Advanced Quantum Chemistry Methods

Release SoSe2021

Grimme group

Apr 07, 2022

SETUP

This is the guide to the practical course for QC2.

```
!> This is your module to write your very own SCF program.
module scf_main
    !> Include standard Fortran environment for IO
    use iso_fortran_env, only : output_unit, error_unit

    ! ------------------------------------------------------------------------
    !> library functions provided by your lab assistents:

    !> interface to LAPACK's double precision symmetric eigenvalue solver (dspev)
    !  examples:
    !  call solve_spev(mat, eigval, eigvec)
    use linear_algebra, only : solve_spev

    !> expansion of slater-functions into contracted gaussians,
    !  coefficients and primitive exponents are taken from R.F. Stewart, JCP, 1970
    !  example:
    !  call expand_slater(zeta, alpha, coeff)
    use slater, only : expand_slater

    !> calculates one-electron integrals and two-electron integrals over
    !  spherical gaussians (s-functions). One-electron quanities supported
    !  are overlap, kinetic energy and nuclear attraction integrals.
    !  Two-electron integrals are provided in chemist notation.
    !  examples:
    !  call oneint(xyz, chrg, r_a, r_b, alp, bet, ca, ca, s, t, v)
    !  call twoint(r_a, r_b, r_c, r_d, alp, bet, gam, del, ca, cb, cc, cd, g)
    use integrals, only : oneint, twoint

    !> prints a matrix quantity to screen
    !  examples:
    !  call write_vector(vec, name='vector')
    !  call write_matrix(mat, name='matrix')
    !  call write_matrix(mat, name='packed matrix')
    use print_matrix, only : write_vector, write_matrix

    !> other tools that may help you jump ahead with I/O-heavy tasks
    !  example:
    !  call read_line(input, line)
    use io_tools, only : read_line

    !> Always declare everything explicitly
    implicit none

    !> All subroutines within this module are not exported, except for scf_prog
    !  which is the entry point to your program
    private
    public :: scf_prog

    !> Selecting double precision real number
    integer, parameter :: wp = selected_real_kind(15)


contains


!> This is the entry point to your program, do not modify the dummy arguments
```

(continues on next page)

(continued from previous page)

```
!  without adjusting the call in lib/prog.f90
subroutine scf_prog(input)

    !> Always declare everything explicitly
    implicit none

    !> IO unit bound to the input file
    integer, intent(in) :: input

    !> System specific data
    !> Number of atoms
    integer :: nat

    !> Number of electrons
    integer :: nel

    !> Atom coordinates of the system, all distances in bohr
    real(wp), allocatable :: xyz(:,:)

    !> Nuclear charges
    real(wp), allocatable :: chrg(:)

    !> Number of basis functions
    integer :: nbf

    !> Slater exponents of basis functions
    real(wp),allocatable :: zeta(:)

    !> Hartree-Fock energy
    real(wp) :: escf

    !  declarations may not be complete, so you have to add your own soon.
    !  Create a program that reads the input and prints out final results.
    !  And, please, indent your code.

    !  Write the self-consistent field procedure in a subroutine.
    write(output_unit, '(a)') 'Here could start a Hartree-Fock calculation'

end subroutine scf_prog


end module scf_main
```

CHAPTER

ONE

SETUP PREREQUISITES

To start with this course, you need the to install necessary programs and setup your system to allow programming. Depending on the platform and operating systems you are using the steps differ but eventually you should be able to use the course material independently of your platform.

Important: If you are working at a CIP pool computer in the Mulliken Center for Theoretical Chemistry we have taken care for you about the setup and you can immediately continue with the next chapter.

Contents

1.1 Windows 10

There are multiple ways to enable your Windows system for development. For this course we will present two methods, either running natively (see *Install MSYS2*) or with a virtualisation of a Linux subsystem (see *Install WSL*).

1.1.1 Install MSYS2

The MSYS2 project provides a package manager and makes many common Unix tools available for Windows. It comes with its own bash-like shell which allows to easily follow this course.

Download the `msys2-x86_64-YYYYMMDD.exe` installer from the MSYS2 webpage and run the installer. MSYS2 will create three new desktop shortcuts: *MSYS terminal*, *MinGW64 terminal* and *MinGW32 terminal* (more information on MSYS2 terminals is available here).

You can either use the *MSYS terminal* or the *MinGW64 terminal* for this course we recommend to use the latter.

Open a new terminal and update your installation with

```
pacman -Syu
```

You might have to update MSYS2 and `pacman` first, restart the terminal and run the above command again to update the installed packages.

If you are using the *MinGW64 terminal* you can install the required software with

```
pacman -S git mingw-w64-x86_64-gcc-fortran mingw-w64-x86_64-make mingw-w64-x86_64-fpm␣
↪vim
```

If you use the *MSYS terminal* leave out the `mingw-w64-x86_64` prefixes to install the required software.

After having installed the necessary software, you need to download the course material. Unzip the `course-material.zip` archive to your home directory and you are setup to start with the next chapter.

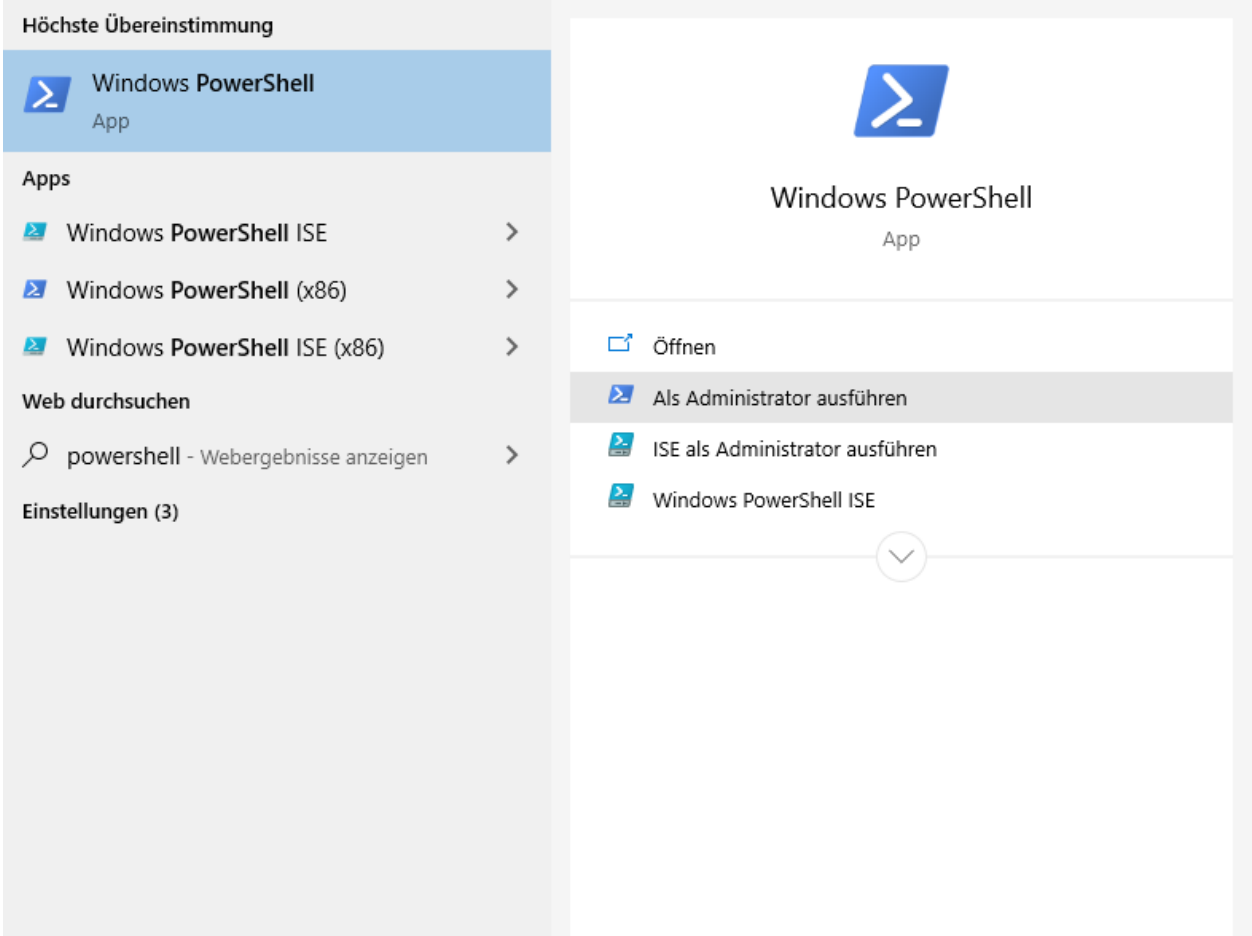
Note: Your home directories `C:\Users\<name>` will be accessible under `/c/Users/<name>` in the MSYS2 terminals, look there if you are searching for your downloaded files.

Tip: The content of the archive might be potentially important as a starting point for your SCF program later.

1.1.2 Install WSL

With Windows 10 comes the possibility to integrate a Linux Console to your Windows PC via the Windows Subsystem for Linux (WSL). This is in particular useful for you as a Windows user to participate in our Linux based course from home. This guide will show you how to use WSL in order to install a Unix-based console that allows you to complete all the tasks of the course (it is basically a shortened version of the Microsoft documentation for installing WSL).

In order to install WSL, you first have to enable the optional WSL feature. Open the Windows PowerShell as administrator (for example by typing `powershell` in the search field of the taskbar).



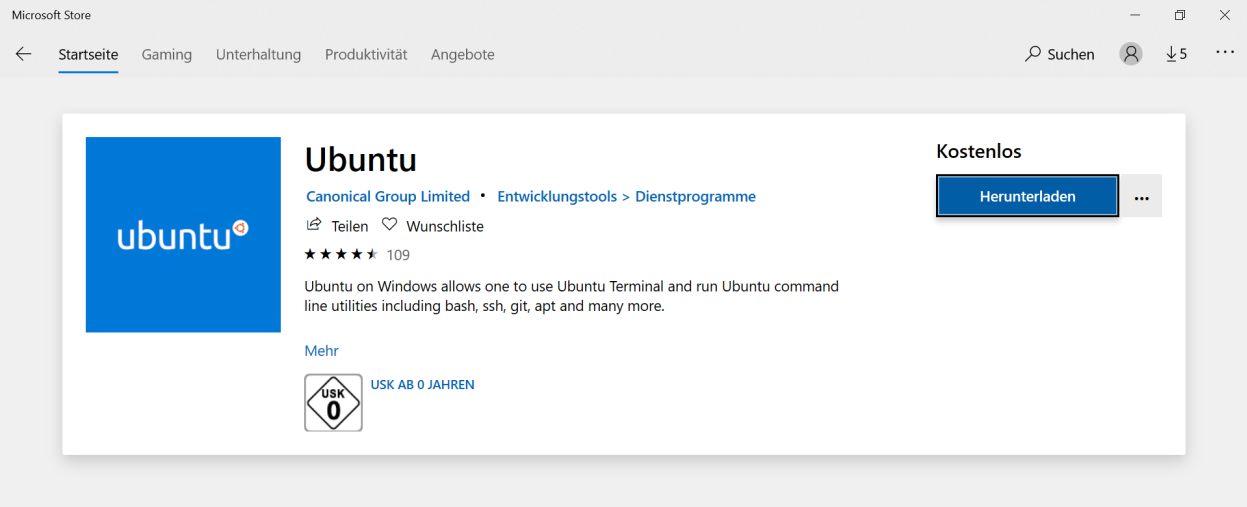
Type the following command in PowerShell and press `<Enter>`:

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux
```

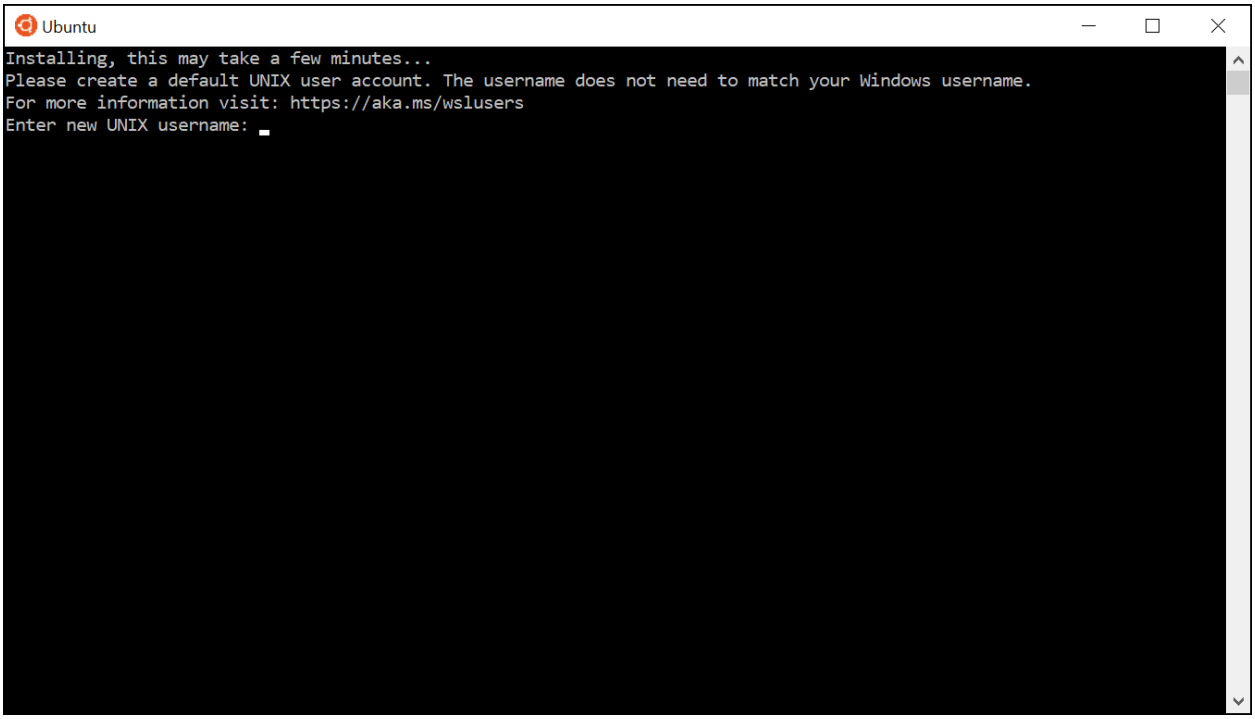
Afterwards please restart your computer if you are asked to.

1.1.3 Install Linux Distribution

You can now install a Linux distribution of your choice to use with WSL. We recommend Ubuntu. You can install it from the Microsoft Store. Just type `Ubuntu` in the search field and download the application.



After the installation was successful, you can start the Linux distribution. Ubuntu will open a console and install itself. You will be ask to choose a username and a password. Those are equivalent to the ones you would choose on a real Ubuntu machine.



Your new console will now show `linuxusername@hostname:/some/path$`, similar to an Ubuntu console. `linuxusername` is your chosen username in the Linux distribution and `hostname` the name of your computer. The directory in which the window is opened (`/some/path`) will probably be your Linux home directory `~` (see *Shell in a nutshell*). Note that this home directory can not simply be accessed via the Windows explorer. But the other way around, you can find your Windows home directory in `/mnt/c/Users/windowsusername`, where `windowsusername` is your username on your Windows computer. If you want to open new files with some Windows program, we recommend to create and save all new files in the mentioned directory or subfolders of it.

You can now also access your Linux console through a Windows console such as CMD. To open CMD, just type `cmd` in the search field of the Windows taskbar and press `Enter`. By entering the command `wsl` or `bash`, you will change to the Linux console in your current directory which is your Windows home directory.

Note: If you feel advanced in handling the shown terminals of different distributions on your Windows computer, you can try the Windows Terminal. It allows you to manage your WSL terminals and may make your home office more comfortable. However, it is still a preview release and thus the installation might require some effort.

You have now successfully installed a Linux console on your Windows computer and can continue with the Ubuntu part of this documentation (see *Ubuntu*).

1.2 Ubuntu

Starting with a fresh version of Ubuntu 18.04 we have to install a few programs first. You have to install the packages: `git`, `gfortran`, `make`, `atom` and `vim`. We will assume you are working with `apt` to install packages, in case you prefer another package manager, feel free to install the packages listed here with this one (see Ubuntu install & remove software).

```
sudo apt install git gfortran make atom vim
```

Note: Some packages, especially `vim` and `make` might already be installed on your system, but it does not harm to include them here again.

For the programming course we recommend to use the Fortran package manager (fpm), which can be downloaded from here. or setup from conda-forge as described in *Fortran package manager from conda*.

After having installed the necessary software, you need to download the course material. Unzip the `course-material.zip` archive to your home directory and you are setup to start with the next chapter.

Tip: The content of the archive might be potentially important as a starting point for your SCF program later.

1.2.1 Fortran package manager from conda

You can install the Fortran package manager (fpm) easily from conda-forge, a large scientific software repository. To do so, download a mambaforge installer from the conda-forge project. For Ubuntu `Mambaforge-Linux-x86_64.sh` is the correct choice. Run the installer as user (no sudo required) to setup your conda base environment:

```
sh Mambaforge-Linux-x86_64.sh
```

Your prompt should now show a `(base)` label in front, which signals you that you now have access to the mamba package manager and the conda environment manager.

Note: Those two tools, mamba and conda, are very powerful to create reproducable development and production environments for scientific work. For this course they allow us to easily install software, that would be difficult to install otherwise or software which is not yet available in all Linux distributions. Many of our groups software is available over conda-forge as well.

To add Fortran package manager (fpm) to your base environment run

```
mamba install fpm
```

Tip: Alternatively, you can create a separate environment for fpm with

```
mamba create -n fpm fpm
conda activate fpm
```

When creating a separate environment, you will always have to activate it after starting a new shell, but it allows you more fine grained control over your installed software.

1.3 MacOS

To setup your MacOS for the course follow this steps

1. Install Xcode from the the App Store
2. Open a terminal from /Applications/Utilities/
3. Install command line tools with

   ```
   xcode-select --install
   ```

4. Install `gfortran` either by installing it from here or by using Homebrew (see *Installing with Homebrew*)
5. Install the Fortran package manager from conda (see *Fortran package manager from conda*) or homebrew (see *Installing with Homebrew*)
6. Download and install atom.

Important: The default shell on MacOS is not `bash`, but a `zsh`, but they should be mostly compatible. In case you want to follow the course with a `bash` start a new `bash` instance after opening your terminal with

```
exec bash
```

After having installed the necessary software, you need to download the course material. Unzip the `course-material.zip` archive to your home directory and you are setup to start with the next chapter.

Note: Some of the keyboard shortcuts in the next chapter are targeted at Linux OS, you probably have to use the `cmd` key instead of the `ctrl` key.

1.3.1 Installing with Homebrew

You can manage packages for your Mac using homebrew. To install homebrew download the installation script from the official homebrew channel at GitHub:

```
curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.sh >␣
↪install-homebrew.sh
/bin/bash install-homebrew.sh
```

The script will ask you a few questions to complete the installation process, for more information on the homebrew installation check their documentation.

To install `gfortran` we will install the `gcc` formula, where it is included, with

```
brew install gcc
```

Additionally, install the Fortran package manager (fpm) with

```
brew tap awvwgk/fpm
brew install fpm
```

CHAPTER

TWO

WORKING ON LINUX

As you participate in this lab exercise, you must have at least a basic knowledge of working with a Linux computer. This chapter shows you the basic commands to get along with your computational environment. First of all, you will be provided with a **username** and a **password**. This represents your user account for the whole lab course and all the data you produce is available with this information.

Note: Some of the keyboard shortcuts in this chapter are targeted at the setup of the CIP Pool computers in the Mulliken Center for Theoretical Chemistry and might not work as expected for you.

In case you use KDE, everything will probably work just fine, if you use another OS or window manager commands might differ slightly.

Contents

2.1 Login

The first thing you will see after booting the computer is the *login-screen*.

Dependent on the Linux version you are using this screen may look slightly different. After you typed in your username and password you press the `<Return>` key and the graphical desktop will be loaded. All further actions will take place in this environment. The next and certainly most important thing is to access the Linux command line. This is usually done by starting a terminal-emulator, which is called **shell** or **terminal**. On the machines you are using there should be a quick start icon directly on the desktop. By clicking on this icon, a window is opened which allows you to communicate with the PC in a command-line mode.

2.2 Shell in a nutshell

After executing the terminal-emulator you will end up with a window, which looks similar to the following image:

```
ehlert@c01:~> pwd
/home/ehlert
ehlert@c01:~> ls
Desktop     Music      QCII
Documents   Pictures   Templates
Downloads   Public     Videos
ehlert@c01:~> cd QCII
ehlert@c01:~/QCII>
```

On the left, you can see the so-called **prompt**. Depending on the default settings of your system it provides you with various information. In a standard configuration, it will show: `username@hostname:~>`, where `username` is your username, `hostname` is the name of the computer and the tilde (`~`) shows that you are currently located in your **home directory** (`/home/username`). The Linux file structure follows the *Filesystem Hierarchy Standard*, which ensures a similar file structure on every version of Linux you can get. As you work with the system you will rapidly gain experience with the different directories and their purposes. For now, you should know that you are in your home directory which is located at `/home/username` and is abbreviated by `~`.

With your user account you have the power to create, edit, and delete files in your home directory at will. But with great power comes great responsibility. You have to be careful with the commands you execute when you delete or overwrite a file it is gone for good. With that in mind, we can now start with the first couple of commands. To see exactly which directory you are in, type `pwd` (print working directory) and press `<Return>`. Since you are in your home directory, this will print the path to that home directory to the screen. Note that all input in the terminal is case-sensitive.

```
ehlert@c01:~> pwd
/home/ehlert
```

Next thing we want to know is what is inside our current location, for this we use the command `ls`, short for list:

```
ehlert@c01:~> ls
Desktop     Music      QCII
Documents   Pictures   Templates
Downloads   Public     Videos
```

We can add options to the `ls` command like `-l` to use the use a long listing format:

```
ehlert@c01:~> ls -l
total 574500
drwxr-xr-x  2 ehlert thch      4096 Jun  7  2018 Desktop
drwxr-xr-x  2 ehlert thch      4096 Jun  7  2018 Documents
drwxr-xr-x  2 ehlert thch      4096 Jun  7  2018 Downloads
drwxr-xr-x  2 ehlert thch      4096 Jun  7  2018 Music
drwxr-xr-x  2 ehlert thch      4096 Jun  7  2018 Pictures
drwxr-xr-x  2 ehlert thch      4096 Jun  7  2018 Public
drwxr-xr-x  4 ehlert thch      4096 Jan 14 09:09 QCII
drwxr-xr-x  2 ehlert thch      4096 Jun  7  2018 Templates
drwxr-xr-x  2 ehlert thch      4096 Jun  7  2018 Videos
```

Alternatively we can provide `ls` with a path, it will then list all the files within this directory

```
ehlert@c01:~> ls QCII
tutorial   scf
```

Of course we can also combine options and paths for `ls`.

You might wonder what a path is, we will go into more detail above them now as they are important for working with Linux. Whenever we refer to a file or a directory on the commandline we are in fact referring to its path. A path is identifying a particular file or directory on the system. Your filesystem starts at the root `/` and can be referenced absolutely from this root or relative from your current working directory. Every directory has at least two links to other directories, to itself `.` (dot) and to its parent `..` (dotdot), which can be used to build paths to reference to any file or directory on your system.

Up to now we only looked around, but we can also change the directory, which is done by the command `cd`, short for change directory.

```
ehlert@c01:~> cd QCII
ehlert@c01:~/QCII> cd .
ehlert@c01:~/QCII> cd ../..
ehlert@c01:/home> cd -
/home/ehlert/QCII
ehlert@c01:~/QCII> cd
ehlert@c01:~>
```

What did just happen?

1. We changed to the QCII directory. Our prompt helpfully reports that we are now in the QCII directory, so usually there is no need to use `pwd`.
2. Next we change to the directory itself using its dot link, and we stay in the same directory as expected.
3. Now we change to the parent directory of the QCII parent directory, which is the parent directory of our home directory. You can easily chain links together using the slash character `/`.
4. In case a change directory brings you to the wrong place you can always go back to the last directory you visited by `cd -`. The absolute path of the directory is also printed so we can be sure to be in the right place.
5. To go back to your home directory use `cd` without an argument.

We differenciated files and directories above, which is not quite true, in Linux everything is a file, also a directory, even your keyboard is a file (one which is only read from), your monitor is also a file (one which is only written to). It will not affect us when working with Linux but it is helps to keep it in mind when trying to understand how Linux manages files and directories.

A standard set of commands is shown in the following table:

command	description
`pwd`	print the working directory
`ls`	lists the files in the current directory
`cd <name>`	change to the directory with `<name>`
`cd ..`	change to the parent directory
`cp <old> <new>`	copy file `<old>` to `<new>`
`cp -r <old> <new>`	copy directory `<old>` to `<new>`
`mv <old> <new>`	move (rename) file/directory
`rm <name>`	remove file with `<name>`
`rm -r <name>`	remove directory recursively (caution!)
`mkdir <name>`	make a new directory with `<name>`
`rmdir <name>`	remove (empty) directory with `<name>`

This is only a very basic list of commands available and some of them have a huge variety of options that can not be listed here, and will hardly concern you. For all options the program can be started with `<command> --help` and a complete summary can be found in its manual page by `man <command>`.

Exercise 1

To get familiar with the shell try to achieve the following task

1. change to the `QCII` directory
2. find or create the `tutorial` directory in `QCII`
3. rename the `tutorial` directory to `shell tutorial`
4. change to the newly created directory

Solutions 1

A sequence of this command would achieve the wished results.

```
username@hostname:~> cd QCII
username@hostname:~/QCII> ls tutorial
tutorial
username@hostname:~/QCII> mv tutorial shell tutorial
mv: cannot move 'tutorial' to a subdirectory of itself, 'tutorial/tutorial'
mv: cannot stat 'shell': No such file or directory
username@hostname:~/QCII> mv tutorial 'shell tutorial'
username@hostname:~/QCII> cd shell\ tutorial
username@hostname:~/QCII/shell tutorial>
```

Note that you have to escape the space in `shell tutorial` in some way.

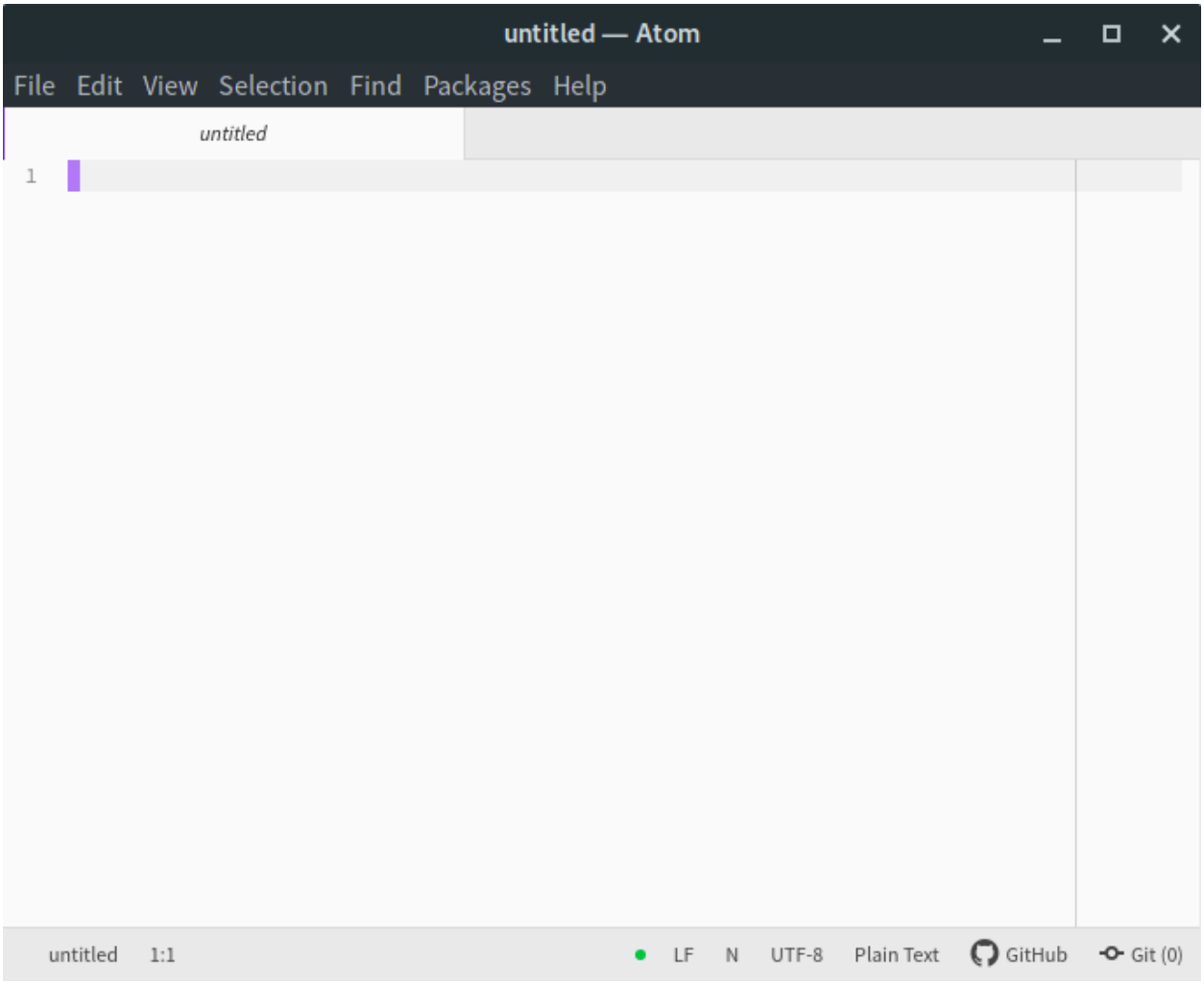
2.3 Editors

To access and edit any text file in Linux you will need an editor. A huge variety of editors exist and your difficult task is to pick the one you are most comfortable with. We introduce the most common ones in this chapter but feel free to work with the editor that fits you the best.

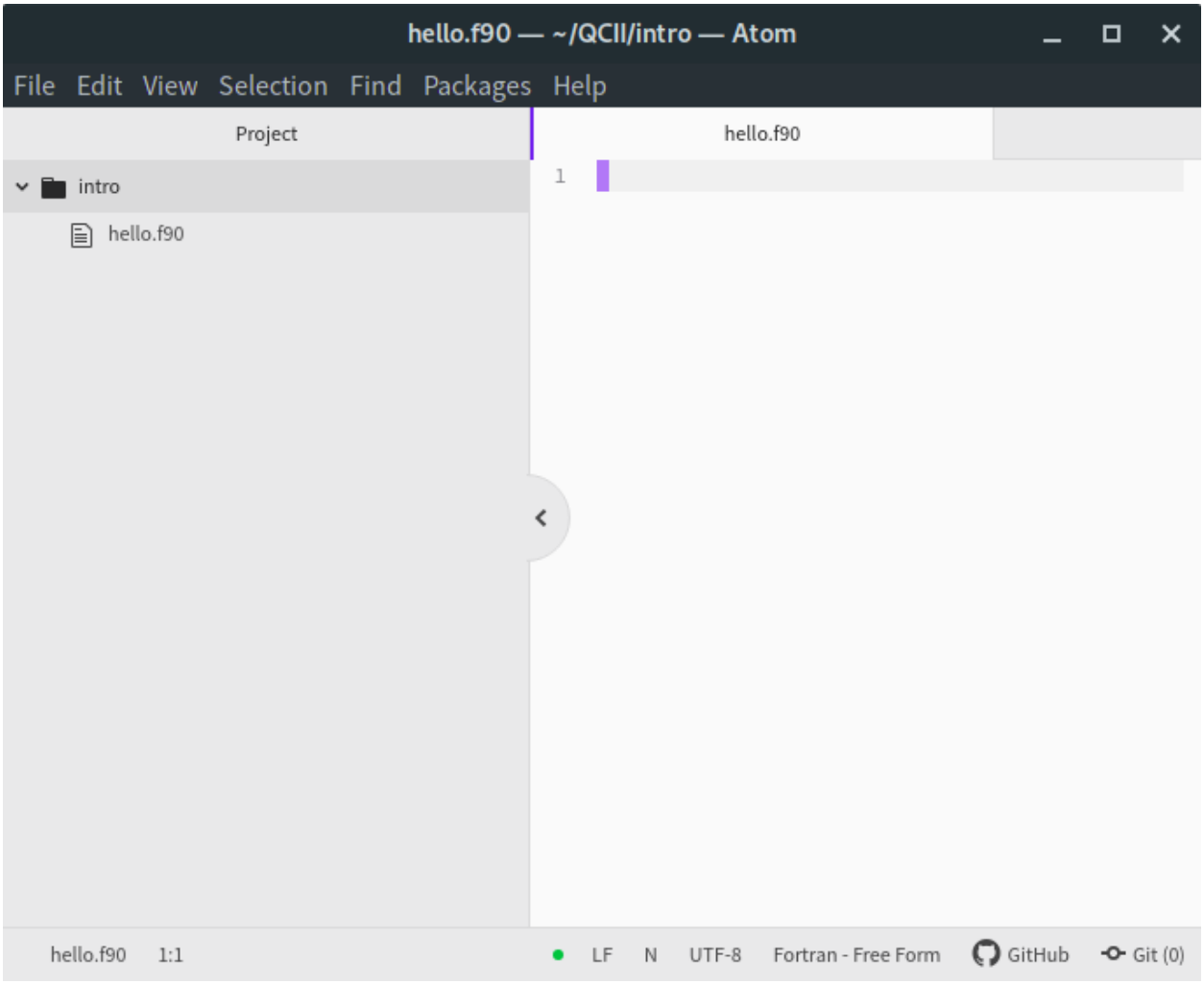
2.3.1 Atom

`atom` is a rather heavyweight but easy-to-use editor, which is built on-top of the `electron` framework and has comparable capabilities to a web browser. It is available for Linux, macOS and Windows. Since we are dealing here with electrons and atoms the choice of programs could not have been better, unfortunately, they do not know much about quantum chemistry. For you can work entirely in `atom`, but you need some extension which might already be installed with your version of `atom`. If not install `language-fortran`, `build`, `build-make` and `terminal-tab` at the setting menu `<ctrl>-<,>` under *install*. `atom` can be easily extended to a complete integrated development environment, but we will assume you are working with a vanilla version including the four additional packages here.

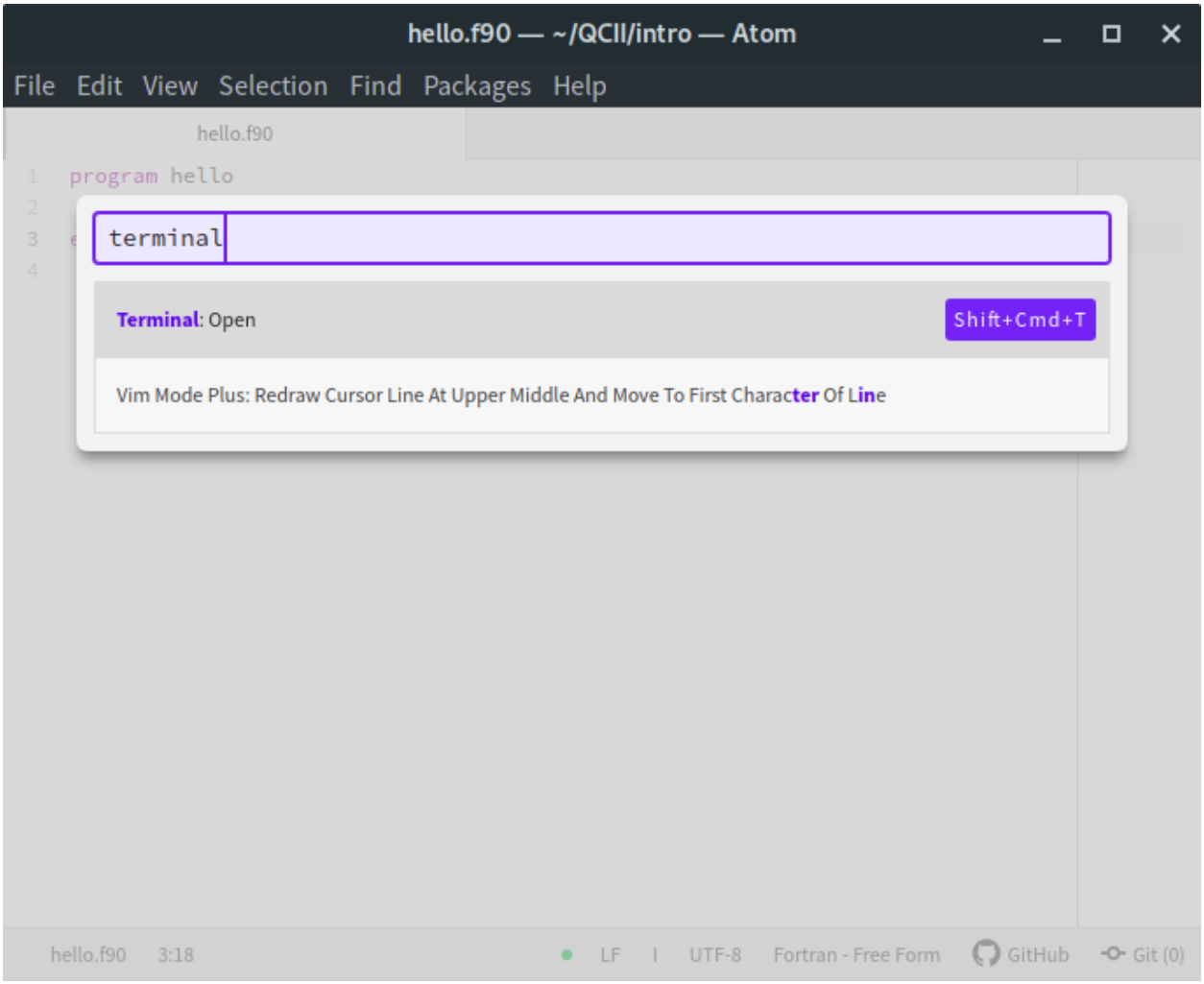
Start `atom` by using `<alt>-<F2>` and typing atom in the quick launch bar or searching the start menu for `atom`.



Having started a new instance of `atom` you either have already an empty file opened or you can open a new file by `<ctrl>-<n>`, save the file with `<ctrl>-<s>` by creating a new directory and giving the file a name there, if you name the file `hello.f90` it will be automatically identified as Fortran source code.

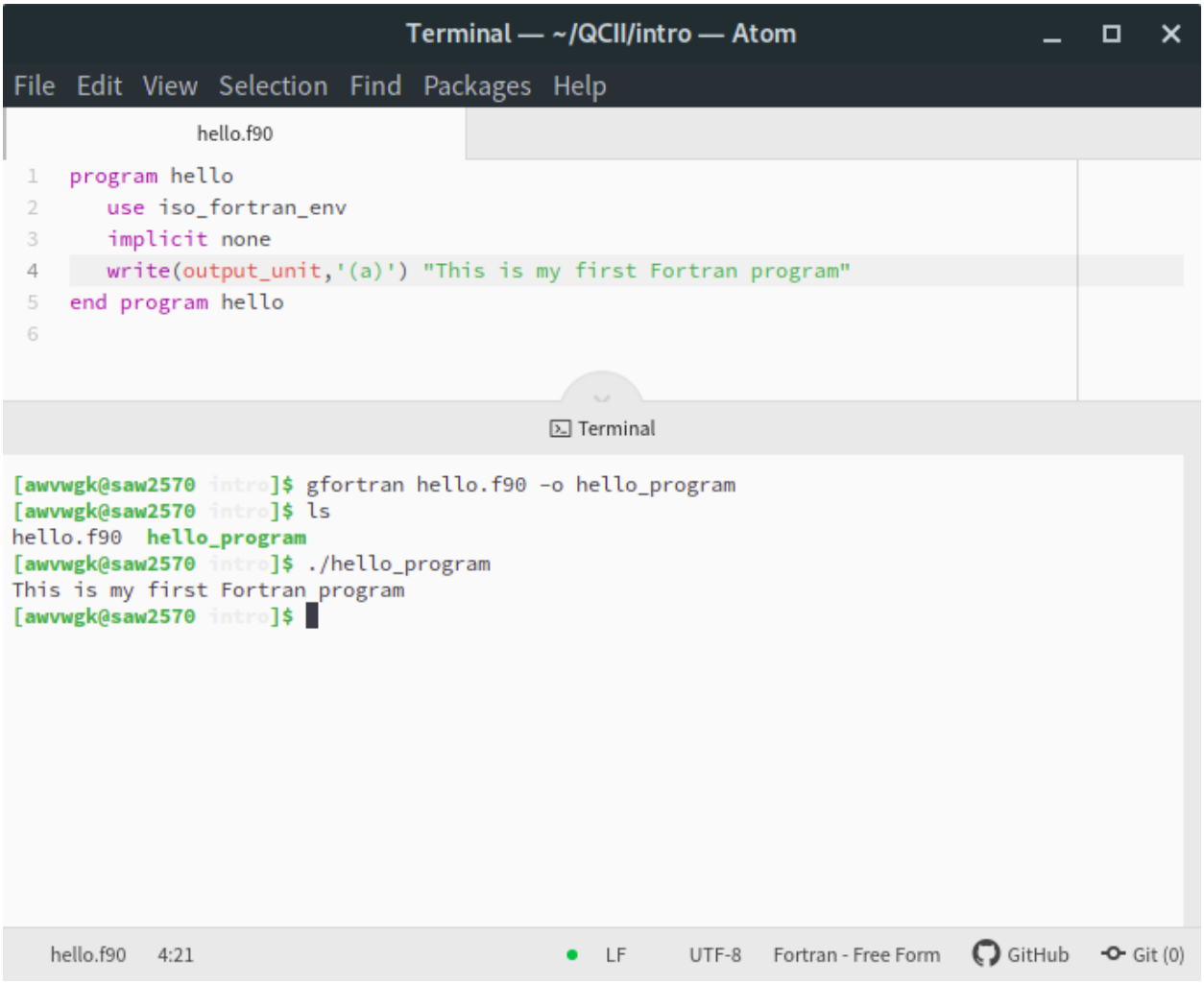


You can start a shell by hitting `<ctrl>-<shft>-<p>` and typing `terminal` in the quick launcher of `atom` the shell can be used for all commands you previously learned.



Note: If you are using atom in Windows and have installed WSL, you can start a Unix shell by typing `wsl` or `bash` in the command line of the terminal you just opened in `atom`.

Later you can use it to compile and execute your programs without leaving your editor. For example, we write a simple Fortran program to print a line to the screen, save it and compile it using `gfortran` in our shell inside `atom`.



2.3.2 Vim

We usually prefer to use `vim` which is a very powerful and lightweight editor once you have mastered the initial steep learning curve. It has the advantage of being installed by default on almost any Linux machine and is even fully usable without a graphical user interface.

However, getting past the initial learning curve can take the better part of a month, but having truly mastered `vim` usually results in a huge performance gain when developing. We encourage you to pick up `vim` instead of `atom`.

To get started with `vim` open a new terminal (type `<alt>-<F2>` for the quick launch menu, then type `konsole` or search for it in the menu) and type `vimtutor`. This will launch an instance of `vim` with an extensive introduction for using it, follow the instructions until you feel confident navigating and editing files with `vim`.

> **Attention:** Don't read past this note without finishing `vimtutor`!

To make working with `vim` easier for you, we provide this `.vimrc` for you:

Listing 1: ~/.vimrc

```
" Defaults for new vim users.
source $VIMRUNTIME/defaults.vim
" Ignore this filetypes in the wildmenu
set wildignore=*.o,*~,*.pyc,*.mod
" Make search case insensitive, but only as long as it contains only lowercase
set ignorecase
set smartcase
" Automatic indentation
set autoindent
set smartindent
set tabstop=4
set shiftwidth=4
set smarttab
set expandtab
" Fortran specific options
let fortran_do_enddo=1
let fortran_free_source=1
```

In case it is not yet in your `.vimrc` we recommend copying, if you are not happy with something we put in here, feel free to modify or replace it, you can also add new configurations if you like.

After you have covered the basics, there are some tricks you might find useful.

Tip: We recommend working with a *single* instance of `vim` in *one* terminal, if used right `vim` can provide all functions from your file navigator and terminal.

---

1. Open your current working directories with `vim .` and you will find yourself in the `netrw` file navigator.
2. Navigate to a file you would like to open and hit `<Enter>`, it will be opened in the same `vim` instance, to get back type `:E` in normal mode and find yourself back in `netrw`.
3. To open a new window type `<ctrl>-w n`, you can close the window again with `<ctrl>-w q` or by typing `:q` as usual.
4. To open a second window you can split your `vim` window by using `<ctrl>-w v` (for vertical splitting) or `<ctrl>-w s` (for horizontal splitting) to have two windows with the same file which can be used independently.

Tip: If your `vim` instance freeze, you hit `<ctrl>-s` by accident, which tells the hosting terminal to freeze, unfreeze it with `<ctrl>-q`.

---

5. If you have your mouse enabled for `vim` you can jump between windows by clicking into another window, the faster way is to use `<ctrl>-w w` to go to the next window.

Make yourself familiar with navigation between multiple windows by creating, closing and jumping between multiple windows. You can yank and paste content between the windows that way, which allows seamless transfer between different files.

6. Now go in one of the windows back to `netrw`, we want to create a new directory without using `:!mkdir ...`, type `d` in normal mode in your `netrw` instance and you should be prompted to provide a name.
7. You can delete it again with `D`, do so by moving your cursor over the file or directory and press `D`, then accept your choice in the prompt.
8. Now we want a new file, the easiest way would be `:e ...`, but this path has to be relative from the working directory we started our `vim` instance in, so we use `netrw` instead and type `%` which prompts as to provide a name

and opens the new file afterward in a new `vim` window.

Let's open a new file `hello.f90` and enter

```
program hello
   implicit none
   write(*, '(a)') "My first Fortran program"
end program hello
```

Tip: In case the syntax highlighting looks strange, `vim` is trying to use Fortran 77 highlighting, add `let fortran_free_source=1` to your `.vimrc` to get the correct Fortran 90 highlighting and restart `vim` for it to take effect.

After saving the file, compile and run it by typing `:!gfortran % && ./a.out`, you should see something like this printout in your terminal:

```
My first Fortran program

Press ENTER or type command to continue
```

The first line is from your program, the second one is produced by `vim`.

Note: To switch between your terminal and `vim` use `<ctrl>-z` to stop `vim` and get it back from the terminal by using the command `fg`.

At this point, you should be ready to use `vim` in production, happy coding.

CHAPTER

THREE

INTRODUCTION TO FORTRAN

Contents

Note: To learn any new programming language good learning resources are important. This introduction covers the basic language features you will need for this course but Fortran offers much more functionality.

Good resources can be found at fortran-lang.org. Especially the learn category offers useful introductory material to Fortran programming and links to additional resources.

The fortran90.org offers a rich portfolio of Fortran learning material. For students with prior experience in other programming languages like Python the Rosetta Stone might offer a good starting point as well.

Also, the Fortran Wiki has plenty of good material on Fortran and offers a comprehensive overview over many language features and programming models.

An active forum of the Fortran community is the fortran-lang discourse. While we offer our own forum solution for this course to discuss and troubleshoot, you are more than welcome to engage with the Fortran community there as well. Just

be mindful when starting threads specific to this course and tag them as *Help* or *Homework*.

3.1 General Principles

Programming is the art of telling a computer what to do, usually to perform an action or solve a problem which would be too difficult or laborious to do by hand. It thus involves the following steps:

1. Understanding the problem.
2. Formulating an approach/algorithm to the problem.
3. Translating the algorithm into a computer–compatible language: *Coding*
4. Compiling and running the program.
5. Analyzing the result and improving the program if necessary or desired.

This manual assumes that steps 1 and 2 have already been taken care of in the Quantum Chemistry I module. We will thus concern ourselves with the problems of translating an algorithm or a formula, spelled out on paper, to something the computer understands first.

3.2 Compiling and Running a Program

Before beginning to write computer code or *to code* in short, one needs to choose a programming language. Many of them exist: some are general, some are specifically designed for a task, e.g. web site design or piloting a plane, some have simple data structures, some have very special data structures like census data, some are rather new and follow very modern principles, some have a long history and thus rely on time-tested concepts.

In the case of this module, the Fortran programming language is chosen because it makes for code that is rather rapidly written with decent performance. It is also in frequent use in the field of quantum chemistry and scientific computing. The name Fortran is derived from Formula Translation and indicates that the language is designed for the task at hand, translating scientific equations into computer code.

Let's take a look at a complete Fortran program.

Listing 1: hello.f90

```
1 program hello
2 write(*, *) "This is probably the simplest Fortran program"
3 end program hello
```

If you were to execute this program, it would simply display its message and exit.

Exercise 1

Open a new file in `atom` and save as `hello.f90` in a new directory in your project directory, you will always create a new directory for each exercise.

Type in the program above. Fortran is case-insensitive, *i.e.* it mostly does not care about capitalization. Save the code to a file named `hello.f90`. All files written in the Fortran format must have the `.f90` extension.

Next, you will check that the file you created is where you need it, translate your program using the compiler `gfortran` to machine code. The resulting binary file can be executed, thus usually called executable.

```
gfortran hello.f90 -o helloprog
./helloprog
 This is probably the simplest Fortran program.
```

Directly after the `gfortran` command, you find the input file name. The `-o` flag allows you to name your program. Try to leave out the `-o helloprog` part and translate `hello.f90` again with `gfortran`. You will find that the default name for your executable is `a.out`. They should produce the same output.

Now that we can translate our program, we should check what it needs to create an executable, create an empty file `empty.f90` and try to translate it with `gfortran`.

```
gfortran empty.f90
/usr/bin/ld: /usr/lib/Scrt1.o: in function `_start':
(.text+0x24): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

This did not work as expected, `gfortran` tells you that your program is missing a *main* which it was about to *start*. The main program in Fortran is indicated by the `program` statement, which is not present in the empty file we gave to `gfortran`.

Important note about errors

Errors are not necessarily a bad thing, most of the programs you will use in this course will return useful error messages that will help you to learn about the underlying mechanisms and syntax.

Just consider it as an error driven development technique!

Let us reexamine the code in `hello.f90`, The first line is the *declaration section* which starts the main program and gives it an arbitrary name. The third line is the *termination section* which stops the execution of the program again and tells the compiler that we are done. The name used in this section has to match the *declaration section*.

Everything in between is the *execution section*, each statement in this section is executed when calling the translated program. We use the `write(*, *)` statement which causes the program to display whatever is behind it until the end of the line. The double quotes enclosing the sentence make the program recognize that the following characters are just that, *i.e.*, a sequence of characters (called a string) and not programming directives or variables.

3.3 The Fortran package manager

The Fortran package manager (fpm) is a recent advancement introduced by the Fortran community. It helps to abstract common tasks when developing Fortran applications like building and running executables or creating new projects.

> **Warning:** The Fortran package manager is still a relatively new project. Using new and recent software always has some risks of running into unexpected issues. We have carefully evaluated fpm and the advantage of the simple user interface outweighs potential issues.
>
> This course is structured such that it can be completed with or without using fpm.

Note: If you are doing this course on a machine in the Mulliken center you have to activate the fpm installation by using the `module` commands:

```
module use /home/abt-grimme/modulefiles
module load fpm
```

Create a new project with fpm by

```
fpm new --app myprogram
```

This will initialize a new Fortran project with a simple program setup. Enter the newly created directory and run it with

```
cd myprogram
fpm run
...
 hello from project myprogram
```

You can inspect the scaffold generated in `app/main.f90` and find a similar program unit as in your very first Fortran program:

Listing 2: app/main.f90

```
program main
  implicit none

  print *, "hello from project myproject"
end program main
```

Modify the source code with your editor of choice and simply invoke fpm run again. You will find that fpm takes care of automatically rebuilding your program before running it.

Tip: By default fpm enables compile time checks and run time checks that are absent in the plain compiler invocation. Those can help you catch and avoid common errors when developing in Fortran.

To read more on the capabilities of fpm check the help page with

```
fpm help
```

3.4 Introducing Variables

The string we have printed in our program was a character constant, thus we are not able to manipulate it. Variables are used to store and manipulate data in our programs, to *declare* variables we extend the *declaration section* of our program. We can use variables similar to the ones used in math in that they can have different values. Within Fortran, they cannot be used as *unknowns* in an equation; only in an assignment.

In Fortran, we need to declare the type of every variable explicitly, this means that a variable is given a specific and unchanging data type like `character`, `integer` or `real`. For example, we could write

Listing 3: numbers.f90

```
program numbers
implicit none
integer :: my_number
my_number = 42
```

(continues on next page)

(continued from previous page)

```
5 write(*, *) "My number is", my_number
6 end program numbers
```

Now the *declaration section* of our program in line 1-3, the second line declares that we want to declare all our variables explicitly. Implicit typing is a leftover from the earliest version of Fortran and should be avoided at all cost, therefore you will put the line `implicit none` in every declaration you write from now on. The third line declares the variable `my_number` as type `integer`.

Line 4 and 5 are the *executable section* of the program, first, we assign a value to `my_number`, then we are printing it to the screen.

Exercise 2

Make a new directory and create the file `numbers.f90` where you type in the above program. Then translate it with `gfortran` with

gfortran

```
gfortran numbers.f90 -o numbers_prog
./numbers_prog
 My number is           42
```

fpm

```
fpm run
 + build/gfortran_debug/app/myproject
 My number is           42
```

Despite being a bit oddly formatted the program correctly returned the number we have written in `numbers.f90`. `numbers_prog` will now always return the same number, to make the program useful, we want to have the program *read* in our number.

Use the `read(*, *)` statement to provide the number to the program, which works similar to the `write(*, *)` statement.

Solutions 2

We replace the assignment in line 4 with the `read(*, *) my_number` and then translate it to a program.

gfortran

```
gfortran numbers.f90 -o numbers_prog
./numbers_prog
31
 My number is           31
```

fpm

```
fpm run
 + build/gfortran_debug/app/myproject
31
 My number is           31
```

If you now execute `numbers_prog` the shell freezes. We are now exactly at the read statement and the `numbers_prog` is waiting for your action, so go ahead and type a number.

You might be tempted to type something like `four`:

gfortran

```
./numbers_prog
four
At line 4 of file numbers.f90 (unit = 5, file = 'stdin')
Fortran runtime error: Bad integer for item 1 in list input

Error termination. Backtrace:
#0  0x7efe31de5e1b in read_integer
   at /build/gcc/src/gcc/libgfortran/io/list_read.c:1099
#1  0x7efe31de8e29 in list_formatted_read_scalar
   at /build/gcc/src/gcc/libgfortran/io/list_read.c:2171
#2  0x7efe31def535 in wrap_scalar_transfer
   at /build/gcc/src/gcc/libgfortran/io/transfer.c:2369
#3  0x7efe31def535 in wrap_scalar_transfer
   at /build/gcc/src/gcc/libgfortran/io/transfer.c:2346
#4  0x56338a59f23b in ???
#5  0x56338a59f31a in ???
#6  0x7efe31867ee2 in ???
#7  0x56338a59f0fd in ???
#8  0xffffffffffffffff in ???
```

fpm

```
fpm run
 + build/gfortran_debug/app/numbers
 Enter an integer value
four
At line 5 of file app/main.f90 (unit = 5, file = 'stdin')
Fortran runtime error: Bad integer for item 1 in list input

Error termination. Backtrace:
#0  0x7fb170e6ffdb in read_integer
   at /build/gcc/src/gcc/libgfortran/io/list_read.c:1099
#1  0x7fb170e73229 in list_formatted_read_scalar
   at /build/gcc/src/gcc/libgfortran/io/list_read.c:2171
#2  0x561f44a562a0 in numbers
   at app/main.f90:5
#3  0x561f44a5637f in main
   at app/main.f90:7
 Command failed
ERROR STOP
```

So we got an error here, the program is printing a lot of cryptic information, but the most useful lines are near to our input of `four`. We have produced an error at the runtime of our Fortran program, therefore it is called a runtime error, more precise we have given a bad integer value for the first item in the list input at line 4 in `numbers.f90`. That was very verbose, Fortran expected an `integer`, but we passed a `character` to our read statement for `my_number`.

We could try to make our program more verbose by adding some information on what kind of input we expect to avoid this sort of errors. A possible solution would look like

Listing 4: numbers.f90

```
program numbers
implicit none
integer :: my_number
write(*, *) "Enter an integer value"
read(*, *) my_number
write(*, *) "My number is", my_number
end program numbers
```

While this will not prevent wrong input it will make it more unlikely by clearly communicating with the user of the program what we are expecting.

3.5 Performing Simple Computing Tasks

Next, you will make your program perform simple computational tasks – in this case, add two numbers.

So let's examine the following code:

Listing 5: add.f90

```
program add
  implicit none
  ! declare variables: integers
  integer :: a, b
  integer :: res

  ! get two values to be stored in a and b
  read(*, *) a, b

  res = a + b  ! perform the addition

  write(*, *) "The result is", res
end program add
```

Again we declare our program and give it a useful name describing the task at hand. The second statement is used to explicitly declare all variables and will be present in any program we write from now on.

The third line is a comment, any text after the exclamation mark is considered to be a comment in Fortran and is ignored by the compiler. Since it does not affect the final program we can use comments to remind ourselves why we choose to do something particular, the intent of the statement or to describe what the program is doing. In the beginning, you should comment your code as much as possible such that you will still understand them in a year from now. It is completely fine to produce more comment lines than lines of code to keep your program understandable. Also notice that Fortran does not care much about leading spaces (indentation) or empty lines, so we can use them to give our code a visual structure, which makes it more appealing and easier to read.

The comment states that we will declare our variables as integers, we have two integer declarations here, once for `a` and `b`, comma-separated on the same line and the next line an integer declaration for `res`. We could put `a, b, res` on

one line, but we might want to separate our input and result variables visually.

The next statement is in line 8 in the *executable section* of the code and reads values into `a` and `b`. Afterward, we perform the addition `a + b` and assign the result to `res`. Finally, we print the result and exit the program again.

Exercise 3

Create the file `add.f90` from the manual and modify it to make it do the following and check

1. Display a message to the user of your program (*via* write statements) about what kind of input is to be entered by them.
2. Read values from the console into the variables `a` and `b`, which are then *multiplied* and printed out. For error checking, print out the values `a` and `b` in the course of your program.
3. What happens if you provide input like `3.14`?
4. Perform a division instead of a multiplication. Attempt to obtain a fraction.

Solutions 3

As before we add a line like `write(*, *) "Enter two numbers to add"` before the read statement. We can do something similar like in numbers for both `a` and `b` to echo their values, the resulting shell history should look similar to this

gfortran

```
gfortran add -o add_prog
./add_prog
 Enter two numbers to add
11 31
 The value of a is           11
 The value of b is           31
 The result is           42
./add_prog
 Enter two numbers to add
-8
298
 The value of a is           -8
 The value of b is          298
 The result is         290
```

fpm

```
fpm run
 + build/gfortran_debug/app/add
 Enter two numbers to add
11 31
 The value of a is           11
 The value of b is           31
 The result is           42
fpm run
 + build/gfortran_debug/app/add
 Enter two numbers to add
```

(continues on next page)

(continued from previous page)

```
-8
298
 The value of a is          -8
 The value of b is         298
 The result is         290
```

The input seems to be quite forgiving and we can also add negative numbers. While this sounds obvious it is a common pitfall in other languages, but in Fortran all integers are signed and there is no unsigned version like in C.

To change the arithmetic operation in our code we have to know the operator used in Fortran to perform anything beyond addition. We can use + for addition, − for subtraction, * for multiplication, / for division and ** exponentiation.

We will skip the resulting output of the multiplication, except for one interesting case (you should have created a new file and a new program for the multiplication, since there is nothing worse than a program called `add_prog` performing multiplications).

gfortran

```
./multiply_prog
 Enter two numbers to multiply
1000000 1000000
 The value of a is     1000000
 The value of b is     1000000
 The result is  -727379968
```

fpm

```
fpm run
 + build/gfortran_debug/app/muliply
 Enter two numbers to multiply
1000000 1000000
 The value of a is     1000000
 The value of b is     1000000
 The result is  -727379968
```

which is kind of surprising. Take a piece of paper or perform the multiplication in your head, you will probably something pretty close to 1,000,000,000,000 instead of -727,379,968, but the computer is not as smart as you. We choose the default kind of the `integer` data types which uses 32 bits (4 bytes) to represent whole numbers in a range from -2,147,483,648 to +2,147,483,648 or -2^{31} to $+2^{31}$ using two's complement arithmetic, since the expected result is too large to be represented with only 32 bits (4 bytes), the result is truncated and the sign bit is left toggled which results in a large negative number (which is called an integer overflow, to understand why that makes sense lookup two's complement arithmetic).

Usually, you do not have to worry about exceeding the 32 bits (4 bytes) of precision since we have data types that can represent such large numbers in a better way.

Finally, think carefully about the result you expect when performing division with integers. Test your hypothesis with your division program. Note for yourself what to expect when trying to obtain fractions from integers.

3.6 Accuracy of Numbers

We already noted in the last exercise that we can create numbers not representable by integers like very large numbers or decimal numbers, therefore we have to resort to real numbers declared by the `real` data type.

Let us consider the following program using `real` variables

Listing 6: accuracy.f90

```
program accuracy
  implicit none

  real :: a, b, c

  a = 1.0
  b = 6.0
  c = a / b

  write(*, *) 'a is', a
  write(*, *) 'b is', b
  write(*, *) 'c is', c

end program accuracy
```

We translate `accuracy.f90` to an executable and run it to find that it is not that accurate

gfortran

```
gfortran accuracy.f90 -o accuracy_test
./accuracy.test
 a is   1.00000000
 b is   6.00000000
 c is  0.166666672
```

fpm

```
fpm run
 + build/gfortran_debug/app/accuracy
 a is   1.00000000
 b is   6.00000000
 c is  0.166666672
```

Similar to our integer arithmetic test, real (floating point) arithmetic has also limitation. The default representation uses 32 bits (4 bytes) to represent the floating-point number, which results in 6 significant digits, before the result starts to differ from what we would expect, by doing the calculation on a piece of paper or in our head.

Now consider the following program

Listing 7: kinds.f90

```
program kinds
  implicit none
  intrinsic :: selected_real_kind
  integer :: single, double
```

(continues on next page)

(continued from previous page)

```
  single = selected_real_kind(6)
  double = selected_real_kind(15)
  write(*, *) "For 6 significant digits", single, "bytes are required"
  write(*, *) "For 15 significant digits", double, "bytes are required"
end program kinds
```

The `intrinsic :: selected_real_kind` declares that we are using a built-in function from the Fortran compiler. This one returns the kind of `real` we need to represent a floating-point number with the specified significant digits.

Exercise 4

1. create a file `kinds.f90` and run it to determine the necessary kind of your floating-point variables.
2. use the syntax `real(kind) ::` to modify `accuracy.f90` to employ what we call double-precision floating-point numbers. Replace kind with the number you determined in `kinds.f90`.

Solutions 4

The output of the second write statement should be `8` on most machines.

But instead of hardcoding our wanted precision we combine `kinds.f90` and `accuracy.f90` in the final program version.

Listing 8: accuracy.f90

```
program accuracy
  implicit none

  intrinsic :: selected_real_kind
  ! kind parameter for real variables
  integer, parameter :: wp = selected_real_kind(15)
  real(wp) :: a, b, c

  ! also use the kind parameter here
  a = 1.0_wp
  b = 6.0_wp
  c = a / b

  write(*, *) 'a is', a
  write(*, *) 'b is', b
  write(*, *) 'c is', c

end program accuracy
```

If we now translate `accuracy.f90` we find that the output changed, we got more digits printed and also a more accurate, but still not perfect result

gfortran

```
gfortran accuracy.f90 -o accuracy_test
./accuracy.test
 a is   1.00000000000000
 b is   6.00000000000000
 c is  0.166666666666667
```

fpm

```
fpm run
 + build/gfortran_debug/app/accuracy
 a is   1.00000000000000
 b is   6.00000000000000
 c is  0.166666666666667
```

It is important to notice here that we cannot get the same result we would evaluate on a piece of paper since the precision is still limited by the representation of the number.

Finally, we want to highlight line 6 in `accuracy`, the `parameter` attached to the data type (here `integer`) is used to declare variables that are constant and unchangeable through the course of our program, more important, their value is known (by the compiler) while translating the program. This gives us the possibility to assign meaningful and easy to remember names to important values.

There is one more issue we have to discuss, look at the following program which does the same calculation as `accuracy.f90`, but with different kinds of literals.

Listing 9: literals.f90

```
1  program literals
2    implicit none
3
4    intrinsic :: selected_real_kind
5    ! kind parameter for real variables
6    integer, parameter :: wp = selected_real_kind(15)
7    real(wp) :: a, b, c
8
9    a = 1.0_wp / 6.0_wp
10   b =    1.0 / 6.0
11   c =      1 / 6
12
13   write(*, *) 'a is', a
14   write(*, *) 'b is', b
15   write(*, *) 'c is', c
16
17 end program literals
```

If we run the program now we find surprisingly that only `a` has the expected value, while all others are off. We can easily explain the result for `c`, the actual calculation is happening in integer arithmetic which yields 0 and is then *cast* into a real number `0.0`.

```
a is  0.16666666666666666
b is  0.16666667163372040
c is  0.00000000000000000
```

But what happens in case of `b`, we perform the calculation with `1.0/6.0`, but those are a real number from the default type represented in 32 bits (4 bytes) and then, as we store the result in `b`, *casted* into a real number represented in 64 bits (8 bytes).

Important: Always specify the kind parameters in floating-point literals!

Here we introduce the concept of *casting* one data type to another, whenever a variable is assigned a different data type, the compiler has to convert it first, which is called *casting*.

Possible Errors

You might ask what happens if we leave out the `parameter` attribute in line 6, let's try it out:

Listing 10: accuracy.f90

```
1  program accuracy
2    implicit none
3
4    intrinsic :: selected_real_kind
5    ! kind parameter for real variables
6    integer :: wp = selected_real_kind(15)
7    real(wp) :: a, b, c
8
9    ! also use the kind parameter here
10   a = 1.0_wp
11   b = 6.0_wp
12   c = a / b
13
14   write(*, *) 'a is', a
15   write(*, *) 'b is', b
16   write(*, *) 'c is', c
17
18 end program accuracy
```

`gfortran` complains about errors in the source code, pointing you at line 7, with several errors following, as usual, the first error is the interesting one:

```
accuracy.f90:7:7:

    7 |   real(wp) :: a, b, c
      |       1
Error: Parameter ‘wp’ at (1) has not been declared or is a variable, which does not␣
↪reduce to a constant expression
accuracy.f90:10:12:

   10 |   a = 1.0_wp
      |            1
Error: Missing kind-parameter at (1)
...
```

There we find the solution to our problem in plain text, the parameter `wp`, which is not a parameter in our program, is either not declared (it is) or it is a variable. `gfortran` expects a `parameter` here, but we passed a variable. All other errors result from either the missing `parameter` attribute or that `gfortran` could not translate line 7 due to the first error.

Therefore, always check for the first error that occurs.

You could also ask how important line 4 with `intrinsic ::` is for our program. You *could* leave it out completely (try it!), but we will always declare all the intrinsic functions we are using here such that you know they are, indeed, intrinsic functions.

3.7 Logical Constructs

Our programs so far had one line of execution. Logic is very fundamental for controlling the execution flow of a program, usually, you evaluate logical expression directly in the corresponding `if` construct to decide which branch to take or save it to a `logical` variable.

Now we want to solve for the roots of the quadratic equation $x^2 + px + q = 0$, we know that we can easily solve it by

$$x = -\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q}$$

but we have to consider different cases for the number of roots we obtain from this equation (or we use `complex` numbers). We have to be able to evaluate conditions and create branches dependent on the conditions for our code to evaluate. Check out the following program to find roots

Listing 11: roots.f90

```
1  program roots
2    implicit none
3    ! sqrt is the square root and abs is the absolute value
4    intrinsic :: selected_real_kind, sqrt, abs
5    integer, parameter :: wp = selected_real_kind(15)
6    real(wp) :: p, q
7    real(wp) :: d
8
9    ! request user input
10   write(*, *) "Solving x² + p·x + q = 0, please enter p and q"
11   read(*, *) p, q
12   d = 0.25_wp * p**2 - q
13   ! discriminant is positive, we have two real roots
14   if (d > 0.0_wp) then
15     write(*, *) "x1 =", -0.5_wp * p + sqrt(d)
16     write(*, *) "x2 =", -0.5_wp * p - sqrt(d)
17   ! discriminant is negative, we have two complex roots
18   else if (d < 0.0_wp) then
19     write(*, *) "x1 =", -0.5_wp * p, "+ i ·", sqrt(abs(d))
20     write(*, *) "x2 =", -0.5_wp * p, "- i ·", sqrt(abs(d))
21   else  ! discriminant is zero, we have only one root
22     write(*, *) "x1 = x2 =", -0.5_wp * p
23   endif
24 end program roots
```

Exercise 5

1. check the conditions for simple cases, start by setting up quadratic equations with known roots and compare your results against the program.
2. Extend the program to solve the equation: $ax^2 + bx + c = 0$.

Fortran offers also a `logical` data type, the literal logical values are `.true.` and `.false.` (notice the dots enclosing true and false values).

Note: Programmer coming from C or C++ may find it unintuitive that Fortran stores a `logical` in 32 bits (4 bytes) similar to an `integer` and that true and false are not 1 and 0.

You already saw two operators, greater than > and less than <, for a complete list of all operators see the following table. They always come in two versions but have the same meaning.

Operation	symbol	.op.	example (var is `integer`)
equals	`==`	`.eq.`	`var == 1`, `var.eq.1`
not equals	`/=`	`.ne.`	`var /= 5`, `var.ne.5`
greater than	`>`	`.gt.`	`var > 0`, `var.gt.0`
greater equal	`>=`	`.ge.`	`var >= 10`, `var.ge.10`
less than	`<`	`.lt.`	`var < 3`, `var.lt.3`
less equal	`<=`	`.le.`	`var <= 8`, `var.le.8`

Note: You cannot compare two logical expressions with each other using the operators above, but this is usually not necessary. If you find yourself comparing two logical expressions with each other, rethink the logic in your program first, most of the time is just some superfluous construct. If you are sure that it is really necessary, use `.eqv.` and `.neqv.` for the task.

To negate a logical expression we prepend `.not.` to the expression and to test multiple expressions we can use `.or.` and `.and.` which have the same meaning as their equivalent operators in logic.

3.8 Repeating Tasks

Consider this simple program for summing up its input

Listing 12: loop.f90

```
program loop
  implicit none
  integer :: i
  integer :: number
  ! initialize
  number = 0
  do
    read(*, *) i
    if (i <= 0) then
      exit
    else
      number = number + i
    end if
  end do
  write(*, *) "Sum of all input", number
end program loop
```

Here we introduce a new construct called `do`-loop. The content enclosed in the `do/end do` block will be repeated until the `exit` statement is reached. Here we continue summing up as long as we are getting positive integer values (coded in its negated form as exit if the user input is lesser than or equal to zero).

Exercise 6

1. there is no reason to limit us to positive values, modify the program such that it also takes negative values and breaks at zero.

Solutions 6

You might have tried to exchange the condition for `i = 0`, but since the equal sign is reserved for the assignment `gfortran` will throw an error like this one

```
loop.f90:9:10:

    9 |     if (i = 0) then
      |          1
Error: Syntax error in IF-expression at (1)
loop.f90:11:8:

   11 |     else
      |        1
Error: Unexpected ELSE statement at (1)
loop.f90:13:7:

   13 |     end if
      |       1
Error: Expecting END DO statement at (1)
```

It is a common pitfall in other programming languages to confuse the assignment operator with the equal operator, which is fundamentally different. While it is syntactically correct in C to use an assignment in a conditional statement, the resulting code is often in error. In Fortran, the assignment does not return a value (unlike in C), therefore the code is logically and syntactically wrong. We are better off using the correct == or `.eq.` operator here.

Now that we know the basic loop construction from Fortran, we will introduce two special versions, which you will encounter more frequently in the future. First, the loop we set up in the example before, did not terminate without us specifying a condition. We can add the condition directly to the loop using the `do while(<condition>)` construct instead.

Listing 13: while.f90

```
1  program while_loop
2    implicit none
3    integer :: i
4    integer :: number
5    ! initialize
6    number = 0
7    read(*, *) i
8    do while(i > 0)
9      number = number + i
10     read(*, *) i
11   end do
12   write(*, *) "Sum of all input", number
13 end program while_loop
```

This shifts the condition to the beginning of the loop, so we have to restructure our execution sequence a bit to match the new logical flow of the program. Here, we save the `if` and `exit`, but have to provide the `read` statement twice.

Imagine we do not want to sum arbitrary numbers but make a cumulative sum over a range of numbers. In this case, we would use another version of the `do` loop as given here:

Listing 14: sum.f90

```
program cumulative_sum
  implicit none
  integer :: i, n
  integer :: number
  ! initialize
  number = 0
  read(*, *) n
  do i = 1, n
    number = number + i
  end do
  write(*, *) "Sum is", number
end program cumulative_sum
```

You might notice we had to introduce another variable `n` for the upper bound of the range, because we made `i` now our loop counter, which is automatically incremented for each repetition of the loop, also you don't have to care about the termination condition, as it is generated automatically by the specified range.

Important: Never write to the loop counter variable inside its loop.

Exercise 7

1. Check the results by comparing them to your previous programs for summing integer.
2. What happens if you provide a negative upper bound?
3. The lower bound is fixed to one, make it adjustable by user input. Compare the results again with your previous programs.

Now, if we want to sum only even numbers in our cumulative sum, we could try to add a condition in our loop:

Listing 15: sum.f90

```
program cumulative_sum
  implicit none
  intrinsic :: modulo
  integer :: i, n
  integer :: number
  ! initialize
  number = 0
  read(*, *) n
  do i = 1, n
    if (modulo(i, 2) == 1) cycle
    number = number + i
  end do
  write(*, *) "Sum is", number
end program cumulative_sum
```

The `cycle` instruction breaks out of the *current* iteration, but not out of the complete loop like `exit`. Here we use it together with the intrinsic `modulo` function to determine the remainder of our loop counter variable in every step and `cycle` in case we find a remainder of one, meaning an odd number.

Note: Programmers coming from almost any language might find it confusing to start counting at 1. It was adopted as

default choice because of it is the natural choice (for non-programmers at least), but Fortran does not limit you there, there are scenarios where counting from -l to +l is the better choice, *i.e.* for orbital angular momenta.

You can also start counting from 0, but please keep in mind that most people also find it unintuitive to start counting from 0.

3.9 Fields and Arrays of Data

So far we dealt with scalar data, for more complex programs we will need fields of data, like a set of cartesian coordinates or the overlap matrix. Fortran provides first-class multidimensional array support.

Listing 16: array.f90

```
program array
  implicit none
  intrinsic :: sum, product, maxval, minval
  integer :: vec(3)
  ! get all elements from standard input
  read(*, *) vec
  ! produce some results
  write(*, *) "Sum of all elements", sum(vec)
  write(*, *) "Product of all elemnts", product(vec)
  write(*, *) "Maximal/minimal value", maxval(vec), minval(vec)
end program array
```

We denote arrays by adding the dimension in parenthesis behind the variable, in this we choose a range from 1 to 3, resulting in 3 elements.

Exercise 8

1. Expand the above program to work on a 3 by 3 matrix
2. The `sum` and `product` can also work on only one of the two dimensions, try to use them only for the rows or columns of the matrix

Usually, we do not know the size of the array in advance, to deal with this issue we have to make the array `allocatable` and explicitly request the memory at runtime

Listing 17: array.f90

```
program array
  implicit none
  intrinsic :: sum, product, maxval, minval
  integer :: ndim
  integer, allocatable :: vec(:)
  ! read the dimension of the vector first
  read(*, *) ndim
  ! request the necessary memory
  allocate(vec(ndim))
  ! now read the ndim elements of the vector
  read(*, *) vec
  write(*, *) "Sum of all elements", sum(vec)
  write(*, *) "Product of all elemnts", product(vec)
```

(continues on next page)

(continued from previous page)

```
  write(*, *) "Maximal/minimal value", maxval(vec), minval(vec)
end program array
```

Exercise 9

1. What happens if you provide zero as dimension? Does the behavior match your expectations?
2. Try to allocate your array with a lower bound unequal to 1 by using something like `allocate(vec(lower:upper))`

Up to now we only performed operations on an entire (multidimensional) array, to access a specific element we use its index

Listing 18: array.f90

```
program array_sum
  implicit none
  intrinsic :: size
  integer :: ndim, i, vec_sum
  integer, allocatable :: vec(:)
  ! read the dimension of the vector first
  read(*, *) ndim
  ! request the necessary memory
  allocate(vec(ndim))
  ! now read the ndim elements of the vector
  read(*, *) vec
  vec_sum = 0
  do i = 1, size(vec)
    vec_sum = vec_sum + vec(i)
  end do
  write(*, *) "Sum of all elements", vec_sum
end program array_sum
```

The above program provides a similar functionality to the intrinsic `sum` function.

Exercise 10

1. Reproduce the functionality of `product`, `maxval` and `minval`, compare to the intrinsic functions.
2. What happens when you read or write outside the bounds of the array?

Solutions 10

Let's try to read one element past the size of the array and add this elements to the sum (`do i = 1, size(vec)+1`):

```
./array_sum
10
1 1 1 1 1 1 1 1 1 1
 Sum of all elements          331
```

Tip: Using fpm would have caught this error by default. You would see the run time error message shown below already at this stage.

Since we provided ten elements which are all one, we expect 10 as result, but get a different number. So what is element 11 of our array of size 10? We have gone out-of-bounds for the array, whatever is beyond the bounds of our array, we are not supposed to know or care.

Checking out-of-bounds errors is not enabled by default, we enable it by recompiling our program and now found a helpful message

```
gfortran array_sum.f90 -fcheck=bounds -o array_sum
./array_sum
10
1 1 1 1 1 1 1 1 1 1
At line 14 of file array_sum.f90
Fortran runtime error: Index '11' of dimension 1 of array 'vec' above upper bound of␣
↪10

Error termination. Backtrace:
#0  0x55fc72c90530 in ???
#1  0x55fc72c9062f in ???
#2  0x7ff5c6e57152 in ???
#3  0x55fc72c9014d in ???
#4  0xffffffffffffffff in ???
```

By using the intrinsic functions like `size` it is garanteered that you will stay inside the array bounds.

3.10 Functions and Subroutines

In the last exercise you wrote implementations for `sum`, `product`, `maxval` and `minval`, but since they are inlined in the program we cannot really reuse them. For this purpose we introduce functions and subroutines:

Listing 19: sum_func.f90

```
1  program array_sum
2    implicit none
3    interface
4    function sum_func(vector) result(vector_sum)
5      integer, intent(in) :: vector(:)
6      integer :: vector_sum
7    end function sum_func
8    end interface
9    integer :: ndim
10   integer, allocatable :: vec(:)
11   ! read the dimension of the vector first
12   read(*, *) ndim
13   ! request the necessary memory
14   allocate(vec(ndim))
15   ! now read the ndim elements of the vector
16   read(*, *) vec
17   write(*, *) "Sum of all elements", sum_func(vec)
18 end program array_sum
19
20 function sum_func(vector) result(vector_sum)
21   implicit none
22   intrinsic :: size
23   integer, intent(in) :: vector(:)
24   integer :: vector_sum, i
```

(continues on next page)

(continued from previous page)

```
  vector_sum = 0
  do i = 1, size(vector)
    vector_sum = vector_sum + vector(i)
  end do
end function sum_func
```

In the above program we have separated the implementation of the summation to an external function called `sum_func` we provided an `interface` to allow our main program to access the new function. We have to introduce a dummy argument (called `vector`) and have to specify its `intent`, here it is `in` because we do not modify it (other intents are `out` and `inout`). When invoking the function we pass `vec` as `vector` to our summation function which returns the sum for us.

Note that we now have *two* declaration sections in our file, one for our program and one for the implementation of our summation function. You might also notice that writing interfaces can rapidly become cumbersome, so there is a better mechanism we want to use here:

Listing 20: sum_func.f90

```
module array_funcs
  implicit none
contains
  function sum_func(vector) result(vector_sum)
    intrinsic :: size
    integer, intent(in) :: vector(:)
    integer :: vector_sum, i
    vector_sum = 0
    do i = 1, size(vector)
      vector_sum = vector_sum + vector(i)
    end do
  end function sum_func
end module array_funcs

program array_sum
  use array_funcs
  implicit none
  integer :: ndim
  integer, allocatable :: vec(:)
  ! read the dimension of the vector first
  read(*, *) ndim
  ! request the necessary memory
  allocate(vec(ndim))
  ! now read the ndim elements of the vector
  read(*, *) vec
  write(*, *) "Sum of all elements", sum_func(vec)
end program array_sum
```

We wrap implementation of the summation now into a `module` which ensures the correct `interface` is generated automatically and made available by adding the `use` statement to the main program.

Exercise 11

1. Implement your `product`, `maxval` and `minval` function in the `array_funcs` module. Compare your results with your previous programs.
2. Write functions to perform the scalar product between two vectors and reuse it to write a function for matrix-vector multiplications. Compare to the intrinsic functions `dot_product` and `matmul`.

When writing functions like the above ones, we follow a specific scheme, all arguments are not modified (`intent(in)`) and we return a single variable. There are cases were we do not want to return a single value (in such a case we would return nothing) or do more complex operations in it. Functions of this kind are called subroutines:

Listing 21: sum_sub.f90

```
module array_funcs
  implicit none
contains
  subroutine sum_sub(vector, vector_sum)
    intrinsic :: size
    integer, intent(in) :: vector(:)
    integer, intent(out) :: vector_sum
    integer :: i
    vector_sum = 0
    do i = 1, size(vector)
      vector_sum = vector_sum + vector(i)
    end do
  end subroutine sum_sub
end module array_funcs

program array_sum
  use array_funcs
  implicit none
  integer :: ndim
  integer, allocatable :: vec(:)
  integer :: vec_sum
  ! read the dimension of the vector first
  read(*, *) ndim
  ! request the necessary memory
  allocate(vec(ndim))
  ! now read the ndim elements of the vector
  read(*, *) vec
  call sum_sub(vec, vec_sum)
  write(*, *) "Sum of all elements", vec_sum
end program array_sum
```

On the first glance, subroutines have several disadvantages compared to functions, we need to explicitly declare a temporary variable, also we cannot use them inline with another instruction. This holds true for short and simple operations, here functions should be prefered over subroutines. On the other hand, if the code in the function gets more complicated and the number of dummy arguments grows, we should prefer subroutines, because they are more visible in the code, especially due to the explicit `call` keyword.

3.10.1 Multidimensional Arrays

We will be dealing in the following chapter with multidimensional arrays, usually in form of rank two arrays (matrices). Matrices are stored continuously in memory following a column major ordering, this means the innermost index of any higher rank array will represent continuous memory.

Reading a rank two array should be done by

Listing 22: array_rank.f90

```
program array_rank
  implicit none

```

(continues on next page)

(continued from previous page)

```
  intrinsic :: selected_real_kind
  ! kind parameter for real variables
  integer, parameter :: wp = selected_real_kind(15)
  integer :: i
  integer :: d1, d2
  real(wp), allocatable :: arr2(:, :)

  read(*, *) d1, d2

  allocate(arr2(d1, d2))

  do i = 1, size(arr2, 2)
    read(*, *) arr2(:, i)
  end do

end program array_rank
```

This ensures that the complete array is filled in unit strides, *i.e.* visiting all elements of the array in exactly the order they are stored in memory. Making sure the memory access is in unit strides usually allows compilers to produce more efficient programs.

Tip: Array slices should preferably used on continuous memory, practically this means a colon should only be present in the innermost dimensions of an array.

```
array2 = array3(:, :, i)
```

Storing data, like cartesian coordinates, should follow the same considerations. It is always preferable to have the three cartesian components of the position close to each other in memory.

3.11 Character Constants and Variables

The `character` data type consists of strings of alphanumeric characters. You have already used *character constants*, which are strings of characters enclosed in single (`'`) or double (`"`) quotes, like in your very first Fortran program. The minimum number of characters in a string is 0.

```
write(*, *) "This is a valid character constant!"
write(*, *) '3.1415936' ! not a number
write(*, *) "{']!=" ! any character can be included, even !
```

A *character variable* is a variable containing a value of the `character` data type:

```
character :: single
character, dimension(20) :: many
character(len=20) :: fname
character(len=:), allocatable :: input
```

- the first variable `single` can contain only a single character
- like before one could try to create an array-like `many` containing many characters, but it turns out that this is not a viable approach to deal with characters
- Fortran offers a better way to make use of the character data type by adding a length to the variable, as is done for `fname`.

- a more flexible way of declaring your character variables is to use a so called *deferred size* character, like `input`.

To write certain data neatly to the screen *format specifiers* can be used, which are character constants or variables. Consider your addition program from the beginning of this course:

```
program add
  implicit none
  ! declare variables: integers
  integer :: a, b
  integer :: res

  ! assign values
  a = 2
  b = 4

  ! perform calculation
  res = a + b

  write(*,'(a,1x,i0)') 'Program has finished, result is', res
end program add
```

Instead of using the asterisk, we now define the *format* for the printout. The format must always be enclosed in parenthesis and the individual format specifier must be separated by commas. Therefore the first format specifier is `a`, which tells Fortran to print a character. The second specifier is one space (`1x`), while the last (`i0`) specifies an integer datatype with automatic width.

The result will look similar to your first run, but now there will only be one space between the characters and the final result. Of course, you can do more: `/` is a line break, `f12.8` is a 12 characters wide floating-point number printout with 8 decimal places and `es12.4` switches to scientific notation with only 4 decimal places.

3.12 Interacting with Files

Up to now you only interacted with your Fortran program by standard input and standard output. For a more complex program a complicated input file might be necessary or the output should be saved for later analysis in a file on disk. To perform this task you need to open and close your files.

```
program files
  implicit none
  integer :: io
  integer :: ndim
  real :: var1, var2
  open(file='name.inp', newunit=io)
  read(io,*) ndim, var1, var2
  close(io)
  ! do some computation
  open(file='name.out', newunit=io)
  write(io,'(i0)') ndim
  write(io,'(2f14.8)') var1, var2
  close(io)
end program files
```

You see that you can interact with your files like with the standard input or output, but instead of the asterisk, you need to give each file a number. Fortunately, you do not have to keep track of the numbers used, as Fortran will do this automatically for you. Of course, you can check the value of `io` after opening a file and will find that it is just a (negative) number used to identify the file opened.

3.13 Application

Finally we have some more elaborate exercises to test what you already learned, it is not mandatory to solve the exercises here, but it will not harm as well.

Exercise 12

Calculate an approximation to π using Leibniz' formula:

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

$$\Rightarrow \quad \pi \approx 4 \sum_{k=0}^{N} \frac{(-1)^k}{2k+1}$$

The number of summands *N* shall be read from command line input. Do a sensible convergence check every few summands, *i.e.* if the summand becomes smaller than a threshold, exit the loop.

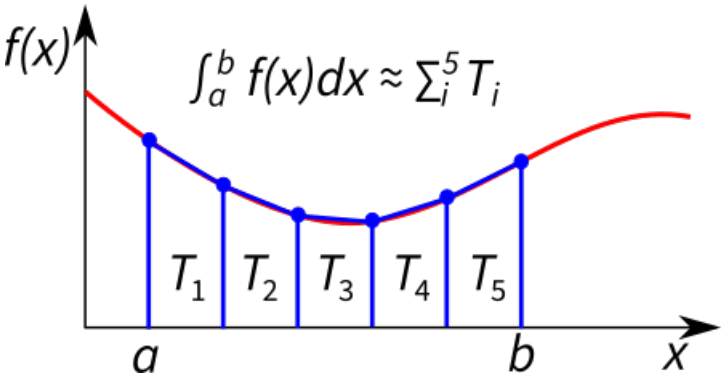
Note that the values in the Leibniz formula alternate in sign, rewrite your program to always add two of the summands (one with positive and one with negative sign) at once. Compare the results, why do they differ?

To adjust the step-length in the loop use

```
do i = 1, nmax, 2
  ...
end do
```

Exercise 13

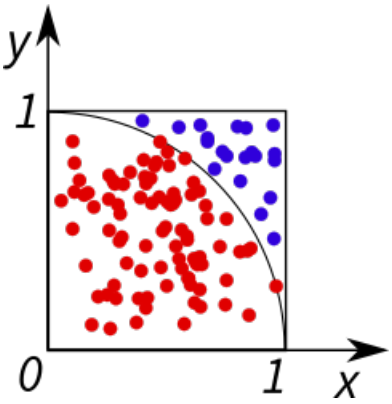
Approximately calculate the area in a unit circle to obtain π using trapezoids. The simplest way to do this is to choose a quarter circle and multiply its area by four.



Recall that the area of a trapezoid is the average of its sides times its height. The number of trapezoids shall be read from command line input.

Exercise 14

Approximately calculate π using the statistical method.



Generate pairs of random numbers *(x,y)* in the interval [0,1]. If they correspond to a point within the quarter circle, count them as "in". For large numbers of pairs, the ratio of "in" to the total number of pairs should correspond to the ratio of the area of the quarter circle to the area of the square (as described by the mentioned interval).

To generate (pseudo-)random numbers in Fortran use

```
real(kind) :: x(dim)
call random_number(x)
```

3.14 Derived Types

When writing your SCF program, you will notice that there are variables that can be grouped or belong to a certain category. To enhance the readability and structure of your code it can be convenient to gather these variables in a so called derived type. A derived type is another data type (like integer or character for example) that can contain built-in types as well as other derived types.

Here is an example of an arbitrary derived type:

```
type :: arbitrary
  integer     :: num
  real        :: pi
  character   :: string
  logical     :: boolean
end type arbitrary
```

The derived type contains the four variables `num`, `pi`, `string` and `boolean`. The syntax to create a variable of type `arbitrary` and access its members is:

```
! create variable "arb" of type "arbitrary"
type(arbitrary) :: arb
! access the members of the derived type with %
arb%num = 1
arb%pi = 3.0
arb%string = 'ok'
arb%boolean = .true.
```

Another advantage of using derived types is that you only need to pass one variable to a procedure instead of all the variables included in the derived type.

Exercise 15

The program below includes a derived type called `geometry`. So far, it contains the number of atoms and the atom positions. In the rest of the code, two atoms and their positions are set. Then the geometry information is printed by calling the subroutine `geometry_info`.

1. In order to clearly specify a chemical structure, it is necessary to assign an ordinal number to each atom. Add a one-dimensional allocatable integer variable to the derived type that will contain the ordinal number for each atom. Allocate memory for your new variable and set the initial value to zero using the source expression.
2. Now add a third atom to the geometry and assign atom types and positions to create a sensible carbon dioxide molecule.
3. Add the ordinal number to the printout in the `geometry_info` subroutine.

Listing 23: derived_types.f90

```
program derived_types
  implicit none
  intrinsic              :: selected_real_kind
  integer, parameter     :: wp = selected_real_kind(15)

! derived type cleary specifying a chemical structure
type :: geometry
  !> number of atoms
  integer                :: n
  !> xyz coordinates of the atoms in angstrom
  real(wp), allocatable :: xyz(:,:)
end type

  type(geometry)         :: geo


! define number of atoms
geo%n = 2

! allocate memory for atoms  and set initial values to zero
allocate(geo%xyz(3, geo%n), source=0.0_wp)

! define atom types and positions
! first atom is oxygen
geo%xyz(1,1) = -1.16_wp
! second atom is carbon

! third atom is oxygen


call geometry_info(geo)


contains

subroutine geometry_info(geo)
  type(geometry)     :: geo
  integer            :: i
  write(*,'(a, i0, a)') 'The input geometry has ', geo%n, ' atoms.'
  write(*,'(a)') 'Writing atom positions:'
  do i=1, geo%n
    write(*,'(3f10.4)') geo%xyz(:,i)
  enddo
```

(continues on next page)

(continued from previous page)

```
end subroutine geometry_info


end program derived_types
```

Solutions 15

Listing 24: derived_types.f90

```
program derived_types
  implicit none
  intrinsic              :: selected_real_kind
  integer, parameter     :: wp = selected_real_kind(15)

! derived type cleary specifying a chemical structure
type :: geometry
  !> number of atoms
  integer                :: n
  !> xyz coordinates of the atoms in angstrom
  real(wp), allocatable :: xyz(:,:)
  !> atom type
  integer, allocatable  :: at(:)
end type

  type(geometry)         :: geo


! define number of atoms
geo%n = 3

! allocate memory for atoms  and set initial values to zero
allocate(geo%xyz(3, geo%n), source=0.0_wp)
allocate(geo%at(geo%n), source=0)

! define atom types and positions
! first atom is oxygen
geo%at(1) = 8
geo%xyz(1,1) = -1.16_wp
! second atom is carbon
geo%at(2) = 6
! third atom is oxygen
geo%at(3) = 8
! carbondioxide should be linear and symetric
! therefore the x-coordinate of the new oxygen should be 1.16 angstrom
geo%xyz(1,3) =  1.16_wp

call geometry_info(geo)


contains

subroutine geometry_info(geo)
  type(geometry)     :: geo
  integer            :: i
  write(*,'(a, i0, a)') 'The input geometry has ', geo%n, ' atoms.'
```

(continues on next page)

(continued from previous page)

```
47    write(*,'(a)') 'Writing atom positions followed by atom type:'
48    do i=1, geo%n
49      write(*,'(3f10.4,4x,i0)') geo%xyz(:,i), geo%at(i)
50    enddo
51  end subroutine geometry_info
52
53
54  end program derived_types
```

CHAPTER

FOUR

INTRODUCTION TO QUANTUM CHEMISTRY

The main objective of this course is to write a working restricted Hartree–Fock program.

Contents

Note: Start by extracting the `course-material.zip` which you downloaded earlier. You should find a `scf/` directory containing the starting code for your program (see `scf/app/main.f90`).

You can just continue to use fpm to work on your program now.

Alternatively, to make building your program easier with just the Fortran compiler and give you more time to focus on the actual programming we put some build automation there as well. To use it just invoke `make` and find everything else taken care for.

For more details on building software and how we automated this process you can find a guide at the fortran-lang learning page.

4.1 Restricted Hartree–Fock

The first task in this course is to code a working restricted Hartree–Fock program.

4.1.1 Getting Input

First, we have to gather which information is important to perform a Hartree–Fock calculation. We need to know about the molecular geometry. There are several ways to represent the geometry, the most simple is to use Cartesian coordinates as a vector of x, y and z coordinates. What maybe is not that obvious, we have to decide on a unit system for the coordinates, usually we would follow IUPAC recommendations and use SI units like meters, but if you think about a molecule on the length scale in meters it becomes quite inconvenient. To get into the right ballpark we could choose Ångström which has an obvious relation to the SI unit, but here we will use Bohr which is the length unit of the atomic units system. Atomic units have the advantage that a lot of constants drop out of the equations we want to code up and we can convert them easily back, in fact, we provide a conversion routine back to SI units for you.

These are the considerations we have to put in the geometry, now we have to identify the atoms/nuclei in our molecule. Chemical identity like the element is given by the nuclear charge and the number of electrons. It is quite popular to specify both with the element symbol which map to the atomic number (which corresponds to the nuclear charge and the number of electrons of the neutral atom) since it is the intuitive choice. We will not follow this approach since it would require you to work with character type variables, instead we will separate the nuclear charge information and the number of electrons. For each element, we will read the nuclear charge in multiples of the electron charge (which is the atomic unit of charge) and specify the number of electrons separately.

Having put some thoughts in the geometric representation of the system, we now have to tackle the electronic representation, that is, we need a basis set to expand our wavefunction. There are many possible choices, like atom centered basis functions (Slater-type, Gaussian-type, …) plain waves, wavelets, … This is one of the most important choices for every quantum chemistry code, usually, a single kind of functions is supported which is limiting the chemical systems that can be calculated with this. The most common distinction is made between codes that support periodic boundary conditions or not, while periodic boundary conditions are naturally included in plain wave and wavelet based programs, extra effort has to be put into codes using Gaussian-type basis functions to support this kind of calculation. Also, most wavefunction centered programs use atom centered orbitals since the resulting integrals are easier to solve. Here the exception from the rule is quite common in our field of research and usually offers a unique competitive edge.

For writing your Hartree-Fock program you do not have to bother with this choice, because we already made it for you by providing the implementation for integrals over Gaussian-type basis functions. We will limit you here to spherical basis functions only, so you can concentrate on coding the self-consistent field procedure and do not have to worry about mapping shells to orbitals to basis functions to primitives.

We will start with the input for the dihydrogen molecule in a minimal basis set. With the input format we provide, the geometrical structure of the system and the basis set are tied together.

Listing 1: h2.in

```
2 2 2
0.0  0.0 -0.7  1.0  1
1.20
0.0  0.0  0.7  1.0  1
1.20
```

This input contains the information necessary to code up your program, we start with the first line, which contains the number of atoms, the number electrons and the total number of basis functions as integer values. None of this information is necessary to read the input file, but is included for convenience, *e.g.* such that you can allocate memory before starting to read the rest of the file.

Starting from the second line we expect a tuple of the three Cartesian coordinates in Bohr, the nuclear charges in multiples of electron charge and the number of basis functions this particular atom. In the lines after position and identity, we find the exponents of our basis functions, the number of lines following corresponds to the number of basis functions for this particular atom.

make

Important: You can compile your program using

```
make
...
gfortran ... -o ./scf
```

The resulting binary will be placed in the current working directory. Run it with

```
./scf
Here could start a Hartree-Fock calculation
```

The starting code provides the possibility to pass the input file as command line argument to your program with

```
./scf molecules/h2.in
Here could start a Hartree-Fock calculation
```

The input file will always be opened to the `input` unit in `app/main.f90`.

fpm

Important: The starting code is already setup as fpm project. The package manifest should contain the following content

Listing 2: fpm.toml

```
name = "scf"

[[executable]]
name = "scf"
main = "prog.f90"
```

You can run your program with

```
fpm run
...
 + build/gfortran_debug/app/scf
Here could start a Hartree-Fock calculation
```

The starting code provides the possibility to pass the input file as command line argument to your program with

```
fpm run -- molecules/h2.in
...
 + build/gfortran_debug/app/scf "molecules/h2.in"
Here could start a Hartree-Fock calculation
```

The input file will always be opened to the `input` unit in `app/main.f90`.

Exercise 1

Before you start coding the input reader for this format, try to write with the specifications inputs for the following systems:

1. A helium atom in a double zeta basis set with exponents 2.5 and 1.0.
2. A helium-hydrogen cation with a bond distance of 2 Bohr in a minimal basis set on both atoms and an exponent of 1.25 for each atom.

Solutions 1

For a single atom the choice of the position is unimportant, so we can write something like

Listing 3: he.in

```
1 2 2
0.0  0.0  0.0  2.0  2
2.5
1.0
```

The helium-hydrogen cation looks similar to the dihydrogen, except for the changed nuclear charge.

Listing 4: heh+.in

```
2 2 2
0.0  0.0 -1.0  2.0  1
1.25
0.0  0.0  1.0  1.0  1
1.25
```

Exercise 2

1. Code an input reader that can read all the provided input files.
2. Make sure the input file is read correctly by printing all data read to the terminal.

4.1.2 Classical Contributions

First, we start by computing the classical nuclear repulsion energy, *i.e.* the Coulomb energy between the nuclei.

Exercise 3

1. Which data is needed in the computation?
2. Code up the nuclear repulsion energy in a separate `subroutine`. Write the resulting energy in a meaningful way to the terminal.
3. Evaluate the Coulomb-law for the dihydrogen molecule at 1.4 Bohr distance and compare it with your program. Can you run your `subroutine` multiple times and get the same result? If not recheck your code.
4. Check what happens if you calculate the nuclear repulsion energy for a single atom. Do you get the expected result?

Classical contributions to the total energy do not dependent on the density or wavefunction and can already be calculated before starting with the self-consistent field procedure. It is usually a good idea to evaluate these contributions first, because they are less expensive to compute and, for the purpose of this course, easier to implement.

4.1.3 Basis Set Setup

This Hartree-Fock program will use contracted Gaussian-type orbitals to expand the molecular orbitals in atomic orbitals. We will use an STO-6G basis set, *i.e.* we use the best representation of a Slater-type orbital by six primitive Gaussian-type orbitals.

This is the first time you will use an external library function, therefore we will clarify to you how to read and use an `interface`. In your program, you will call a provided `subroutine` to perform the expansion from the Slater orbital to six primitive Gaussian-type orbitals.

The final call in your program might look somewhat similar to this:

```
call expand_slater(ng, zeta, exponents, coefficients)
```

To understand why the `subroutine expand_slater` takes four arguments, we look up its `interface`:

```
interface
subroutine expand_slater(ng, zeta, exponents, coefficients)
import wp
integer,  intent(in)  :: ng   !< number of primitive Gaussian functions
real(wp), intent(in)  :: zeta !< exponent of slater function
real(wp), intent(out) :: exponents(:)    !< of primitive Gaussian functions
real(wp), intent(out) :: coefficients(:) !< of primitive Gaussian functions
end subroutine expand_slater
end interface
```

An `interface` provides the necessary information on how to invoke a `subroutine` or `function` without concerning you with the implementation details.

Note: for programmers coming from C or C++, it is similar to an `extern` declaration in a header file for a function.

Usually, you do not have to write an `interface` since they are conveniently created and handled for you by your compiler.

Exercise 4

1. Create a `parameter` storing the information about the number of primitive Gaussian functions you want to expand your Slater function in.
2. What is the optimal layout for saving the exponents and coefficients of the primitive Gaussian functions? Which quantities from the input do you need to determine the amount of memory to store them?
3. Allocate enough space to store all the primitive exponents and coefficients from the expansion for calculating the integrals later.
4. Loop over all basis functions, perform the expansion for each and save the resulting primitive Gaussians to the respective arrays.

4.1.4 One-Electron Integrals

Note that the basis set we have chosen is very simple, we only allow spherical basis function (*s*-functions), also the contraction depth of each function is the same. Usually one would choose more sophisticated basis sets for quantitative calculations, but the basic principle remains the same.

Integral calculations can quickly be very obscure depending on the way a basis set is stored and mainly handle implementation-specific details of the program to perform the integral evaluation in some clever way. We use a simple basis set here to teach you the basic principle of integral evaluation.

We start with the simple one-electron integrals, for Hartree-Fock we need two-center overlap integrals, two-center kinetic energy integrals and three-center nuclear attraction integrals. To make things easier we provide the implementation for all three integrals over contracted Gaussian orbitals, let's check out the `interface`:

```
interface
!> one electron integrals over spherical Gaussian functions
subroutine oneint(xyz, chrg, r_a, r_b, alp, bet, ca, cb, sab, tab, vab)
   import wp
   real(wp), intent(in)  :: xyz(:, :) !< position of all atoms in atomic units, dim:␣
↪[3, nat]
   real(wp), intent(in)  :: chrg(:) !< nuclear charges, dim: nat
   real(wp), intent(in)  :: r_a(:) !< aufpunkt of orbital a, dim: 3
   real(wp), intent(in)  :: r_b(:) !< aufpunkt of orbital b, dim: 3
   real(wp), intent(in)  :: alp(:) !< Gaussian exponents of the primitives at a
   real(wp), intent(in)  :: bet(:) !< Gaussian exponents of the primitives at b
   real(wp), intent(in)  :: ca(:) !< contraction coeffients of primitives at a
   real(wp), intent(in)  :: cb(:) !< contraction coeffients of primitives at b
   real(wp), intent(out) :: sab !< overlap integral <a|b>
   real(wp), intent(out) :: tab !< kinetic energy integral <a|T|b>
   real(wp), intent(out) :: vab !< nuclear attraction integrals <a|Σ z/r|b>
end subroutine oneint
end interface
```

The most important information is we need *two* centers for the calculation, meaning we have to implement it as a loop over all orbital pairs (= pairs of basis functions).

Exercise 5

1. Which matrices can you compute from the one-electron integrals?
2. Allocate space for the necessary matrices.
3. Loop over all pairs and calculate all the matrix elements.

Exercise 6

Symmetric matrices, like the overlap, can be stored in two ways, as full $N \times N$ matrix with `dimension(n,n)` or as packed matrix in a one-dimensional array with `dimension(n*(1+n)/2)`, like:

$$\begin{pmatrix} 1 & 2 & 4 & 7 & \cdots \\ & 3 & 5 & 8 & \\ & & 6 & 9 & \\ & & & 10 & \\ \vdots & & & & \ddots \end{pmatrix} \Leftrightarrow \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ \vdots \end{pmatrix} \qquad \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \cdots \\ & a_{22} & a_{23} & a_{24} & \\ & & a_{33} & a_{34} & \\ & & & a_{44} & \\ \vdots & & & & \ddots \end{pmatrix} \Leftrightarrow \begin{pmatrix} a_{11} \\ a_{12} \\ a_{22} \\ a_{13} \\ \vdots \end{pmatrix}$$

1. Check if the matrices you have calculated are symmetric.
2. Pack all symmetric matrices.

The eigenvalue solver provided can only work with symmetric packed matrices, therefore you should have an unpacked to packed conversion routine ready for the next exercise.

4.1.5 Symmetric Orthonormalizer

The eigenvalue problem we have to solve is a general one given by

$$\mathbf{FC} = \mathbf{SC}\boldsymbol{\varepsilon}$$

where **F** is the Fock matrix, **C** is the matrix of the eigenvectors, **S** is the overlap matrix and **ε** is a diagonal matrix of the eigenvalues. While this is in principle possible with a more elaborated solver routine, we want to solve the following problem instead:

$$\mathbf{F}'\mathbf{C}' = \mathbf{C}'\boldsymbol{\varepsilon}$$

For this reason, we have to find a transformation from **F** to **F'** and back from **C'** to **C**. We choose the Loewdin orthonormalization or symmetric orthonormalization by calculating $\mathbf{X} = \mathbf{S}^{-1/2}$, to invert (or perform any operation with) a matrix we have to diagonalize it first and perform the operation on the eigenvalues **s**.

$$\mathbf{X} = \mathbf{S}^{-1/2} = \mathbf{U}\mathbf{s}^{-1/2}\mathbf{U}^{\top} \qquad \text{where} \quad \mathbf{s} = \mathbf{U}^{\top}\mathbf{S}\mathbf{U}$$

Exercise 7

1. Use $\mathbf{F'} = \mathbf{X}^{\mathrm{T}}\,\mathbf{F}\mathbf{X}$ and $\mathbf{C} = \mathbf{X}\mathbf{C'}$ to show that the general eigenvalue problem and the transformed one are equivalent.
2. Write a `subroutine` to calculate the symmetric orthonormalizer **X** from the overlap matrix.
3. Make sure that $\mathbf{X}^{\mathrm{T}}\,\mathbf{S}\mathbf{X} = \mathbf{1}$ and that you are not overwriting your overlap matrix in the diagonalization.
4. Do you see why **X** is called symmetric orthonormalizer?

To solve the actual eigenvalue problem we will use the linear algebra package (LAPACK), namely the subroutine `dspev`, which stands for double precision, symmetric packed eigenvalue problem, for more details you can look up the documentation. Since LAPACK routines can be somewhat unintuitive to work with on the first encounter we provide a wrapper called `solve_spev`. As usual we checkout the interface in `src/linear_algebra.f90`:

```
interface
subroutine solve_spev(matrix, eigval, eigvec, stat)
  import wp
  !> symmetric packed matrix to diagonalize, dim: n*(n+1)/2
  !  matrix is overwritten by the LAPACK solver
  real(wp), intent(inout) :: matrix(:)
  !> contains eigenvalues on exit, dim: n
  real(wp), intent(out) :: eigval(:)
  !> contains eigenvectors on exit, dim: [n, n]
  real(wp), intent(out) :: eigvec(:, :)
  !> error status, if not provided the routine will stop the program
  integer, intent(out), optional :: stat
end subroutine solve_spev
end interface
```

Note: The linear algebra package (LAPACK) and the basic linear algebra subprograms (BLAS) are commonly used libraries in many quantum chemical programs, as they provide a computationally efficient and uniform interface to many linear algebra problems.

4.1.6 Initial Guess

There are now two possible ways to initialize your density matrix **P**.

1. Provide a guess for the orbitals
2. Provide a guess for the Hamiltonian

If you happen to have converged orbitals around, the first method would be the most suitable. Alternatively, you can provide a model Hamiltonian, usually a tight-binding or extended Hückel Hamiltonian is used here. The simplest possible model Hamiltonian we have readily available is the core Hamiltonian $\mathbf{H}_0 = \mathbf{T} + \mathbf{V}$.

The initial density matrix **P** is obtained from the orbital coefficients by

$$\mathbf{P} = \mathbf{C}\,\mathbf{n}_{\mathrm{occ}}\mathbf{C}^{\top}$$

where $\mathbf{n}_{\mathrm{occ}}$ is a diagonal matrix with the occupation numbers of the orbitals. For the restricted Hartree-Fock case $n_{*\mathrm{i}*,\mathrm{occ}}$ is two for occupied orbitals and zero for virtual orbitals.

Exercise 8

1. By using $\mathbf{F} = \mathbf{H}_0$ as an initial guess for the Fock matrix we effectively set the density matrix **P** to zero. What does this mean from a physical point of view?
2. Add the initial guess to your program.
3. Diagonalize the initial Fock matrix (use the symmetric orthonormalizer) to obtain a set of guess orbital coefficients **C**.
4. Calculate the initial density matrix **P** resulting from those orbital coefficients.

4.1.7 Hartree-Fock Energy

The Hartree-Fock energy is given by

$$E_{\mathrm{HF}} = \frac{1}{2}\mathrm{Tr}\{(\mathbf{H}_0 + \mathbf{F})\mathbf{P}\}$$

where Tr denotes the trace.

Exercise 9

You should have now an initial Fock matrix **F** and an initial density matrix **P**.

1. Write a `subroutine` to calculate the Hartree-Fock energy.
2. Calculate the initial Hartree-Fock energy.

4.1.8 Two-Electron Integrals

We have ignored the two-electron integrals for a while now, up to now they were not important, but we will now need to calculate them to perform a self-consistent field procedure. The four-center two-electron integrals are the most expensive quantity in every Hartree-Fock calculation and there exist many clever algorithms to avoid calculating them all together, we will again go the straight-forward way and calculate them the naive way.

Again we will check the `interface` of the `twoint` routine in `src/integrals.f90`.

Exercise 10

1. Allocate enough space to store the two-electron integrals.
2. Create a (nested) loop to calculate all two-electron integrals.

Exercise 11

The four-center integrals have an eight-fold symmetry relation we want to use when calculating such an expensive integral to speed up the calculation. Rewrite your loops to only calculate the unique integrals:

$$(\mu\nu|\kappa\lambda) = (\nu\mu|\kappa\lambda) = (\mu\nu|\lambda\kappa) = (\nu\mu|\lambda\kappa) = (\kappa\lambda|\mu\nu) = (\kappa\lambda|\nu\mu) = (\lambda\kappa|\mu\nu) = (\lambda\kappa|\nu\mu)$$

1. Write down all the indices for the four-center integrals and figure out unique relation between the indices (it is similar to packing matrices).
2. Implement this eight-fold-symmetry in your calculation.
3. You can also pack the two-electron integrals like you packed your matrices (you need to pack them three times, the bra and the ket separately, and then the packed bra and ket again). Note that this will make it later more difficult to access the values again, therefore it is optional.

4.1.9 Self Consistent Field Procedure

Now you have everything together to build the self-consistent field loop. Remember, the Fock matrix in terms of the two-electron integrals is given by

$$F_{\mu\nu} = H_{\mu\nu} + \sum_{\lambda}\sum_{\kappa}\left(P_{\lambda\kappa}\cdot(\mu\nu|\kappa\lambda) - \frac{1}{2}P_{\lambda\kappa}\cdot(\mu\lambda|\kappa\nu)\right)$$

Make sure your definition of the density matrix matches the definition of the Fockian.

Exercise 12

1. First, you need to construct a new Fock matrix from the density matrix.
2. Construct a loop that performs your self-consistent field calculation.
3. Copy or move (whatever you find more appropriate) the necessary steps inside the SCF loop.
4. Define a convergence criterion using a conditional statement (`if`) based on the energy change and/or the change in the density matrix and `exit` the SCF loop.
5. Compare your final HF energies with

Input	E(RHF) / E_h
H_2	–1.127785613
He	–2.860251227
Be	–14.568567143

6. Calculate the proton affinity of H_2. H_3^+ has a D_{3h} structure with R_{HH} = 1.7 Bohr

---

4.1.10 Dissociation Curves

To calculate a dissociation curve we could either implement this functionality in the program itself or use an external script to automatically write input files and read the program output to collect the necessary data. Since this approach is common for computional chemistry, we will use a scripting approach here as well.

Modify the provided `bash` script to calculate a dissociation curve:

Listing 5: h2-diss.bash

```
#!/usr/bin/env bash

# stop on errors
set -eu

# Ensure non-localized printout
expot LC_NUMERIC=en_US.UTF-8

# put the name of your program here ("./scf" or "fpm run --")
program=echo
# unique pattern to find the final energy
pattern='final SCF energy'
# output file for plotting
datafile=plot.dat
```

(continues on next page)

(continued from previous page)

```
# scan distances
start_distance=1.4
last_distance=5.0
step=0.1

read -r -d 'END' input <<-EOF
   2 2 2
   0.0  0.0   0.0  1.0  1
   1.20
   0.0  0.0  DIST  1.0  1
   1.20
   END
EOF

tmpinp=temporary.inp
tmpout=temporary.out

# cleanup
[ -f $datafile ] && rm -v $datafile

steps=$(seq $start_distance $step $last_distance | wc -l)
printf "Scanning from %.3f Bohr to %.3f Bohr in %d steps\n" \
   $start_distance $last_distance $steps

for distance in $(seq $start_distance $step $last_distance | sed s/,/./)
do
   # generate the input file
   echo "$input" | sed s/DIST/$distance/ > $tmpinp
   # perform the actual calculation on the input file
   2>&1 $program $tmpinp > $tmpout
   # get the energy from the program output
   energy=$(grep "$pattern" $tmpout | awk '{printf "%f",$(NF)}' | tail -1)
   # if there is no energy to be found, we complain
   if [ -z "$energy" ]
      then
         1>&2 printf "ERROR!\n"
         1>&2 printf "'%s' cannot be found in '%s' output\n" "$pattern" "$program"
         1>&2 printf "please inspect '%s' and '%s'\n" "$tmpinp" "$tmpout"
         exit 1
   fi
   # otherwise we write to the logfile
   printf "Current energy is %.8f Hartree for distance %.3f Bohr\n" \
      $energy $distance
   printf "%8.3f %12.8f\n" $distance $energy >> $datafile
done

# cleanup
[ -f $tmpinp ] && rm $tmpinp
[ -f $tmpout ] && rm $tmpout
```

Exercise 13

1. The above script only contains dummies and can be executed without performing a calculation. Perform such a dry-run to understand how the script is working.
2. Modify it to match your program and plot the resulting dissociation curve.

Note: In case the script exits an error message like

```
script.bash: Zeile 34: printf: 1.4: Ungültige Zahl.
script.bash: Zeile 34: printf: 5.0: Ungültige Zahl.
Scanning from 1,000 Bohr to 5,000 Bohr in 37 steps
```

your language settings might be set to something other then English (in this case German). To overwrite the localisation settings run the script with

```
LC_NUMERIC=en_US.UTF-8 bash script.bash
```

4.2 Properties

With a working RHF program we have access to a wavefunction and we can use it to calculate certain properties.

4.2.1 Partial Charges

The total number of electrons in a system is given by

$$N_{el} = \sum_{\mu}^{N}\sum_{\nu}^{N} P_{\mu\nu}S_{\nu\mu} = \sum_{\mu}^{M}(\mathbf{PS})_{\mu\mu}.$$

If your SCF wavefunction does not return its input number of electrons, the wavefunction is most certainly not properly normalized. In this case recheck your SCF and integral implementation.

Based on the above formula, Mulliken concluded that the number of electrons associated with a particular nucleus is equal to the number of electrons associated with its basis functions. Thus the partial Mulliken charge of atom A is defined as:

$$q_A = Z_A - \sum_{\mu\in A}(\mathbf{PS})_{\mu\mu}$$

Exercise 13

1. Code a subroutine to perform a Mulliken atomic population analysis
2. Determine the Mulliken atomic partial charges in LiH using the input file provided.

4.2.2 Charge Density

In terms of real contracted (Gaussian) basis functions, the electron density is given by

$$\rho(\mathbf{r}) = \sum_{\mu}\sum_{\nu} P_{\mu\nu}\psi_{\mu}(\mathbf{r})\psi_{\nu}(\mathbf{r}).$$

To calculate the product of the two basis functions the Gaussian product theorem can be used

$$\phi(\alpha, \mathbf{r}-\mathbf{R}_A)\cdot\phi(\beta, \mathbf{r}-\mathbf{R}_B) = K_{AB}\cdot\phi\left(\alpha+\beta, \mathbf{r}-\frac{\alpha\mathbf{R}_A+\beta\mathbf{R}_B}{\alpha+\beta}\right)$$

where K_{AB} is given by

$$K_{AB} = \left(\frac{2\alpha\beta}{(\alpha+\beta)\pi}\right)^{\frac{3}{4}} \exp[-\alpha\beta/(\alpha+\beta)\cdot R_{AB}^2]$$

Note that the norming constants of the Gaussians have been absorbed into the contraction coefficients already.

Exercise 14

1. Code a subroutine to calculate the electron density ϱ at a given point in cartesian space.
2. Calculate and plot the electron density along the axis connecting the bonds in H_2 and through the Be and He atoms. Plot them together.

4.3 Geometry Optimization

The next task is to expand your program to perform a simple geometry optimization.

4.3.1 Numerical Derivatives

The simplest way to perform geometry optimizations is by using the information from the energy derivative, since we do not want to code analytical derivatives of the Hartree–Fock energy expression, we will resort to numerical derivatives instead. This requires to evalulate several SCF energies in one program run, since you coded your SCF in a subroutine this should not be an issue. Nevertheless, try to run the subroutine several times to check if the code is correctly allocating and initializing its variables, they might show up now and you can fix them before adding much more code to your program.

Copy and modify your input files such that for each parameter (cartesian atomic coordinates and Slater exponents) θ_{i} it contains an integer indicating whether a numerical gradient with respect to θ_{i} is to be calculated. Create a new subroutine that will allow the variation of one parameter θ_{i} at a time as necessary for each element of the numerical gradient

$$\frac{\delta E}{\delta\theta_i} \approx \frac{E(\theta_1,\ldots,\theta_i+\Delta\theta,\ldots,\theta_n) - E(\theta_1,\ldots,\theta_i-\Delta\theta,\ldots,\theta_n)}{2\Delta\theta}.$$

Exercise 15

1. Code a numerical gradient that allows the calculation of the derivative of the HF energy with respect to atom positions and Slater exponents.
2. Save your gradient components to arrays analogous to the ones for the atom positions and Slater exponents. We shall call the conceptual combination of them the gradient vector **g**.
3. By definition, in which direction does a gradient point? How is this different from the force?

4.3.2 Steepest Decent

Similar to the SCF, parameter optimizations are performed iteratively and depend on a convergence criterion concerning energy and/or size of gradient.

Code a variant of the "steepest descent" optimization routine as given by:

$$\Theta^{k+1} = \Theta^k + \eta \mathbf{g}^k$$

k denotes the number of the optimization cycle, Θ is the parameter set for an iteration. Choose η to get smooth, fast convergence.

Exercise 16

1. Optimize the geometry of HeH^+ in a minimal basis set.
2. Optimize the Slater exponents of Be and H_2 (R_{HH} = 1.4 Bohr) in a full double-ζ basis set. Compare to energies for the est. HF basis set limit:

System	Energy/E_h
H_2	–1.134
Be	–14.573

4.4 Unrestricted Hartree-Fock

Copy and modify your RHF subroutine to create an UHF program. The input files will need to be modified such that they contain the number of α and the number of β electrons. By definition, the number of α electrons is bigger then the number of β electrons. You can check the correctness of your results against the Li atom (Slater exponents 3.5, 2.0, 0.7 and 0.3: E = –7.419629 E_h) and the H atom.

For UHF you use one set of **F**, **C** and **P** matrices for each spin to solve the two eigenvalue problems

$$\mathbf{F}^\alpha \mathbf{C}^\alpha = \mathbf{S}\mathbf{C}^\alpha \boldsymbol{\varepsilon}^\alpha \quad \text{and} \quad \mathbf{F}^\beta \mathbf{C}^\beta = \mathbf{S}\mathbf{C}^\beta \boldsymbol{\varepsilon}^\beta$$

concurrently. They are coupled only through the formation of the Fock matrix *via* the Coulomb interaction as demonstrated for $\mathbf{F}^\alpha$ below:

$$F^\alpha_{\mu\nu} = h_{\mu\nu} + \sum_\lambda \sum_\kappa \Big((P^\alpha_{\lambda\kappa} + P^\beta_{\lambda\kappa}) (\mu\nu|\kappa\lambda) - P^\alpha_{\lambda\kappa} (\mu\lambda|\kappa\nu) \Big)$$

Note that the calculation of **P** differs in the occupation number and for closed-shell test cases provided your UHF program must give RHF results. For this reason, you have to break the spatial symmetry of your system to obtain the UHF solution, if it is available. The easiest way to do this is through the initial guess. If you use a symmetric guess (like **P** = 0), you will only find the RHF solution.

The UHF Energy is given by:

$$E = \frac{1}{2} \sum_{\mu\nu} \Big(P^\alpha_{\mu\nu} (h_{\nu\mu} + F^\alpha_{\nu\mu}) + P^\beta_{\mu\nu} (h_{\nu\mu} + F^\beta_{\nu\mu}) \Big)$$

Exercise 17

1. Calculate and plot the dissociation/potential curves for $^3H_2$, $^1H_2$ and $^1Li_2$.

2. The exchange reaction $H_2 + H \rightarrow H + H_2$ has a linear symmetric transition state. Find it and its relative energy.
3. Calculate the first ionization potential of Be. The experimental value is 9.3 eV.
4. Are He_2^+ or $^3He_2$ bonded?

4.4.1 Spin Contamination

Recall that spin contamination can occur in UHF calculations, *i.e.* deviations of the expectation value of the square of the total spin angular momentum operator S^2 from the ideal value:

$$\langle \hat{S}^2 \rangle_{\text{UHF}} = \langle \hat{S}^2 \rangle_{\text{exact}} + \Delta$$
$$\langle \hat{S}^2 \rangle_{\text{exact}} = \frac{N_{el}^{\alpha} - N_{el}^{\beta}}{2} \cdot \left(\frac{N_{el}^{\alpha} - N_{el}^{\beta}}{2} + 1 \right)$$

Calculate the spin contamination according to

$$\Delta = N_{el}^{\beta} - \sum_{i}^{N_{el}^{\alpha}} \sum_{j}^{N_{el}^{\beta}} \left| \langle \chi_i^{\alpha} | \chi_j^{\beta} \rangle \right|^2$$

where *i*, *j* indicate α and β-MOs, respectively.

Exercise 18

1. For the system:

```
h 0.0  0.0  -1.0
h 0.0  0.0   0.0
h 0.0  0.2   1.0
```

   and a Slater exponent of 1.24, you should find a spin contamination of 0.004682 and a UHF energy of -1.265643.
2. Plot the spin contamination along both RHF and UHF dissociation curves of H_2.
3. Why is the spin contamination of RHF always 0?

4.5 Møller–Plesset Perturbation Theory

Using your RHF program, code a subroutine to calculate the RMP2 energy after your RHF procedure has converged. The RMP2 correction to the HF energy (assuming real MOs *a, b, r, s*) can be written as

$$W_2 = \sum_{a,b=1}^{N_{el}/2} \sum_{\substack{r,s= \\ N_{el}/2+1}}^{M} \frac{(ar|bs)\left[2\,(ar|bs) - (as|br)\right]}{\varepsilon_a + \varepsilon_b - \varepsilon_r - \varepsilon_s}.$$

As for all correlation methods, you will need to have two-electron integrals over MOs (denoted by Roman letters) instead of over AOs (Greek letters). The process to obtain them is called AO-MO-Transformation. There are different algorithms possible, among them are one to scale with M^8 and one with M^5 that takes approximately twice as much memory. For now, code up the straigthforward algorithm to transform the two-electron integrals. This is the one that scales with M^8.

Input	E(RHF) / E_h	E_c(MP2) / E_h
H_2	–1.127785613	–0.012541746
He	–2.860251227	–0.012686549
Be	–14.568567143	–0.014939565

In contrast to the M^8 variant, where you transform from $(\mu\nu|\lambda\kappa)$ to $(ar|bs)$ at once, you need to transform the two-electron integrals step-by-step:

$$(\mu\nu|\lambda\kappa) \rightarrow (a\nu|\lambda\kappa) \rightarrow (ar|\lambda\kappa) \rightarrow (ar|b\kappa) \rightarrow (ar|bs)$$

Exercise 19

1. Copy your RMP2 routine and modify it such that your integral transformation scales with M^5.
2. For the provided examples H_2 and LiH compare the efficiency of both algorithms by counting the number of cycles passed through in the most inner loops. What happens if you increase the number of Slater functions by a factor of two?
3. Calculate the dissociation curve of H_2 with RMP2. Plot both the RHF and RMP2 curves on a relative energy scale (in kcal/mol) in a minimal STO basis with Slater exponent of $\zeta = 1.2$. As in the provided input for H_2, the Hartree-Fock energy of a hydrogen atom is –0.4798356 E_h. What behavior do you observe compared to RHF close to the equilibrium and at far distances $R = 10$ Bohr?
4. Calculate the dissociation curve of He_2 between 4 and 10 Bohr with RHF and RMP2 and plot them on a relative energy scale (in kcal/mol). You will find that there is a minimum in each curve. From your knowledge about HF and MP2, did you expect this behavior? What effect could be the cause for the minimum in the RHF curve?

CHAPTER

FIVE

RECOMMENDATIONS

5.1 General Recommendations

5.1.1 Working with this Script

1. Work on the exercises in the given successive order. In the first exercises you will learn some basic routines and procedures which you will need again later but which will not be explained once more.
2. Read the whole exercise before you start working on it. Often technical hints are given at the end.
3. Programs can crash. So check your outputs as soon as possible to make sure your calculations actually did work. And sometimes preparing the input and running the program is much faster than finding the right number in the output.
4. Prepare an LibreOffice sheet (or similar) with a collection of your results. Checking them this way is much easier for us.

5.1.2 Trouble Shooting

Many programs may cause many problems, therefore you should follow some simple guidelines to identify their origins:

- "Crap in, crap out": Always check your input (input structures, file formats, input file, chosen keywords etc.) before you start a calculation.
- If a calculation stops abnormally check the output (*e.g.* orca.out, job.last etc.) and error files first. Always make sure that you pipe all needed output data into files if its not done by default.
- Read your output and error files carefully. Especially check the last lines of the output file for error messages that give a hint what may caused the problem.
- If you identified the problem (maybe you have to start at the first point again), check the program manual for additional options or trouble shooting help, fix the problem and restart your calculation.
- If the calculation still stops abnormally and all other possibilities and options are exhausted, prepare a detailed description of the problem, the output/error messages and contact one of the tutors.

5.2 Software Recommendations

5.2.1 Logging in to your work machine

At the beginning of the course you got a number of the computer you will work on.

First you need access to the intranet. The only way to access any machine in the facility is by tunneling via the ssh-server. You will receive an account on one of the ssh-servers as well.

Important: The ssh-server is not only used by you, but also by your fellow students and possibly also by members of the institute. We expect responsible behaviour from you on the server, because many people depend on those servers running reliably.

Whatever you do, never copy large files to the ssh-server, it has only very limited disk space. Also do not run any resource consuming program on the server (anything that needs a GUI is per definition resource consuming if used on shared resources).

First, we will work with three machines in this tutorial, your local machine (`M-Bot`), the ssh-server instance (`ssh3`) and the CIP computer (`c00`). `c00` is an existing machine, you can log in there as well, it is also the least powerful machine, therefore, just do not use this machine for computations, use the one you were assigned. Your local username might also be different from the one in the facility, we will use `stahn` for the user on the local machine as well as for the user at the facility.

Attention: When following the steps described afterward, you have to change the respective names (both usernames and hostnames), of course. You will *not* be able to use `ssh3` to log in the ssh server, but a different machine (most likely `ssh5`).

Please read the tutorial and the code snippets carefully, understand what is shown and adapt the commands accordingly. *A copy and paste approach on this tutorial will fail!*

We will always show a prompt with username and hostname to illustrate who and where we are. To setup a similar prompt in your shell set the following in your bashrc (note that you will have several bashrcs, one on each machine).

```
PS1='[\u@\h \W] \$ '
```

Note for Windows Users

Don't worry if you are using WSL under Windows. You can do all the following steps as described. Just find the notes when things behave a bit special.

The easy way

If you log in for the first time you will be asked for confirming the host identity, do so. The request will be gone if you log in a second time.

```
1 stahn@M-Bot:~/.ssh$ ssh -Y stahn@ssh3.thch.uni-bonn.de
2 The authenticity of host 'ssh3.thch.uni-bonn.de (131.220.44.130)' can't be␣
  ↪established.
3 ECDSA key fingerprint is SHA256:eEdQpqyV6oP0Ddra7H2QDI6kC9rX3XQRAlWxX6LfA6U.
4 Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
```

(continues on next page)

(continued from previous page)

```
Warning: Permanently added 'ssh3.thch.uni-bonn.de' (ECDSA) to the list of known hosts.
Password:
stahn@ssh3:~> ssh -Y stahn@c00
The authenticity of host 'c00 (172.17.3.20)' can't be established.
ECDSA key fingerprint is SHA256:ozq72tQ9gROvzDwv+ZFQ7wc+L/Dmu9Fptbfhf2zfd1M.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'c00,172.17.3.20' (ECDSA) to the list of known hosts.
Password:
Have a lot of fun...
stahn@c00:~> logout
Connection to c00 closed.
stahn@ssh3:~> logout
Connection to ssh3.thch.uni-bonn.de closed.
stahn@M-Bot:~/.ssh$ ssh -Y stahn@ssh3.thch.uni-bonn.de
Password:
Last login: Thu Feb 17 16:39:19 2022 from 131.220.44.207
stahn@ssh3:~> ssh -Y stahn@c00
Password:
Last login: Thu Feb 17 16:39:35 2022 from 131.220.44.130
Have a lot of fun...
stahn@c00:~>
```

Note: In the following guide we will highlight every line, which requires user input

From here you have everything you need to work on the machines, but it might get somewhat inconvenient because you have to type your password every time. Also copying files back to your machine is not easily possible, because you shall not copy big files to the ssh-server.

The following guide is a bit lengthy, but you only have to do it once and you can easily work and move files between your local computer and your work machine.

The right way

We start on your local machine, we create the ssh directory in your home by

```
stahn@M-Bot:~$ cd ~
stahn@M-Bot:~$ mkdir .ssh
stahn@M-Bot:~$ chmod 700 .ssh
```

The last step ensures that you and only you have access to your ssh keys, never allow anyone else access to this directory!

Note for Windows Users

Using WSL, you might have two `.ssh` directories. The Linux one is the same as above and found in:

```
~/.ssh
```

The Windows one can be found in your Windows home directory (assuming `stahn` is your Windows username):

```
/mnt/c/Users/stahn/.ssh
```

Don't get confused by that and decide upon one of these directories (*e.g.* the Linux one) for the next steps. If something doesn't work, check if there are perhaps doubled files interfering each other.

We enter the ssh directory to create a new ssh-keypair, we recommend using elliptic curve keys because they are short and fast:

```
stahn@M-Bot:~/.ssh$ ssh-keygen -t ed25519
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/stahn/.ssh/id_ed25519): id_ssh3
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in id_ssh3
Your public key has been saved in id_ssh3.pub
The key fingerprint is:
SHA256:bDVv26H9hIx1K21pFRZXF2pqfD8Mw9osb2K5opLeOHU stahn@M-Bot
The key's randomart image is:
+--[ED25519 256]--+
|               o*|
|              . +|
|          o  o o |
|        . ..o+ . .|
|         S  +o=o o|
|        o E..=O*++|
|       o .  o=+=X.|
|     +o  . +o.+o.|
|    .ooo. o.+.  .|
+----[SHA256]-----+
```

The key-generator will prompt you a to enter a filename, we will name the key `id_ssh3`, choose any name you find appropriate.

Tip: A very good read on the generation of ssh-keypairs is the Arch Linux wiki page on ssh-keys.

Now we need to copy the public key to the ssh-server. Since you log in for the first time, you have to provide your password in line 5:

```
stahn@M-Bot:~/.ssh$ ssh-copy-id -i id_ssh3 stahn@ssh3.thch.uni-bonn.de
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "id_ssh3.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out␣
↪any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted␣
↪now it is to install the new keys
Password:

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'stahn@ssh3.thch.uni-bonn.de'"
and check to make sure that only the key(s) you wanted were added.
```

You can check, if your key was succesfully added by logging into the machine. The ssh-server will probably be unknown to your local machine, therefore, you have to add it to your known hosts list first, type yes when prompted in line 4.

```
stahn@M-Bot:~/.ssh$ ssh stahn@ssh3.thch.uni-bonn.de
The authenticity of host 'ssh3.thch.uni-bonn.de (131.220.44.130)' can't be␣
↪established.
ECDSA key fingerprint is SHA256:eEdQpqyV6oP0Ddra7H2QDI6kC9rX3XQRAlWxX6LfA6U.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'ssh3.thch.uni-bonn.de' (ECDSA) to the list of known hosts.
Last login: Thu Feb 17 13:56:18 2022 from 131.220.44.207
stahn@ssh3:~>
```

We need to register the ssh-server now in our configuration file

```
stahn@M-Bot:~/.ssh$ vim config
```

We will use `vim` here but feel free to edit the file with your preferred editor and add the lines:

```
Host ssh3.thch.uni-bonn.de
   IdentityFile ~/.ssh/id_ssh3
```

Now we will try again, to see if our connection is correctly established.

```
stahn@M-Bot:~/.ssh$ ssh stahn@ssh3.thch.uni-bonn.de
stahn@M-Bot:~/.ssh$
```

If you are prompted for a password your setup is wrong and you have to retry.

Tip: You can also optionally add your username to the ssh config file and set up a custom Hostname for the ssh-server.

```
Host ssh3
   Hostname ssh3.thch.uni-bonn.de
   User stahn
   IdentityFile ~/.ssh3/id_ssh3
```

This will allow you to easily connect to the ssh-server by just typing:

```
stahn@M-Bot:~/.ssh$ ssh ssh3
Last login: Thu Feb 17 13:57:03 2022 from 131.220.44.207
stahn@ssh3:~>
```

Now we have to repeat the same steps for the machine at the facility, but first we need to be able to directly connect to it from our local working machine. We do so by altering the ssh-config and adding the following lines:

```
Host c00
   ProxyJump ssh3
   User stahn
```

We just told our system, that it needs to use the ssh-server as a proxy for connecting to our remote working machine. This enables us to connect to our remote working machine at the facility by a single ssh command:

```
stahn@M-Bot:~/.ssh$ ssh c00
The authenticity of host 'c00 (<no hostip for proxy command>)' can't be established.
ECDSA key fingerprint is SHA256:ozq72tQ9gROvzDwv+ZFQ7wc+L/Dmu9Fptbfhf2zfd1M.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'c00' (ECDSA) to the list of known hosts.
Password:
Have a lot of fun...
stahn@c00:~>
```

Now we generate another keypair (always use a new keypair for each connection) and register the connection like before:

```
stahn@M-Bot:~/.ssh$ ssh-keygen -t ed25519
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/stahn/.ssh/id_ed25519): id_c00
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in id_c00
```

(continues on next page)

(continued from previous page)

```
Your public key has been saved in id_c00.pub
The key fingerprint is:
SHA256:mUBCFiGUc6kqbb1fspxwQ0k9V0eT8sg59bV80w7jPTM stahn@M-Bot
The key's randomart image is:
+--[ED25519 256]--+
| .oo*+.      ..+. |
|  oooo .    ...o..|
|   +  o o .. *..+|
|  .  . o =  = +++|
| o .  o S    o =o|
|o o ..        .Eo|
|..  ..+ .       +|
|    .+ *         |
|     .=          |
+----[SHA256]-----+
stahn@M-Bot:~/.ssh$ ssh-copy-id -i id_c00.pub c00
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "id_c00.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out␣
↪any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted␣
↪now it is to install the new keys
Password:

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'c00'"
and check to make sure that only the key(s) you wanted were added.
```

Finally we want to include the new ssh-key to our ssh-config by adding the following lines to our ssh-config:

```
Host c00
   ProxyJump ssh3
   User stahn
   IdentityFile ~/.ssh/id_c00
```

Now try to login to the work machine again (remember to specify the X forwarding).

```
stahn@M-Bot:~/.ssh$ ssh -Y c00
Last login: Thu Feb 17 15:00:38 2022 from 131.220.44.130
Have a lot of fun...
stahn@c00:~>
```

Again, if you have to enter your password, the setup was not correct and you have to retry. From now on, you can also copy files from and to your work machine.

```
stahn@M-Bot:~/.ssh$ scp Lehre/lect3_htm.doc c00:Documents/.
stahn@M-Bot:~/.ssh$ scp c00:Lehre/QC2.pdf ~/Lehre/QC2/.
```

As a short recap, you should now be able to log in with a single command.

```
stahn@M-Bot:~/.ssh$ ssh c00
Last login: Thu Feb 17 15:08:55 2022 from 131.220.44.130
Have a lot of fun...
stahn@c00:~>
```

Tips and Tricks

For the three machine setup we had, a configuration file like the following would be appropriate:

```
Host c00
  User stahn
  IdentityFile ~/.ssh/id_c00
  ProxyJump ssh3

Host ssh3
  Hostname ssh3.thch.uni-bonn.de
  User stahn
  IdentityFile ~/.ssh/id_ssh3
```

If you are working remotely over ssh, any process you start with the shell will be terminated as soon as you log out. Keeping your process alive, requires that you detach the process from your terminal. You can create a completely detached process by:

```
stahn@M-Bot:~/.ssh$ setsid xtb h2o.xyz > xtb.out
```

However, keep in mind, that you have no control at all over this process after starting it. Normally, setting the process to ignore Hangup Signals and rerouting the output of the process is enough to keep it alive. You can do so by using nohup.

```
stahn@M-Bot:~/.ssh$ nohup xtb h2o.xyz
```

Any output created by the process will be printed to nohup.out.

Note: `nohup` is a useful to run commands on your work machine that should continue even if you log out from the ssh-session.

More lengthy calculations with quantum chemistry software are a potential target for this approach. But think first before adapting the above command, because you probably want to keep the output instead of scrapping it to `/dev/null`. Also, you won't have to kill your program in the end, because it will terminate on its own.

If you like the prompt style and want to use it for your bash as well, there is also a colorful version available. Just add this lines to your bashrc (if you always want a full path use `\w` instead of `\W`).

```
if ${use_color} ; then
  if [[ ${EUID} == 0 ]] ; then
    # show a red prompt if we are root
    PS1='\[\033[01;31m\][\h\[\033[01;36m\] \W\[\033[01;31m\]] \$\[\033[00m\] '
  else
    PS1='\[\033[01;32m\][\u@\h \W] \$\[\033[01;37m\] '
  fi
else
  if [[ ${EUID} == 0 ]] ; then
    # show root@ when we don't have colors
    PS1='[\u@\h \W] \$ '
  else
    PS1='[\u@\h \W] \$ '
  fi
fi
```

Note: If you want other colors, play a bit around with the last number in the bracktes ([033[01:**31** m]). If you want your username in different color than your path you can also specify this. Play a bit around with it.

5.2.2 X-Server or How to make your graphical connection work (optional)

Sometimes it is easier to directly have a look at structures or plots, instead of copying everything to your local computer. Therefore, we recommend an application that enables you to open graphical interfaces on the CIP Pool computers in the Mulliken Center and see the opened windows on your home computer. For everyone, who is interested, just google "X-Server connection windows linux" or some similar combination and try to install this on your own. For all others: Install Xming, a free Windows stand-alone program, and follow the setup there. Afterwards, always ensure that `Xming` is running, when you open a shell and try to open some visualization software. For that, you only have to start `Xming` (press the Windows button, type `Xming` and press enter), then the `Xming` symbol will appear at your taskbar. Now open a shell and type:

```
echo "export DISPLAY=localhost:0.0" >> ~/.bashrc
source ~/.bashrc
```

Now you can login as described above (remember to have `Xming` running).

5.2.3 Software for Visualization of Molecules

A quantum chemical calculation always needs a structure as input (and will often result in a modified structure as output), so you need some kind of visualization program to create the desired molecule or to look at it. We recommend the use of the program Avogadro to generate and manipulate molecules. Next, you will need the program molden for some exercises (we recommend the version `gmolden`). You can open an input file (*e.g.* `molden.input` or a `*.xyz` file) by typing:

```
gmolden <input>
```

For Windows users that have unpacked the above linked .rar file, we recommend opening the input file (`molden.input` or `*.xyz`) by right-clicking on it and selecting "Open with", then choose the unpacked `gmolden.exe` file. You can also use `gmolden` for generation and manipulation of molecular structures, but we recommend the use of `Avogadro`. Of course you can also use any other visualization software you know. Please remember that for some exercises it is important to keep the atom count during the manipulation of the molecule geometry, which some of the more common programs do not do (`Avogadro` keeps it).

Note: During testing `gmolden` with Windows 10, we encountered problems if the path contains blanks or umlauts (*e.g.* C:Program Filesmolden). If you cannot open `gmolden` on your windows computer, copy the *molden folder* to you desktop and try again.

5.2.4 Plotting

For some exercises you have to create proper plots. In our group we usually use `gnuplot` for this, a powerful program if you can handle it. `gnuplot` scripts for any plotting problem you can imagine (and much more) are easy to find on the Internet. In general, you tell the program via a small script in which format you want your final picture, you name your axis and then plot directly from an external file. In the following, you will find a small script called `plot.gp` to plot your data points as a line with `gnuplot`.

```
1 set terminal pdf color font 'Times-Roman, 30'    # Produce files in pdf format as␣
  ↪output, you can also choose jpeg, eps, or whatever you like
2 set output 'NAME.pdf'                            # your final file is named "NAME.pdf"
3 set encoding iso_8859_1                          # Sometimes needed for e.g. the
  ↪"angstrom" symbol
4
5 set key font "Times-Roman, 20"                   # Sets a legend for your plot.
```

(continues on next page)

(continued from previous page)

```
6
7  set xlabel "X-AXIS" font",20"                    # Sets name for the X-axis (don't␣
   ↪forget the unit!)
8  set xtics nomirror                               # Tells gnuplot, that the scale is␣
   ↪only shown on one side
9  set xtics font 'Times-Roman, 20'                 # Sets font for the x-scale
10 set xzeroaxis                                    # Draws a line at y=0
11 set ylabel "Y-AXIS" font",20"                    # Same as for the X-axis, just for␣
   ↪the y-axis
12 set ytics nomirror
13 set ytics font 'Times-Roman, 20'
14
15 plot \                                           # Finally the plot command. The "\"␣
   ↪tells gnuplot to also plot the next line. Remove the out-commented description␣
   ↪before plotting, as it can cause errors.
16 'file.txt' u 1:2 w l lw 2, \                     # "file.txt" is the File which will␣
   ↪be plotted. "u 1:2" means literally "use column 1 and 2", "w l" = with lines ("w lp
   ↪" = with line points, prints a line with points at the respective data points), "lw␣
   ↪2" = linewidth 2. You can do many more things here, these are just some exemplary␣
   ↪points. Remove this comment before plotting.
```

Copy this file in your working directory, if you want to plot something with `gnuplot`. For actually plotting your data, change at least `file.txt` to however your file with the data points is called, and then type:

```
gnuplot plot.gp
```

Now you can find your graphic `NAME.pdf` in the directory, where you executed your plot script. To look at it, you can either copy the file to your local computer (and use whatever pdf reader you use to open it), or you can open it with e.g. *Okular* (preinstalled on the MCTC computers) by typing:

```
okular NAME.pdf
```

Remember that you need a graphical connection for the latter. If you now want to change something in your plot, you just have to modify the script `plot.gp` and plot it again as described above.

Instead of `gnuplot`, you can also use any other plotting program (Microsoft's *Excel*, LibreOffice's *Calculator*, *SciDavis*, you name it). In the end, it is only important that the plots follow some simple rules:

1. Axes are labeled with the correct expression and unit (e.g. **time / h**).
2. Axes are divided with markings/tics and numbers.
3. All lines in a plot should look different. Different colors are one possibility, which breaks down by printing the protocols in black and white. You can, of course, use colors, but if you are plotting more than one line, you must also make sure that each line is distinguishable without color (e.g. by using different markers).
4. Remember: the first thing you usually look at in publications are pictures. Writing protocols prepares you for writing scientific papers, so it is also important to learn how to create nice figures. Every letter (title, axes, etc.) and also the lines should be printed in a size that we can see them at a glance without a magnifier. Avoid similar colors and markings if possible. Name your curves with meaningful expressions.

All figures in your final report must have captions that adequately describe the illustration. Captions should describe the contents of a figure in as few words as possible.

Hint: If you do not immediately understand your own plot after two days, it is probably bad. Rethink.

5.2.5 Summary

Check the `.bashrc` of your local Linux distribution and add `export DISPLAY=localhost:0.0`, if you want to use a graphical interface to the MCTC computers.

Program	local / MCTC	Links (if local installation needed)	optional?
Xming	local	https://xming.en.softonic.com	yes
avogadro	local / MCTC	https://avogadro.cc/	no
molden	local / MCTC	https://uni-bonn.sciebo.de/s/XxSEG5DHbzitX7Z	no
gnuplot	MCTC	[-]	yes

CHAPTER

SIX

QUANTUM CHEMISTRY SOFTWARE

This section will give a short introduction and an overview of the Quantum Chemistry programs that will be used in this practical course.

Contents

6.1 Setting up the Software

In order to gain access to the needed software packages, you need to make some changes to your system. The `.bashrc` file located in your home directory is sourced every time you open a new shell. While you can directly execute any program by giving the full path, it is more convenient to tell the system where to look for the binaries by saving the location in the `$PATH` variable. Additionally, some programs need global variables. All those are usually set in the `.bashrc`. In order to gain access to all the needed software, add the following lines to your `.bashrc`:

```
# AKbin
export PATH=/home/abt-grimme/AK-bin:$PATH
export PATH=/home/$USER/bin:$PATH

module use /software/modulefiles
module load turbomole orca
alias molden='/software/bin/molden'

# XTB
export OMP_NUM_THREADS=2
export MKL_NUM_THREADS=2
export OMP_STACKSIZE=500m
ulimit -s unlimited
```

Be sure to create a directory called `bin` in your home directory by typing:

```
mkdir ~/bin
```

Important: All changes apply to shells opened afterwards.

If you want to apply the changes to your current shell, you need to run:

```
source ~/.bashrc
```

6.2 Program Packages

6.2.1 TURBOMOLE

To run a basic calculation with TURBOMOLE, you will only need the following two files:

- `coord`: Molecular geometry in atomic units
- `control`: All data required for the calculation (method, parameters, …)

For more sophisticated calculation settings, TURBOMOLE provides an interactive input generator called `define` which only needs the `coord` file to create the `control` file. Basis set and orbital files, that are also necessary for the calculation, are created during the calculation or by `define`, respectively. These are:

- `basis` (and `auxbasis`): Basis set (and auxiliary basis set for RI)
- `mos` or `alpha` and `beta`: Orbitals (MO coefficients) for restricted and unrestricted treatment, respectively

In this course, all calculations can be prepared manually by only providing a `coord` and a `control` file containing all the necessary information.

TURBOMOLE has different binaries and scripts for different jobs. While they do not need an explicit input file when called, you should **always** pipe the output into a file, *e.g.*:

```
ridft > ridft.out &
```

The most important scripts that come along with the TURBOMOLE program package are listed in the following table.

TURBOMOLE script	Functionality
Most important scripts for calculations	
`ridft`	DFT and HF SCF calculations with the RI-approximation
`dscf`	DFT and HF SCF calculations without the RI-approximation
`ricc2`	Module for second-order correlated WF methods (MP2, CC2)
`ccsdf12`	Module for coupled cluster methods (CCSD, CCSD(T), …)
`rdgrad`, `grad`	Calculate gradients (with and without RI)
`aoforce`	Calculate analytical vibrational frequencies
`statpt`	Coordinate/Hessian update for stationary point searches
`jobex`	Script for geometry optimizations
Scripts for visualization purposes	
`eiger`	Show the orbital energies and the HOMO-LUMO gap
`x2t`	Convert a *.xyz (in Å) file to coord (in bohr)
`t2x`	Convert a coord (in bohr) file to *.xyz (in Å)
`tm2molden`	Generate a molden input

Important: Each TURBOMOLE calculation needs its own directory.

The `control` file

While `coord` stores the molecular geometry, the `control` file contains all the specifications and settings for the desired calculation. It contains keywords indicated with a `$` symbol followed by some setting. Related specifications sometimes follow in the next line and are indented. Every `control` file must end with the `$end` keyword in the last line. An example input for a simple DFT calculation on the BLYP/def2-TZVP level of theory can look as follows:

```
$coord file=coord
$atoms
  basis = def2-TZVP
$dft
  functional b-lyp
$end
```

In the following exercises, some proper TURBOMOLE input will always be given (at least partially) in the text. Additionally, you can find a short list of all keywords needed in this course in the *Keywords in control* section below.

6.2.2 ORCA

ORCA needs an input file (*e.g.* `myinput.inp`) to run. A typical call to perform a calculation with ORCA would be

```
orca myinput.inp > myinput.out &
```

The input file is generally structured as follows:

```
# Comment lines are marked with an '#' and are possible everywhere
! Method Basis and further options

# Input is organized in blocks which start with '%'
# e.g.
%scf
        MaxIter 150 #maximum number of iteration steps in the scf,
                    #default = 50
end
# definition of input geometry
* xyz <charge> <multiplicity>
        cartesian coordinates (Angstroms)
*
or:
* int <charge> <multiplicity>
        Z-Matrix
*
or:
* xyzfile <charge> <multiplicity> <filename.xyz>
```

Important: Multiplicity = 2S+1 with S being the total spin.

A short reference of ORCA keywords can be found in the section *Short ORCA Reference*. Further information is accessible from: https://sites.google.com/site/orcainputlibrary/.

6.3 Keywords in `control`

The `control` file contains all specifications and settings for a calculation with TURBOMOLE. Keywords start with `$` and sub-settings are indented. The last line of the file must always be `$end`. The following table shows the most important keywords that are interesting for this course.

Command	Functionality
essential for all calculations	
`$coord file=coord`	Defines the `coord` file to be the one containing the molecular structure information.
`$atoms` `  basis=<bas>`	Defines the basis set for the calculation to be `<bas>`.
always recommended	
`$eht charge=<chrg> unpaired=<uhf>`	Defines the charge `<chrg>` and the number of unpaired electrons `<uhf>` for the extended Hückel guess and the entire rest of the calculation.
`$symmetry <sym>`	Use the symmetry of pointgroup `<sym>`. If not stated otherwise, in the scope of this course it is always recommended to use C_1 symmetry to avoid technical issues (choose `c1`).
`$rij`	Use the resolution of the identity (RI) approximation. Note that you then have to use `ridft` for single-point calculations and the `-ri` option for `jobex`. We recommend using the RI approximation for all exercises in this course.
`$energy file=energy` `$grad file=gradient`	For geometry optimizations: The energies and gradient of all optimization cycles will be saved in the files `energy` and `gradient`.
DFT calculations	

`$dft`
`  functional <func>`
`  grid <grid>`
	Perform a DFT calculation using the functional `<func>`. Note that the BYLP, B3YLP and B2PLYP functionals are named `b-lyp`, `b3-lyp` and `b2-plyp`, respectively. Define the integration grid `<grid>` (optional, the default is `m4`).
	For double-hybrid functionals, also include the settings for MP2 calculations listed below (`$ricc2` and `$denconv` blocks).

`$disp3 -bj`	Use the D3 dispersion correction with Becke-Johnson damping.
Note: If the `$dft` block is missing, a HF calculation will be performed.	
Post HF calculations	
`$scfconv 7` `$denconv 1.0d-7`	For MP2 and CC calculations, a well-converged SCF run is needed. Therefore, set the convergence threshold of the SCF and the density matrix to 10^{-7} or less.

`$ricc2`
`  mp2`
`  geoopt model=mp2`
	Perform an MP2 single-point calculation.
	The `geoopt model=mp2` keyword is only necessary if a geometry optimization on the MP2 level is desired.

`$ricc2`	Perform a CCSD(T) single-point calculation.

6.4 Short ORCA Reference

You can find a rough summary of the most important ORCA keywords in the following table. For a complete reference, consult the manual at https://orcaforum.kofo.mpg.de/.

Keyword	Explanation
RHF	Restricted Hartree-Fock
UHF	Unrestricted Hartree-Fock
TPSS	DFT with the functional TPSS (can be any valid functional)
MP2	Do an MP2 calculation.
CCSD(T)	Do a CCSD(T) calculation.
TZVP	Use a TZVP basis. Can be any valid basis set definition
Opt	Do a geometry optimization.
RI	Use the resolution of the identity approximation.
NumFreq	Calculate second derivatives (vibrational frequencies). Also gives an IR spectrum and thermal corrections + ZPE.
NMR	Calculate nuclear magnetic shielding tensors.
TightSCF	Increases the convergence criterion for the SCF.

CHAPTER

SEVEN

EXERCISES

Contents

7.1 Electron Correlation

7.1.1 Multireference Methods

Exercise 1.1

Electron correlation is very important in dissociation processes to get qualitatively and quantitatively correct results. Calculate the potential energy curves for the dissociation of HF with the single-reference methods RHF, UHF, MP2, CCSD(T) and CASSCF, which is a multi-reference method. Compare the results.

Approach

1. In order to easily calculate potential energy curves, we use ORCA. Create the following inputs for the given methods and save them in different directories. (*e.g.* `rhf.inp`, `uhf.inp`, etc.).

   Input for RHF:

```
! RHF def2-TZVP TightSCF

%paras R= 4.0,0.5,35 end

* xyz 0 1
H 0 0 0
F 0 0 {R}
*
```

   Modifications for

   UHF:

```
! UHF def2-TZVP TightSCF

%scf BrokenSym 1,1 end
```

   MP2:

```
! RHF MP2 def2-TZVP TightSCF
```

   CCSD(T):

```
! RHF CCSD(T) def2-TZVP TightSCF
```

   CASSCF (2 active electrons in 2 orbitals):

```
! RHF def2-TZVP TightSCF Conv

%casscf
  nel 2
  norb 2
  switchstep nr
end
```

   Calculate the potential energy curve by a CASSCF calculation with 6 electrons (`nel`) in 6 active orbitals (`norb`) as well.

2. Call ORCA with the command

```
orca <file>.inp > <file>.out
```

3. Plot the resulting potential energy curves using *e.g.* with `gnuplot` (see section *Plotting*). To do so, delete the first line in the files `<filename>.trj*.dat` to read them (find out which file is the right one yourself).
4. Calculate the energies for hydrogen and fluorine atoms for all given methods. Which methods will yield the identical energies for the hydrogen atom?
5. Plot the curves relative to the energies of the individual atoms and discuss your results (particularly the energies at large distances).

Hint: Have a closer look at the UHF dissociation curve. Does it look as you would expect it? Try to explain the "strange" behavior in terms of symmetry breaking.

Hint: If you encounter convergence problems in the CAS(6,6) calculations, try increasing the maximum number of macro-iterations for CASSCF by adding the following keyword in the `%casscf` block:

```
6  maxiter 1000
```

7.1.2 Carbenes

All following calculations will be done with TURBOMOLE if not stated otherwise. Also, if not specified otherwise we will use the RI approximation and C_1 symmetry throughout.

Exercise 1.2

Calculate the singlet-triplet splitting of methylene and *p*-benzyne with HF, MP2, DFT and CCSD(T).

Approach

1. Create the file `coord` with starting geometries for Methylene and *p*-Benzyne.

   The syntax is:

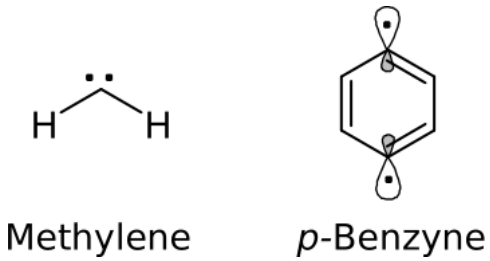
```
1  $coord
2  x y z atom1
3  x y z atom2
4  ...
5  ...
6  $end
```

   You can either create the files by hand or use the program `Avogadro` for this purpose (see section *Software for Visualization of Molecules*). `Avogadro` uses Å as unit, but the unit for the `coord` file has to be bohr (atomic units). To convert a *.xyz file into a coord file you can use the command

```
x2t *.xyz > coord
```

   This also works the other way round:

```
t2x coord > *.xyz
```



2. **Methylene**: Optimize the geometries of the singlet and the triplet state with the given methods (HF, TPSS, B3LYP, PW6B95, MP2) and the basis set def2-TZVP. The following input is an example of a `control` file with the correct options for a B3LYP/def2-TZVP calculation of the triplet (= 2 unpaired electrons) state.

```
$coord file=coord
$eht charge=0 unpaired=2
$symmetry c1
$atoms
  basis=def2-TZVP
$dft
  functional b3-lyp
$rij
$energy file=energy
$grad file=gradient
$end
```

To generate the input for the other calculations, please look at the table provided in section *Keywords in control*. For the MP2 calculations, exchange the `$dft` with the proper `$ricc2` block and don't forget to specify `$denconv` small enough.

Geometry optimizations (DFT, HF) are done with the program `jobex`:

```
jobex -ri > jobex.out
```

Note that for MP2 geometry optimizations, you have to add the `-level cc2` option. Energies after geometry optimizations can be found in the file `job.last`. HF and DFT energies for each SCF-cycle are additionally written in the file `energy`. In the case of CCSD(T) (also defined via the `$ricc2` block), do not perform a geometry optimization, but do a single-point calculation on the MP2 optimized geometries. Before performing the actual CCSD(T) calculation, you have do run a HF-SCF:

```
ridft > ridft.out
ccsdf12 > ccsdf12.out
```

The energies can then be found in `ccsdf12.out`.

3. In order to measure the angles of the optimized structures, you can use `Avogadro` or a small TURBOMOLE script called `bend`:

```
bend i j k
```

with atom numbers `i`, `j`, `k`.

Note down the HCH-angle and total energies for each method, as well as the singlet-triplet splitting ($\Delta E_{\text{S-T}} = E_{\text{singlet}} - E_{\text{triplet}}$). The experimental value for the splitting is 9.0 kcal·mol^{-1}. The experimentally found angles are 102.4° for the singlet and 135.5° for the triplet.

4. **p-Benzyne**: Repeat the same calculations for *p*-benzyne. The experimental value for the splitting is -4.2 kcal·mol^{-1}.
5. Discuss your findings and compare them to the experiment.

Hint: A HF calculation can be performed by simply omitting the `$dft` block in the above input file. For post HF calculations, add the `$ricc2` block:

```
$ricc2
  mp2                 # or ccsd(t)
  geoopt model=mp2    # only for MP2 geometry optimizations
```

Hint: If you encounter convergence problems in the MP2 calculation, try to increase the maximum number of SCF cycles (e.g. 100) by adding the following keyword:

```
  $scfiterlimit <limit>
```

7.2 Basis Set Convergence

7.2.1 Formic Acid Dimer

Exercise 2.1

Investigate the basis set convergence behavior of different methods for the formic acid dimer.

Approach

1. Create separate directories for the formic acid dimer and monomer and set up geometry optimizations on the TPSS-D3/def2-TZVP level of theory. To do so, create structures using *e.g.* `Avogadro` and convert them to `coord` files. Prepare the calculations in the same way as in exercise 1.2. You can use a similar `control` file and add the `$disp3 -bj` keyword for the D3 dispersion correction. **DFT** geometry optimizations are performed via:

   ```
   jobex -ri > jobex.out
   ```

2. Using the optimized geometries, calculate the dimerization energy (energy difference of one dimer and two monomers) with HF, TPSS-D3 and MP2 employing the cc-pVXZ (X = D, T, Q) basis sets and their augmented counterparts (aug-cc-pVXZ). Refer to the table of input options given in section *Keywords in control*.
3. Tabulate your results and plot the total energies versus the cardinal number of the basis set for each method (*e.g.* with `gnuplot`).
4. Discuss your findings with respect to the basis set superposition error (BSSE) and the basis set incompleteness (BSIE). Which methods can be considered as converged towards the basis set limit when used with a quadruple-ζ basis?

Hint:

- The calculations with quadruple-ζ basis can be quite time consuming. Exploiting the symmetry by choosing the correct point group in the `$symmetry` block might accelerate the calculations.

- Remember that you have to perform a HF single-point calculation (using `ridft`) before you can start the MP2 calculation with `ricc2` in the same directory.

7.3 Thermochemistry

7.3.1 Reaction Enthalpies of Gas-Phase Reactions

Exercise 3.1

For small molecules, highly accurate thermochemical results are reachable in quantum chemistry. This means *chemical accuracy* with an error of less than 1 kcal/mol. Calculate the reaction enthalpies at 298 K for the following, industrially important reactions:

$CH_4 + H_2O \rightleftharpoons CO + 3\ H_2$ (steam reforming of methane)
$N_2 + 3\ H_2 \rightleftharpoons 2\ NH_3$ (Haber-Bosch process)

The experimental data are:

Reaction	$\Delta H_r(298\,\mathrm{K})$ / kcal·mol^{-1}
Steam reforming	+49.3
Haber-Bosch	-22.5

Approach

1. Optimize the reactants and products using TPSS-D3/def2-TZVP (see earlier exercises and section *Keywords in control*, keep in mind the `$disp3 -bj` keyword).
2. In order to get the thermal corrections from energy to enthalpy at 298 K, do a frequency calculation first. Use the program `aoforce` to calculate the vibrational frequencies in TURBOMOLE:

   ```
   aoforce > aoforce.out
   ```

3. Then, calculate the thermal enthalpy corrections ΔH_{298} with the program `thermo`. It needs a `.thermorc` input file from your home directory. Create this file by typing:

   ```
   echo "0.0  298.15  1.0" > ~/.thermorc
   ```

   The first number is an internal threshold, the second the temperature in Kelvin and the third the scaling factor for the vibrational frequencies (1.0 for TPSS). Pipe the output into a separate file, *e.g.*:

   ```
   thermo > thermo.out
   ```

4. Repeat the optimization for the molecules involved in the Haber-Bosch process with MP2/def2-TZVP. Calculate the deviation of these differently optimized structures by computing the root mean square deviation of the coordinates:

   ```
   rmsd <tpss-geometry> <mp2-geometry>
   ```

5. Calculate single-point energies with the hybrid functional B3LYP-D3/def2-TZVP and with MP2/def2-TZVP. Use the TPSS-D3 geometries and thermal corrections to calculate the reaction enthalpies.
6. Calculate single-point energies with the double hybrid B2PLYP-D3/def2-QZVP and with CCSD(T)/def2-QZVP. Use the TPSS geometries and thermal corrections to calculate the reaction enthalpies. Keep in mind that you have to run an SCF first with `ridft`. Afterwards, use `ricc2` for the double-hybrid and `ccsdf12` for the coupled cluster calculation. The energies can be found in the respective output.
7. Tabulate your results and compare to the experimental values. Which method would you expect to show the smallest deviation to the experiment? Do your findings match your expectation? Disscuss your outcome.

7.3.2 Heat of Formation of C_{60} (optional)

Exercise 3.2

Calculate the heat of formation ΔH_f^0 of the C_{60} molecule by using different methods.

Approach

1. Optimize the geometry of C_{60} on the TPSS-D3/def2-SVP level in I_h symmetry. In order to do so you can build a `coord` file on your own or search for a proper input on the internet. The symmetry can be defined with the `$symmetry ih` keyword in `control`.
2. Calculate the energy of C_{60} on TPSS/def2-SVP level without D3 corrections, use the TPSS-D3/def2-SVP optimized geometry.
3. Calculate the frequencies of C_{60} and the thermal corrections the same way as in exercise 3.1.
4. Now, calculate the energy of a single carbon atom on the TPSS/def2-SVP level of theory and the thermal corrections to $\Delta H_{298.15}$ (use C_1 symmetry).
5. Calculate ΔH_f^0 of C_{60} and compare to experimental results (599 / 635 kcal/mol). You will need the experimental ΔH_f^0 of a carbon atom: 170.89 kcal/mol.
6. Calculate single-point energies (without dispersion correction) for carbon and C_{60} with TPSS and HF employing the def2-TZVP and the def2-QZVP basis sets. Use the results to calculate the heat of formation. (Use the TPSS-D3/def2-SVP geometries and corrections to $\Delta H_{298.15}$ for this purpose.)
7. Calculate the D3 dispersion correction to the TPSS/def2-QZVP energy and calculate ΔH_f^0 again. Use the standalone program `dftd3`:

```
dftd3 coord -func tpss -bj
```

8. Discuss your results.

7.4 Kinetics

7.4.1 Kinetic Isotope Effect

Exercise 4.1

Calculate the kinetic isotope effect for the reaction $CH_4 + HO\cdot \rightarrow \cdot CH_3 + H_2O$. From transition state theory, it is known that

$$\frac{k_{\mathrm{H}}}{k_{\mathrm{D}}} = e^{-\frac{\Delta H_{\mathrm{H}}^{\neq} - \Delta H_{\mathrm{D}}^{\neq}}{RT}}.$$

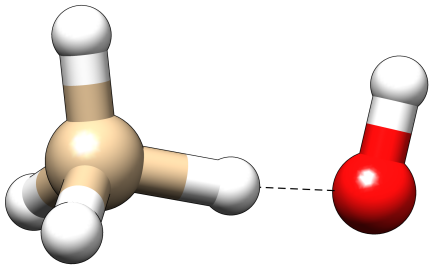


Fig. 1: Geometry of the transition state.

Approach

1. Calculate the geometry of the transition state for the hydrogen transfer. In order to do this, create a `coord` file with a starting geometry that is similar to the one in the picture, with $R_{C-H} \approx 1.2$ Å and $R_{O-H} \approx 1.3$ Å.

   In order to find the transition state, use the following steps:

   (a) Prepare a `control` file. Use the B3LYP-D3/def2-TZVP level of theory. Look at section *Keywords in control* if you are unsure. You might need to increase the `$scfiterlimit` in this exercise.

   (b) Consecutively, calculate energy, gradient and hessian:

   ```
   ridft > ridft.out
   rdgrad > rdgrad.out
   aoforce > aoforce.out
   ```

   (c) Verify that there is at least one, relatively large imaginary frequency in the output of `aoforce` (it also appears in the `vibspectrum` file). Then, add the following block to the `control` file.

   ```
   $statpt
     itrvec 1
   ```

   (in general the frequency mode describing the motion of the reaction)

   (d) Start the transition state search:

   ```
   jobex -ri -trans > jobex.out
   ```

2. When the search is successful (a `GEO_OPT_CONVERGED` file has been created in the directory), calculate the vibrational frequencies of the transition state (`aoforce`) and verify that there is only one imaginary frequency. You can have a look at that corresponding normal mode by calling:

   ```
   tm2molden
   ```

   Choose your desired options in the short interactive experience. You do not need to pick a name for the input file or to save the MO data, the latter will make the file rather large (but obviously save the frequency data). You can open the resulting file (default: `molden.input`) with `gmolden`. The normal modes can be visualized by clicking on "Norm. Mode" on the right side of the menu.

3. Call the program `thermo` and note down the thermal corrections to the enthalpy.

4. Repeat the transition state search with CD_4 and OH. Therefore, take the initial `control` file and add an additional non-default atom weight definition to the 4 affected hydrogen atoms in the `$atoms` block. You need to look in the `coord` file and identify the indices of the `h` atoms you want to change to deuterium. Say these hydrogen atoms are atoms 2, 3, 4 and 5, then change the `$atoms` block to:

```
$atoms
  basis=def2-TZVP
h 2,3,4,5
  mass=2.014
```

5. With this input, repeat all aforementioned steps.
6. Calculate the energies and thermal corrections for CH_4, CD_4 and OH.
7. Finally, calculate k_H/k_D.

7.5 Solvation

7.5.1 S_N2-Reaction

Exercise 5.1

Calculate the potential energy curve for the S_N2-reaction of chloromethane with a flouride anion in the gas-phase and in methanol (ε = 32) between $r(C-F)$ = 2.25 and 8.00 bohr with ε being the dielectric constant of the solvent.

Approach

1. Create structures and calculate the energies of the reactants (one calculation for each reactant) in the gas-phase and at ε = 32 (methanol). Use the hybrid functional PW6B95 with a def2-TZVP basis and D3 dispersion correction. Keep in mind that you also have to specify the charge and the solvent model in the `control` file, *e.g.*:

```
$eht charge=-1 unpaired=0
$cosmo
  epsilon=32.0
```

2. To create the potential energy curves, use the shell script below. The script loops over all distances. For each distance it creates a new directory, performs the constrained geometry optimization and writes the electronic energy (not necessarily your final reaction energy) into a file called `results.dat`. Create a new directory and copy and paste the script to a file named `run-scan.sh`.

```
#!/usr/bin/env bash

# Choose directory here
calc_dir=scan_vac

cd $calc_dir
if [ -f ./results.dat ]
then
  rm results.dat
fi

read -r -d 'END' template <<-EOF
\$coord
   0.00000000      0.00000000      0.00000000  c f
```

(continues on next page)

(continued from previous page)

```
   0.00000000      0.00000000     -3.36989165  cl
   0.00000000      0.00000000      DIST        f f
  -1.00404366      1.73905464     -0.62462166  h
  -1.00404366     -1.73905464     -0.62462166  h
   2.00808733      0.00000000     -0.62462166  h
\$end
END
EOF

for dist in $(seq 2.25 0.25 8.00 | sed s/,/./)
do

  # Check for existence of folder
  if [ -d $dist ]
  then
    rm -r $dist
  fi
  mkdir $dist
  cp control $dist
  pushd $dist
  echo "$template" | sed "s/DIST/$dist/" > coord

  jobex -ri -c 50

  # Get final energy
  e=$(sdg energy | tail -1 | gawk '{printf $2}')

  # Write energy to a file
  echo $dist $e >> ../results.dat
  popd

done
```

A template for the `coord` file is given directly inline in the script, we will repeat it here to explain a few details. The `f` after the atom specification tells TURBOMOLE to keep the coordinates fixed for that atom. `DIST` ist a placeholder which will be substituted by the C–F distance.

```
$coord
   0.00000000      0.00000000      0.00000000  c f
   0.00000000      0.00000000     -3.36989165  cl
   0.00000000      0.00000000      DIST        f f
  -1.00404366      1.73905464     -0.62462166  h
  -1.00404366     -1.73905464     -0.62462166  h
   2.00808733      0.00000000     -0.62462166  h
$end
```

In order to use the script, you have to make it executable by typing:

```
chmod +x run-scan.sh
```

Create subdirectories (*e.g.* `scan_vac` and `scan_cosmo`) for each potential energy curve and place a proper `control` file in each of these subdirectories. You will have to adapt the script to your directory names (name in line 4). The `results.dat` file will be written to the respective subdirectory. Execute the script by typing:

```
./run-scan.sh
```

3. Plot the two curves together (normalize the curves reasonably) and discuss the results. Estimate the activation

barrier for both cases.

Hint: If the execution of such a script takes some longer time, consider calling it with:

```
nohup ./run-scan.sh &
```

Then, you can log out of your shell without killing the calculation.

7.6 Activation Energies

7.6.1 Rearrangement and Dimerization Reactions

Exercise 6.1

Estimate the activation energy for the Claisen rearrangement of allyl-vinyl ether and the dimerization of cyclopentadiene to *endo*-dicyclopentadiene (Diels-Alder).

Approach

1. Construct the geometry of reactant(s) and product for each reaction (*e.g.* using `Avogadro`). Ensure that the sequence of atoms is the same in every pair of reactant and product structure.

   Hint: Preparing good input structures for transition state searches is absolutely essential, often you can easily create a product structure from your reactant. Furthermore, this generally eases the sorting of the atoms.

2. Optimize the geometries using GFN2-xTB. GFN2-xTB is a semi-empirical tight-binding based method that employs a minimal valence basis set and is very efficient for calculating **G**eometries, vibrational **F**requencies and **N**oncovalent interactions. For running the geometry optimization, call

   ```
   xtb start.xyz --opt > opt.out
   ```

   The optimized geometry is written to the file `xtbopt.xyz`.

3. Verify that the sequence of atoms is still the same in every pair of reactant and product structure.

4. Perform a reaction path search with the double-ended Growing String Method (GSM) at the GFN2-xTB level:

   (a) Create a directory for each reaction.

   (b) Have your optimized reactant and product structure sorted and available in xyz format (*e.g.* starting structure `start.xyz`, ending structure `end.xyz`).

   (c) Create a directory `scratch` and store the starting and ending structures in the file `initial0000.xyz`:

   ```
   cat start.xyz end.xyz >> scratch/initial0000.xyz
   ```

   (d) Place a file named `ograd` in the respective directory with the following content:

   ```
   #!/bin/bash
   ofile=orcain$1.in
   ofileout=orcain$1.out
   molfile=structure$1
   ```

(continues on next page)

(continued from previous page)

```
ncpu=$2
basename="${ofile%.*}"
########## XTB/TM settings: #################
cd scratch
wc -l < $molfile > $ofile.xyz
echo "Dummy for XTB/TM calculation" >> $ofile.xyz
cat $molfile >> $ofile.xyz

xtb $ofile.xyz -grad  > $ofile.xtbout

tm2orca.py $basename
rm xtbrestart
cd ..
```

The `ograd` file has to be made executable:

```
chmod u+x ograd
```

e) Place a file named `inpfileq` in the respective directory with the following content:

```
# FSM/GSM/SSM inpfileq

------------- QCHEM Scratch Info ------------------------
$QCSCRATCH/    # path for scratch dir. end with "/"
GSM_go1q       # name of run
-------------------------------------------------------

------------ String Info --------------------------------
SM_TYPE                 GSM    # SSM, FSM or GSM
RESTART                 0      # read restart.xyz
MAX_OPT_ITERS           160     # maximum iterations
STEP_OPT_ITERS          30     # for FSM/SSM
CONV_TOL                  0.0005 # perp grad
ADD_NODE_TOL              0.1    # for GSM
SCALING                                1.0    # for opt steps
SSM_DQMAX               0.8    # add step size
GROWTH_DIRECTION        0      # normal/react/prod: 0/1/2
INT_THRESH              2.0    # intermediate detection
MIN_SPACING             5.0    # node spacing SSM
BOND_FRAGMENTS          1      # make IC's for fragments
INITIAL_OPT             0      # opt steps first node
FINAL_OPT               150    # opt steps last SSM node
PRODUCT_LIMIT           100.0  # kcal/mol
TS_FINAL_TYPE           0      # any/delta bond: 0/1
NNODES                                 15      # including endpoints
-------------------------------------------------------
```

For the Claisen rearrangement, you can change the `TS_FINAL_TYPE` option from 0 to 1 to force GSM to break a bond upon the transition state search. The number of nodes (keyword `NNODES`) can be increased for a more refined search, which (of course) leads to longer computing time. In most cases, 15 nodes should be sufficient to find a good guess for the transition state.

(f) Start the transition state search by typing:

```
gsm.orca
```

(g) After the calculation, you can find the reaction path in the file `stringfile.xyz0000`, and the transition

state in `scratch/tsq0000.xyz`.

5. Prepare relative energy diagrams for both reactions (relative energy vs. reaction coordinate), depict the molecular structures of both transition states and highlight the most important bond distances.
6. Calculate the activation energy for each reaction.
7. Perform single-point calculations on the starting structure and transtion state structure with PW6B95-D3/def2-TZVP using TURBOMOLE. Compare the DFT activation energy with the GFN2-xTB energy and discuss the differences.
8. How would you proceed further to gain more reliable numbers?
9. How feasible is this approach? Where do you see its limits in applicability and usefulness?

7.7 Noncovalent Interactions

7.7.1 Noble Gas · · · Methane

Exercise 7.1

Calculate potential energy curves of the "weak" interactions between the noble gases Ar or Kr and methane.

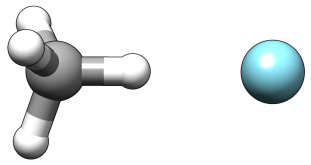


Fig. 2: Geometry of the $CH_4 \cdots Ar$ complex.

Approach

1. Calculate the potential energy curve at the BLYP-D3/def2-QZVP level for $Ar \cdots HCH_3$ by performing a geometry optimization with a fixed Ar and H atom. Do this for $R_{(Ar\text{-}H)}$ = 4.5 - 15.0 bohr with a stepsize of 0.25 bohr.

   Use the `run-scan.sh` script from exercise 5.1 and adopt it to this task. Substitute the template molecular geometry by the following one:

```
$coord
   0.00000000000000      0.00000000000000      DIST                  ar f
   0.00000000000000      0.00000000000000      0.00000000000000      h f
   0.00000000000000      0.00000000000000     -2.06945348098289      c
   0.97576020317533      1.69006623336522     -2.75977586481614      h
   0.97576020317533     -1.69006623336522     -2.75977586481614      h
  -1.95152040635065      0.00000000000000     -2.75977586481614      h
$end
```

   You also have to modify the directory names and the distances. Prepare a `control` file for the BLYP-D3/def2-QZVP calculations including the `$disp3 -bj` and `$symmetry c1` keywords.

2. Repeat the calculations for BLYP/def2-QZVP and MP2/def2-QZVP. For BLYP, just omit the `$disp3` block. For MP2, clear the `$dft` and `$disp3` blocks and insert proper settings under `$ricc2` and `$denconv`. In the latter case, don't forget to change the `jobex` command in the script to:

```
37  jobex -ri -level cc2 -c 50
```

   To get the final MP2 energy for each distance, also change the `sdg` command to:

```
40  e=$(grep "Total Energy" job.last | gawk '{print $4}')
```

3. Repeat the calculations for Kr $\cdots$ HCH_3 (substitute `ar` by `kr` in the template) with BLYP-D3/def2-QZVP, BLYP/def2-QZVP and MP2/def2-QZVP.
4. Plot the curves (**normalize to the dissociation limit**) and discuss your findings.

7.8 Spectroscopy

7.8.1 IR-Spectrum of 1,4-Benzoquinone

Exercise 8.1

Calculate the IR-spectrum of 1,4-benzoquinone using DFT and HF, and compare the results to the experimental spectrum given below.

Approach

1. Create a `coord` file for 1,4-benzoquinone.
2. Optimize the geometry with TURBOMOLE on the HF-D3/def2-SVP level of theory.
3. Calculate the normal modes with `aoforce`.
4. Call `tm2molden` and check the normal modes with `gmolden` the same way as in exercise 4.1.
5. Assign each dipole-allowed normal mode to an experimental one and calculate the scaling factor $f_{\text{scal}} = \nu_{\text{exp}}/\nu_{\text{calc}}$.
6. Repeat everything with TPSS-D3/def2-SVP and discuss your findings.

Hint: If you obtain imaginary frequencies, try to start the geometry optimization from a slightly distorted structure. Check if the imaginary frequencies vanish.

7.8.2 The Color of Indigo

Exercise 8.2

Calculate the color of indigo with three different methods: time-dependent Hartree-Fock (TD-HF) and time-dependent DFT (TD-DFT) with two different functionals (PBE and PBE0).

Approach

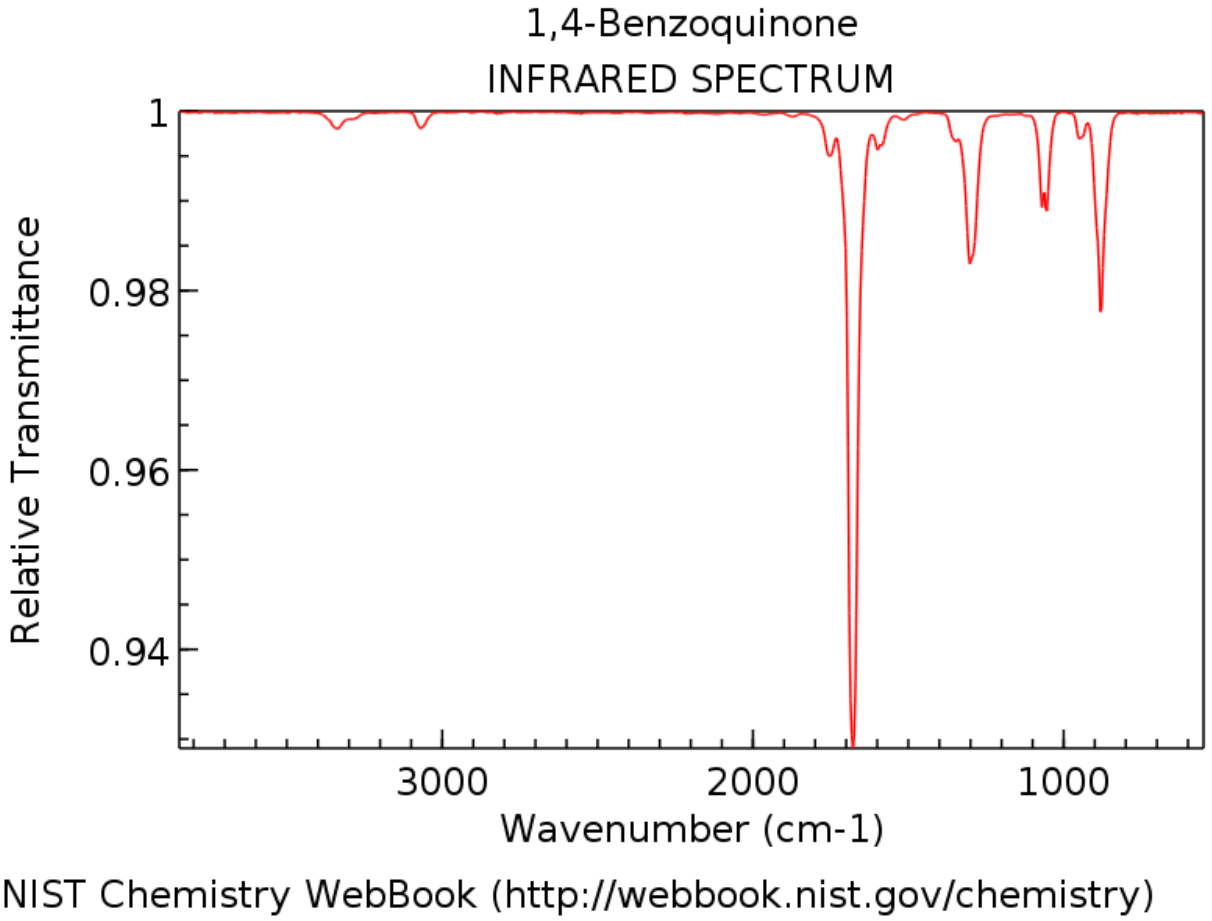


Fig. 3: IR spectrum of 1,4-benzoquinone in KBr.

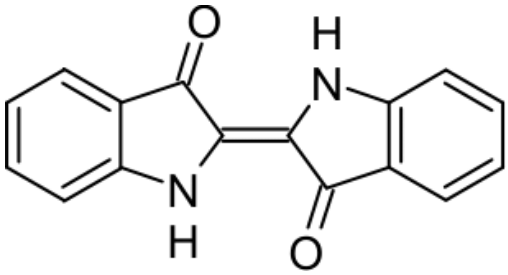


Fig. 4: Structure of indigo.

1. In this exercise, please use TURBOMOLE's interactive input generator `define` to create the `control` file. To do so, create the geometry of an indigo molecule (figure below) and place the `coord` file in its own directory.

   To start the input generator, navigate to the directory containing the `coord` file, type the following command and answer all questions that appear.

   ```
   define
   ```

   Prepare an input on the TPSS-D3/def2-SVP level and optimize the geometry. In the define dialogue, you can skip the first two questions, then choose the input geometry file via:

   ```
   a coord
   ```

   In this menu, you will find a header telling you the number of atoms of your molecule and its symmetry point group (Schoenflies symbol). If this is not yet correct (the molecule does not have C_1 symmetry), type `desy 1d-3` or some larger value to loosen the symmetry determination threshold until the correct point group is recognized. If the symmetry is still not recognized correctly, you can set the point group manually with the `sy` command. In this case make sure TURBOMOLE does not add atoms to your `coord` file.
   Leave the molecular geometry menu and then do not choose internal coordinates. In the atomic attribute definition menu, you can choose the same basis for all atoms by typing *e.g.*:

   ```
   b all def2-SVP
   ```

   Afterwards, choose an extended Hückel guess for the MOs with default parameters (don't worry about a small HOMO/LUMO gap). In the last general menu, go to the `dft` submenu and type *e.g.*:

   ```
   on
   func tpss
   ```

   Don't forget to also choose the `bj` option in the `dsp` submenu for the D3 disperion correction (activating the RI approximation in the `ri` submenu is also recommended). When you are finished, you will find a `control` file in your directory containing all keywords you already know and more. If everything is correct, you are now ready to run the geometry optimization.

   Hint: Try to navigate through the `define` dialogue a few times and explore it by yourself to get familiar with it. If you still can't figure out a certain aspect, feel free to ask your lab assistant.

   ---

2. Prepare a HF-D3/def2-SVP single-point calculation in the same way by using `define`, but do not turn on the `dft` option. Do not use the RI approximation here and run:

   ```
   dscf > dscf.out
   ```

   In order to do the subsequent TD-HF (or a TD-DFT) calculation, add the following two blocks so the `control` file:

   ```
   $scfinstab rpas
   $soes
     bu 1
   ```

   Then, run:

```
escf > escf.out
```

3. For the TD-DFT calculations repeat the above procedure with PBE-D3/def2-SVP and PBE0-D3/def2-SVP (now with RI approximation). Use proper settings during the `define` dialogue as well as in `control` and run:

```
ridft > ridft.out
escf > escf.out
```

4. Discuss the excitation energies for all three methods; which method would predict (at least approximately) the correct color for indigo? How do you explain the errors?

7.8.3 NMR Parameters

The computation of NMR parameters can be done with the ORCA program package. A simple input for the calculation of the NMR chemical shielding of CH_3NH_2 with the PBE functional and a pcSseg-2 basis set is presented below. The pcSseg-*n* basis sets are special segmented contracted basis sets otimized for the calculation of NMR shieldings. (For more information see: Jensen, F., *J. Chem. Theory Comput.* **2015**, *11*, 132 - 138.)

```
!PBE RI pcSseg-2 def2/J NMR

*xyzfile 0 1 input.xyz
```

At the end of the ORCA output, a summary of the calculated NMR absolute chemical shieldings can be found.

Exemplary output for CH_3NH_2:

```
--------------------------
CHEMICAL SHIELDING SUMMARY (ppm)
--------------------------


  Nucleus  Element    Isotropic     Anisotropy
  -------  -------  ------------   ------------
      0       C          152.992         46.821
      1       H           29.025          8.165
      2       H           28.567         10.286
      3       N          242.339         43.497
      4       H           29.033          8.176
      5       H           31.095         13.580
      6       H           31.120         13.593
```

Exercise 8.3

Calculate the ^{13}C-NMR chemical shifts δ for a number of organic compounds and compare the results to experimental data. In addition, investigate the correlation of the π-electron density with δ.

Approach

1. Optimize the geometries of the compounds **A – G** and the reference molecule $Si(CH_3)_4$ (TMS) on the PBEh-3c level of theory (how to use PBEh-3c is explained in the ORCA manual).
2. Calculate the ^{13}C-NMR chemical shieldings and shifts δ for compounds **A – G** with TMS as reference at PBE/pcSseg-2 level of theory (use this level in all the following calculations if not stated otherwise). Given the isotropic NMR shielding constants σ of the compound (c) and the reference (ref.), the chemical shift $\delta_{c,ref.}$ is

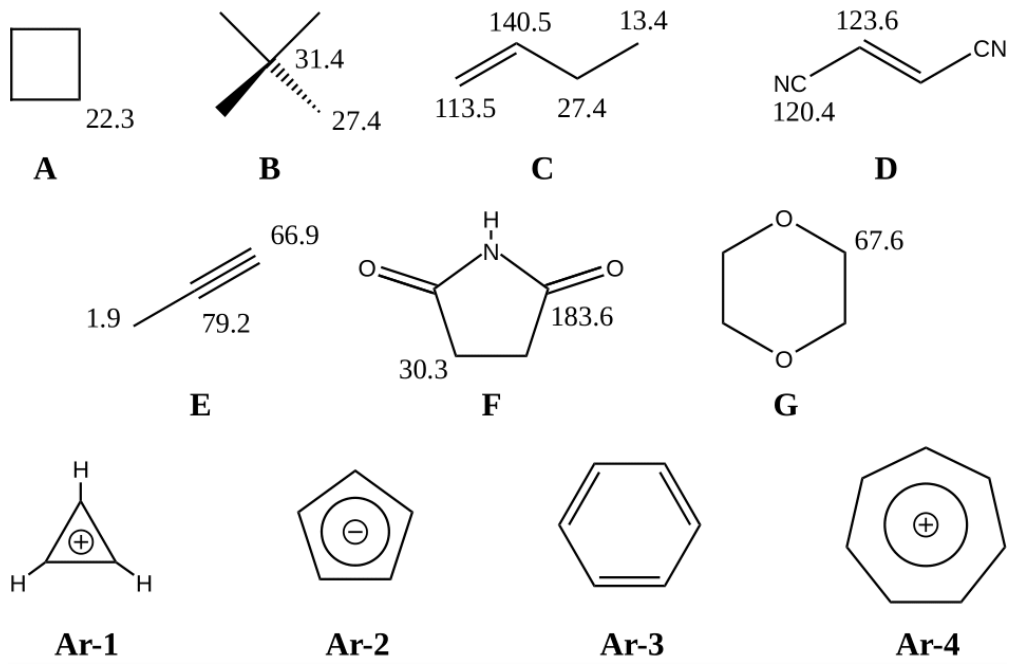


Fig. 5: Lewis structures of seven organic compounds **A – G** with their experimentally obtained chemical shift as well as four aromatic compounds **Ar-1** - **Ar-4**.

defined as

$$\delta_{\text{c,ref.}} = \sigma_{\text{ref.}} - \sigma_c.$$

Hint: Computational time can be saved by calculating the shieldings only for carbon atoms instead of all atoms. How to do so is described in the ORCA manual.

---

3. Compare your results with the experimental values and calculate the mean deviation (MD) and the mean absolute deviation (MAD).

$$\text{MD} = \frac{1}{N}\sum_{i=1}^{N}(\delta_{\text{calc.}} - \delta_{\text{exp.}}) \qquad \text{MAD} = \frac{1}{N}\sum_{i=1}^{N}(|\delta_{\text{calc.}} - \delta_{\text{exp.}}|)$$

4. Repeat the calculations for the four aromatic compounds **Ar-1 – Ar-4** and plot the calculated chemical shift against the formal π-electron density ($\rho_\pi = n_{el^\pi}/n_{at^\pi}$). Discuss your results.

5. The experimental ^{17}O- and ^{13}C-NMR chemical shifts of the carbonyl function in acetone are shifted by 75.5 and -18.9 ppm, respectively, if an acetone molecule is transferred from the gas phase to aqueous solution. Try to reproduce these values by considering solvation effects by the implicit solvent model CPCM. You can switch on the implicit solvent by adding the `CPCM(<solvent>)` keyword (example: `CPCM(toluene)`) to your ORCA input. For carbon, the reference is TMS, for oxygen it is water.

6. Try to estimate which effect is larger – the inclusion of an implicit solvation in the NMR calculation or the inclusion of the implicit solvent in the geometry optimization.

Hint: There are two options (gas/solution) for each calculation, the geometry optimization and the shielding calculation. Compare all possibilities.

---

7. Calculate the ^{1}H-NMR chemical shifts for **H** and **I** in the gas-phase at the PBE/pcSseg-2//PBEh-3c level of theory (again use TMS as the reference). Discuss your observations regarding the chemical shift of the methine proton in both compounds. Give a short explanation of your findings.

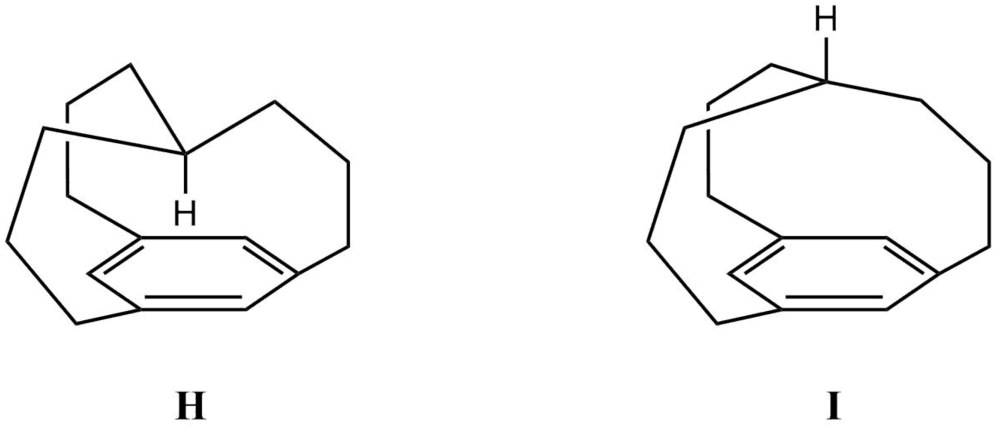


Fig. 6: Lewis structures of *in*- (**H**) and *out*- (**I**) [$3^{4,10}$][7]Metacyclophane.

Hint: The NMR chemical shielding calculations for **H** and **I** may be time consuming, consider to run them over night.