule name=1bna
distance matrix=true
overlap population=false
reduced overlap population=true
charge matrix=false
wave functions=false
net charges=false
overlap=false
hamil=false
electrostatic=false
levels=false
fermi=false
orbital energy=false
orbital coeff=false
orbital mapping=false
dump overlap=true
dump hamil=true
dump sparse=false
dump matrix market=true
```

Listing 2.2: Sample configuration file for the deoxyribonucleic acid

2.2 User Interface

Users have two possibilities for running the application:

1. running a Python script in command line,
2. using a Jupyterhub notebook.

In either of the cases the *bind* and the *input_pdb* executables need to be in the same folder as the *.py* or *.ipynb* file.

2.2.1 Python script

Users may use a joint version of the modules by using the Python 3 script created for the application. There is, however, an opportunity to use the modules separately. In that case the option needs to be specified as an argument.

The following command supposes that a Python 3 version is currently in the path of the system.

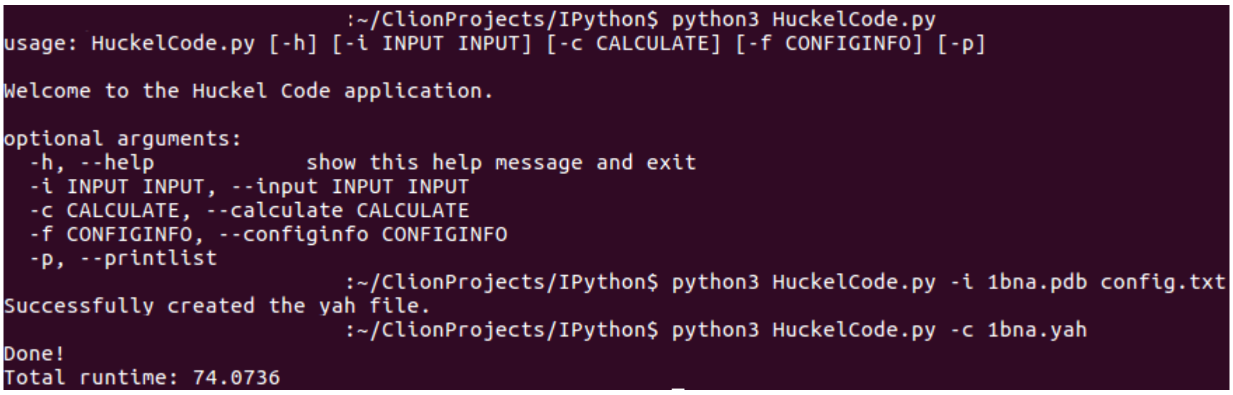
```
python HuckelCode.py -i 1bna.pdb config.txt
```

Listing 2.3: Command for using HuckelCode.py

Available options for the application:

- default: Calls the input and the calculate modules, needs a molecule file and a configuration file as input
- -i, –input: Calls only the input module on arguments, needs a molecule file and a configuration file as input
- -c, –calculate: Calls only the calculation module, needs a .yah file as input
- -p, –printlist: Lists the printing options that can be specified in the configuration file
- -f, –configinfo: Information regarding the configuration file
- -h, –help: Show this brief help message

Users may interrupt the running process by hitting Ctrl+C.



```
                           :~/ClionProjects/IPython$ python3 HuckelCode.py
usage: HuckelCode.py [-h] [-i INPUT INPUT] [-c CALCULATE] [-f CONFIGINFO] [-p]

Welcome to the Huckel Code application.

optional arguments:
  -h, --help            show this help message and exit
  -i INPUT INPUT, --input INPUT INPUT
  -c CALCULATE, --calculate CALCULATE
  -f CONFIGINFO, --configinfo CONFIGINFO
  -p, --printlist
                           :~/ClionProjects/IPython$ python3 HuckelCode.py -i 1bna.pdb config.txt
Successfully created the yah file.
                           :~/ClionProjects/IPython$ python3 HuckelCode.py -c 1bna.yah
Done!
Total runtime: 74.0736
```

Figure 2.1: User interface of a terminal running the Python script of the application

2.2.2 Jupyter Notebook

Allowing the use of an interactive environment, the Jupyter Notebook offers users possibilities to easily integrate other applications into the same Python environment as this application. Furthermore, mathematical data can be visually represented for example using the **Matplotlib** module. It also allows use of **bash** commands by using the *%%bash* magic command.

```
jupyter notebook --notebook-dir="/home/user/huckelcode/"
```

Listing 2.4: Command for starting Jupyter Notebook at target path

After the execution of this command, the resulted URL needs to be copied to the address bar of a browser. Having navigated into the directory containing the ipynb file, it can be launched to start the application.

2.2.3 System messages

- Please provide two filenames: one for your molecule file and one for your configuration file: the input for the application consists of two files, the user only gave one as input
- Formats not currently available in OpenBabel: Format1 and Format2: the conversion between molecular formats is done with the help of OpenBabel, a format was given that is not one of the more than 100 supported chemical formats
- Cannot open input file (*fileName*) for conversion with OpenBabel, please check if it exists in the current folder.: non-existent input file was given for conversion
- Cannot open output file for conversion with OpenBabel, please check if you have rights to write the file.: the OpenBabel API was unable to access the file specified at the beginning of program run
- Cannot open output file for .yah conversion, please check if you have rights to write the file.: the input module was unable to access the file specified at the beginning of program run
- No atoms found in the PDB file. Please check if it has correct lines starting with the ATOM or HETATM keyword.: the PDB parser was unable to find standard PDB atomic structure in the file
- Configuration file (*fileName*) does not exist.: the system was unable to locate the given configuration file
- Cannot open (*fileName*) for writing the properties, please check if it exists in the current folder.: the settings could not be written as the file is non-existent or the application has no rights to write it
- Done! Total runtime: (*time*): after the calculation module finishes, it outputs the total time spent
- Successfully created the yah file.: the input module created the input file for the calculation module

Chapter 3

Developer Documentation

3.1 Solution plan

3.1.1 System plan

The goal of this thesis was to create an improved extended Hückel code and discuss the observations made during the development of the software. Aims for the extensive program were for it to be able

- to consume standard chemical input formats,
- to handle large molecules with thousands of atoms,
- to utilize calculations with higher performance on multi-core machines,
- to output large matrices in sparse format for later processing.

As mentioned in the introductory part of the thesis, a need for the creation of such a program arose because the existing pieces of code were written using outdated forms of technology and architecture.

Following the main aims for creating this specific program, the plan for the whole system can be described easily. The development of this software constitutes as an extension and redesigning to the existing YAeHMOP application. On the one hand, this decision of integrating an altered version of the previously produced software and not recreating it was made due to the complex nature of the concerned field. Another aspect for that was to remain in accordance with the idea of using tested software with the exact same functionalities rather than creating new that may lead to possible difficulties.

Consequently, various architectural and implementation decisions were made in order to keep to the extant technologies. However, this phenomenon did not limit or hinder in any way the opportunities to make use of contemporary concepts used during software development, such as the complete separation of integrated modules or possible opportunities for future maintenance.

Version control system

During the work, the development process needed control in some way and so the issue of easily updating the previous version was to be solved. For this purpose, the use of a contemporary version control system was needed. This also solved the backing up of the code base that was being created from time to time, allowing the restoration of previous versions if needed.

Input

Following the ideas of separation of integrated modules and maintenance opportunities a stand-alone module of handling the input was needed. A plan for such a unit was created with the intention of it tending to two main types of tasks. The first one was the parsing of files with chemical formats. Due to the common use cases of working with several formats of the field, this criterion was to be fulfilled so that the module supported a great number of formats in use. As any widespread chemical format has a standardized, yet different way of containing data, the conversion of properties contained in these data sets was the task. Since the completion of this requirement posed a robust problem to be solved manually, the possible use of a conversion tool appeared to be feasible.

After parsing the inputs, the purpose of the input module was the creation of a file that can serve as input for the calculation module. It did not only need to process the data obtained from the file just read, but also include the printing options that were to be used during the run of the program. The separation of these tasks was essential and so, the printing options could be configured in some form of configuration unit. This fundamental decision in design gave users the ability to run calculations for the same file with different configuration files.

Redesigning YAeHMOP

The main module responsible for carrying out the extended Hückel calculations was in need of redesigning due to reasons mentioned in the introductory sections of this document. This part of the application was created based on a chemical program called YAeHMOP produced for computations in the field of quantum chemistry.

While the reasons for creating a rework for this product were mentioned previously, the ultimate aim was enhancing usability and development opportunities with the use of newer technologies. Two fundamental examples to these were the use of 64 bit architecture used by contemporary computers and the exploitation of multi-core processors.

When considering the modules of the entire YAeHMOP, the tightbind module is the part of the vast program which performs the complex calculations. Further references to YAeHMOP will refer to the tightbind module.

Output

The output of the YAeHMOP program can be specified with the use of keywords. Considering the several useful output opportunities provided by the initial design, one may conclude that some of the important ones are still lacking. In order to fulfill expectations that emerged throughout the years, such as a need for memory management, brevity and usability of the files created by the application, a demand for extended or replaced output options arose. The main reason for such reconsiderations was that the previously described Hamiltonian and overlap matrices in many cases belong to the set of sparse matrices and so may be subject to sparse format representation. This would yield end files of smaller size.

3.1.2 C++

The C++ programming language was chosen as the main technology for creating the software, allowing easy integration with the rework of YAeHMOP, which was initially written in C with calls to Fortran modules. This decision was made on the one hand partly because this language supports modern architectures and computer designs, on the other hand it has an evergrowing development in terms of language capabilities.

In 2011 and then in 2014 new standards of C++ were introduced and approved by the International Organization for Standardization (ISO), introducing new language concepts such as move semantics, lambda expressions and support for concurrency. Apart from these entirely new elements in the language, other new keywords have emerged as well, replacing or extending the former functionalities previously used language elements had. The specific language elements used will be specified at the relevant parts later on in this document.

3.1.3 Make and CMake

Make is a tool used for compiling and recompiling of programs useful for handling larger programs with many source files. It uses specific files called makefiles that describe how the different files of the program are related to each other and contains all commands that need to be run. In this terminology, there are two main types of files: targets that are to be updated by make and the preexisting prerequisites upon which targets depend in this process. [17] By calling the make command, users may compile and link the program. This tool is independent of programming languages, compiler directives are included in makefiles.[18]

Cmake is a building tool for C++ source files using makefiles to create executables. It has the major advantage of being cross-platform, as well as open-source. Users may create CMakeLists.txt configuration files, in which they can specify source files and link libraries statically and dynamically to the program by using predefined commands.[19]

During the development of the thesis work the CLion Integrated development environment was used, which is suited for creating software in C or C++ with the use of CMake files.

3.1.4 System architecture

Regarding the system architecture of the entire program, the previously introduced modules required a top layer or some other kind of integration, so that the communication between them was enabled. This interaction entails the provision of the file created by the input module for the calculation module. Once this module is finished with its task, the results of computations need to be forwarded to the output module. Later on, this is the part of the program, which creates the end product ready for further usage.

Some of these interaction cases were solved with the use of C++ integration, whereas others required scripting done with Python or Shell Script. The user interface is provided using these latter scripting technologies. Specific cases for the solutions will be discussed later on when expanding on the implementation of modules.

3.1.5 Structure of modules

Input module

- **main() function of the Input module**:
  1. **Input**: file of chemical format and configuration file of specific format
  2. **Output**: file of YAeHMOP format
  3. **Action**: instantiates classes of the input module and solves the passing of data between the objects

- **convert() function of the OpenBabelUtils class**:
  1. **Input**: file of chemical format
  2. **Output**: file of standard PDB format
  3. **Action**: converts a file with a supported format to standard PDB format, leaves it intact if originally PDB

- **convert() function of the PdbParser class**:
  1. **Input**: file of standard PDB format
  2. **Output**: partial file of YAeHMOP format
  3. **Action**: writes the atoms and their basic descriptions to a newly created file

- **readSettings() function of the SettingsParser class**
  1. **Input**: file of specific format containing configuration properties
  2. **Output**: properties data contained in the SettingsParser object
  3. **Action**: reads the configuration file and stores the properties described in it

- **writeSettings() function of the SettingsParser class**
  1. **Input**: properties data contained in the SettingsParser object
  2. **Output**: complete file of YAeHMOP format
  3. **Action**: completes the file previously created by the PdbParser with the stored properties

Redesigned YAeHMOP module

- **main() function of the Redesigned YAeHMOP module**

  1. **Input**: file of YAeHMOP format
  2. **Output**: possible output files of ASCII or binary format containing various dense formatted matrices and further details of the extended Hückel operation
  3. **Action**: carries out various calculations on the molecule contained in the input file, based on the initial printing options defined creates output file(s)

Output module

- **createBinaryFile() function of the Output module**

  1. **Input**: sparse matrix
  2. **Output**: binary file containing the nonzero elements and coordinates of the matrix in a sparse Matrix Market format
  3. **Action**: iterates through the sparse matrix only keeping details of nonzero elements, eventually creating a sparse Matrix Market formatted file

- **createDumpFile() function of the Output module**

  1. **Input**: sparse matrix
  2. **Output**: binary file containing the dump of nonzero elements and coordinates of the matrix
  3. **Action**: iterates through the sparse matrix only keeping details of nonzero elements and creates a binary dump of these data

The following Unified Modeling Language diagram contains the functionalities for the entire program.

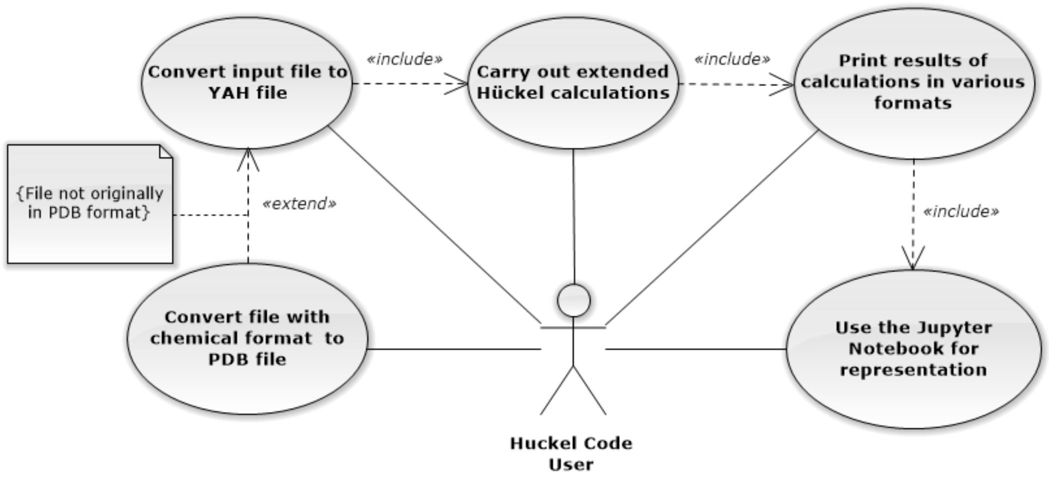
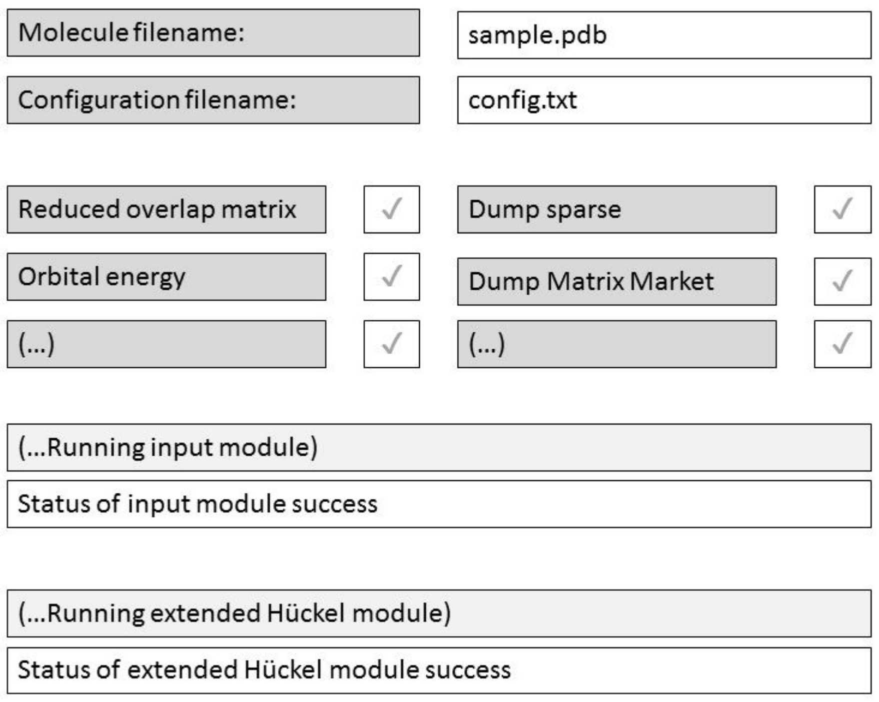

Convert input file to YAH file
«include»
Carry out extended Hückel calculations
«include»
Print results of calculations in various formats
{File not originally in PDB format}
«extend»
«include»
Convert file with chemical format to PDB file
Use the Jupyter Notebook for representation
Huckel Code User

Figure 3.1: Application use cases

3.1.6 Plan for user interface

The completed program required a simple user interface that is user-friendly and allows easily comprehensible running options. For this purpose, two platforms were chosen. One of these is the **Python 3 script**, easily fit for running the application with explanatory messages. Another, more advanced interface is the **Jupyter Notebook** providing users with a python kernel, while also granting them further programming and representation opportunities.



Molecule filename:	sample.pdb
Configuration filename:	config.txt

Reduced overlap matrix	✓	Dump sparse	✓
Orbital energy	✓	Dump Matrix Market	✓
(...)	✓	(...)	✓

(...Running input module)

Status of input module success

(...Running extended Hückel module)

Status of extended Hückel module success

3.2 Solution of version controlling

For the purpose of version controlling, the Git system was used. Work was stored in a private repository throughout the development process, the end product can be found in the open repository previously mentioned in the User Documentation part. With the use of Github, a platform for easy communication was also provided, thus allowing messaging with other participants assisting the project. This was important to specify the particular user demand for the software being made and to clarify and evaluate all implementation that came up during development.

3.3 Implementation of the input module

The implementation of the input module entailed the solution of two different problems: conversion between molecular formats and parsing a PDB formatted file. The latter came into existence as the PDB format was chosen as a fix intermediary during the software use, thus all non-PDB files are first converted into PDB, and then to the desired YAH format.

The main.cpp file of the input module is responsible for the instantiation of related objects and their communication. It also checks for the correct number of arguments.

3.3.1 ErrorHandler namespace

To avoid unexpected behavior during runtime, the ErrorHandler namespace was introduced, consisting of functions with error messages. These functions are called for each case when during the functioning of the class such a case emerges that would yield unsuccessful execution. The namespace contains the following functions with related messaging structures:

- **invalidNumberOfArguments()**: Handles errors of unknown filename when too few arguments were given
- **notAvailableFormats(const char * inputFormat, const char * outputFormat)**: Handles errors of unknown formats given for OpenBabel
- **openBabelInputFileNotOpen()**
- **openBabelInputFileNotOpen(std::string fileName)**: Informs users about inability to open input file for OpenBabel conversion

- **openBabelOutputFileNotOpen()**: Informs users about inability to open output file for OpenBabel conversion
- **yahInputFileNotOpen()**: Informs users about inability to open input file for .yah conversion
- **yahOutputFileNotOpen()**: Informs users about inability to open output file for .yah conversion
- **noAtomsFound()**: Informs users about no atoms found in PDB file
- **noSettingsInputFile(std::string filename)**: Informs users about non existent settings file
- **settingsOutputFileNotOpen(std::string)**: Informs users about inability to open settings output file

3.3.2 OpenBabelUtils class

For the conversion of the chemical formats, the OpenBabel API was chosen. Written in C++, it was integrated into the input module to yield the opportunity of converting from all its supported formats to PDB.

For the integration two prerequisites are to be met. First, the source code of the OpenBabel API was placed into the *include/openbabel* directory of the input module, and was included in the relevant source files.

Second, following the installation of OpenBabel, the shared object created in this process was linked to the project of the input module with the use of CMake. Thus the following line was included in the CMakeLists.txt file of the project:

```
target_link_libraries(input_pdb /home/user/openbabel/lib/
    libopenbabel.so)
```

Openbabel API

In the input module, the OpenBabel API's OBConversion class [16] was used.

- **OBConversion (std::istream *is=NULL, std::ostream *os=NULL)**: Takes two data streams and prepares them for input and output
- **OBConversion::SetInAndOutFormats(const char * inID, const char* outID)**:

  - *"Sets the formats from their ids, e g CML.*
  - *Sets the formats from their ids, e g CML. If inID is NULL, the input format is left unchanged. Similarly for outID Returns true if both formats have been successfully set at sometime"*[16]

- **OBConversion::Convert()**:
  - *"Conversion with existing streams.*
  - *Actions the "convert" interface. Calls the OBFormat class's ReadMolecule() which*
    - *makes a new chemical object of its chosen type (e.g. OBMol)*
    - *reads an object from the input file*
    - *subjects the chemical object to 'transformations' as specified by the Options*
    - *calls AddChemObject to add it to a buffer. The previous object is first output via the output Format's WriteMolecule(). During the output process calling IsFirst() and GetIndex() (the number of objects including the current one already output. allows more control, for instance writing <cml>and </cml >tags for multiple molecule outputs only.)*
  - *If ReadMolecule returns false the input conversion loop is exited.*" [16]

For further details please see the full OpenBabel API documentation: `http://openbabel.org/dev-api/`

Structure of the OpenBabelUtils class

The OpenBabelUtils class stores data needed for file manipulation activities and contains calls to the OpenBabel API function detailed above. The class was written in a way that forms a separate unit in the module and may be easily extended or used for future similar purposes.

- **OpenBabelUtils(std::string fileName)**: Constructor of the class, initializes the *inputFileName* private member with the *fileName* argument
- **initInputFile(std::string filename)**: Opens the specified input file for reading and informs the user if the opening was not successful

- **initOutPutFile(std::string filename)**: Opens the specified input file for writing and informs the user if the opening was not successful
- **obtainInputFileFormat()**: Obtains file format from inputfile name and creates output filename
- **OpenBabelConversion()**: Creates *OpenBabel::OBConversion* object responsible for conversion and sets the conversion formats, checking if they are available in OpenBabel
- **convert()**: initiates the files for carrying out conversion actions and then executes the conversion

The class further includes getters for testing purposes.

3.3.3 PDBParser class

Seeking a C/C++ native solution for PDB parsing, first the **dsr-pdb parser (A simple C++ PDB reader)** `http://graphics.stanford.edu/~drussel/pdb/` was taken into consideration. It provides users with basic functionalities for handling files of this specific format. As there were several ambiguities in the API and documentation of the library as a whole, it was deemed best to look for alternative packages as well.

Easy Structural Biology Template Library

After this, the **Easy Structural Biology Template Library (ESBTL)** was considered and it was integrated into the input module. This library is well fit for PDB parsing with several possibilities of specifying the particular inputs.

Details of important classes and structures of ESBTL used in the input module with descriptions from the Reference Manual of ESBTL [25] :

- **template<class System>**
  class ESBTL::All_atom_system_builder<System >: *"Class responsible for building a system (..) System is a system as ESBTL::Molecular_system."*
- **struct ESBTL::PDB::Mandatory_fields_default**: *"Default class to specify to ESBTL::PDB::Line_format which PDB fields of a coordinate line are mandatory."*

- **ESBTL::PDB_line_selector**: *"This class is a simple line selector: all atoms and hetero-atoms are in the one system."*
- **int ESBTL::PDB_line_selector::keep (const Line_format & line_format, const std::string & line, Occupancy_handler & occupancy)**: Member function of the PDB_line_selector class determining whether or not to keep a line of the PDB file based on the keyword and the particular occupancy in the line
- **ESBTL::Accept_none_occupancy_policy <Line_format>**: *"This object provides a restrictive occupancy policy: All atoms are discarded."*
  This means that if the occupancy value of an atom is not 1, then it will be discarded.

The following is the description to reading a PDB file:

1. *"A system type and a container for the systems have to be defined*
2. *A line selector (see Line selectors) is used to define what will the systems be made of*
3. *A builder is used to fill the system container*
4. *An occupancy policy has to be chosen"* [25]

For further details please see the full ESBTL Reference Manual: `http://esbtl.sourceforge.net/refman/doc/html/index.html`

Structure of the PDBParser class

The PDBParser class uses the ESBTL to parse the PDB file as described before. The application of this thesis contains alterations to this library, that were implemented to meet the specific requirements of the software. To easily integrate the calls to the library, template functions and synonyms, in forms of typedefs were introduced to the PDBParser class.

The following are the **private data members** of the class:

- ***typedef ESBTL::Accept_none_occupancy_policy <ESBTL::PDB::Line_format <CustomMandatoryFields >> Accept_none_occupancy_policy***: synonym for the use of Accept_none_occupancy_policy with CustomMandatoryFields

- ***struct CustomMandatoryFields***: structure containing the attributes that the given PDB file have
- ***ESBTL::PDB_line_selector sel***: determines which lines of the PDB file to keep
- ***std::vector<ESBTL::Default_system>systems***: array of molecular system objects representing molecules
- ***ESBTL::All_atom_system_builder <ESBTL::Default_system >builder***: this object is responsible for building a system for the molecule
- ***std::string inputFileName, std::string moleculeName, std::string outputFileName***: molecule name, input and output names
- ***std::ofstream outputFile***: output file stream

The following are the **private member functions** of the class:

```
template<ESBTL::Reading_mode mode,class mandatoryFields,
  class Line_selector ,class Builder ,class Occupancy_handler
  >
    bool readPdb(const std::string& filename,Line_selector&
        sel,Builder& builder,const Occupancy_handler&
        occupancy,char altloc='␣')
```

Listing 3.1: readPdb function of the PDBParser class with specifiable reading mode

The above function creates a **Line_reader** object for the filename using the specified reading mode and occupancy handling method, with the help of a mandatoryFields structure and Line_selector as well as Builder objects. After this, the read function of the Line_reader object is called for the given filename, occupancy and altloc character (default is non breaking space)

```
template<class mandatoryFields, class Line_selector ,class
  Builder ,class Occupancy_handler>
    bool readPdb(const std::string& filename,Line_selector&
        sel,Builder& builder,const Occupancy_handler&
        occupancy,char altloc='␣')
```

Listing 3.2: readPdb function of the PDBParser class without specifiable reading mode

The above function calls the readPDB function with ASCII read mode

The following are the **public member functions** of the class:

- **PDBParser(std::string fileName, std::string moleculeName)**: Constructor of the PDBParser class, instantiates the private data members line _selector, systems, builder (using the systems object and the maximum number of systems value of line_selector) and further data members inputFileName and moleculeName using the constructor arguments. It also creates and sets the outputFilename based on the inputFileName member
- **readPdb()**: calls the private readPDB function by specifying ASCII read mode and CustomMandatoryFields as mandatoryFields class. It also uses the inputFileName, line_selector, builder, Accept_none_occupancy_policy() as arguments
- **convert()**: opens the output file for conversion, then calls the public readPDB member function of the class. After parsing, it writes to the output file the molecule name, the number of atoms and the coordinates in YAH format

It is important to note that as YAeHMOP only uses the atomic coordinates in it's input file format, not all characteristics of a PDB file needs to be taken advantage of. Important properties, however, include the number of atoms as well as the name and the x, y, z coordinates for each atom.

An extension to the ESBTL package was the counting of atoms. This number is needed in a YAH file, and as there may be ATOMs and HETATMs in a PDB file, a new counter was introduced.

When implementing this counter, certain aspects needed to be considered. For input files with several atomic models, the number of atoms needed to be calculated only once. This was achieved by creating certain conditions to be met when taking into consideration the modelnumber that was being parsed.

Another aspect was the occupancy level for atoms. Based on the occupancy policy used during parsing, some atoms may be discarded. This, however, might lead to situations where not all atoms are included in the atomic system representing the molecule. After this, the issue of having different number of atoms then counted may arise when writing to a YAH formatted file.

Troubleshooting

In some cases, when the user tried converting files not entirely adhering to the standards of the given format, OpenBabel alerted the user of such cases. This, however, resulted in the output file of the conversion not being closed properly, even though future conversion to YAH format may have taken place and then the PDB parser was unable to function well. Consequently, closing of the files was implemented, for the cases when this phenomenon was experienced.

3.3.4 SettingsParser class

The Settingsparser class is responsible for the processing of printing options of the application. The separation of the configuration settings file from the other input file and its creation was carried out for two main reasons.

One is the possible use of the application in bulk mode. This means, that several configuration files can be created beforehand for the same molecule, and run in different dedicated environments (such as virtual machines or Docker containers).

Another reason was the idea of separation of components in the application having a fundamental influence during the creation of the entire software.

- **SettingsParser(std::string configFileName, std::map <std::string, std::string>& props)**: Constructor of the SettingsParser class, sets the name of the configuration file and the reference of the map data structure. The map data structure of the standard library was chosen, as it provides an easy to use and efficient representation of a container storing key-value pairs.
- **readSettings()**: after initializing input file stream and setting the "=" character as delimiter, it stores properties in the map data structure according to the content of the configuration file
- **writeSettings(std::string outputFileName)**: appends to a file the printings options that were parsed

  The class further includes getters for testing purposes.

3.4 Redesigning YAeHMOP

3.4.1 Flow of calculations

The following is a description of the source of the YAeHMOP program.

The most important header of the program is called *bind.h.* All structure definitions and global constants are introduced here. The original pieces of code contained many cases when function declarations were separated in one single header file (*prototypes.h*). These functions were declared as *extern* so that the compiler is informed about their definition being in another source file.

The general flow of operations for the redesigned parts of YAeHMOP are the following (similar to that of YAeHMOP):

First, the input files are parsed into the cell and details structures with the help of the *read_inputfile* function.

Then, the *map_orb_num_to_name* function is called which determines the number of orbitals and their names

After these calls, the general procedure follows by looping over the points of the Walsh diagram. The following actions are taken during the course of this looping:

- Opening files for band and FMO output (if applicable)
- Setting up positions for a Walsh diagram (if applicable)
- Generating a distance matrix with *build_distance_matrix*
- If the execution mode is set to *"FAT"* and the R overlap matrices will be stored, or in case of a molecular execution, the following function is called for building the R overlap matrix:
  - *R_space_overlap_matrix*
  - After this, if an FMO computation was set, then calculations for that are carried out by using the following functions:
    - *full_R_space_Hamiltonian* (building molecular type of Hamiltonian)
    - *build_FMO_overlap*
    - *build_FMO_hamil*
    - *diagonalize_FMO*
    - *gen_FMO_tform_matrices*
  - *full_R_space_Hamiltonian* (building the original Hamiltonian matrix)

- If the execution mode is set to *"FAT"* but the R overlap matrices will not be stored, then calls to the following functions are done instead:
  - build_all_K_overlaps
  - *full_R_space_Hamiltonian* (building the original Hamiltonian matrix)
- Finally, if the execution mode is set to *"THIN"*, then only diagonal elements are sought by using the *R_space_Hamiltonian* function.
- After these particular cases, the ***loop_over_k_points*** is called, a function that iterates through the k points (if more than one). This function is particularly important as:
  - First, the *build_k_overlap_FAT* and *build_k_hamil_FAT* or *build_k_overlap_THIN* and *build_k_hamil_THIN* functions are called according to execution mode (matrix sparsification may take place as well) to create the Hamiltonian and Overlap matrices
  - Then, with the help of the *print_labelled_mat* function these matrices are written in ASCII mode to the output file (if applicable):
    - ***dump_hermetian_mat***
    - ***dump_sparse_mat_matrix_market*** (newly introduced)
    - ***dump_sparse_mat*** (redesigned)
    - ***Cboris*** (optional egeinvalue calculator routine)
    - ***zhegv*** (computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem)
    - ***zheev*** (computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A)
  - if moleculer execution: call *eval_electrostatics* to evaluate the electrostatic term for molecular optimizations (if applicable)
  - do the average properties calculations (if applicable):
    - call *sort_avg_prop_info* and *find_crystal_occupations* to sort the average properties information in increasing energy level and find the crystal occupations
    - *calc_avg_occups* to determine net charges and orbital occupations
    - update the zeta values for the self-consistent procedure with *update_zetas* (if applicable)
    - do the charge iteration (if applicable)

* ∗ get the average overlap population, reduced OP matrices, and average OP's (if applicable)
* ∗ get the density of states
* ∗ print all further data that were specified by user input to the bottom of the output file

The ***dump_sparse_mat_matrix_market*** option was added to the output alternatives of the original program. With this option, both Hamiltonian and overlap matrices are written to file in binary conforming to the Matrix Market format. It uses the output module to create the file.

The ***dump_sparse_mat*** option was redesigned to resemble the Matrix Market format, however, in a way of only writing the coordinates and the specific values of a matrix. This output option also uses the output module.

For further details please see the full YAeHMOP tightbing User Manual: `https://github.com/greglandrum/yaehmop/blob/master/docs/bind_manual.pdf`

3.4.2 Refactoring

The original code base contained architectural and data structure solutions that generate obstacles for maintenance and code reuse. Exceeding more than 18,000 lines in C and in Fortran, however, it was deemed that the entire refactoring of the program was beyond the scope of this thesis.

This way, certain parts and certain types of the code pieces were altered during creation of the end product. The most important intention was the creation of C++ code that could communicate with other, newly created modules easily.

An example for an issue was with function definitions. Earlier versions of the C programming language supported certain function definitions. These weren't supported later on anymore, but in some projects were still in use. C++, however, is unable to process such functions, this is why refactoring needed to take place.

```
void loop_over_k_points(cell,details,overlapR,hamilR,overlapK,hamilK,
                         cmplx_hamil,cmplx_overlap,
                         eigenset,work1,work2,work3,cmplx_work,
                         properties,avg_prop_info,
                         num_orbs,orbital_lookup_table)
  cell_type *cell;
  detail_type *details;
  hermetian_matrix_type overlapR,hamilR;
  hermetian_matrix_type overlapK,hamilK;
  complex *cmplx_hamil,*cmplx_overlap;
  eigenset_type eigenset;
  real *work1,*work2,*work3;
  complex *cmplx_work;
  prop_type *properties;
  avg_prop_info_type *avg_prop_info;
  int num_orbs;
int *orbital_lookup_table;
{
(...)
}
```

Listing 3.3: Variables are declared after function signature (C)

```
void loop_over_k_points(cell_type *cell, detail_type *details,
   hermetian_matrix_type overlapR,
                         hermetian_matrix_type hamilR,
                            hermetian_matrix_type overlapK,
                         hermetian_matrix_type hamilK,
                         complex *cmplx_hamil, complex *cmplx_overlap,
                         eigenset_type eigenset, real *work1, real *
                            work2, real *work3,
                         complex *cmplx_work,
                         prop_type *properties, avg_prop_info_type *
                            avg_prop_info,
                         long num_orbs, int *orbital_lookup_table)
{
(...)
}
```

Listing 3.4: Variables are declared as part of function signature (C++)

Difficulties arose when turning to C++ syntax in the case of iteration through enumerations as well. This was so, because in C, the language supported the syntax of iterating. An example for that was in the *symmetry.cpp*, postfix ++ operator definitions of *possible_sym_op& operator++(possible_sym_op &c)*, *possible_sym_op* *operator++(possible_sym_op &c, int),possible_axis& operator++(possible_axis &c)* and *possible_axis operator++(possible_axis &c, int)* were solving this issue.

Trigonometric functions such as sine, cosine and further simple mathematical functions such as fabs(), ceil(), floor() were altered in a way that instead of the previously used sin(), cos(), fabs(), etc. functions from a C library, their counterparts from the C++ standard library were used.

An example for the difference between C and C++ linking created issues as well. The LAPACK functions *zhegv* and *zheev* could be declared as extern functions to link the Fortran code. However, C++ requires a difference syntax, and thus, these function declarations were introduced as *extern "C"*.

3.4.3 64 bit architecture

As in some cases for large molecules, the number of orbitals were exceeding the numerical capabilities of storing integers, in these cases runtime errors arose. This meant that the 32 bit version of integer variables of the C++ language were used instead of 64 bit version. On Unix platforms, these are the *int* and *long* respectively, for real numbers the *float* and *double*.

In an attempt to introduce a transition to the 64 bit counterparts of numerical variables, for real values, this change was carried out successfully throughout the project. However, for integer numbers, a basic *search and replace* method could not be followed as the code base entailed POSIX file handlers, which store integer file descriptors, as values. Changing the type of these variables results in unwanted behavior and segmentation fault.

Considering the size of the code base, as mentioned before, vast refactoring procedures could not be applied. Consequently, certain types for integer values were replaced throughout the project. These included variables related to the number of orbitals, and tabulator variables used for accessing certain values in a matrix. A further action of this area was the changing of execution mode from *"FAT" "THIN"*.

3.5 Multi-threading and higher performance

3.5.1 Introduction

The use cases of running the extended Hückel method for large molecules containing thousands of atoms and the slow runtime these bred motivated this part of the work. The initial idea while taking into consideration the YAeHMOP program and its running was the enhancement of performance while carrying out robust calculations. As discussed in the scientific background part of this document, the initial problem of molecular energy levels involving the Schrödinger equation leads to a vastly sized eigenvalue equation. Consequently, calculations related to linear algebra and numerical methods are completed such as the Cholesky or the QR factorization. Difficulties with such calculations arise because of the high number of molecular orbitals.

To achieve better performance during the run of the program, the multi-core architecture of contemporary computers was considered for use.

3.5.2 Linear Algebra software

YAeHMOP may process linear algebra calculations in two different ways, based on the configurations specified in the CMakeLists.txt or Makefile. Before running the application, certain options need to be set in either of the files. The use of available supporting software and system architecture needs to be taken into consideration before deciding which method to use.

As described in the YAeHMOP tightbind documentation, the first type of these is the default **cboris** routine. Dating back to 1974, this subroutine was written in Fortran to solve the complex eigenvalue problem. It calls other Fortran subroutines. YAeHMOP may use these source files in two ways: it either compiles and links them with the other source files originally of the C programming language or uses a translation software called **F2C**. With this translation tool, users may create C code from Fortran source, serving as the basis for programs that were not written originally in this language and thus are not optimized in any way. For these reasons in many cases using these routines with great matrices have significantly high computation times.

Coming as the alternative to cboris, the usage of contemporary software developed since the 1990s is an option as well. The brief introductory descriptions to the functionalities of these are specified on their official website as follows:

- *"**LAPACK (Linear Algebra PACKage)** is written in Fortran 90 and provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems."*[20]

- *"**BLAS (Basic Linear Algebra Subprograms)** are routines that provide standard building blocks for performing basic vector and matrix operations."*[21]

YAeHMOP uses the **zhegv** Fortran routine from LAPACK, making several calls to BLAS routines during runtime.

3.5.3 Chosen package: OpenBlas

When considering the possibilities for enhancing the performance of the program, execution time measurements were carried out distributed for certain parts of the software. Based on the results it was concluded that the optimization of the software came down to the method in which the linear algebra equations are carried out. As stated in the YAeHMOP documentation, in general cboris yields much slower execution than using LAPACK and BLAS. The advantage of this routine, however, remains that it needs no further packages to be installed as sources are distributed as a part of the YAeHMOP. Attempts for exploiting multi-threading in the cboris routine were made using **OpenMP**, but during development it was decided that this task was not within the scope of this thesis.

The original distributions of LAPACK and BLAS were compared with cboris and a highly optimized package called **OpenBlas**. While further relevant software including ATLAS (Automatically Tuned Linear Algebra Software) or vendor-supplied versions of BLAS such as Intel's Math Kernel Library exist, OpenBlas was chosen for being an example of open source software, having highly optimized routines and supporting multi-threading in some cases.

The OpenBlas package includes a LAPACK distribution as well as extending original routines with some optimized ones. Using the integrated modules of **FindBLAS** and **FindLAPACK** in CMake, the original CMakeLists.txt file of the redesigned YAeHMOP module was extended:

```
set(BLAS_DIR /usr/lib/)
find_package(BLAS REQUIRED)
set(LAPACK_DIR /usr/lib/)
find_package(LAPACK REQUIRED)
```

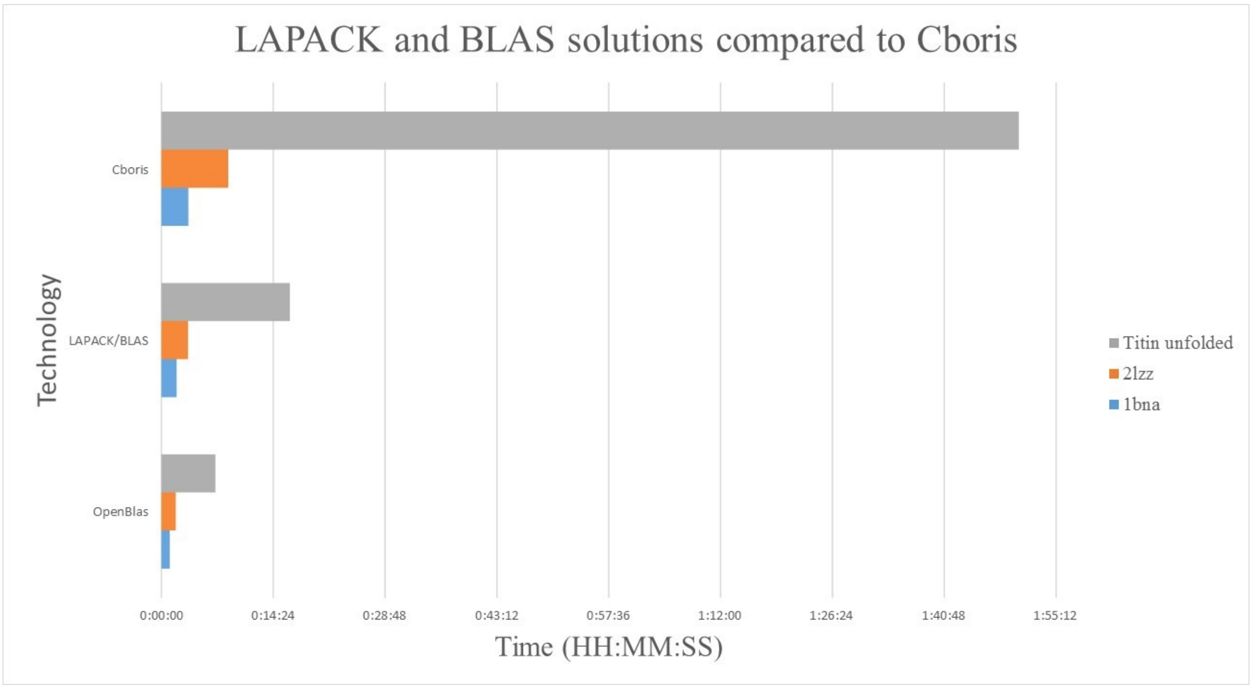
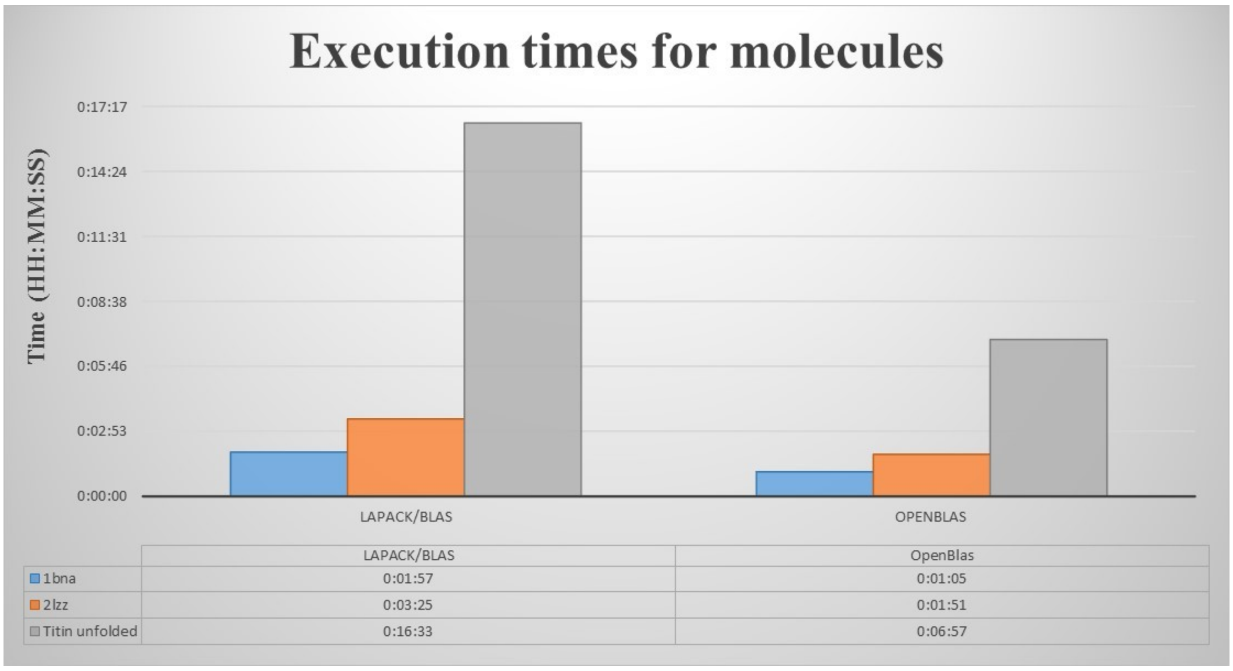


Figure 3.2: Execution times of technologies including Cboris on the three observed molecules



	LAPACK/BLAS	OpenBlas
🟦 1bna	0:01:57	0:01:05
🟧 2lzz	0:03:25	0:01:51
▨ Titin unfolded	0:16:33	0:06:57

Figure 3.3: Original LAPACK and BLAS performance compared to OpenBlas

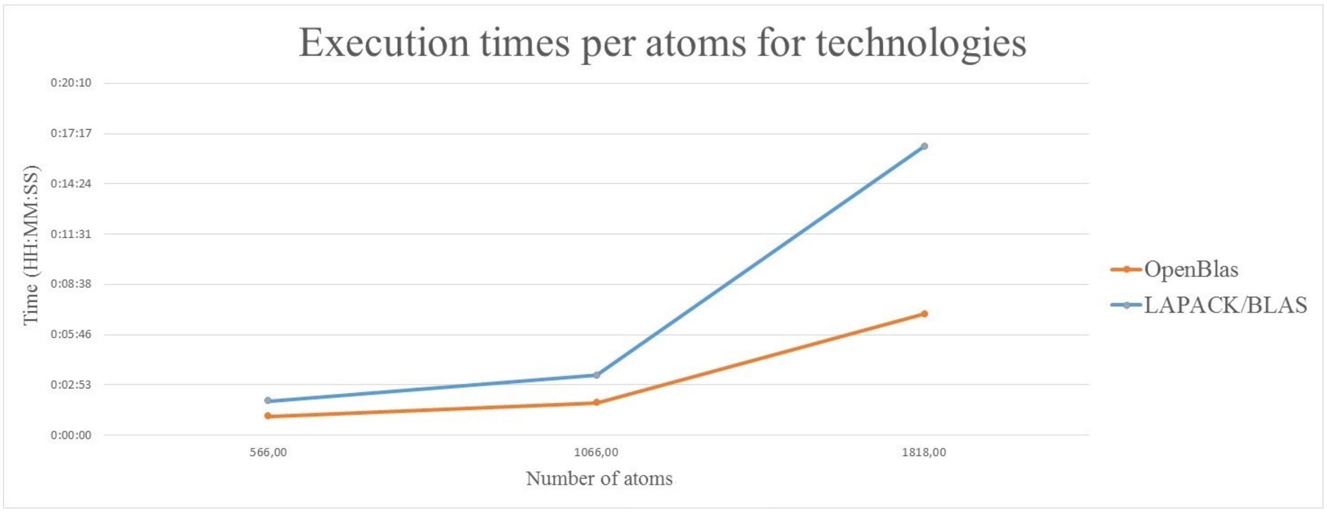


Figure 3.4: Execution times affected by number of atoms

As it can be interpreted from the diagrams, the conclusion of personal measurements is that OpenBlas was the most efficient of the three technologies that were under testing. When applicable, it used multi-threading during the execution of some linear algebra routines.

3.5.4 Measuring execution time

In the process of measuring time of executions, several test files were given as an input for the application, staying within the capabilities of a quad-core personal computer. The results of these calculations were profiled with the Linux open-source profiler called **oprofile**. The main advantage of this tool is that it creates little overhead during the running of the application thus ensuring real profiling.

Due to the long execution times in many cases, executions were carried out overnight. Therefore commands entailing the run of test executables were scheduled by specifying the start date and adding them in the **crontab** list.

Scripts akin to the following were used for timing:

```
for VARIABLE in 1 2 3 4 5
do
SECONDS=0
now=`date +'%m_%d'`
mkdir "$now"_orig_$VARIABLE
./bind 1bna.yah > ./"$now"_orig_$VARIABLE/log_1bna.txt
duration=$SECONDS
echo "$(($duration / 60)) minutes and $(($duration % 60))
    seconds elapsed." > "$now"_orig_$VARIABLE/log_1bna.txt
done
```

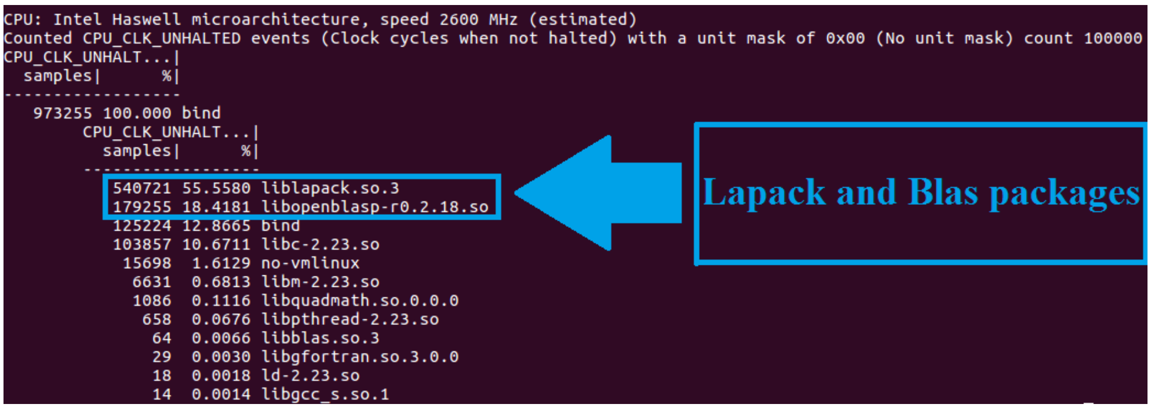
Listing 3.5: Measuring Huckel Code execution time on the DNA molecule

During the timing phase of the thesis, various molecules were tested several times and with several technologies. The **1bna** (566 atoms and 2264 orbitals), **2lzz** (1066 atoms and 2662 orbitals) and **unfolded titin** (1818 atoms and 4575 orbitals) molecules were subject to most of the testing procedures.

All tests were carried out with the following printing options:

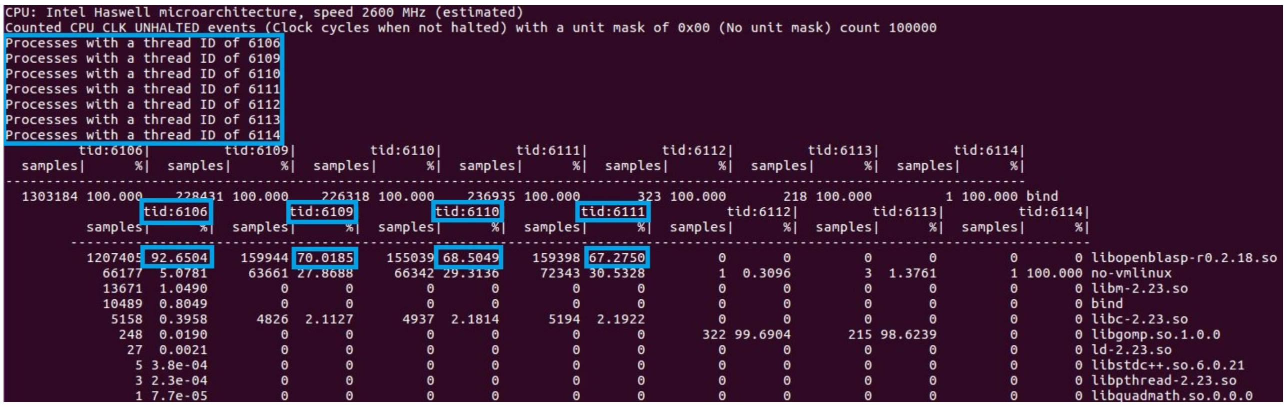
- Orbital Mapping
- Overlap
- Hamil
- Dump Hamil
- Dump Overlap

The following execution measurements were carried out on a Ubuntu 16.06 LTS system with system specifics of 64 bit Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz, 6GiB System Memory.



```
CPU: Intel Haswell microarchitecture, speed 2600 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask of 0x00 (No unit mask) count 100000
CPU_CLK_UNHALT...|
  samples|      %|
------------------
   973255 100.000 bind
        CPU_CLK_UNHALT...|
          samples|      %|
        ------------------
           540721 55.5580 liblapack.so.3
           179255 18.4181 libopenblasp-r0.2.18.so
           125224 12.8665 bind
           103857 10.6711 libc-2.23.so
            15698  1.6129 no-vmlinux
             6631  0.6813 libm-2.23.so
             1086  0.1116 libquadmath.so.0.0.0
              658  0.0676 libpthread-2.23.so
               64  0.0066 libblas.so.3
               29  0.0030 libgfortran.so.3.0.0
               18  0.0018 ld-2.23.so
               14  0.0014 libgcc_s.so.1
```

Figure 3.5: Majority of total execution time spent for LAPACK and BLAS calls



```
CPU: Intel Haswell microarchitecture, speed 2600 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask of 0x00 (No unit mask) count 100000
Processes with a thread ID of 6106
Processes with a thread ID of 6109
Processes with a thread ID of 6110
Processes with a thread ID of 6111
Processes with a thread ID of 6112
Processes with a thread ID of 6113
Processes with a thread ID of 6114
         tid:6106|          tid:6109|          tid:6110|          tid:6111|          tid:6112|          tid:6113|          tid:6114|
  samples|      %|  samples|      %|  samples|      %|  samples|      %|  samples|      %|  samples|      %|  samples|      %|
------------------------------------------------------------------------------------------------------------------------------------
  1303184 100.000    228431 100.000    226318 100.000    236935 100.000       323 100.000       218 100.000         1 100.000 bind
                 tid:6106|          tid:6109|          tid:6110|          tid:6111|          tid:6112|          tid:6113|          tid:6114|
          samples|      %|  samples|      %|  samples|      %|  samples|      %|  samples|      %|  samples|      %|  samples|      %|
        ------------------------------------------------------------------------------------------------------------------------------------
          1207405 92.6504    159944 70.0185    155039 68.5049    159398 67.2750         0       0         0       0         0       0 libopenblasp-r0.2.18.so
            66177  5.0781     63661 27.8688     66342 29.3136     72343 30.5328         1  0.3096         3  1.3761         1 100.000 no-vmlinux
            13671  1.0490         0       0         0       0         0       0         0       0         0       0         0       0 libm-2.23.so
            10489  0.8049         0       0         0       0         0       0         0       0         0       0         0       0 bind
             5158  0.3958      4826  2.1127      4937  2.1814      5194  2.1922         0       0         0       0         0       0 libc-2.23.so
              248  0.0190         0       0         0       0         0       0       322 99.6904       215 98.6239         0       0 libgomp.so.1.0.0
               27  0.0021         0       0         0       0         0       0         0       0         0       0         0       0 ld-2.23.so
                5 3.8e-04         0       0         0       0         0       0         0       0         0       0         0       0 libstdc++.so.6.0.21
                3 2.3e-04         0       0         0       0         0       0         0       0         0       0         0       0 libpthread-2.23.so
                1 7.7e-05         0       0         0       0         0       0         0       0         0       0         0       0 libquadmath.so.0.0.0
```

Figure 3.6: Multiple threads running the LAPACK and BLAS calls

As an extreme input file, for the *4bip* molecule (it has 2643 atoms) the execution time with the original LAPACK and BLAS distributions was a little more than **192 minutes**. Using OpenBlas, however, resulted in having an application run with **92 minutes**.

Unfortunately, for molecules having more than 5000 atoms, the configuration of the computer that was used proved to be insufficient to complete tests.

3.6 Implementation of the output module

3.6.1 Introduction

A separate module was decided to be created that manages the output sparse matrix format after calculations took place in the core module of the program. This decision was made because of the principles of maintenance and importance of possible future reuse mentioned throughout this thesis. The task of this module was the creation of an additional output format extending the preexisting possibilities. This was needed, because though the calculation module was capable of creating output files in

ASCII as well as in binary format, this meant the writing of entire matrices. Having mentioned at an earlier part of this document, the idea of having a sparse matrix output format emerged. For this, a wide-spread format, the Matrix Market format was chosen. This format is particularly useful, because there exist various APIs and easy to integrate modules connected to this format in programming languages such as Python.

Regarding the form of writing to the file, it is important to determine what purpose the output file would have in the case of the program. The intention of recording the results of complex mathematical computations in order to describe certain relationships and energy levels of atoms in a molecule, breeds that large amounts of data needs to be written to file. This does not only carry possible issues regarding how much space these files would have to take up, but also difficulties arise when managing and processing these collections of data. Consequently, it might be advisable to rethink if it is a preference for the output file to be human-readable and thus easily comprehensible by users. If not, then a change in the output formatting could have its obvious benefits.

The use of the Matrix Market format also entailed memory management aspects. This aspect of the format plays a significant role for larger input data. Specific examples will be mentioned in this section.

Using a binary output format, however, might generate issues with endianness. [22] In other words, a file created on one machine that is big endian, the most significant byte is stored in the smallest address would be subject to difficulties when processed by a machine having the little endian characteristics. Following that, issues may arise with the size of certain variables and how they are stored on the computers exchanging the files. The use of outputting the machine specific details regarding these characteristics is highly advised as otherwise binary files may not be interpreted by target programs.

3.6.2 MatrixMarketWriter class

For the MatrixMarketWriter class, first the ANSI C library for Matrix Market Input and Output was considered, however, as it was deemed that this library needed refactoring as well, a newly created module was preferred. The principle of using newly introduced elements of C++ whenever it was feasible was applied. An example for that was the use of *enum classes.* During the creation of this module, the idea of future extension opportunities were kept in mind (for example for dense matrix output). Adhering to the data structures used by the calculation module, matrices are represented as pointers to double values. Following this idea, writing to the file was solved by using POSIX calls (similar to that in the calculation part). This means that integers for file descriptors were introduced. Thus, the writing to the output file in binary format was achieved with the POSIX *write* function. The class entails variables with 64 bit representation for Unix systems.

Created for the format keywords, the following are the enumeration classes of the MatrixMarketWriter class:

- FormatType
- SparsityType
- FieldType
- SymmetryType

The MatrixMarketWriter class contains the output file details, the matrix to be recorded and its dimensions, and the cutoff vale (optional) as private data members. Apart from these, it also contains a value for each aforementioned Matrix Market category (e.g.: FormatType: matrix, SparsityType: sparse, FieldType:real, SymmetryType:symmetric).

Private member functions of the class writing to the output file:

- writeFormat(): writes Matrix Market format category
- writeSparsity(): writes Matrix Market sparsity category
- writeField(): writes Matrix Market field category
- writeSymmetry(): writes Matrix Market symmetry category
- writeSparseDetails(): iterates through the matrix to determine the number of non-zero elements, then writes this value
- writeCoordinates(): writes the coordinates and values non-zero elements, uses whitespace characters as delimiters between values
- dumpCoordinates(): writes the coordinates and values non-zero elements one after the other
- writeToFileBinary(variable): writes variable (contains three overloads for std::string, double or long types) to file using POSIX write function

Public member functions of the class writing to the output file:

- `MatrixMarketWriter(std::string fileName, double* mat, long row, long column, FormatType ft = FormatType::matrix, SparsityType st = SparsityType::sparse)`

  Constructor of the MatrixMarketWriter class, initializes output filename, matrix, number of rows and columns (overload for square matrices exists)

- `MatrixMarketWriter(std::string fileName, double cutoff, double* mat, long row, long column, FormatType ft = FormatType::matrix,SparsityType st = SparsityType::sparse ):outputFileName(fileName), matrix(mat), rowNumber(row), columnNumber(column), cutoff(cutoff), formatType (ft), sparsityType(st)`

  Constructor of the MatrixMarketWriter class, initializes output filename, matrix, number of rows and columns and also uses cutoff value (elements of the matrix smaller are considered to be zero) (overload for square matrices exists)

- createBinaryFile(): writes the matrix to a binary file entirely in Matrix Market format

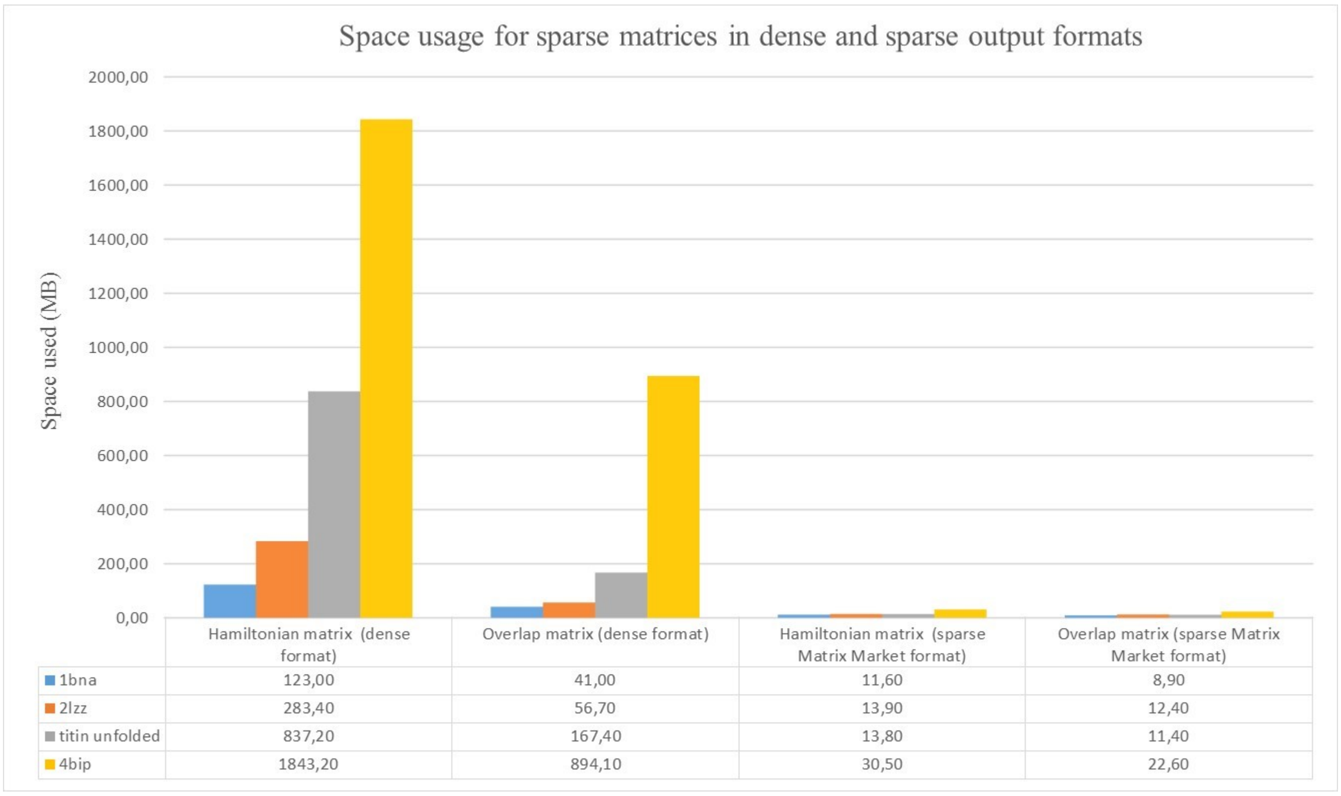
- createDumpFile(): writes the matrix to a binary file with resembling the Matrix Market format (without any text or characters as delimiters)

Following the format of a Matrix Market file, the number of non-zero elements needs to be determined before actually disclosing their coordinates and values. This issue was solved by first iterating through the matrix counting non-zero elements, and then in a following loop, writing their coordinates and values. In order to achieve better performance for vast matrices, the execution of the first loop was made parallel with the use of *OpenMP*. By using OpenMP, one can create pragma directives in the source code, that will inform the compiler that parallelized blocks are introduced. If the C++ compiler supports OpenMP, this would mean that from the point where this pragma was introduced, operations would be forked and based on the specific command a possible multi-threaded execution would commence on machines with multi-core processors. Each thread would have unique local variables for each variable introduced after the start of the parallelized block. However, those outside of this block would constitute as shared variables and thus would be subject to unwanted behavior such as race conditions.

```
long numberOfNonZero = 0;
    #pragma omp parallel for shared(numberOfNonZero)
    for (long i = 0; i < rowNumber; i++) {
        long itab = i * columnNumber;
        for (long j = 0; j < columnNumber; j++){
            if (fabs(matrix[itab + j]) > cutoff)
            {
                #pragma omp atomic
                ++numberOfNonZero;
            }
        }
    }
```

Listing 3.6: Multi-threaded counting of non-zero elements in a sparse matrix

The class further contains setters for the categories to be set when objects are created outside of the class.



	Hamiltonian matrix (dense format)	Overlap matrix (dense format)	Hamiltonian matrix (sparse Matrix Market format)	Overlap matrix (sparse Matrix Market format)
🟦 1bna	123,00	41,00	11,60	8,90
🟧 2lzz	283,40	56,70	13,90	12,40
⬜ titin unfolded	837,20	167,40	13,80	11,40
🟨 4bip	1843,20	894,10	30,50	22,60

Figure 3.7: Observations of space usage for output files created for sparse matrices

As the diagram shows, all of the cases prove that the use of the Matrix Market sparse matrix format is more beneficial in terms of space allocation.

3.7 Implementation of the user interface

3.7.1 Python script

The Python 3 script was created to aggregate the input and the calculation modules into one executable. It uses the **ArgumentParser** class of the *argparse* module that allows the implementation of an argument parser program. When calling the executables for each module, it uses the **Popen** function of the *subprocess* module to open the files.

3.7.2 Jupyter Notebook

During university research activities I became acquainted with the Jupyter Notebook. It offers various data transformation, numerical simulation and modeling functionalities and contributes to the open science movement with easy to share programming environments.[23]

Another technology I had the chance to work with in the course of a university project is a collaborative platform providing users with plenty of opportunities for called **SageMathCloud**. It allows the creation and sharing of online documents that can be modified by various users, among many possibilities such as LaTeX editing and creating interactive documents one can create Jupyter Notebooks as well.[24]

In this particular Jupyter Notebook, designed for the Huckel Code application, ipython widgets were used. These widgets entailed checkboxes for user input, then with the help of similar functions used in the Python script, the files were processed.

3.8 Tests

3.8.1 Unit tests

Introduction

Through the process of unit testing, developers may have the chance to check if their software indeed works the way they expect it to. This is achieved by taking the smallest parts, units of the code and testing their functionalities. Ensuring that every tiny method in a robust application operates correctly may seem as an overly time-consuming task, but it can lead to the avoidance of several serious errors and thus to cost efficiency in the long run. For this reason, this type of testing comes as paramount while and after developing software in the corporate world.

Each unit is to be tested completely separately and independently from other parts of the application. Consequently, the outcome of a unit test may in no way be influenced by another one. If this is not the case then the content of these methods need serious reconsideration.

Investigation

When taking into consideration preexisting packages used in this thesis work, it can be concluded that such software require no further unit testing as we presuppose that they were developed correctly and function well. This way tests of this kind were created exclusively to newly created pieces of code that form larger unity in the entirety of the project. In order to avoid duplicated code and for easy maintenance of application, the original modules were included in the test projects using the CMake file. Test classes for performing operations on and with the existing classes and a main source file running the unit tests were introduced for the important modules. This method allowed tests to be taken in a fashion that is most similar to the release running of the application. During this type of work no further test frameworks were used, tests were prepared manually bearing in mind the most important use cases.

During the course of making tests, the principles of building up a test object and environment was given emphasis. Thus in general, test classes were made with a function responsible for any tasks that were connected to the creation of a simulation environment. As examples, input/output methods, database calls are often functionalities included in such parts of the test class. Instantiation of objects of the classes under test happen just before the call of these functions or constitute as the first operation inside of them.

Following the principles of unit testing and thus aiming at not leaving any states or files created throughout this process, functions responsible for restoring the state before the testing commenced were introduced as well. Such tear down functions may entail the closing of open filestreams as well as the removal of temporary files that were created in the process of unit testing. This functionality is required so that ultimately, as previously mentioned, no resources or files are created.

TestOpenBabelUtils class

Files related to the checking of the input module are located in the tests folder. Testing the OpenBabelUtils class was carried out in a way that a TestOpenBabelUtils test class was created in this folder. The integration of the OpenBabelUtils class was achieved by introducing a pointer to an OpenBabelUtils object as private member of the class.

This architectural decision was made because of many reasons. On the one hand this instance is solely used for test purposes and thus one object is used throughout the testing process (thus resembling singleton objects used in C++), hence there was no need for copying existing objects. Another reason was simple memory management, storing only a pointer to an object has the clear advantage of less allocated memory. The member was made private in order to adhere to the paradigm of object oriented design. Further test classes will follow these principles as well.

The main.cpp file of test projects contain asserted calls to the test functions. In order to test all parts of if statements, in some cases more than one test was introduced for the same function.

The following test functions were introduced in the **TestOpenBabelUtils** class:

- **buildUp(filename)**: this function is responsible for the aforementioned building up of test environment in the way of dynamically instantiating an OpenBabelUtils object and creating an input file of the format specified by the argument
- **obtainInputFileFormat()**: call to the obtainInputFileFormat() function of private OpenBabelUtils object
- **openBabelConversion()**: call to the openBabelConversion() function of private OpenBabelUtils object
- **convert()**: call to the convert() function of private OpenBabelUtils object
- **initInPutFileSuccessFileOpen(filename)**: calls the private member's initInputFile function with filename argument, then checks if the input file is open
- **initOutPutFileSuccessFileOpen(filename)**: calls the private member's initOutPutFile function with filename argument, then checks if the output file is open
- **obtainInputFileFormatSuccessFileName()**: may be called after an outer call to private member's obtainInputFileFormat(), after such a call determines if the name of the file without the end string containing the file format was successfully acquired from the initial name

- **obtainInputFileFormatSuccessFormat()**: may be called after an outer call to private member's obtainInputFileFormat(), after such a call determines if the newly created file name has the correct string as an end to (this specifies the format of the file)
- **openBabelConversionSuccessInputClosed()**: this function determines after an openBabelConversion() function invoke, if the input file was closed successfully
- **openBabelConversionSuccessOutputClosed()**: this function determines after an openBabelConversion() function invoke, if the output file was closed successfully
- **tearDown(fileName)**: as previously mentioned, this function ensures that temporary files are removed and as the OpenBabelUtils class has filestream initializer functions solely responsible for opening files, it also closes both input and output filestream should they have remained open during the test

TestPdbParser class

- **buildUp(filename,molecule)**: this function creates a PdbParser object, and then populates a temporary PDB formatted file with random data
- **readSuccessFileExists()**: in this class, a private fileChecker member was introduced. This function uses this filestream to check if the output file is existing, by trying to open it
- **readSuccessFileNotempty()**: checks if the file created by the PdbParser is non-empty
- **tearDown(fileName)**: removes the files created by the PdbParser object during test and destroys the private member of the class

TestSettingsParser class

Compared to the previous test classes, TestSettingsParser class was extended with a reference to a map data structure as an additional private member. This allows the possible injection and opportunity for minor redesign in case a user would like to work test with an existing properties data set.

- **buildUp(filename)**: creates the SettingsParser class by using the testProps member and then populates a temporary file with random property data

- **readSettingsSuccessPropsNotEmpty()**: function checking if the properties member is non-empty
- **writeSettingsSuccessFileNotempty()**: function checking if the output file created by the SettingsParser class is non-empty
- **tearDown(fileName)**: removes the files created by the SettingsParser object during test and destroys the private member of the class

TestMatrixMarketWriter class

Using the TestMatrixMarketWriter class, several cases were tested including the use of cutoff values when working with the MatrixMarketWriter class. Two other cases were when writing to the file strictly adhering to the standards of the MatrixMarket format and when only the coordinates and values of the matrix were recorded in a sparse format.

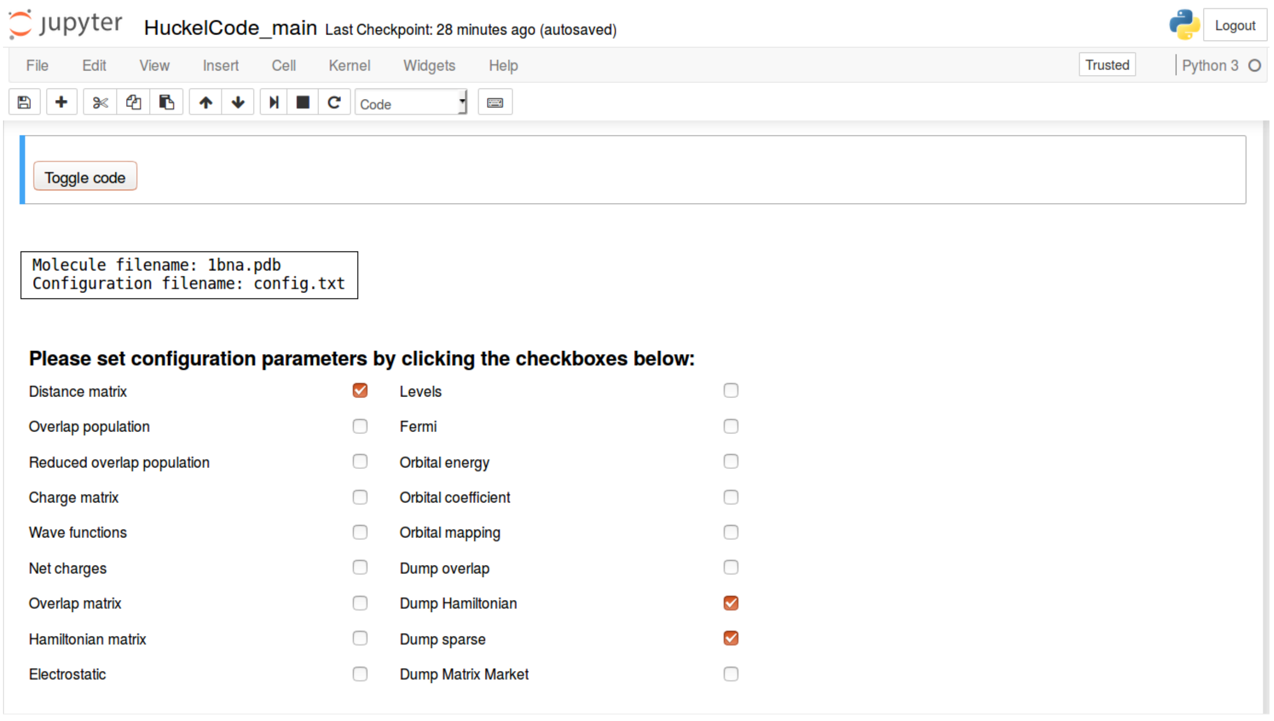
As mentioned before in this thesis, when deciding about the redesigning of the YAeHMOP program and the modules that would interact with the newly emerged version, an aspect was the conformation of the original and the newly created code. As an example for that, the representation of matrices was not altered and the MatrixMarketWriter module used the same ways of handling these mathematical structures as did the YAeHMOP. However, when taking into consideration the testing of this module, newer language elements (for example the vector data structure of the C++ standard library) were chosen over the other, if equal conversion in such cases between the representations existed. These changes created opportunities for easier testing mechanisms, for current and future testing purposes as well.

- **buildUp(filename)**: creates a temporary sparse matrix, then instantiates a MatrixMarketWriter object using the filename argument and the temporary matrix
- **buildUp(filename, cutoff)**: an overloaded version of the buildup function, additionally a cutoff value may be added as well to be used for MatrixMarketWriter object instantiation
- **tearDown(fileName)**: removes the file created by the MatrixMarketWriter object during test and destroys the private member of the class
- **writeSuccessFileNotempty()**: checks if after a writing function call, the created file in non-empty

3.8.2 System tests

Several previously mentioned implementational decisions were carried out after testing took place. Examples for that were the counting of atoms in the process of parsing a PDB (occupancy was not considered correctly first), using the ESBTL so that it does not discard hydrogen atoms and the closing files after OpenBabel accessed them, in case something unexpected happened.

The following is a system test showing the run of the application in a Jupyter Notebook with the input file 1bna.pdb:

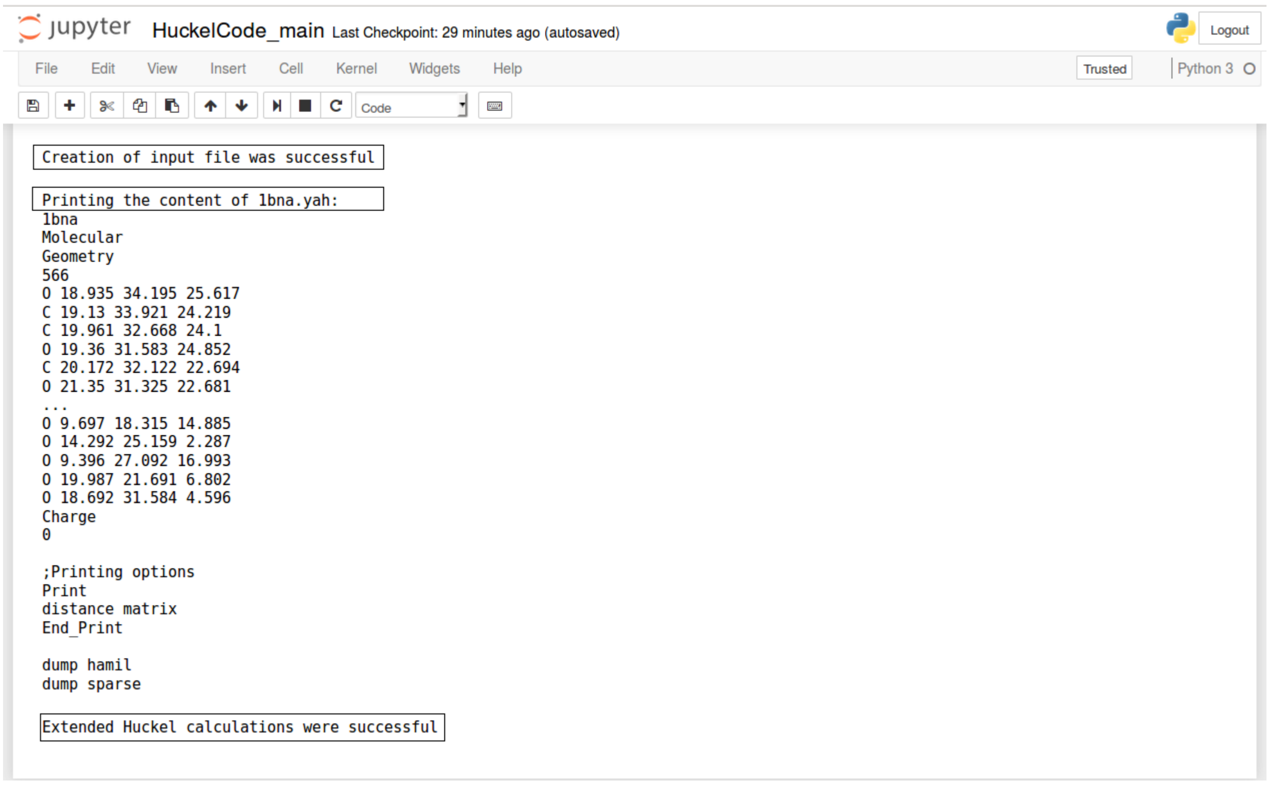

jupyter
HuckelCode_main Last Checkpoint: 28 minutes ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Code
Toggle code

```
Molecule filename: 1bna.pdb
Configuration filename: config.txt
```

Please set configuration parameters by clicking the checkboxes below:

Distance matrix	✓	Levels	☐
Overlap population	☐	Fermi	☐
Reduced overlap population	☐	Orbital energy	☐
Charge matrix	☐	Orbital coefficient	☐
Wave functions	☐	Orbital mapping	☐
Net charges	☐	Dump overlap	☐
Overlap matrix	☐	Dump Hamiltonian	✓
Hamiltonian matrix	☐	Dump sparse	✓
Electrostatic	☐	Dump Matrix Market	☐

Figure 3.8: Setting configuration parameters through the User Interface



```
Creation of input file was successful

Printing the content of 1bna.yah:
1bna
Molecular
Geometry
566
O 18.935 34.195 25.617
C 19.13 33.921 24.219
C 19.961 32.668 24.1
O 19.36 31.583 24.852
C 20.172 32.122 22.694
O 21.35 31.325 22.681
...
O 9.697 18.315 14.885
O 14.292 25.159 2.287
O 9.396 27.092 16.993
O 19.987 21.691 6.802
O 18.692 31.584 4.596
Charge
0

;Printing options
Print
distance matrix
End_Print

dump hamil
dump sparse

Extended Huckel calculations were successful
```

Figure 3.9: Showing progress of modules and content of intermediary file

3.9 Conclusion

The main topic of this thesis was the creation of new software in the field of quantum chemistry and observing possible changes in performance related to previously existing applications.

During the course of the work, software was made to assist calculations for the extended Hückel method. The end product created during this thesis can process more than 100 formats that are supported by OpenBabel. After carrying out a conversion between formats, differently formatted files can be transformed into an input for the calculation module. Following the results of measurements taken, it can be concluded that performance and thus execution time of the calculation module greatly comes down to the LAPACK and BLAS distributions used. When choosing a specific one, a package with optimized and possible multi-threaded routines may be preferable. When considering the output format to be used, most of the times it can be a favorable decision to use a sparse format because it is more beneficial than a dense matrix format in terms of memory management. Other aspect for this choice may also be the easier visual representation of such a matrix.

Chapter 4

Bibliography

[1] Khanacademy: The quantum mechanical model of the atom https://www.khanacademy.org/science/physics/quantum-physics/quantum-numbers-and-orbitals/a/the-quantum-mechanical-model-of-the-atom April 27, 2017

[2] Purdue university, College of Science, Chemical Education Division Groups: Molecular Orbital Theory 2004 https://chem.libretexts.org/Textbook_Maps/Physical_and_Theoretical_Chemistry_Textbook_Maps/Map%3A_Quantum_States_of_Atoms_and_Molecules_(Zielinksi_et_al.)/10%3A_Theories_of_Electronic_Molecular_Structure/10.06%3A_Semi-Empirical_Methods%3A_Extended_H%C3%BCckel April 27, 2017

[3] Roald Hoffmann:"An Extended Hückel Theory. I. Hydrocarbons.", 09/1963 J. Chem. Phys. 39 (6): 1397–1412. Bibcode:1963JChPh..39.1397H. doi:10.1063/1.1734456 (further details unknown)

[4] J. P. Lowe and K. A. Peterson, Quantum Chemistry, 3rd Edition: The Extended Huckel Method, (Chapter 10), 1 April 2014, https://dasher.wustl.edu/chem478/reading/extended-huckel-lowe.pdf April 27, 2017

[5] G.A.Landrum and W.V.Glassey, Yet Another Extended Hückel Molecular Orbital Package (YAeHMOP) Version 3.0 User Manual, March 2001, [58]

[6] G.A.Landrum and W.V.Glassey, bind (ver 3.0). bind is distributed as part of the YAeHMOP extended Hückel molecular orbital package and is freely available on the WWW at http://sourceforge.net/projects/yaehmop/

[7] Worldwide Protein Data Bank: Introduction 2010 http://www.wwpdb.org/documentation/file-format-content/format33/sect1.html April 16, 2017

[8] Worldwide Protein Data Bank: Title Section 2010 http://www.wwpdb.org/documentation/file-format-content/format33/sect2.html April 16, 2017

[9] Worldwide Protein Data Bank: Coordinate Section 2010 http://www.wwpdb.org/documentation/file-format-content/format33/sect9.html April 16, 2017

[10] RCSB PDB: Guide to Understanding PDB Data, date of submission unknown https://pdb101.rcsb.org/learn/guide-to-understanding-pdb-data/dealing-with-coordinates

[11] April 16, 2017 Worldwide Protein Data Bank: Connectivity Section 2010 http://www.wwpdb.org/documentation/file-format-content/format33/sect10.html April 16, 2017

[12] National Institute of Standards and Technology, Mathematical and Computational Sciences Division: Text File Formats 14 August 2013 http://math.nist.gov/MatrixMarket/formats.html April 29, 2017

[13] Ronald F. Boisvert, Roldan Pozo, Karin A. Remington, National Institute of Standards and Technology December 1996 The Matrix Market Exchange Formats: Initial Design http://math.nist.gov/MatrixMarket/reports/MMformat.ps April 29, 2017

[14] N M O'Boyle, M Banck, C A James, C Morley, T Vandermeersch, and G R Hutchison. "Open Babel: An open chemical toolbox." J. Cheminf. (2011), 3, 33. DOI:10.1186/1758-2946-3-33 April 18, 2017

[15] The Open Babel Package, version 2.3.1, Oct 2011 http://openbabel.org April 18, 2017

[16] The Open Babel API Documentation, Open Babel, version 2.3., February 28 2012, http://openbabel.org/dev-api/, April 18, 2017

[17] GNU Operating System: GNU Make Manual, May 22, 2016 http://www.gnu.org/software/make/manual/make.html, May 8, 2017

[18] The IEEE and The Open Group: make utility page, 2016 http://pubs.opengroup.org/onlinepubs/9699919799/utilities/make.html#tag_20_76_13_04 May 8, 2017

[19] Kitware: CMake, date of submit unknown, https://cmake.org/ May 8, 2017

[20] University of Tennessee, University of California, Berkeley; University of Colorado Denver and NAG Ltd.: Lapack, February 16, 2017 http://www.netlib.org/lapack/ May 11, 2017

[21] University of Tennessee, University of California, Berkeley; University of Colorado Denver and NAG Ltd.: BLAS (Basic Linear Algebra Subprograms), February 16, 2017 http://www.netlib.org/blas/ May 11, 2017

[22] Standard C++ Foundation: Serialization and Unserialization, 2017, https://isocpp.org/wiki/faq/serialization#serialize-binary-format May 2, 2017

[23] Project Jupyter, date of submission unknown http://jupyter.org/ May 13, 2017

[24] SageMath Inc: SageMathCloud 2017, https://cloud.sagemath.com May 13, 2017

[25] Julie Bernauer, Frédéric Cazals and Sébastien Loriot: ESBTL, a PDB parser and data structure for the structural and geometric analysis of biological macromolecules Dec 1 2011 http://esbtl.sourceforge.net/refman/doc/html/index.html May 13, 2017