,

A Modern Update of Fortran Software DVR3D

Runyu Zhang

A Dissertation submitted
for the Degree
of Master of Science of University College London
Supervised by Professor: Tennyson, Jonathan

PHAS0077: Scientific Computing Individual Research Project

UNIVERSITY COLLEGE LONDON

Department of Physics and Astronmy

Submitted data: 17/09/2022

Word count: 7428

ABSTRACT

The program about the discrete variable representation for three dimensions (DVR3D) was written by Fortran 90 with some old version libraries in Fortran 72. The DVR3D program suite is developed for calculating the physical quantities such as energy levels, wavefunctions for triatomic molecules. Continuous updates and improvements from Jonathan Tennyson and other members allow the program to have subroutines that quickly solve situation-specific exceptions. In addition, some new features merge and replace the original program. However, the long-term development period and a large number of developers with different software styles cause code redundancy and low readability. On the one hand, it is difficult for new staff to quickly locate the focus code section even though the program structure is clear. On the other hand, if not a physics major, it is challenging to master the whole project and improve and optimize it from the programmer's direction. In this report and the whole scientific computing individual research project, the main focus is on project code optimization, syntax, and readability improvement from the role of modern programming.

Declaration

This report (include relevant software, source code and oral presentation materials) is entirely by mine, except for the statement reference already in the citation. This dissertation is worked by Runyu Zhang(ID: 20069941) for Scietific Computing Individual Research Project (Module code PHAS0077) by department Physics and Astronmy in University College London. The source code is based on the original DVR3D project (https://github.com/ExoMol/dvr3d). My own updated version (https://github.com/RunyuZhang-toma/DVR3D). After checking and testing, the updated version will merge to the original git repository.

Acknowledgment

Thanks for the Professor Tennyson Jonathan, Dr. Pezzella Marco.

Thanks for my parents caring for my dissertation.

TABLE OF CONTENTS

Page

CHAPTER

LIST OF TABLES

Table Page

LIST OF FIGURES

Chapter 1 Introduction

1.1 Background

The discrete variable representation for three dimensions (DVR3D) is a long-term program crossing over 20 years. The DVR3D program suite is developed to calculate the physical quantities such as energy levels and wavefunctions for triatomic molecules. The original paper has been cited more than 290 times, and the scientists who cited this paper are widely distributed in universities and research institutes worldwide[1]. The program contains some 90's Fortran 72 codes for supplying and is still being updated and some key features added.

However, many coders with different coding styles are involved in the project, which causes difficulty in handling the entire program. Most of them update and add features based on the former code structure, resulting in coupled and redundant code. At the same time, the advanced physics and mathematics content is another problem for recent students to understand and deploy the program correctly in a short time. The development of the Fortran 90 code standard and the user-friendly coding style, influenced by the current modern coding language, significantly improved the Fortran program design mode and readability compared with the old program. Some new features such as the **module** highly enrich Fortran language flexibility.

1.2 File Structure Illustration

The file for this new program suite is structured as follows: The main directory contains the dissertation Report (Latex files), issue sheet, and DVR3D manual.MD, i.e., the basic quick guidance, European standards for writing and documenting exchangeable Fortran 90 code [2], and README.MD. The folder DVR3D contains files as shown in **Table. B.1.**

Table 1.1: Descriptions about some files.

var.dvr	the compile parameters definition
README.md	the project code, script and explanation
makefile script	contains all used code files
obj folder	contains the modules .mod and .o files if no folder, plase run the run.bash once
ChangedNotApply	contains the advanced improvement and not tested codes.

There are four sample folders to handle different problems explained in **Table. B.2.** In these folders, they contain a **makefile** for compile directory and parameter setting, a running script called **run.sh**, and three task folders: jobs, results and source. Special notice, some updated files were not from GitHub. They are replaced or added to the DVR/source folder without compile and conflict errors with the original GitHub code.

Table 1.2: Descriptions about some files in the sampled folders

H3+Jacobi	calculation for H3+ in Jacobi coordinate giving a sample spectrum.
H3+Radau	calculation for H3+ in Radau coordinates inculding rotational excitation
HCN	mcalculation for HCN in Jacobi coordinates give a sample spectrum
Water	calculation for water in Radau coordinates giving a sample spectrum and doing a property calculation.

In the appendix, **Fig A.1** and **Fig A.2** show the sample **HCN** folder structure before and after the running, respectively.

The total file structure is shown in **Fig A.3**.

The dependency table is also shown in **Table. A.1**.

1.3 Problems

- *Unformatted code.* There are many different coding styles throughout the project source code with the different developers because of no unified formatting doc and long development term. The current code solved the problem quickly; however, it greatly hindered later developers from reading consistency and modifications, making the next deployment cycle even more time-consuming. Therefore, the code will format by using the format doc in the root user manual markdown file, which bases on the European Standards for Writing and Documenting Exchangeable Fortran 90 Code [2].

- *Outdated coding standards.* Fortran 90 continues to evolve itself and consider the trends in computer development and other computer languages, such as Python and C++. The update of relational operators could highly improve the code readability and continuity. However, the logical operators are still following the old standard, and the Fortran logical operators will replace when the new standard updates release.

- *Code redundancy.* Due to the old-fashioned way of variable declaration and assignment, the subroutine and function need more space to duplicate the constants and global variables in every section using the keywords **common** and **namelist**. It discourages later developers from updating the overall code because the transformation cost is much higher than simply updating new features. In this declaration strategy, it accounts for over 10% of the actual code section; for example, in the main file dvr3drjz.f90 with 4300 lines in total with comments, the common sections occupy over 322 lines. In the subroutines, it is necessary to declare variables multiple times conducive to programmer thinking connection.

These declarations not only influence the development speed that every functions and subroutine need to declare once at the beginning but also highly reduce the code readability and consistency. Even more critical, if users are involved in adding and deleting variables, all duplicate declarations and assignments must be checked, which is not easy and is not suggested in the current software engineering principles. This coding strategy violates one of the fundamental principles of software engineering, modifiability. The purpose of a design choice may be to structure a system so a change's effects can predict with assurance. [3] However, in this case, the declaration strategy highly relies on the stability of the programmers. This dangerous strategy easily leads to omissions and missing problems that could cause compile exceptions and errors.

Incomplete add and delete operations can lead to problems not detected at the compilation stage, or even more significant computational errors. According to Westley Weimer, detecting and preventing run-time errors is much more difficult than compile-stage errors [4]. Here, runtime errors are highly random and unpredictable, and it is possible to run safely for a long time without error, leading to logical errors in the code and incorrect user reliance on the incorrectly calculated results of the code.

- *Outdated* **I/O** *performance.* The code files in this project require reading a large amount of data and exporting the calculation results. As the size of the calculation file grows, job files and data could reach 1-2 GB, which may run out of the memory of the old machine or compute slowly. The project extensively uses the keyword **format** with an old file manipulation method to read and write files. The more urgent need for this project is to improve **I/O** performance. Also, take the main file

dvr3drjz.f90 code with 4270 lines as an example, the read/write operations alone account for 500 lines, more than 10%, so updating the read/write is also crucial for the whole project. To reduce the complexity of the code to some extent, this project will use a more accessible way to read and write files.

- *Makefile error.* In this program, the **makefile** file handles the compile connection and dependency and the **run.sh** is responsible for the code running. While in the command **make clean** and **make dvr.out** could occur the ignored errors. But this will not influence the compilation and running, sometimes will be ignored.

1.4 Solutions

- *New features.* Some new features will be applied in this project such as deleting the keyword **format** for the reading and writing method, replacing the relation operator for example .NE., .EQ. and .GE. The code will be formatted according to the rules we discussed below and inspired by the Fortran 90 style [2]. These methods will greatly increase the readability and be user-friendly to the programmers, further narrowing the grammatical difference between other modern coding languages.

- *Guidance and environment.* The paper briefly explains the guide for version control and software environment deployment. The hardware section will introduce the minimal requirements and development environments. The software environment part will introduce a simple guide to lead the new developers to set up the environment and run the code. Some official documents contain the advanced set-up. In the meantime, one piece of the user manual markdown file is place in the root directory.

- *Constant.f90 documents.* A Constants.f90 document include all the constants grouped by the different module names (from the common group) and contains all the detailed comments. It could replace the pdf version developed before to make it user-friendly for coding.

- *Modulization.* It mainly focuses on the old version constants declaration in the **common group**. The new assignment method merges into modules, and some out-of-data commons in the block data will be deleted and migrated.

- *Read/Write update.* The current file process method with the **format** keyword largely influences the readability and code consistency. The new strategies will be discussed below.

- *Testing.* Finally, the code will be tested on three different platforms: the Ubuntu Linux virtual machine based on the Intel brand chipsets, the real windows platform and UCL GNU/Linux system. The three types of systems could have abundant tests for performance and program reliability. However, the lack of older machine test makes this test impossible to be compatible with older models and configurations.

Chapter 2 Development Platform

2.1 Hardware

In this section, we will discuss the hardware firstly, including the hardware used in this project, the specific equipment used in the later test section, and the configuration requirements summarized in my development process.

To simulate the best general environment for this project, and personal equipment limitations, Intel's server-level CPU is used for this project, which has better support compared to other brands of CPU and traditional consumer-level CPU. It has better support for the old instruction sets, old code and VMWARE software. Besides, Intel Inc offers an optimized Fortran compiler : **Intel Fortran**.

For srorage, 8 GB RAM and 50GB disk space are recommended since less than 6GB RAM or 40 GB disk space may cause the failure of Intel® Math Kernel Library instalments and virtual machine lag in some cases. Once filed the instalments of the kernel library, the modification of disk and RAM does not work and the fast way is to reinstall the whole virtual machine.So one recommends allocating at least 6G of RAM and 50G of hard disk space when creating the virtual machine, though at a later stage it does not really consume more than 40G of disk space. Overall, the minimal configuration for this guidance in a virtual machine is 4 cores, 6GB virtual

RAM (not physics memory) and 50 GB virtual disk space.

The test machine with real windows platform configuration: Intel NUC9 platform NUC9VXQNX 8 cores Xeon E-228M processor, 64GB DDR4 RAM, 1T SSD, windows10 with bash. The activity monitor could check the specific configuration or use the manager of system. In the UCL GNU/Linux system, the parameters are Intel Xeon Silver 4215R CPU, 128GB Memory, 550G disk for all users. The command lines to check the Linux configuration are as follow: "**lscpu**" for a CPU, and "**/proc/meminfo**", "**free**" for RAM, and "**df -h**", "**lsblk**" for checking the disk.

2.2 Software

The software environment will be introduced, and there will be simple guidance and code sample to lead the new developers to set the environment and run the code.

- *Windows 11 up-to-date.* This is the physics machine environment, and if you want to use it as the long-term physics machine system, the windows server [5] or windows enterprise lstc [6]) is the better choice for stability. Windows 10 and Windows 11 could get stuttering and non-recurring bugs after a long period of uninterrupted operation, and it needs to be solved by restarting the virtual or physical machine.

- *VMware Workstation.* It allows users to set up virtual machines (VMs) on a single computer and use them simultaneously with the host operating system. This software is free for non-commercial use and cross-platform [7], which means users can directly share or copy the virtual machine as a file on different computers and platforms. It is easy to migrate your virtual machine from your own computer to anywhere and quickly batch production of deployed virtual machines. The shortage is that it only allows one virtual machine at the same time; if you want to open more on one single physics machine, you should upgrade to the commercial version.

- *Ubuntu Linux LTS 20.04 x64 [8].* This LTS system is a free and long-term maintenance version with the most stability and clean environment. Besides, it supports pure command line mode and an easy-to-use graphical user interface. As for the use of virtual machines system, your own preferences should prevail, but for first-time developers, GUI and initial command line software are very user-friendly. It could suggest similar and correct commands and give more relevant information.

- *GNU make [9].* GNU make is a well-known project build and management tool for the Linux environment, allowing us to compile, link, and execute with a single command. Gnu make's scripts do the work for us automatically.

- *GCC/G++ compiler [10].* The make and will help us control the clean roles, include the folder relative path and record the dependency information. In the meantime, the running bash script relies on this software. The GCC/G++ compiler is the required environment for the Ifortran compiler and Intel math kernel kit tools.

- *Intel fortran [11].* Intel Fortran is the specified compilation environment for this project, and it works best with intel chipsets. It is integrated into intel's one API platform currently, making it easier to install and use [12]. In the appendix **??**, there will be a specific GUI installation and installation guide. You could use the commands line to add it to the path and start when you open your terminal if you select the default install path. For special requirements and settings, choose the user setting and installment. It's better to check the Inten Fortran hand manual for help [11, 13]. The **bashrc** setting is attaching in the **Appendix ??**.

- *Intel math kernel Library(standalone version) [13, 14].* This is the support library for the Intel Fortran and this project. For new developers, if one wants to check the environment and code availability, use the following commands to test. This is an example of task **HCN.** The complete bash command is attached in **Appendix ??**.

Chapter 3 Methodology

3.1 Relation Operators Update

The new relation operators are introduced in Fortran90, and the update will focus on all f90 files in the source folder. The replacement chart is shown in **Table. B.3**. On the one hand, the new standard is easier to operate and will be more operational and convenient in operating an extensive range of codes. On the other hand, it is more in line with current programmers' reading habits and logic. Moreover, when we use the new standard operator, it significantly improves long logic expression compared with the older version. In special cases, the missing format blanks could mislead the coder to the wrong code logic. Although capitalization makes it somewhat less difficult to read and emphasizes the logical sequence, it is still tough to read code with many consecutive conditions plus if-else structures. An example of one line code with two different operators is shown below.

It should be noted that the comment symbol for Fortran is **!**, not / . If we use **!=** to represent a non-equality operator, it will comment out the rest of the code in the line. If the editor is smart enough to check for errors, then the editor may be able to warn of the error. Otherwise, incorrect comment notation may result in a compilation error. Even if it compiles correctly, it may cause the code to run with an error.

Also, in the current version of Fortran90, the logical operators have not been updated as shown in **Appendix. A.2**. So the logical operations are left unchanged but marked in the comments so they can be quickly replaced later using lookups and other methods.

Table 3.1: Table about the replacement

New	Old	Example	meanings	times
==	.eq.	(A == B) is not true.	Check whether the values of the two operands are equal or not, if yes, the condition becomes true.	1570
/=	.ne.	(A ! = B) is true.	Check whether the values of the two operands are equal or not, if not equal, then the condition becomes true.	340
>	.gt.	(A >B) is not true.	Check if the value of the left operand is greater than the right one, if so then the condition is true.	640
<	.lt.	(A <B) is true .	Check if the value of the left operand is less than the right one, if so then the condition is true.	150
$\geq$	.geq.	(A $\geq$ B) is not true .	if the value of the left operand is greater than or equal to the right operand, the condition becomes true.	300
$\leq$	.leq.	(A $\leq$ B) is true .	if the value of the left operand is less than or equal to the right operand, the condition becomes true.	320

3.2 Declaration Standard

The real type is a very complex declaration problem, except for arrays declared separately for special types such as real*4 in this project. In this project, all real (kind=dp) types are unified as double precision::. In addition, we have standardized the declaration of arrays.

Listing 3.1: Example code

```
module example
    save
    integer :: n, int_1, int_2(n)
    logical :: istrue, isfalse(n)
    double precision :: real_1, real_2(n)
    double precision, dimension(:) :: real_3
    double precision, dimension(n, n) :: real_4
end module example
```

3.3 Constants Documents

In the original document, many constants and comments were distributed in many files. Most constants appear only when they are first used or declared at the large scope. To understand these constant variables, the developers or the users need to search for the variables and comment them in the entire documentation. This situation is therefore very unfriendly to developers or users.

To solve the above problem, a constant.f90 file has been prepared to contain all constant variables. Of course, there is also the problem of so-called cross-declarations, where the same constant is declared in different modules. I have added them to all appearing modules and noted them.

This **module.f90** file is developer-friendly and convenient, and the constant variables in the documentation can be quickly located. However, it is worth noting that the declared module cannot be directly copied to existing files for use. If usable, change the module name relative to the files; otherwise, the compiled .mod file may crash with another file compiled .mod file. Also, the constant name in different file modules may have different value types, so please check the constant meaning within the process file. For example, the same common group in different files may have different numbers and different names of constants. Here again, the cross-declaration is explained. In this situation, the same constant appears in different common groups, and of course, the same constant appears multiple times in the same common group within different files.

After switching to a module, a compilation error will occur if the cross-declared constant variable is introduced (used) simultaneously. However, it is not usually possible to change the entire module documentation because of an error in a single variable. The non-simultaneous use of crossed variables can result in undefined variables, which can also cause compilation errors as shown in the table above. The details of how to solve this problem are described in the next section on constant modulization. In this case, the module size and dim from the old version **common/size** and **common /dim** cannot appear in the same file. It may cause the reference error at the compile stage. Currently, the module works well and developers need to check the module declaration in the different file to make sure the delete and add process safe.

Indent specification also needs to be documented. It is widely known that indent is critical to code formatting and readability, especially for the if-else nesting structure. The program contains different indent strategies such as no indent, random indent, and jump indent. So the unified rules for Fortran language are defined as follows, a sample code is attached in **Appendix ??.**

- The default indent value is 4;
- The new line after the line break should add one more indent compared to the original line;
- The **goto, format, and do** jump to the new line, and the line number should indent as well and close to the normal code.
- The same role as previous one if there are **if-else** nest structures.
- The keyword: **program, subroutine, function, end** should have no indent.

Inspired by the Fortran 90 ES document [2], for the complete new files, the comment should quickly reflect the basic properties of the file and explain the usage of the file. For modified file blocks, the basic declarations and usage are placed at the beginning of the block. Deleting unnecessary comments and reformatting the disorder comments are fundamental tasks for the existing comments. The existing code was not in a uniform format, so comment split lines are used to split the different subroutines/functions/modules/programs and standardize the split line. The full comments appear in the module at the beginning of each f90 file.

3.4 Constants Modulization

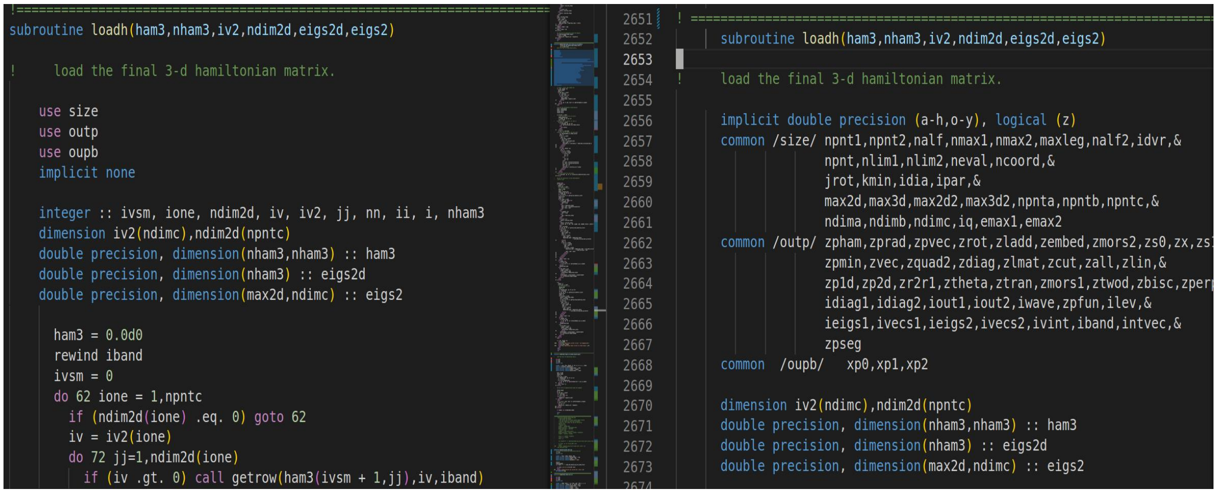
In the original Fortran code, the old pattern of **implicit+common+data** was used to define, declare, and assign values to constant type variables. This approach results in redundant code and low maintenance performance. The software principle indicates that the data structure should be designed before the start of programming according to the project requirements and specifications. Redesign and migration costs may over the redevelopment. The cost of redesigning migrating software can far exceed the cost of doing a new project from scratch, especially when new features are constantly being developed. In the later development stage just like the current project, the more features add-in, the more difficulties to modify. The developer would rather use the outdated data structure.

• According to the current temporary data declaration method: **implicit double precision(a-h, o-y), logical (z)**, all the data should be named with the corresponding first letter. The variable names of the entire file should be standardized by modifying the first letter of the variable name.

• For all the codes in the program file, the **common** sections that exist in function and subroutine, respectively, need to be standardized. This outdated way could take up 10% of the space of a single file.

• If the number of parameters and variable types in the same group are different between different files in the definition domain, then there is a problem with cross-declaration.

- When adding or deleting changes to a constant, it is necessary to synchronize all the changes to the common involved, even to the extent that different files and folders are involved. Developers who are not proficient with the entire project can cause many compilation errors. Even worse it could lead to the problems detected run time error. A snapshot of the subroutine is shown in **Fig. 3.1**.



```
!==========================================================================
subroutine loadh(ham3,nham3,iv2,ndim2d,eigs2d,eigs2)

!     load the final 3-d hamiltonian matrix.

    use size
    use outp
    use oupb
    implicit none

    integer :: ivsm, ione, ndim2d, iv, iv2, jj, nn, ii, i, nham3
    dimension iv2(ndimc),ndim2d(npntc)
    double precision, dimension(nham3,nham3) :: ham3
    double precision, dimension(nham3) :: eigs2d
    double precision, dimension(max2d,ndimc) :: eigs2

      ham3 = 0.0d0
      rewind iband
      ivsm = 0
      do 62 ione = 1,npntc
        if (ndim2d(ione) .eq. 0) goto 62
        iv = iv2(ione)
        do 72 jj=1,ndim2d(ione)
          if (iv .gt. 0) call getrow(ham3(ivsm + 1,jj),iv,iband)
```

```
! ========================================================================
    subroutine loadh(ham3,nham3,iv2,ndim2d,eigs2d,eigs2)

!     load the final 3-d hamiltonian matrix.

    implicit double precision (a-h,o-y), logical (z)
    common /size/ npnt1,npnt2,nalf,nmax1,nmax2,maxleg,nalf2,idvr,&
                  npnt,nlim1,nlim2,neval,ncoord,&
                  jrot,kmin,idia,ipar,&
                  max2d,max3d,max2d2,max3d2,npnta,npntb,npntc,&
                  ndima,ndimb,ndimc,iq,emax1,emax2
    common /outp/ zpham,zprad,zpvec,zrot,zladd,zembed,zmors2,zs0,zx,zs
                  zpmin,zvec,zquad2,zdiag,zlmat,zcut,zall,zlin,&
                  zp1d,zp2d,zr2r1,ztheta,ztran,zmors1,ztwod,zbisc,zper
                  idiag1,idiag2,iout1,iout2,iwave,zpfun,ilev,&
                  ieigs1,ivecs1,ieigs2,ivecs2,ivint,iband,intvec,&
                  zpseg
    common  /oupb/   xp0,xp1,xp2

    dimension iv2(ndimc),ndim2d(npntc)
    double precision, dimension(nham3,nham3) :: ham3
    double precision, dimension(nham3) :: eigs2d
    double precision, dimension(max2d,ndimc) :: eigs2
```

Figure 3.1: The subroutine snapshot before and after modulization.

There are four strategies in the modulization process, with different solutions and the first three unacceptable performance problems. So they are abandoned, and the fourth approach is adopted with feature 5.

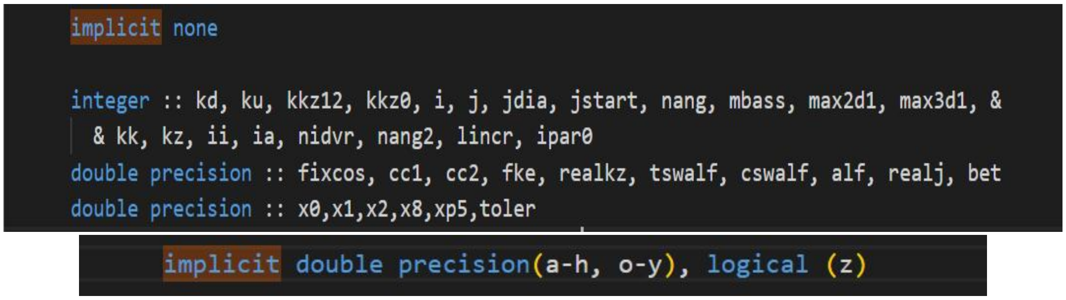
- Initially, the first strategy was to use a single file: the current **constants.f90** description file controls all constants by **include** this file in all source files. Nevertheless, such an approach will cause a series of errors as follows. 1. Cross-declaration compilation problems. 2. The same joint group has different variables. 3. Variables with the same name are declared differently in different files. At the development stage, it is impossible to determine which files will be used in the same application. In addition, groups with the same name have a high degree of similarity, although they may differ. It would be unacceptable to redeclare an utterly new module because of some individual variables. In this case, whether the current development or later improvement, this way is very complex and inefficient, while the requirements for developers are also very high.

- The second option is to remove the cross-declared constants and declare them in the temp module. The rest of the regular constants store in the current module file. This approach causes a lot of **use** commands in specific functions and subroutines. By doing so, the current code does not improve development efficiency but also makes it more difficult for developers to read and manipulate the constants. Suppose all of these unique variables are not explicitly commented in functions and subroutines but placed in the **use** module. In that case, it is difficult for developers to locate them quickly. Managing many temporary variables in a module outside the program is also very inefficient. In summary, this solution is abandoned.

- Temporary variables declare using the old **implicit double precision (a-h, o-y), logical (z)** so that they can be used directly in functions and subroutines; however, this approach violates the original purpose of this task. One of the key ideas for modulization is to change the implicit method if we want to be able to declare, define, and assign variables freely, so we have abandoned this approach.

- By studying the **dipolez.f90** file structure, we have come up with a final solution, which is to place each file's module declaration before the file's beginning program. This approach can avoid cross-declarations to a large extent so that the same group between different files can have its perfect module. Moreover, the number of module usages in subroutines and functions reduces apparently. Developers could assign the default value directly in the **module** and ensure that the initial default value of each file is unique to the corresponding functions and subroutines and is not affected by other files. Removing the **block data** section improves the code readability. This strategy also facilitates adding, deleting, and modifying processing and does not increase the cost of searching, which benefits developers and users. In addition, considering that each file's module name is the same, if we compile and run the file directly, the same name will cause a reference error. The solution is to add a particular file tag to the same module name in each file. Based on the above analysis and the methods adopted, the file structure becomes module define + main program + subroutines/functions mode.

• Final solution. Based on solution 4, the **implicit double precision(a-h, o-y), logical(z)** was introduced back in some extreme functions and subroutines, which contain plenty of temporary variables. Without affecting the naming rules of the module and the overall files, it is possible to reduce the amount of code by 50% just by declaring some extreme conditions. We use the command line as shown in **Fig. 3.2**.Newly developed features and files do not affect all existing files, they need to be developed in the same format, and subsequent development will remain unaffected.



```
implicit none

integer :: kd, ku, kkz12, kkz0, i, j, jdia, jstart, nang, mbass, max2d1, max3d1, &
  & kk, kz, ii, ia, nidvr, nang2, lincr, ipar0
double precision :: fixcos, cc1, cc2, fke, realkz, tswalf, cswalf, alf, realj, bet
double precision :: x0,x1,x2,x8,xp5,toler
```

```
implicit double precision(a-h, o-y), logical (z)
```

Figure 3.2: The two types of temporary variable declaration.

3.5 Code Format Standardization

Code quality is multi-faceted and covers, for instance, testability, maintainability, and readability [15]. Due to the age of the project and the large number of people involved in programming, the code has different formats. So the unification of the code format could significantly improve the code quality in the aspects mentioned above. The following are the specific update strategies.

- Tighten the code and remove unnecessary spaces and lines. The multiple blank lines to separate the code block could be replaced by the comment line.

- Use lowercase for all text. In Fortran syntax, two different forms of the same word, uppercase and lowercase, represent the same meaning, and variable names are not case-sensitive. Using all lowercase makes reading the code the same as reading ordinary text, in agreement with people's reading habits. In addition, we could use uppercase words and sentences in the comment section to notify the new developers.

- Declarations follow a fixed order. This order is the module, name list, implicit none, temporary variable declaration, and variable assignment to ensure that it makes the code look neat.

- Update the assignment method. The old method uses **block data** to store constant values. Here is the new version of the programming language Fortran 90. We drop using **common/data** and assignments and use modules instead.

- Updates of the module. When modules in different files have the same name but are actually declared and initialized differently, the module name needs to be changed to distinguish between different modules.

- Use tab before the expression of one-line **if** statements.
- Revision of **do** and **goto** statement indentation. In older versions, line flags were placed at the beginning of a line or randomly, which could cause reading alignment problems. Below we give specific examples

Listing 3.2: Fortran version

```
! From
         do 1 i=1, 3
1        continue
! To
         do 1 i=1, 3
         1 continue
```

where “1” stands for the line flag.

- Aligning conditional statements. In older versions **if**, **else**, **endif**, etc. were not aligned, which affected reading speed and made it easier to make mistakes. Here, these keywords are aligned, especially the several times nested keywords.

3.6 Read/Write Update

This project process the read and write stream frequently, so reading and writing methods are essential. It not only affects the efficiency of the code but also influences the coders reading consistency and handling of the source code. The earliest implementation of file operations combines the format sentence with the reading and writing method. Since the format still uses a syntax similar to **goto**, we need to keep jumping when looking for the format, which is very slow and affects the reading continuity and readability if there is no support from the advanced editor to remind us. Too many jump statements are nested in **if else**, **do statements**; in addition, the jump format statements are not always close to the reading and writing expression. So this case makes it even more challenging to read. Take the main file **dvr3drjz.f90**, for example, there are over 250 **read/write** statements, which means almost the same number of go statements; for a 4000-line file, it has taken up more than 10% of the space for text control. Moreover, the distance between the line express and format **go-to** number further increases the difficulty of reading. The solution is the same as point 8 of the code formatting specification. Jump numbers are indented with the code.

There is also a more appropriate form of the format in the code; that is, the format. Though it only needs to jump when querying the specific text content (because the variables follow the read and write command. The format position does not affect the variables themselves, but only the text format). Nevertheless, the problem arises when checking and adjusting the text formatting. Combining these drawbacks with the fortran90 specification update, we have a new way to avoid these centralized problems: the new way of writing below, which eliminates the format keyword. Furthermore, it provides a higher level of flexibility. Under this writing style, the format is integrated directly into the control command after the first solution for formatting.

Compared to the previous way, it can save twice the amount of code. Also, for reading operations, we can use the third line of code to increase the read operation's flexibility further. In this way, not only do we not need to consider the type and dimension of variables, but we can also reduce the code and read operation requirements for reading documents. The second line of code is susceptible to the document's formatting requirements, such as the number of spaces, which may cause the reading task to fail. Taking the third line of code can circumvent this problem.

3.7 Comment

I used the comment method mentioned above [2] for the file I completed alone. These comments quickly reflect the basic properties of the file and explain the usage of the file. For blocks that are modified on top of the original file, I write the basic declarations and usage at the beginning of the block, just like everyone else. The read/write statements are too fragmented to be commented on.

Listing 3.3: Constants.f90

```
!=========================================================
!Copyright(C). 2022, University College London
!File name: model.f90
!Author: Runyu Zhang & Jonathan Tennyson
!Version: 1.1
!Data: 7th/july/2022
```

```
!Description: this Fortran90 file contains the module
!template for the source folder common
!  constants divided by the common group
!Dependency: Folder source
!=========================================================
```

Listing 3.4: Module defintion

```
!========================================================
!By Runyu Zhang & Tennyson Jonathan 1/Sep/2022
!Contains size, outp, oupb, timing, split1, split2, mass
!Special notice:: The module name contains the file name,
!it means cannot directly paste to other
!files and use. There are some difference
!between the constant numbers, types and
!                 default value.
!=========================================================
```

For the existing comments, I have formatted them and removed any unnecessary or repeated comments. The existing code was not in a uniform format, so I used a comment split line to split the different **subroutines/functions/modules/programs** and standardized the split line. The full comments appear in the module at the beginning of each f90 file, in addition to the constants module.

Chapter 4 Analysis

4.1 Constants Modulization

• The final solution adopted can reduce the coupling between codes to a great extent and can completely disentangle the modules with different functions. It completely solves the cross-declaration reference error by separating the module into its own file. It significantly reduces the possibility of human uncertain factors. Reducing the mortal error aspect through code is an essential part of programming and software engineering.

• It completely solves the cross-declaration reference error by separating the module into its own file. It drastically reduces the need for programmers, the difficulty of development and the chance of future compilation and runtime errors. Adding new constants to an old module without deleting old and non-use parameters will result in low usage of module variables and bloat, as well as many invalid module constants consuming memory.

• For larger modules with higher reusability, the code size can be reduced by 50%. The overall code size can reach 12% due to the overall code format update and useless deletion based on the main function **dvr3drjz segmented.f90**, which has the largest code size. One sample figure is shown in **Fig. 3.1**.

- Every used **.f90** file will generate corresponding .mod files, instead of one **.f90** file or module, to perfectly grasp all the constants. If we process a global constant, we still need to process corresponding files. However, it is advantageous compared with all subroutines, functions, and programs. In addition, this strategy is the fastest way to develop new code to use the module feature.

- There are issues of concern in the final solution. Since modules divide into different source files, the final compilation produces many module files that are saved in the obj folder by default in the Makefile configuration. However, for most developers, .mod files do not require special attention. The most important thing is that untraceable runtime errors may occur during the modulize process due to differences in individual file editing and actual project compilation, which is probably the biggest concern for developers. It requires the developers to check every constant before compiling the full project

- Current strategy didn't convert all the implicit declarations in the subroutine, functions to none. In this case, the plenty of temporary variables function and subroutine need much space to declare. In turn, it makes the code bulkier.

All the current source code can be used as a template for the next development and new Fortran features can be used, and because of the significant reduction in code coupling, we can do advanced development work with less knowledge of the project. It's easy to write new files according to personal code habits, or modify and add features according to the modified file format and modules, without fully understanding the entire project. This allows projects to be developed and deployed quickly while keeping the project neat and tidy.

As for future work, there is another firstly, checking all the temporary variable names, deleting useless integer count variables, and need to circumvent the name of variables that have been used or may be used in future expansions. Secondly, update all implicit commands to implicit none, declare all temporary variables, and separate them from important ones. Finally, merge the same module across the different files and remove the initial value assignment inside the module. Return the assignment function to the files themselves and check if the required initial values are the same in that file.

As is known to all, development efficiency and code running efficiency always contradict each other. Every coin has two sides and can't do perfectly balanced. The final solution taken in this paper can speed up the development efficiency; the in-file module can isolate the difference between different files and reduce the interaction with other files during the development process. However, it will add many mod .mod files. On the contrary, the unified module approach makes it much more difficult to manage modules and requires frequent interaction with other files during development. But it generates only a minimal number of mod files and reduces the reference rate in the files.

As far as I'm concerned, the current approach adopted in this paper is a more favourable strategy for developing this software, since it will continue to be developed, requiring a lot of debugging and modification, and will continue to integrate new developers. However, for some small general-purpose modules similar to the module **timing** in this project, the unified module mode can be used completely. This block has been successfully implemented but placed in the Not apply folder.

4.2 Read/Write

In the new method, for the write method, if the argument or format is too long, the single format control string makes the line break symbol & out of work and treats it as a character. So the Fortran single line limitation restricts the format string length. Unfortunately, the symbol is the only line break space and cannot be used in the string.

So there are three faulty solutions for the long format statements.

- Use the original format and jump statement, which is complex but good for centralizing grouped write operations.

- Use the splitting for splicing, which is also a very unreasonable operation, but compared to the first one, it omits to jump and a lot of formatting.

- Use the character substitution **Aw** instead of a long string. It converts a long output string into an argument to the write method's second parameter. The long output string is a parameter following the complete write expression. In that case, the line break symbol is available outsides the format string. Special notice, if the newline operator is used to concatenate sentences of more than 2 lines, then not only at the end of the line but also at the beginning of each line, the symbol is still required for concatenation.

Furthermore, operations still need to be improved. In the future, read operations will be extracted from subroutines and functions in the file and operated in a separate module, which can even reduce the problem of too many temporary variables in the constants module. All subroutines and functions only exchange data with the read/write module and do not perform stream processing operations. This also centralizes the read/write operations to quickly locate bugs, improve text processing efficiency and reduce the probability of errors. Centralized processing of text files and good annotation can greatly improve the efficiency of programmers' operations.

4.3 Readability

In this section, the readability analyses will be based on the main file dvr3drjz.f90. The module section replaces the block data section, constant comments sections, and Common declarations. The total lines for module is 140. The three sections are 85, 335 and 36 lines respectively. In addition, there are nearly 200 lines deleted for read/write format. Only the modulization and text processing, the single file reduce by over 500 lines. If we use a single line to declare the uninitialized constants, it will dramatically reduce over 550 lines. In this aspect, the coding experience increases visibly.

4.4 Test

The code compile and running stages are tested on the three platforms discussed in chapter 3. By compiling the HCN sample, all platforms compile correctly. All of them appeared the makefile ignored errors which are not influenced in the testing stage.

The current corresponding file overall size for all dvr3drjz.f90 is 295KB(.f90 file) + 3.7MB (.o file + .mod file). The old version is 308 (.f90 file) KB+3.4MB (.o file + .mod file).The file sizes are recorded in the Linux system.

For the total running time(directly use the command: **bash run.sh**), the data are shown in the below take. The table clearly shows the old version program has a Marginal advantage which might be influenced by the new version files' need to compile more .mod files in the same platform. If we increase the test sample size, the influence of .mod file compilation time will become smaller. For the current computer system, it can be ignored.

On the contrary, there is a huge difference between different platforms. The virtual machine has the longest running time which could be because of the low efficiency of the virtual machine compared with the physics machines. As for the Windows platform, based on the current knowledge, the causes of this result are more complex. The analysis can only be done based on the existing equipment because of the inability to control the variables completely. First, the Windows platform is a physical platform with the highest CPU frequency and memory response speed, which will be why it has the least overall time and is substantially ahead of other platforms. Secondly, due to problems with the CPU scheduling of the Windows platform system, his frequency variation would be very noticeable, which could be the reason for his significant unstable compilation time. The UCL cluster has combined the physics machine and low-frequency factors and the running time is in the middle. It can conclude that there is still a big gap between virtual and actual machines.

Table 4.1: HCN sample test result.

	VM. Linux (seconds)	Windows 10 PC (seconds)	UCL. theory cluster (seconds)
Old version	33.924 34.234 34.025	25.214 26.376 23.796	28.316 27.983 28.502
New version	35.032 34.987 34.850	24.136 25.103 25.831	29.326 28.832 28.516

Chapter 5 Conclusion

This project's overall software development targets specified at the beginning stage are complete within three months of development. The project focuses on improving the readability and formatting of the code; it is essential to manipulate the overall code and confirm the formatting requirements for the Fortran language base on its feature and current project coding style.

Compared to the wide range of other popular languages, many coding plugins do not support the Fortran language, even the auto-alignment and layout features. So almost every line was edited in the entire project by hand. The change of the main file **dvr3drjz.f90** could even be over as many as ten times. The amount of code can be reduced by more than 10%. In addition, the current code can primarily reduce the difficulty of reading and understanding for fresh developers.

According to the test results, there is not much improvement in performance, which is in line with expectations. Fortran, as a numeric computation and scientific computing language, in the premise of using the traditional syntax, there is little possibility of speed improvement, especially in the entire project; scientific computing occupies most of the running time. If the core algorithm is not modified or nearly optimized, there will not be much improvement. The compilation time of modules can even be considered an experimental error. In contrast, the development performance improvement will be very significant compared to the runtime performance improvement. During the last development period, the new code strategy was efficient enough compared with the beginning of the project. Both the modification and deletion of global variables and the declaration and modification of temporary variables are user-friendly. At the same time, the files that apply the above strategies provide a high degree of ease of use and demonstration when used as templates for new files.

It is a massive challenge to modify and redesign a physics scientific computing software in software engineering. Handling the entire program requires higher code understanding, analysis ability and learning the new code language. Compared with the previous project, which could have its idea and design the program structure, understanding the whole project and considering the problem as the designer is essential. In the early stage, the aim was unclear and consumed plenty of time to continue changing the strategies and modifying the dvr3drjz.90 file. With the improvement of project understanding, the final strategy was determined. Besides, the unsuitable schedule makes the final report a late penalty. Due to the lack of proper version control and branches, multiple version rollbacks, and incorrect copying and loss of local files, the project was far behind schedule.

In the future, the project plan is to convert all support .f(Fortran 72 file) files into Fortran90 files. The current files are great templates for the new files in formatting and data structure. In addition, the relevant updates of new standards about Fortran 90 for the discussed section before is another exciting thing. This project will be continually focused on and be contributed to if possible.

REFERENCES

[1] Jonathan Tennyson, Maxim A. Kostin, Paolo Barletta, Gregory J. Harris, Oleg L. Polyansky, Jayesh Ramanlal, and Nikolai F. Zobov. Dvr3d: a program suite for the calculation of rotation–vibration spectra of triatomic molecules. *Computer Physics Communications*, 163(2):85–116, 2004.

[2] Andrews Phillip, Cats Gerard, Dent David, Gertz Michael, and Ricard Jean, Louis. European standards for writing and documenting exchangeable fortran 90 code, https://wiki.c2sm.ethz.ch/pub/cm/codingadvicescosmo2/europsd.pdf.

[3] J. Goodenough, D. Ross, and C. Irvine. Software engineering: Process, principles, and goals. *Computer*, 8(05):17–27, 1975.

[4] Westley Weimer and George Necula. Finding and preventing run-time error handling mistakes. *ACM SIGPLAN Notices*, 39, 08 2004.

[5] Windows server,
https://www.microsoft.com/en-us/evalcenter/evaluate-windows-server-2022.

[6] Windows enterprise,
https://www.docs.microsoft.com/en-us/deployoffice/ltsc2021/overview.

[7] Vmware, www.vmware.com/.

[8] Ubuntu linux 20.04 lts x64, https://releases.ubuntu.com/focal/.

[9] Gnu make, https://www.gnu.org/software/make/manual/make.html.

[10] Gcc, https://gcc.gnu.org/.

[11] Intel ifortran (standalone version)
https://www.intel.com/content/www/us/en/developer/tools/oneapi/fortran-compiler.htmlgs.cj9cs5.

[12] Intel fortran guide,
https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-fortran-compiler/top.html.

[13] Intel math kernal library (standalone version),
https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-download.html.

[14] Intel math kernal library document (standalone version),
https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-mkl-for-dpcpp/top.html.

[15] Haris Mumtaz, Shahid Latif, Fabian Beck, and Daniel Weiskopf. Exploranative code quality documents. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):1129–1139, 2020.

Appendix A

Listing A.1: bash version

```
source /opt/intel/oneapi/compiler/latest/env/vars.sh
# default location
source /home/toma/intel/oneapi/setvars.sh
# default location
\label{list1}
```

Listing A.2: bash version

```
make dvr.out # in the project location
sh run.sh # in the HCN file
ll -htr
tail -n20 result.HCN.
tail -n20 result.HCN.*
```

Listing A.3: Module defintion

```
!========================================================
!By Runyu Zhang & Tennyson Jonathan 1/Sep/2022
!Contains size, outp, oupb, timing, split1, split2, mass
!Special notice:: The module name contains the file name,
!it means cannot directly paste to other
!files and use. There are some difference
!between the constant numbers, types and
!                 default value.
!=========================================================
```

Listing A.4: Constants.f90

```
!=========================================================
!Copyright(C). 2022, University College London
!File name: model.f90
!Author: Runyu Zhang & Jonathan Tennyson
!Version: 1.1
!Data: 7th/july/2022
```

Table A.1: Descriptions about some files in the sampled folders

H3+Jacobi	dvr3drjz.f90 rotlev3.f90 spectra.f90 dipole3.f90
H3+Radau	dvr3drjz.f90 dipolez.f90 rotlev3z.f90
HCN	dvr3drjz.f90 rotlev3.f90 spectra.f90 dipole3.f90
Water	dvr3drjz.f90 rotlev3b.f90 xpect3.f90 dipole3.f90

Table A.2: Table about logical variables

New	Example	
.and.	(A .and. B) is false.	The logical and operator. If both operands are non-zero, the condition becomes true
.or.	(A .or. B) is true.	The logical OR operator. If either of the two operands is non-zero, the condition becomes true.
.not.	!(A.and. B) is not true	The logical non-operator, is used to invert the logical state of its operands.
.eqv.	(A .eqv. B) is true	Logical equivalence operator for checking the equivalence of two logical values.
.neqv.	(A .neqv. B) is not true	Logical non-equivalence operator for checking the equivalence of two logical values.

```
!Description: this Fortran90 file contains the module
!template for the source folder common
!  constants divided by the common group
!Dependency: Folder source
!==========================================================
```

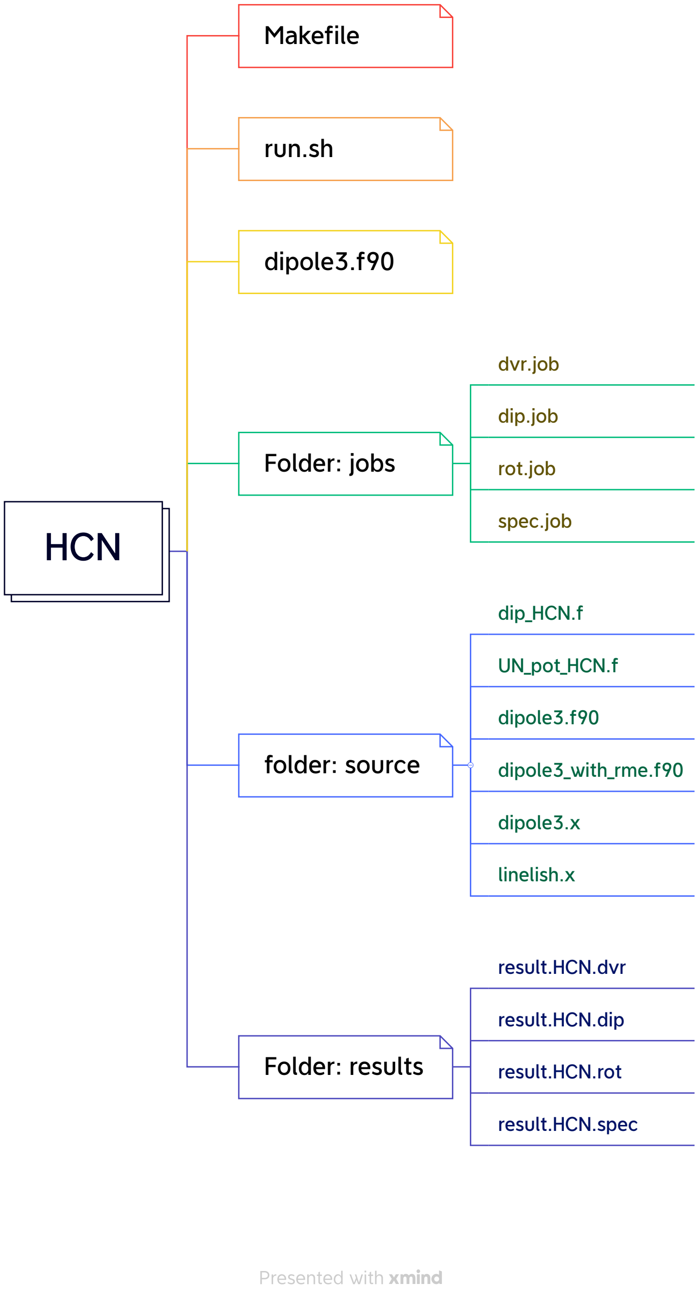


Figure A.1: HCN folder structure illustrating figure before running

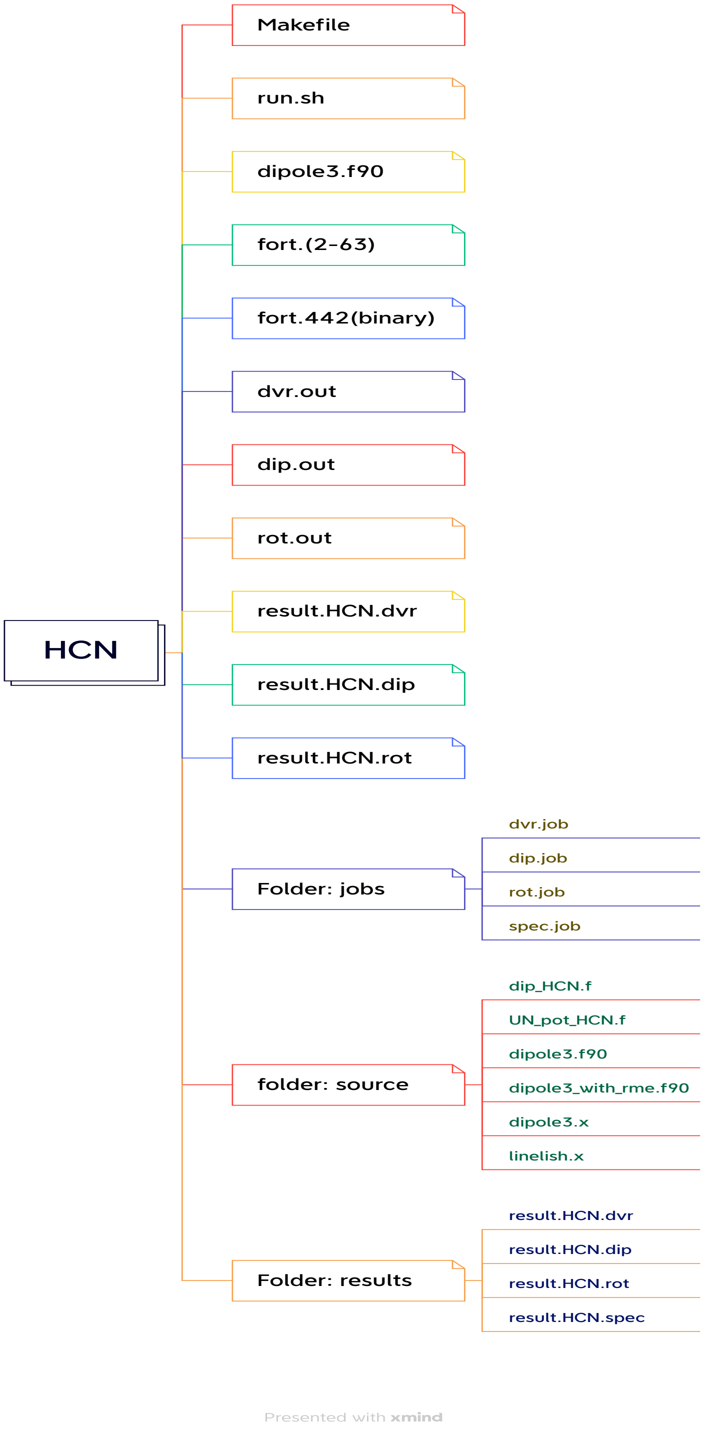


Figure A.2: HCN folder structure illustrating figure after running

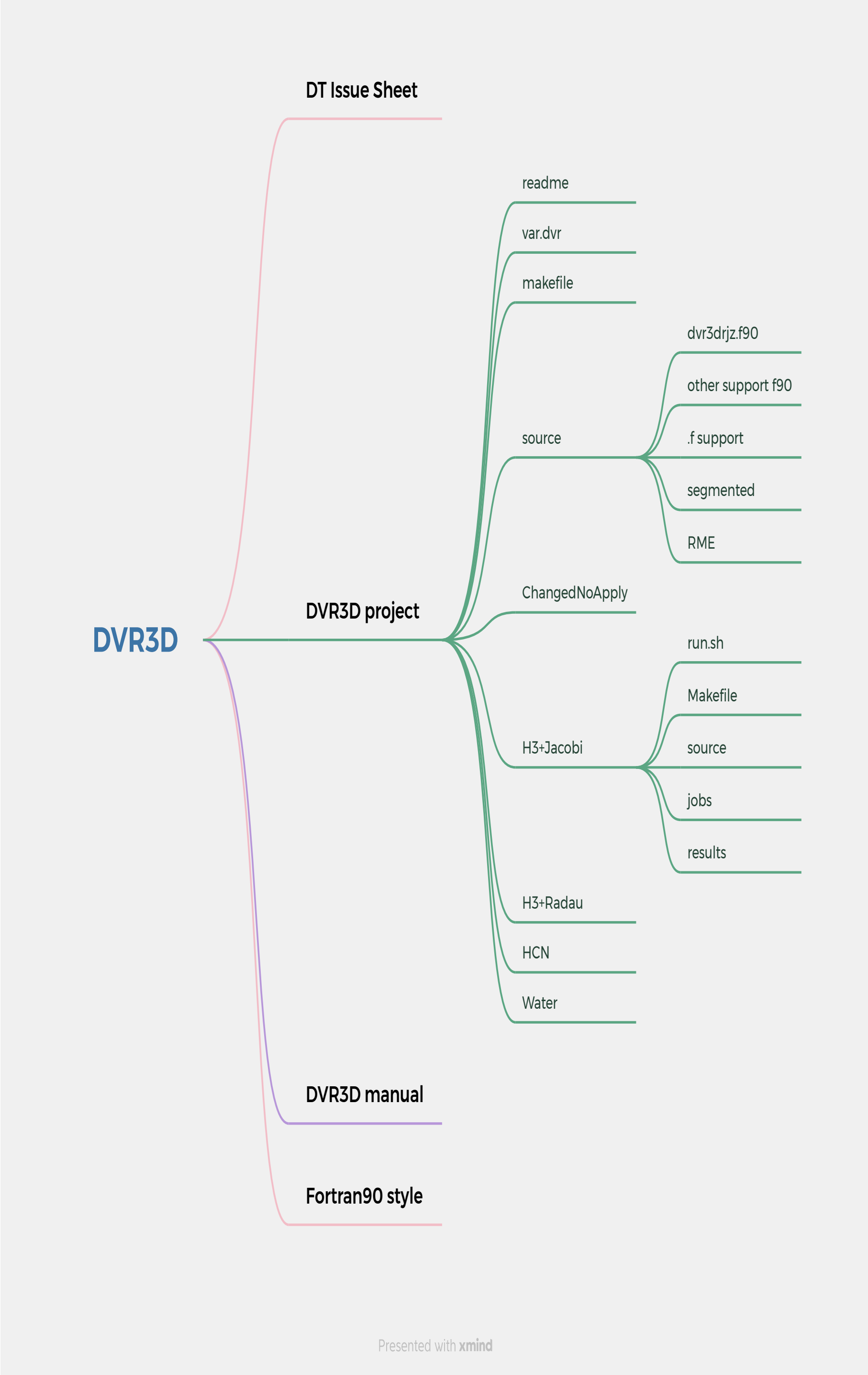


Figure A.3: The whole file structure of the program DVR3D

Appendix B

Table B.1: Descriptions about some files.

var.dvr	the compile parameters definition
README.md	the project code, script and explanation
makefile script	contains all used code files
obj folder	contains the modules .mod and .o files if no folder, plase run the run.bash once
ChangedNotApply	contains the advanced improvement and not tested codes.

Table B.2: Descriptions about some files in the sampled folders

H3+Jacobi	calculation for H3+ in Jacobi coordinate giving a sample spectrum.
H3+Radau	calculation for H3+ in Radau coordinates inculding rotational excitation
HCN	mcalculation for HCN in Jacobi coordinates give a sample spectrum
Water	calculation for water in Radau coordinates giving a sample spectrum and doing a property calculation.

Table B.3: Table about the replacement

New	Old	Example	meanings	times
==	.eq.	(A == B) is not true.	Check whether the values of the two operands are equal or not, if yes, the condition becomes true.	1570
/=	.ne.	(A ! = B) is true.	Check whether the values of the two operands are equal or not, if not equal, then the condition becomes true.	340
>	.gt.	(A >B) is not true.	Check if the value of the left operand is greater than the right one, if so then the condition is true.	640
<	.lt.	(A <B) is true .	Check if the value of the left operand is less than the right one, if so then the condition is true.	150
≥	.geq.	(A ≥ B) is not true .	if the value of the left operand is greater than or equal to the right operand, the condition becomes true.	300
≤	.leq.	(A ≤ B) is true .	if the value of the left operand is less than or equal to the right operand, the condition becomes true.	320

Table B.4: HCN sample test result.

	VM. Linux	Windows 10 PC	UCL. theory cluster
	(seconds)	(seconds)	(seconds)
Old version	33.924	25.214	28.316
	34.234	26.376	27.983
	34.025	23.796	28.502
New version	35.032	24.136	29.326
	34.987	25.103	28.832
	34.850	25.831	28.516