January 20, 2025

Quantum description of the photochemistry of molecules trapped in an optical cavity : numerical and methodological developments of the MolecCav library

Author :
Mathéo SEGAUD[†]

[†] *Université Paris-Saclay, M2 Physical, Inorganic and Solid-State Chemistry, 91190, Gif-sur-Yvette, France.*

matheo.segaud@etu-upsaclay.fr

1 Introduction and Research background

1.1 Research background

Theoretical studies in photochemical process and spectroscopy have great interest in describing quantically the dynamics of a light-interacting molecular system using the most efficient algorithms possible in order to be able to support computations with a high potential number of degrees of freedom (DOF). In the case of an optical cavity-caged system, the Jaynes-Cummings model [1] is a good starting point. It describes a system composed of a two-energy level atom, confined in an optical cavity and interacting with one of this cavity mode. The major point of this model is that the Hamiltonian of the cavity i.e. the photon electromagnetic field Hamiltonian, is described as a quantum harmonic oscillator. Nonetheless this model is limited to one atom interacting with one mode of the cavity, and therefore had to be extended to reach photochemistry applications. More particularly, in the so called Extended Molecular Jaynes-Cummings Model (EJCM) [2], the molecular properties - dipole moments and electronic potential energy - are computed for a free molecule before being used to compute the effects of the couplings with the cavity.

This project takes place as a first step of a larger project which will constitute the M2 internship that will follow, and they are thus both realized under the supervision of David Lauvergnat, reaserch professor at the CNRS and the Université Paris-Saclay [3]. The aim is to provide numerical and methodological methods for a quantum description of an optical cavity-caged molecule, including the molecular modes with non-adiabatic couplings, and a quantum description of the electric field of the cavity modes and its couplings with the molecule. The latter "cavity-field" point may be numerically computed using a grid-based approach, but this way turns out not to be efficient enough for a system with a high number of DOFs. Therefore, numerical developments of a analytically-based computation of the cavity-matter couplings have to be carried out, and this is, in some extent, the part this project focuses on. Its aim is to provide a Modern Fortran library [4] to compute the effects of the cavity and the couplings between a single-mode cavity and the molecule, neglecting the magnetic contribution of the field, and based on analytical properties of the cavity model. The molecular part of the system is assumed to be computed by the code calling the library, and will be considered in later parts of the internship, coupling the library to quantum dynamics programs. The library will be then generalized to N-dimensions during the internship, and adapted to treat more realistic systems.

1.2 Theoretical points - the models

As said before, the Jaynes-Cummings (JC) Model describes the interactions of a two-energy level atom with one mode of the optical cavity it is trapped in, which is modeled as a quantum harmonic oscillator (HO) of mass 1 a.u. and eigenpulsation ω_c. This model's total Hamiltonian of the system $\hat{H}_{tot}^{JC}$ can be written as the sum of three terms $\hat{H}_{tot}^{JC} = \hat{H}_{ato}^{JC} + \hat{H}_{cav}^{JC} + \hat{H}_{CA}^{JC}$, being respectively the Hamiltonian associated with the free atom, the one associated with the free electromagnetic field of the cavity (without the atom), and the one associated with the cavity-atom interactions. The cavity being a HO, the photonic part of the system is expressed in the second quantization notations as $\hat{H}_{cav}^{JC} = \hbar w_c \left(\hat{a}^\dagger \hat{a} + \frac{1}{2}\right)$. We recall here that in that formalism, the two operators $\hat{a}^\dagger$ and $\hat{a}$ are respectively the excitation quanta creation and annihilation of the HO. They are defined such as, for a eigenstate $|n\rangle$ of the HO, $\hat{a}^\dagger |n\rangle = |n+1\rangle$, and $\hat{a}^\dagger |n\rangle = |n-1\rangle ifn > 0; 0otherwise.$

The extensions [2][5] of this model describe, in the dipolar approximation, a (usually non-relativistic) molecular system interacting with several modes of the cavity it is confined in. As before, the global system Hamiltonian $\hat{H}_{tot}$ is expressed as a sum of three terms :

$$\hat{H}_{tot}(\overrightarrow{r}, \overrightarrow{x}, \overrightarrow{R}) = \hat{H}_{mat}(\overrightarrow{r}, \overrightarrow{R}) + \hat{H}_{cav}(\overrightarrow{x}) + \hat{H}_{CM}(\overrightarrow{r}, \overrightarrow{x}, \overrightarrow{R}) \tag{1}$$

In this equation, $\overrightarrow{r}$ represent the coordinates of the electrons, that is a vector of $\left(\mathbb{R}^3\right)^{N_e}$ with N_e the number of electrons; $\overrightarrow{x}$ is the displacement coordinate of the photons of the N_c cavity modes; and $\overrightarrow{R}$

is the coordinates of the N_N Nuclei. Please note that all equations are henceforth written in atomic units (a.u.) as well as are computed the results in the code. Similarly to the JC model, the $\hat{H}_{mat}$ term corresponds to the free matter Hamiltonian. It is expressed here in the Born-Oppenheimer approximation :

$$\hat{H}_{mat}(\overrightarrow{r},\overrightarrow{R}) = \hat{T}_N(\overrightarrow{R}) + \hat{H}_{elec}(\overrightarrow{r},\overrightarrow{R}) \tag{2}$$

with respectively the nuclei kinetic energy operator and the electronic Hamiltonian. The second term is again the free cavity modes Hamiltonian expressed as the sum of the HO associated with each of the cavity mode ν and formulated with photon displacement coordinates $\overrightarrow{x}$:

$$\hat{H}_{cav}(\overrightarrow{x}) = \sum_{\nu=1}^{2N_c} (\frac{1}{2}\left(\hat{p}_\nu^2 + \omega_\nu^2 \hat{x}_\nu^2\right) \tag{3}$$

where ν is the photon mode, ranging from 1 to $2N_c$ to account for the two polarizations of the electromagnetic field. $\hat{x} = \frac{1}{\sqrt{2\omega_c}}\left(\hat{a} + \hat{a}^\dagger\right)$ and $\hat{p}_\nu = -i\sqrt{\frac{\omega_c}{2}}\left(\hat{a} - \hat{a}^\dagger\right)$ are respectively the photon displacement operator and the momentum operator. Finally, the last term is the Hamiltonian accounting for the cavity-matter interactions and can be decomposed into several terms between electronic-matter and nuclei-matter couplings :

$$\hat{H}_{CM}(\overrightarrow{r},\overrightarrow{x},\overrightarrow{R}) = \sum_{\nu=1}^{2N_c} \lambda_\nu\omega_\nu\hat{\mu}_e(\overrightarrow{r})\hat{x}_\nu + \lambda_\nu\omega_\nu\hat{\mu}_N(\overrightarrow{R})\hat{x}_\nu + \hat{H}_{DSE,\nu}(\overrightarrow{r},\overrightarrow{R}) \tag{4}$$

$$\hat{H}_{DSE,\nu} = \frac{1}{2}{\lambda_\nu}^2\hat{\mu}_e(\overrightarrow{r}) + \frac{1}{2}{\lambda_\nu}^2\hat{\mu}_N(\overrightarrow{R}) + {\lambda_\nu}^2\hat{\mu}_N(\overrightarrow{R})\hat{\mu}_e(\overrightarrow{r}) \tag{5}$$

λ_ν being the strength of the coupling between the matter and the ν^{th} mode, $\hat{\mu}_e$ the dipole moment of the electronic part of the system, and $\hat{\mu}_N$ the dipole moment of the nucleic one. $\hat{H}_{DSE,\nu}$ is the dipole self-energy term regarding the ν^{th} cavity mode, and have to be taken into account in case of larges coupling effects [2].

More particularly and among these extensions, one must note the Extended Molecular Jaynes-Cummings Model (EJCM) [2]. It relies on the assumption that the molecular subsystem's properties as the dipole moments and the electronic potential energies $E_{elec}(\overrightarrow{R})$ can be computed in a first phase for the free molecule, and used afterwards in the computation of the effects of the couplings with the cavity. This molecular part is computed using the standard Born-Oppenheimer approximation, so that the electronic Time-independent Shrödinger equation is solved at fixed geometry of the nuclei for a given electronic state i, and the $E_{elec,i}(\overrightarrow{R})$ thus obtained is injected in the matter Hamiltonian. The total Hamiltonian becomes for a given electronic state :

$$\hat{H}_{tot,i}^{EJCM}(\overrightarrow{x},\overrightarrow{R}) = \hat{H}_{mat,i}^{EJCM}(\overrightarrow{R}) + \hat{H}_{cav}^{EJCM}(\overrightarrow{x}) + \hat{H}_{CM,i}^{EJCM}(\overrightarrow{x},\overrightarrow{R}) \tag{6}$$

The cavity Hamiltonian remains the same as in the general model (3) but the two others simplifies :

$$\hat{H}_{mat,i}^{EJCM}(\overrightarrow{R}) = \hat{T}_{N,i}(\overrightarrow{R}) + E_{elec,i}(\overrightarrow{R}) \tag{7}$$

$$\hat{H}_{CM,i}^{EJCM}(\overrightarrow{x},\overrightarrow{R}) = \sum_{\nu=1}^{2N_c} \lambda_\nu\omega_\nu\langle\hat{\mu}_{M,i}\rangle(\overrightarrow{R})\hat{x} + \frac{1}{2}{\lambda_\nu}^2\langle\hat{\mu}_{M,i}^2\rangle(\overrightarrow{R}) \tag{8}$$

where $\langle\hat{\mu}_{M,i}^2\rangle(\overrightarrow{R})$ represent the dipolar moment operator of the matter system under electronic state i.

2 Theory and Methodology

In this section will be described in details what the library computes, the theoretical approximations of the aforementioned models used to do so, as well as its structure and some key points of its algorithm.

2.1 The chosen theoretical models

As said before, the MolecCav library is designed, in a first time, to compute the effects of the interactions between **one** molecule and a **single** mode of the cavity it is confined in using analytical properties of the cavity, considering only the electric part of the photon field. The system is considered non-relativistic and in the dipole approximation. As a first approach, the cavity will be considered having infinite dimensions over the y and z-axis and small over the x-axis. No quantisation of the cavity field will thus be observed on the two former axis, and the considered single stationary cavity mode will resonate only along the latter one. As a consequence, the afore-written photon displacement coordinate $\overrightarrow{x} \in \mathbb{R}^3$ can be reduced to a one-dimensional coordinate along this very axis $x \in \mathbb{R}$. Within the framework of the models developed earlier, this cavity mode will be modeled by a quantum HO of mass $m = 1a.u.$, eigenpulsation ω_c and force constant $k_c = m\omega_c$, each eigenstate $|n\rangle, \forall n \in \mathbb{N}$ of which i.e. each excitation quanta of which corresponding to the number of photons n resonating in the mode. As for the total system Hamiltonian, it is expressed as written in eq. (9), in a similar way as the general extension of the JC model described in eq. (6), as well as the expression of the photon field Hamiltonian in eq. (11) that comes from the HO character of the electric field model. One goes from eq. (11) to eq. (12) - which is the adaptation of eq. (3) to a single-mode cavity - taking into account the HO properties (m=1a.u.) and using atomic units. The matter (eq. 10) and the cavity-matter coupling Hamiltonians (eq. 13) are written directly following the EJCM formalism (eq. 7 and 8), and $\hat{H}_{CM}$ is adapted to a single-mode cavity. Yet, one is actually not concerned with the expression of $\hat{H}_{mat}$ since it is not computed in the library and left to the quantum chemistry code calling MolecCav, nor is the action of the matter total dipole moment $\hat{\mu}_M$. Furthermore, since from the MolecCav viewpoint the quadratic part of the couplings, left in gray in eq. (13), is just a constant that does not depend on the photonic operators and is not crucial for several cases far from the edge of the model (strong coupling effects) it will henceforward be omitted and likely to be added to the code only in the internship part of the project.

$$\hat{H}_{tot}(x, \overrightarrow{R}) = \hat{H}_{mat}(\overrightarrow{R}) + \hat{H}_{cav}(x) + \hat{H}_{CM}(x, \overrightarrow{R}) \tag{9}$$

$$\hat{H}_{mat}(\overrightarrow{R}) = \hat{T}_N(\overrightarrow{R}) + E_{elec}(\overrightarrow{R}) \tag{10}$$

$$\hat{H}_{cav}(x) = \hbar\omega_c\left(\hat{a}^\dagger\hat{a} + \frac{1}{2}\right) = \frac{\hat{p}_x^2}{2m} + \frac{1}{2}k_c\hat{x}^2 \overset{\langle x|}{=} -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + \frac{1}{2}k_c x^2 \tag{11}$$

$$= \omega_c\left(\hat{a}^\dagger\hat{a} + \frac{1}{2}\right) = \frac{1}{2}\left(\hat{p}_x^2 + \omega_c^2\hat{x}^2\right) \overset{\langle x|}{=} \frac{1}{2}\left(-\frac{\partial^2}{\partial x^2} + \omega_c^2 x^2\right) \tag{12}$$

$$\hat{H}_{CM}(x, \overrightarrow{R}) = \lambda\omega_c\hat{\mu}_M(\overrightarrow{R})\hat{x} + \frac{1}{2}\lambda^2\hat{\mu}_M^2(\overrightarrow{R}) \tag{13}$$

2.2 Practical details

Starting from this section will be discussed the code of the library, its structure and its methods. The "users-centered" part of the manual, such as the installation, requirements, and running procedure, is rather to be found in the README.md [4].

Please note nonetheless that the library is designed to be used within a GNU/Linux operating system, and the compilation and execution files (makefile and run.sh) were written to use the gfortran compiler for Fortran90, since its the language used for the whole code except for the Make and Shell files management parts.

2.3 Overview of the library

The MolecCav Library is structured over several subdirectories within the main MolecCav one, as illustrated in Fig. 1. Its different versions over time are managed using git, and available online in a GitHub repository [4].

```
MolecCav/
├── APP/
│   └── App_MolCav.f90
├── Ext_Lib/
│   ├── QDUtilLib_loc/
│   ├── QDUtilLib-main/
│   ├── .gitignore
│   ├── cleanlib
│   └── get_Lib.sh
├── OBJ/
│   └── obj/
├── OUT/
├── SRC/
│   ├── MC_cavity_mode_m.f90
│   ├── MC_total_hamiltonian_m.f90
│   └── MC_operator_1D_m.f90
├── TESTS/
│   ├── test_cavity_mode.f90
│   └── test_construct_op.f90
├── .gitignore
├── data_app.nml
├── data_tests.nml
├── data.nml
├── libMolecCav.a
├── makefile
├── README.md
└── run.sh
```

Figure 1: MolecCav arborescence

- The directory that holds the core of the library is the **SRC/** one. It contains all the source files of the modules that compose the library.
- The others source files of the code are either from the test programs, or the application program. They respectively belong either to the **TESTS/** directory, or to the **APP/** one.
- Three data files are expected to be used with MolecCav as Fortran namelist files : **data_app.nml**, **data_tests.nml** and **data.nml**, placed for now in the main directory. They contains information about the cavity parameters (cf. System initialisation section 2.4). For now only one namelist can be found in each of them since the code supports only one cavity mode, but more will be added when the procedures will be generalized to N cavity modes. The data_app.nml and data_tests.nml are not intended to be modified by the user and reserved for the application and tests programs respectively. One should instead used the **data.nml**. Yet, whatever program is running, one can select the datafile used by providing its name as a command-line argument of the run.sh script.
- To compile, link, and when required execute the code from the source files, one should use one of the two proposed methods : the **makefile** or the shell script **run.sh**. Both have more or less the same features, at the main exception that the script creates the asked files without checking whether they need to be, while the make will do so only if the asked files are missing or older than the files they depend on. Both are placed in the current directory.
- When a source file is compiled into an object file, that new file is put away in one of the **OBJ/** subdirectories. If the source file was a module, the .mod file is stored at the same place. By default, only the **OBJ/obj/** is created and used but, if one wants to separate the files from a compilation not

to have the previous ones deleted, their destination can be changed through the OBJ_DIR variable of the makefile, or passing its name as an argument when run.sh is run (cf README.md [4] for more).

- The **Ext_Lib/** directory contains the external libraries used in MolecCav. For now, the QDUtilLib library by David Lauvergnat [6] is mandatory for some convenient writings and will be used later for diagonalization procedures. It is automatically downloaded from GitHub (with **get_Lib.sh**), installed and compiled when the make or run.sh files are used. Note that, although run.sh doesn't check the creation date of the files as make does and compile the library again each time, that does not apply to the external library. They are compiled through a command that calls their own makefile, which does check the dates. The external libraries have their own **.gitignore** file.

- Then the object files of the modules and the External libraries static library files are gathered into the static library file **libMolecCav.a**. It is automatically generated when the makefile or run.sh files are run and left in the current directory.

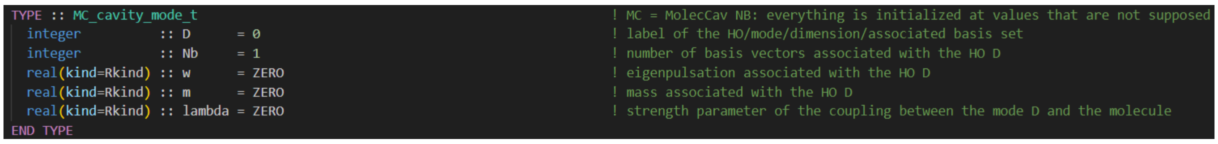
- After the object file of a program (test or application) is linked with the static library files, the executable file is placed in the current directory. It is most of the time automatically run by the make or run.sh files can also be run directly by command-line. The output files (.log) of these executables, named after the name of the corresponding source file, are stored in the **OUT/** directory.

- Finally the **.gitignore** file of the current directory exclude all the object, module, executable, and output files.

The following sections detail the procedure of the simulation as written in the code.

2.4 MC_cavity_mode_m.f90 - System initialization

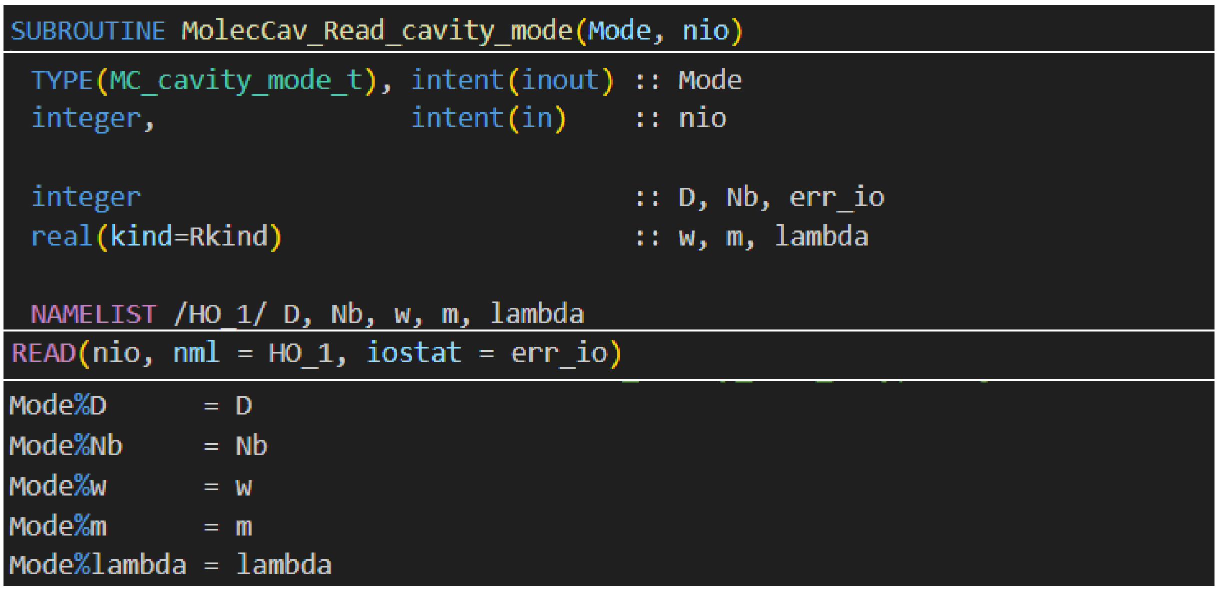
To begin with, the single-mode cavity part of the system is built in the **MC_cavity_mode_m.f90 module**. It is in the form of the derived type **MC_cavity_mode_t** which is structured with the cavity physical parameters as shown Fig. 2. Each object of this type represent one mode of the cavity, so when the code will be extended to N dimensions, the same procedure will be used for all the modes.



```fortran
TYPE :: MC_cavity_mode_t                                                       ! MC = MolecCav NB: everything is initialized at values that are not supposed
  integer           :: D      = 0                                              ! label of the HO/mode/dimension/associated basis set
  integer           :: Nb     = 1                                              ! number of basis vectors associated with the HO D
  real(kind=Rkind) :: w      = ZERO                                           ! eigenpulsation associated with the HO D
  real(kind=Rkind) :: m      = ZERO                                           ! mass associated with the HO D
  real(kind=Rkind) :: lambda = ZERO                                           ! strength parameter of the coupling between the mode D and the molecule
END TYPE
```

Figure 2: MC_cavity_mode_t

Note that all the parameters of the type are initialized to default values that are not supposed to be used, so that the well-construction of a cavity mode can easily be tested by checking its parameters values. It is built by calling the **MolecCav_Read_cavity_mode subroutine** (Fig. 3). It initializes the object **Mode** of type MC_cavity_mode_t by reading a namelist from the data file of Fortran unit **nio**, and assigning its values to the Mode's parameters.



```fortran
SUBROUTINE MolecCav_Read_cavity_mode(Mode, nio)
  TYPE(MC_cavity_mode_t), intent(inout) :: Mode
  integer,                intent(in)    :: nio

  integer                               :: D, Nb, err_io
  real(kind=Rkind)                      :: w, m, lambda

  NAMELIST /HO_1/ D, Nb, w, m, lambda
READ(nio, nml = HO_1, iostat = err_io)
Mode%D      = D
Mode%Nb     = Nb
Mode%w      = w
Mode%m      = m
Mode%lambda = lambda
```

Figure 3: MolecCav_Read_cavity_mode subroutine

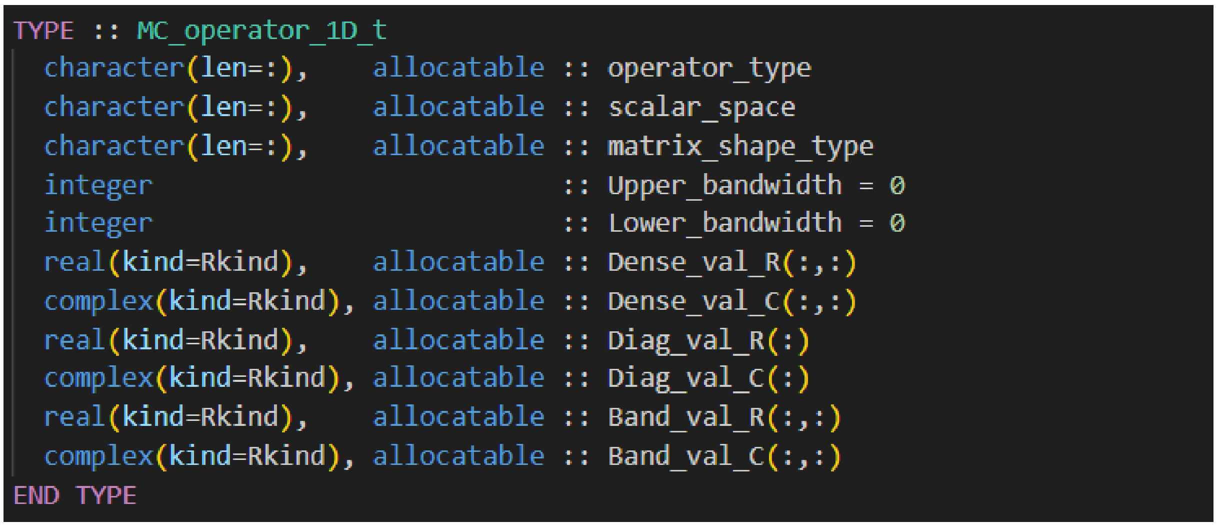
2.5 MC_operator_1D_m.f90 - Construction of the operators

Now the electric field mode of the cavity is built, one have to construct one by one the operators defining the observable phenomena, as the **MC_operator_1D_m.f90** propose to do. This module allows the construction of the operators related to the cavity mode, so they are the well-known 1D HO operators of quantum mechanics : the 1D HO Hamiltonian $\hat{H}_{cav}(x)$, position $\hat{x}$, and number of excitation quanta i.e. number of photons $\hat{N} = \hat{a}^{\dagger}\hat{a}$ operators. Note that the momentum operator $\hat{p}_x$ is not constructed here since it is not needed yet, but is likely to be implemented in later releases of the library.

These operators are built as numerical objects of the type **MC_operator_1D_t** (Fig. 4), which gathers all information about an operator. The first string **operator_type** is the physical meaning of that operator ("Hamiltonian", "Position", etc), on which depends the shape of its matrix and so the subroutine called to construct it. **scalar_type** informs about whether its matrix elements has to be declared as real or complex numbers, but up to now, only reals are used. Then the **matrix_shape_type** contains the information about how is supposed to be its analytical matrix. Most of the time, these operators have sparse matrix, so knowing where are the zeros make it possible to optimize the storage of the matrix and later the action computation. This string is expected to take "Dense", "Diagonal", or "Band" values. The "Dense" possibility allows to store the whole matrix without any optimization, as it were a dense matrix, and is mainly used for tests purposes (cf. 2.9 section). "Band" is a flexible possibility, because it allows to inform two more parameters : **Upper_bandwidth** and **Lower_bandwidth**, which indicates the number of diagonals above and below the main one to store. **Upper_bandwidth** = **Lower_bandwidth** = 0 would give the same result as the "Diagonal" case. Then the several matrices that follows in the derived type corresponds to the different possibilities to store matrix elements according to the **matrix_shape_type** and **scalar_space** of the operator.

To construct an object of that derived type, one would call the **MolecCav_Construct_Operator** general subroutine (Fig. 5. It begins the initialization of the **Operator** variable by assigning to its parameters the type of the operator and the scalar space, and left the remaining of the initialization to the specific subroutines that it calls, according to the operator's nature. The last three arguments are separated for now so that this module is independent on the previous one, but they will be soon replaced by one argument of the MC_cavity_mode_t type i.e. the cavity mode associated to this operator.

All the three specific subroutines realize the same thing (Fig. 6: they initialize the parameter of the Operator corresponding to the shape of its matrix, and then build that matrix into one of the Operator's parameters, using its analytical expression. For each, two modes are possible. Either the argument **matrix_shape_type** says "Opt" and the storage of the matrix is optimized i.e. built such as to take

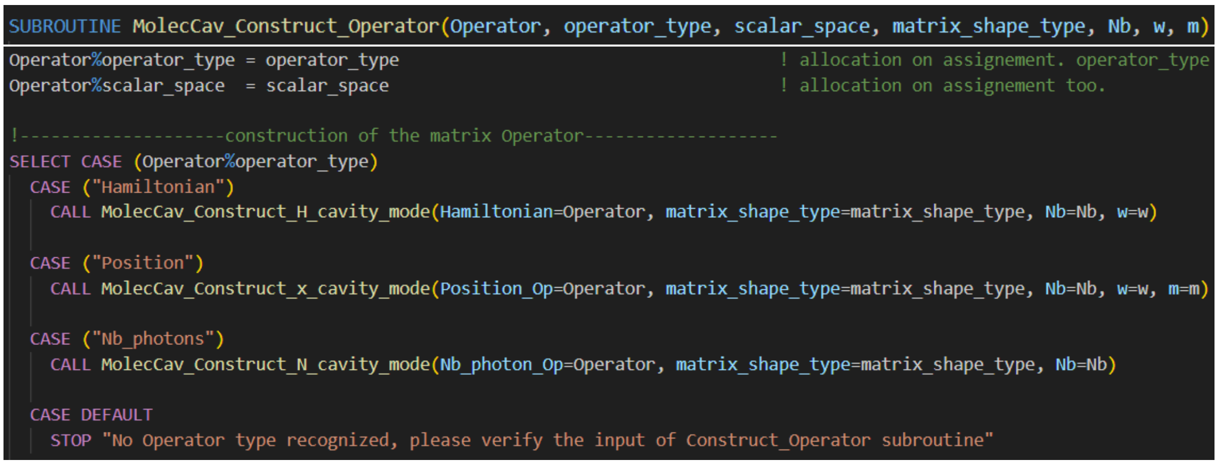


```
TYPE :: MC_operator_1D_t
  character(len=:),    allocatable :: operator_type
  character(len=:),    allocatable :: scalar_space
  character(len=:),    allocatable :: matrix_shape_type
  integer                          :: Upper_bandwidth = 0
  integer                          :: Lower_bandwidth = 0
  real(kind=Rkind),    allocatable :: Dense_val_R(:,:)
  complex(kind=Rkind), allocatable :: Dense_val_C(:,:)
  real(kind=Rkind),    allocatable :: Diag_val_R(:)
  complex(kind=Rkind), allocatable :: Diag_val_C(:)
  real(kind=Rkind),    allocatable :: Band_val_R(:,:)
  complex(kind=Rkind), allocatable :: Band_val_C(:,:)
END TYPE
```

Figure 4: MC_operator_1D_t derived type

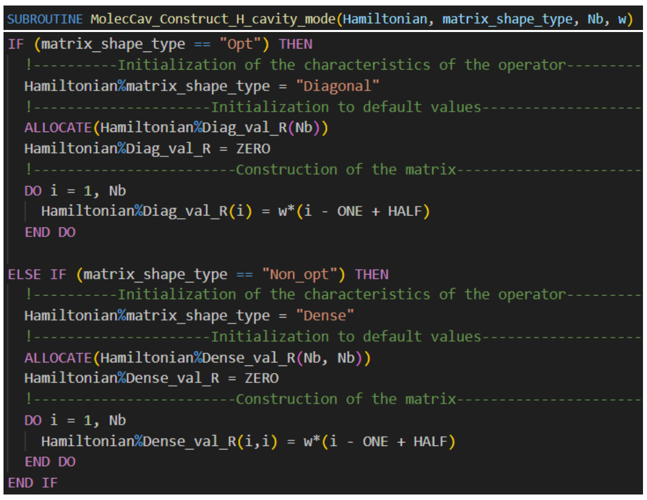
advantage of the sparse nature of the matrices, or it says "Non_opt" and the whole matrix is constructed as it were a dense one.

A word about the basis vectors though. As was said, the library is design to make use of the analytical properties of the HO system in aim not to have to work on any spatial grid. The basis vectors are thus not explicitly defined in any module. They are *implicitly* taken as the corresponding HO eigenvectors basis set $|n\rangle_{n\in\mathbb{N}}$, so that the operators' matrices have known expression, and especially, the Hamiltonian and number of photons operator have diagonal matrices.

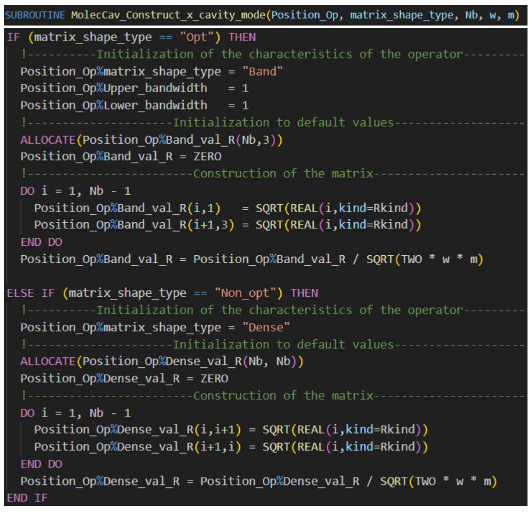


```
SUBROUTINE MolecCav_Construct_Operator(Operator, operator_type, scalar_space, matrix_shape_type, Nb, w, m)
Operator%operator_type = operator_type                          ! allocation on assignement. operator_type
Operator%scalar_space  = scalar_space                           ! allocation on assignement too.

!------------------construction of the matrix Operator------------------
SELECT CASE (Operator%operator_type)
  CASE ("Hamiltonian")
    CALL MolecCav_Construct_H_cavity_mode(Hamiltonian=Operator, matrix_shape_type=matrix_shape_type, Nb=Nb, w=w)

  CASE ("Position")
    CALL MolecCav_Construct_x_cavity_mode(Position_Op=Operator, matrix_shape_type=matrix_shape_type, Nb=Nb, w=w, m=m)

  CASE ("Nb_photons")
    CALL MolecCav_Construct_N_cavity_mode(Nb_photon_Op=Operator, matrix_shape_type=matrix_shape_type, Nb=Nb)

  CASE DEFAULT
    STOP "No Operator type recognized, please verify the input of Construct_Operator subroutine"
```

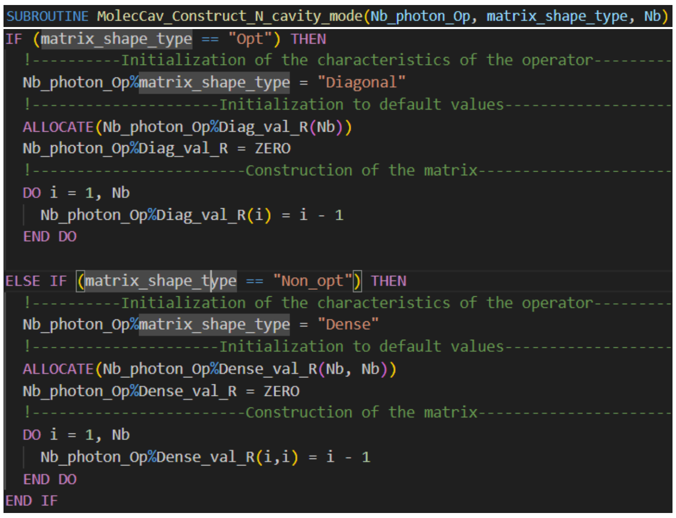
Figure 5: MolecCav_Construct_operator subroutine



```fortran
SUBROUTINE MolecCav_Construct_H_cavity_mode(Hamiltonian, matrix_shape_type, Nb, w)
IF (matrix_shape_type == "Opt") THEN
  !---------Initialization of the characteristics of the operator--------
  Hamiltonian%matrix_shape_type = "Diagonal"
  !-------------------Initialization to default values------------------
  ALLOCATE(Hamiltonian%Diag_val_R(Nb))
  Hamiltonian%Diag_val_R = ZERO
  !----------------------Construction of the matrix---------------------
  DO i = 1, Nb
    Hamiltonian%Diag_val_R(i) = w*(i - ONE + HALF)
  END DO

ELSE IF (matrix_shape_type == "Non_opt") THEN
  !---------Initialization of the characteristics of the operator--------
  Hamiltonian%matrix_shape_type = "Dense"
  !-------------------Initialization to default values------------------
  ALLOCATE(Hamiltonian%Dense_val_R(Nb, Nb))
  Hamiltonian%Dense_val_R = ZERO
  !----------------------Construction of the matrix---------------------
  DO i = 1, Nb
    Hamiltonian%Dense_val_R(i,i) = w*(i - ONE + HALF)
  END DO
END IF
```



```fortran
SUBROUTINE MolecCav_Construct_x_cavity_mode(Position_Op, matrix_shape_type, Nb, w, m)
IF (matrix_shape_type == "Opt") THEN
  !---------Initialization of the characteristics of the operator--------
  Position_Op%matrix_shape_type = "Band"
  Position_Op%Upper_bandwidth   = 1
  Position_Op%Lower_bandwidth   = 1
  !-------------------Initialization to default values------------------
  ALLOCATE(Position_Op%Band_val_R(Nb,3))
  Position_Op%Band_val_R = ZERO
  !----------------------Construction of the matrix---------------------
  DO i = 1, Nb - 1
    Position_Op%Band_val_R(i,1)   = SQRT(REAL(i,kind=Rkind))
    Position_Op%Band_val_R(i+1,3) = SQRT(REAL(i,kind=Rkind))
  END DO
  Position_Op%Band_val_R = Position_Op%Band_val_R / SQRT(TWO * w * m)

ELSE IF (matrix_shape_type == "Non_opt") THEN
  !---------Initialization of the characteristics of the operator--------
  Position_Op%matrix_shape_type = "Dense"
  !-------------------Initialization to default values------------------
  ALLOCATE(Position_Op%Dense_val_R(Nb, Nb))
  Position_Op%Dense_val_R = ZERO
  !----------------------Construction of the matrix---------------------
  DO i = 1, Nb - 1
    Position_Op%Dense_val_R(i,i+1) = SQRT(REAL(i,kind=Rkind))
    Position_Op%Dense_val_R(i+1,i) = SQRT(REAL(i,kind=Rkind))
  END DO
  Position_Op%Dense_val_R = Position_Op%Dense_val_R / SQRT(TWO * w * m)
END IF
```

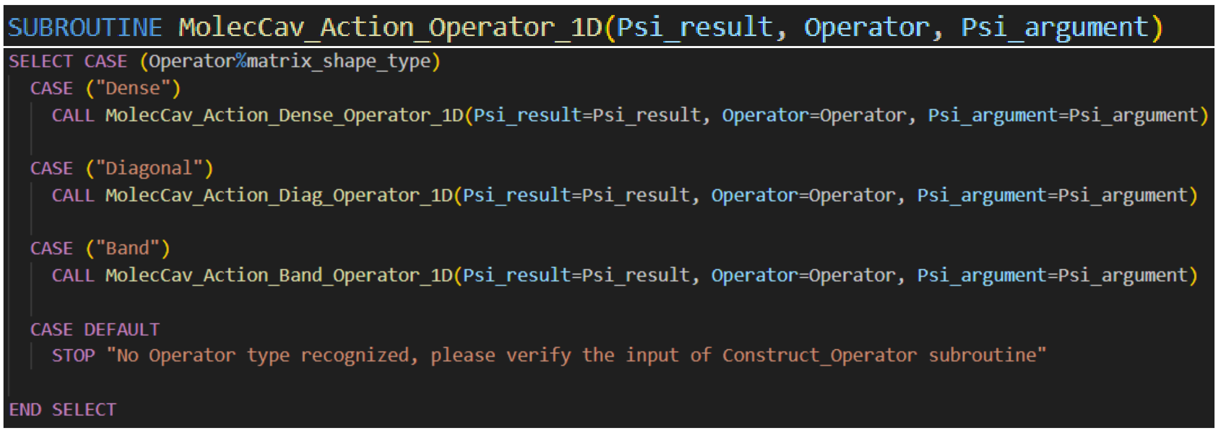


```fortran
SUBROUTINE MolecCav_Construct_N_cavity_mode(Nb_photon_Op, matrix_shape_type, Nb)
IF (matrix_shape_type == "Opt") THEN
  !---------Initialization of the characteristics of the operator--------
  Nb_photon_Op%matrix_shape_type = "Diagonal"
  !-------------------Initialization to default values------------------
  ALLOCATE(Nb_photon_Op%Diag_val_R(Nb))
  Nb_photon_Op%Diag_val_R = ZERO
  !----------------------Construction of the matrix---------------------
  DO i = 1, Nb
    Nb_photon_Op%Diag_val_R(i) = i - 1
  END DO

ELSE IF (matrix_shape_type == "Non_opt") THEN
  !---------Initialization of the characteristics of the operator--------
  Nb_photon_Op%matrix_shape_type = "Dense"
  !-------------------Initialization to default values------------------
  ALLOCATE(Nb_photon_Op%Dense_val_R(Nb, Nb))
  Nb_photon_Op%Dense_val_R = ZERO
  !----------------------Construction of the matrix---------------------
  DO i = 1, Nb
    Nb_photon_Op%Dense_val_R(i,i) = i - 1
  END DO
END IF
```

Figure 6: MolecCav_Construct_H_cavity_mode, x_cavity_mode, and N_cavity_mode subroutine

2.6 Action of the operators

After the operators were constructed, one will need to be able to compute their action on a wavefunction to eventually compute the action of the total Hamiltonian. As the construction, the action calculation starts by the call of a general subroutine **MolecCav_Action_Operator_1D**, which does nothing but calls the specific subroutine corresponding to the shape of the operator's matrix. The **MolecCav_Action_Dense_Operator_1D** subroutine computes the result of the Operator's action upon the wavefunction's vector by simple matrix multiplication, while **MolecCav_Action_Diag_Operator_1D** and **MolecCav_Action_Band_Operator_1D** does so by term-wise multiplication of their respective elements.

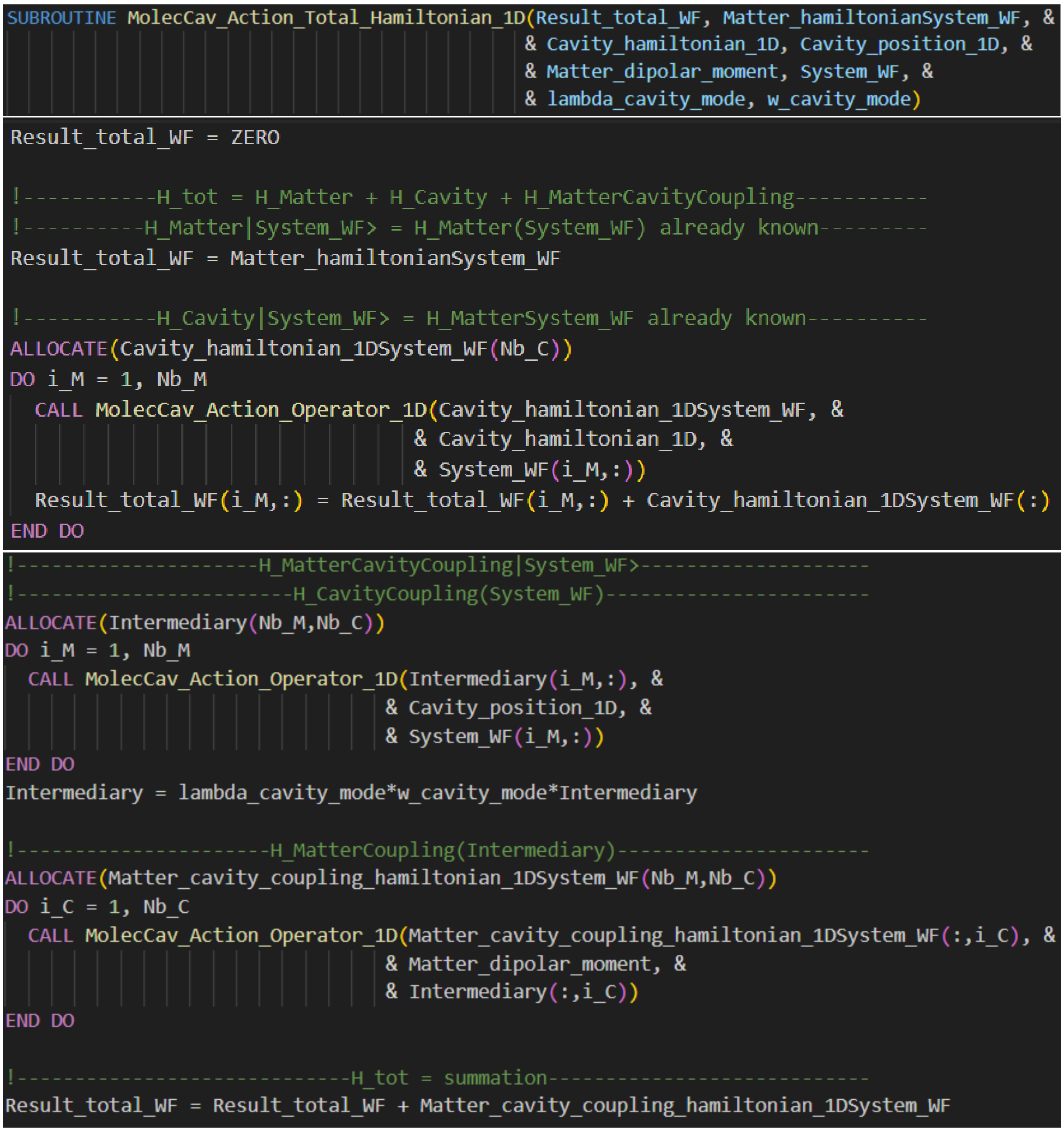


```
SUBROUTINE MolecCav_Action_Operator_1D(Psi_result, Operator, Psi_argument)
SELECT CASE (Operator%matrix_shape_type)
  CASE ("Dense")
    CALL MolecCav_Action_Dense_Operator_1D(Psi_result=Psi_result, Operator=Operator, Psi_argument=Psi_argument)

  CASE ("Diagonal")
    CALL MolecCav_Action_Diag_Operator_1D(Psi_result=Psi_result, Operator=Operator, Psi_argument=Psi_argument)

  CASE ("Band")
    CALL MolecCav_Action_Band_Operator_1D(Psi_result=Psi_result, Operator=Operator, Psi_argument=Psi_argument)

  CASE DEFAULT
    STOP "No Operator type recognized, please verify the input of Construct_Operator subroutine"

END SELECT
```

Figure 7: MolecCav_Action_Operator_1D subroutine

2.7 Action of the total Hamiltonian

The action of $\hat{H}_{tot}(x, \overrightarrow{R})$ can now be computed from the **MC_total_hamiltonian_m** module, with the **MolecCav_Action_Total_Hamiltonian_1D** subroutine (Fig. 8). For a single-mode cavity, it computes and constructs the resulting wavefunction from the system (total : matter⊗cavity) wavefunction in a few steps, corresponding to the separated action of the matter Hamiltonian, the cavity Hamiltonian, and the cavity-matter couplings Hamiltonian. For now and for making it possible to test the library without the calling program, the computation relies on the representation of the system wavefunction by a two-variables function - one for the matter, one the photon displacement coordinate -, and thus a matrix with one index for each. As a result, the operators taking action only on the matter (resp. photonic) part operates on the line-vectors (resp. column-vectors) of the matrix System_WF(i_M,:) (resp. System_WF(:,i_C)). Nontheless, the result wavefunction of the action of the matter Hamiltonian on the system one has to be computed by the calling program and in hence an argument of the subroutine. As in the construction subroutine, the last two parameters are about to be gathered into a variable of type MC_Cavity_Mode_t.



```fortran
SUBROUTINE MolecCav_Action_Total_Hamiltonian_1D(Result_total_WF, Matter_hamiltonianSystem_WF, &
                                                & Cavity_hamiltonian_1D, Cavity_position_1D, &
                                                & Matter_dipolar_moment, System_WF, &
                                                & lambda_cavity_mode, w_cavity_mode)

Result_total_WF = ZERO

!-----------H_tot = H_Matter + H_Cavity + H_MatterCavityCoupling----------
!----------H_Matter|System_WF> = H_Matter(System_WF) already known---------
Result_total_WF = Matter_hamiltonianSystem_WF

!-----------H_Cavity|System_WF> = H_MatterSystem_WF already known----------
ALLOCATE(Cavity_hamiltonian_1DSystem_WF(Nb_C))
DO i_M = 1, Nb_M
  CALL MolecCav_Action_Operator_1D(Cavity_hamiltonian_1DSystem_WF, &
                                 & Cavity_hamiltonian_1D, &
                                 & System_WF(i_M,:))
  Result_total_WF(i_M,:) = Result_total_WF(i_M,:) + Cavity_hamiltonian_1DSystem_WF(:)
END DO
!-------------------H_MatterCavityCoupling|System_WF>-------------------
!----------------------H_CavityCoupling(System_WF)----------------------
ALLOCATE(Intermediary(Nb_M,Nb_C))
DO i_M = 1, Nb_M
  CALL MolecCav_Action_Operator_1D(Intermediary(i_M,:), &
                                 & Cavity_position_1D, &
                                 & System_WF(i_M,:))
END DO
Intermediary = lambda_cavity_mode*w_cavity_mode*Intermediary

!--------------------H_MatterCoupling(Intermediary)---------------------
ALLOCATE(Matter_cavity_coupling_hamiltonian_1DSystem_WF(Nb_M,Nb_C))
DO i_C = 1, Nb_C
  CALL MolecCav_Action_Operator_1D(Matter_cavity_coupling_hamiltonian_1DSystem_WF(:,i_C), &
                                 & Matter_dipolar_moment, &
                                 & Intermediary(:,i_C))
END DO

!----------------------------H_tot = summation---------------------------
Result_total_WF = Result_total_WF + Matter_cavity_coupling_hamiltonian_1DSystem_WF
```

Figure 8: MolecCav_Action_Total_Hamiltonian subroutine

2.8 Construction of the total Hamiltonian's matrix - mapping

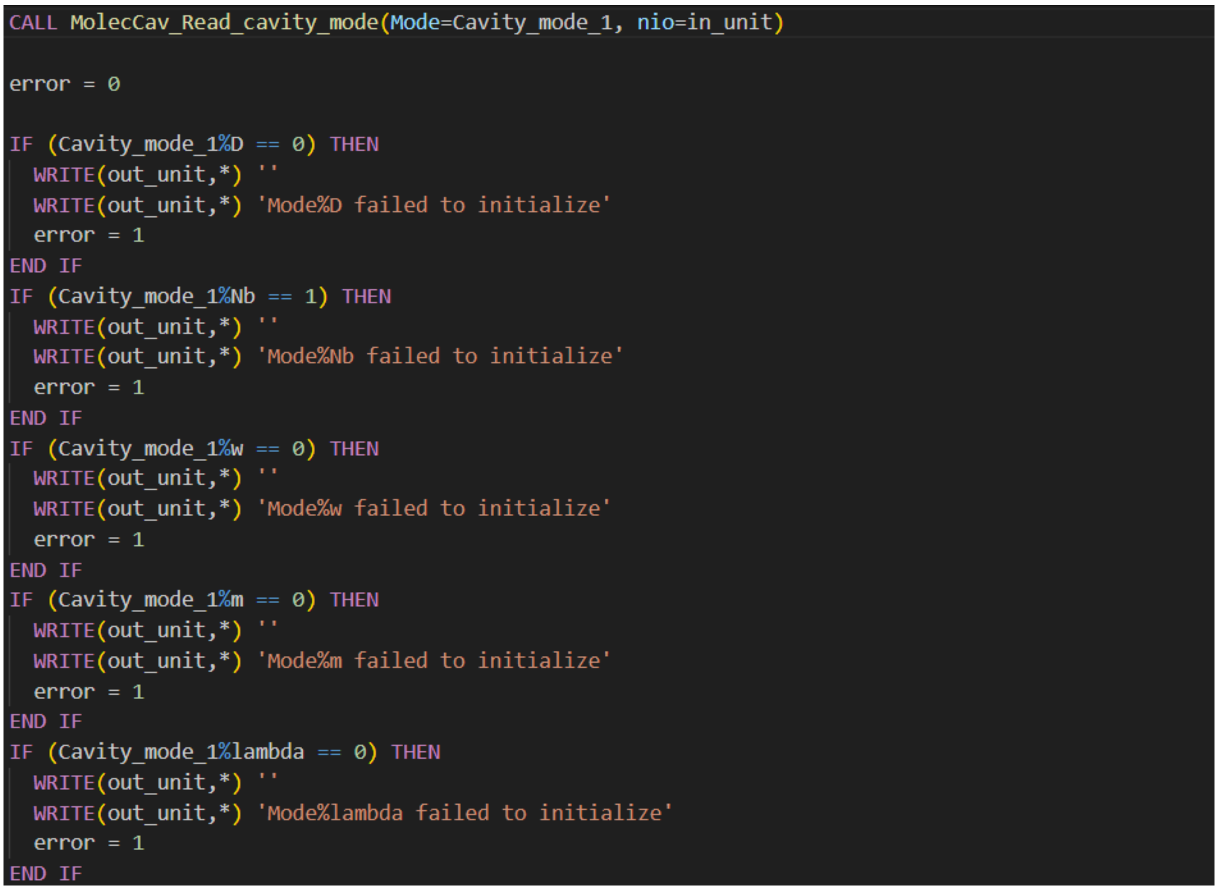
Even though this procedure is still in progress, the calculation currently developed is the building of the total Hamiltonian matrix. This steps uses a mapping method that maps a system wavefunction from a matrix of dimensions $Nb_{matter} \times Nb_{cavity}$ (Nb the number of basis vectors of the associated part of the system) to a vector of dimension $NB = Nb_{matter}.Nb_{cavity}$. This mapping makes it possible to build the total Hamiltonian as a square matrix of dimensions $NB \times NB$, then to diagonalize using DQUtilLib's procedures. This will eventually lead to the Hessian computation and thus the eigenstates and pulsations of the total system.

2.9 The tests

Making it possible for the users to test automatically the library after installation or modification if they want to modify it by themself is required for a safe use. Therfore programs have been implemented and will be implemented to test its different features. For now are available **test_action_op.f90** that test the well-initialization of a cavity mode from a data file, and **test_construct_op.f90** that test the well-construction of the different operators. Both can be executed by hand but one would rather use the "ut" commands of the makefile or the run.sh script, that compile, link, execute, and display the sum-up message written at the end of the output file of each test program.

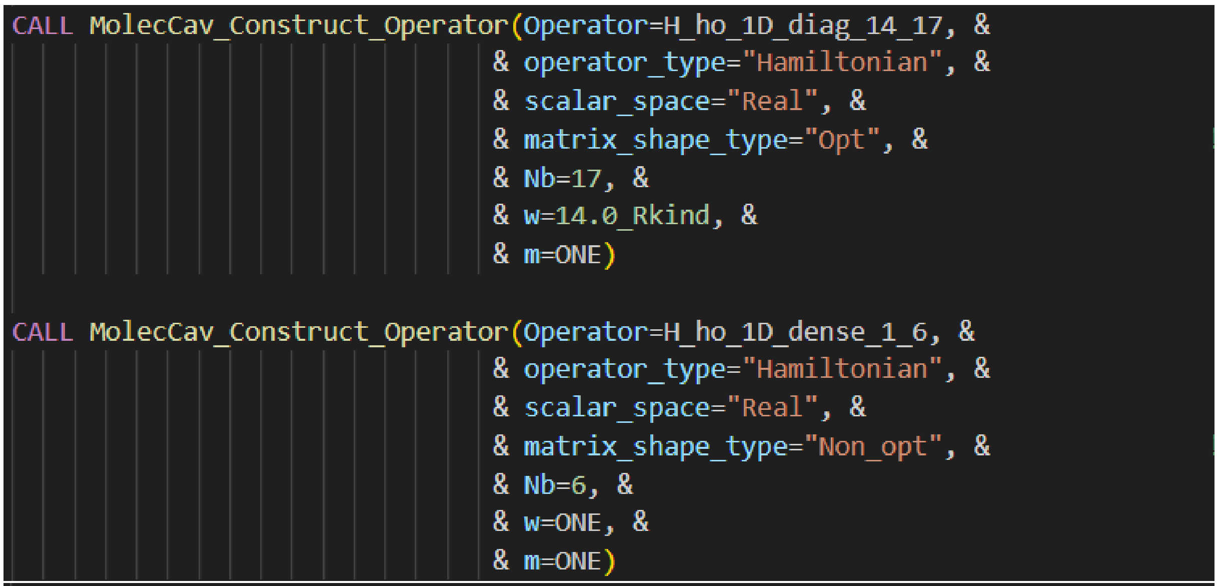
The first one (Fig. 9) initializes a variable of type MC_cavity_mode_t from the data file data_tests.nml calling the previously discussed subroutine and displays error messages for each parameter of the variable still set to its default values.

For each kind of operator $(\hat{H}, \hat{x}, \hat{N})$, the second one initializes several variables of type MC_operator_1D_t with varying parameters (number of basis vectors, eigenpulsation, mass, shape-optimized or not) as well as a few hard-coded reliable matrices determined analytically and corresponding to some of the previous ones (Fig. 10). Then these matrices are compared one with another, as one of these tests is illustrated in Fig. 11, and an error message is displayed for each one found badly initialized.

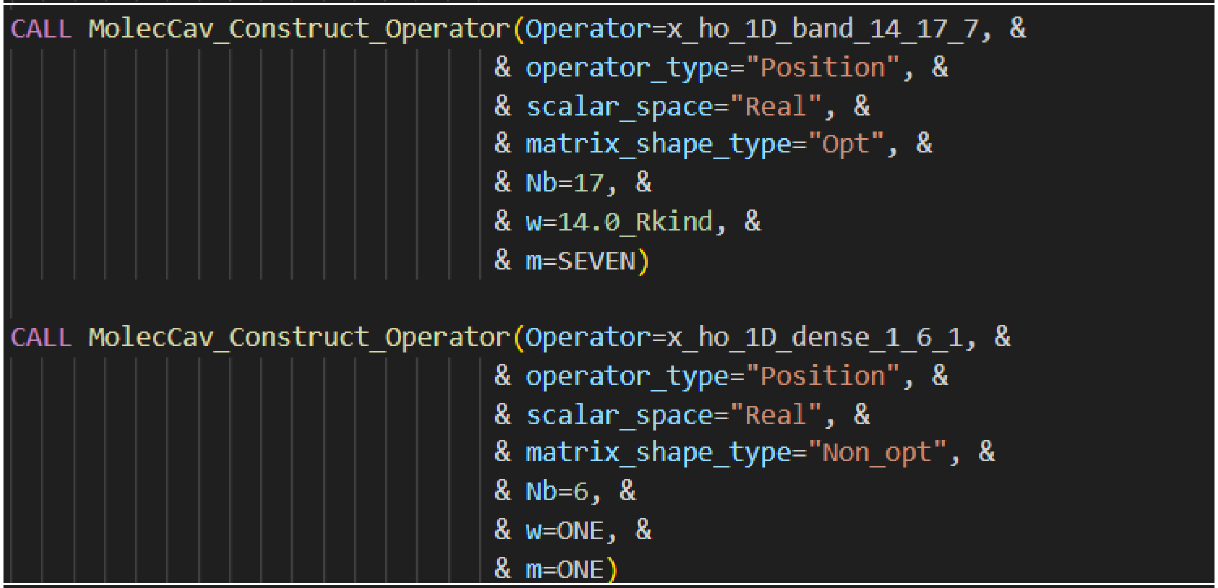


```
CALL MolecCav_Read_cavity_mode(Mode=Cavity_mode_1, nio=in_unit)

error = 0

IF (Cavity_mode_1%D == 0) THEN
  WRITE(out_unit,*) ''
  WRITE(out_unit,*) 'Mode%D failed to initialize'
  error = 1
END IF
IF (Cavity_mode_1%Nb == 1) THEN
  WRITE(out_unit,*) ''
  WRITE(out_unit,*) 'Mode%Nb failed to initialize'
  error = 1
END IF
IF (Cavity_mode_1%w == 0) THEN
  WRITE(out_unit,*) ''
  WRITE(out_unit,*) 'Mode%w failed to initialize'
  error = 1
END IF
IF (Cavity_mode_1%m == 0) THEN
  WRITE(out_unit,*) ''
  WRITE(out_unit,*) 'Mode%m failed to initialize'
  error = 1
END IF
IF (Cavity_mode_1%lambda == 0) THEN
  WRITE(out_unit,*) ''
  WRITE(out_unit,*) 'Mode%lambda failed to initialize'
  error = 1
END IF
```

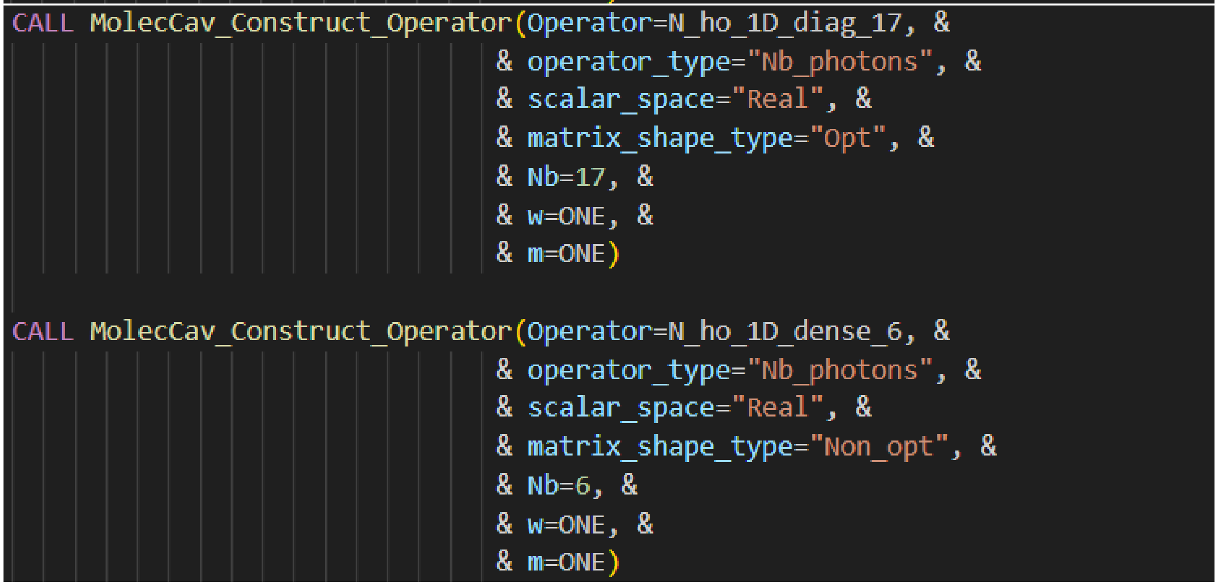
Figure 9: test_cavity_mode test



```fortran
CALL MolecCav_Construct_Operator(Operator=H_ho_1D_diag_14_17, &
                               & operator_type="Hamiltonian", &
                               & scalar_space="Real", &
                               & matrix_shape_type="Opt", &
                               & Nb=17, &
                               & w=14.0_Rkind, &
                               & m=ONE)

CALL MolecCav_Construct_Operator(Operator=H_ho_1D_dense_1_6, &
                               & operator_type="Hamiltonian", &
                               & scalar_space="Real", &
                               & matrix_shape_type="Non_opt", &
                               & Nb=6, &
                               & w=ONE, &
                               & m=ONE)
```

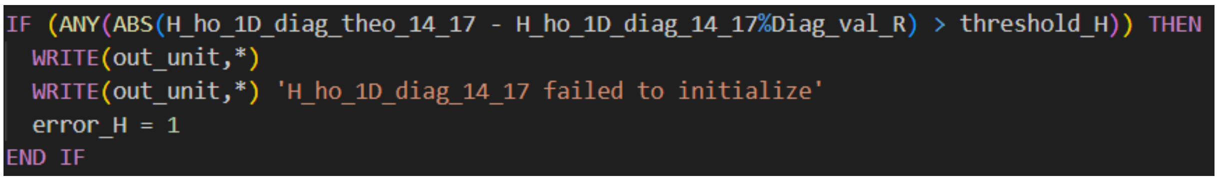


```fortran
CALL MolecCav_Construct_Operator(Operator=x_ho_1D_band_14_17_7, &
                               & operator_type="Position", &
                               & scalar_space="Real", &
                               & matrix_shape_type="Opt", &
                               & Nb=17, &
                               & w=14.0_Rkind, &
                               & m=SEVEN)

CALL MolecCav_Construct_Operator(Operator=x_ho_1D_dense_1_6_1, &
                               & operator_type="Position", &
                               & scalar_space="Real", &
                               & matrix_shape_type="Non_opt", &
                               & Nb=6, &
                               & w=ONE, &
                               & m=ONE)
```



```fortran
CALL MolecCav_Construct_Operator(Operator=N_ho_1D_diag_17, &
                               & operator_type="Nb_photons", &
                               & scalar_space="Real", &
                               & matrix_shape_type="Opt", &
                               & Nb=17, &
                               & w=ONE, &
                               & m=ONE)

CALL MolecCav_Construct_Operator(Operator=N_ho_1D_dense_6, &
                               & operator_type="Nb_photons", &
                               & scalar_space="Real", &
                               & matrix_shape_type="Non_opt", &
                               & Nb=6, &
                               & w=ONE, &
                               & m=ONE)
```

Figure 10: test_construct_op test - creation of the operators



```
IF (ANY(ABS(H_ho_1D_diag_theo_14_17 - H_ho_1D_diag_14_17%Diag_val_R) > threshold_H)) THEN
  WRITE(out_unit,*)
  WRITE(out_unit,*) 'H_ho_1D_diag_14_17 failed to initialize'
  error_H = 1
END IF
```

Figure 11: test_construct_op test - comparisons

2.10 The Application

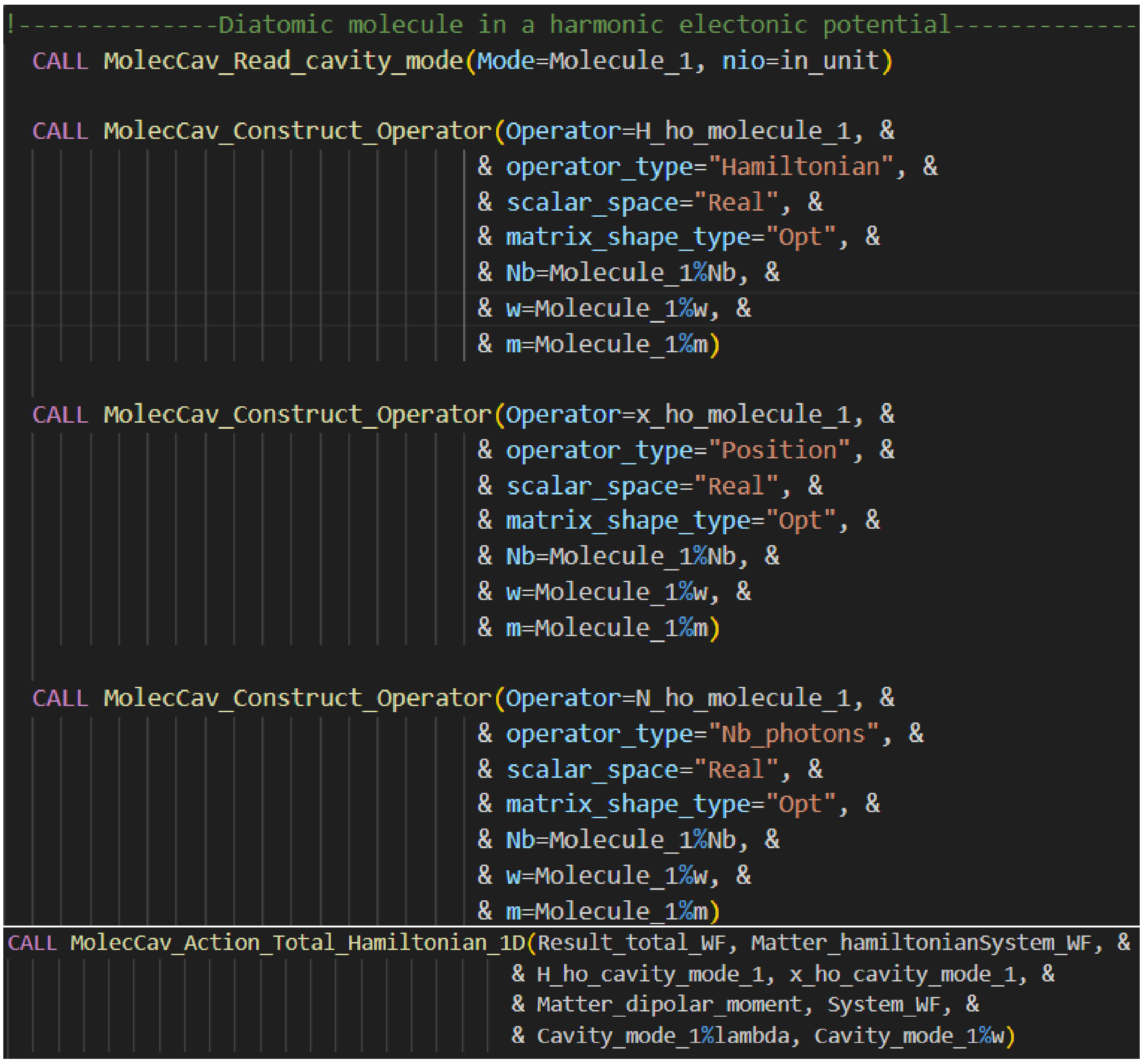
In addition to the tests, an application file is provided as a way to illustrate how the library can be used as well as an additional test. The aim is here to show how to set up in a program the computation of the total Hamiltonian. However, since the computation of the matter part of the system and its Hamiltonian's action is not supposed to be part of the library, the test has to be conducted in a specific case.

The model computed in this application is a diatomic molecule with harmonic electronic potential trapped in our single-mode cavity. This model allows us to use our modules and procedures to construct the molecule and its operators the very same way as if it were a cavity mode, as illustrated Fig. 12. The only difference is the mass that is not 1 a.u. anymore but the reduced mass of the HF molecule. Besides, to be able to compute the action of the dipole moment of the molecule, another approximation is made : it is developed by a Taylor development at the first order 14 with x_0 the position where the harmonic potential of the molecule is centered. Then, knowing that the equilibrium value of μ is mainly required for rotational properties and not for our calculations, it is set at 0 a.u. as well as the equilibrium position of the potential well R_0, hence 15. Finally, the partial derivative is considered as a constant assuming a linear behaviour of the dipole moment around its equilibrium position 16. The dipole moment can thus be computed as a parameter times the matter position operator. Moreover, since the total system wavefunction is not provided by the calling program, it is arbitrarily initialized.

$$\mu(R) = \mu_0 + \frac{\partial \mu}{\partial R}(R - R_0) \tag{14}$$

$$\mu(R) = \frac{\partial \mu}{\partial R}.R \tag{15}$$

$$\mu(R) = Cte.R \tag{16}$$



```
!-------------Diatomic molecule in a harmonic electonic potential------------
  CALL MolecCav_Read_cavity_mode(Mode=Molecule_1, nio=in_unit)

  CALL MolecCav_Construct_Operator(Operator=H_ho_molecule_1, &
                                  & operator_type="Hamiltonian", &
                                  & scalar_space="Real", &
                                  & matrix_shape_type="Opt", &
                                  & Nb=Molecule_1%Nb, &
                                  & w=Molecule_1%w, &
                                  & m=Molecule_1%m)

  CALL MolecCav_Construct_Operator(Operator=x_ho_molecule_1, &
                                  & operator_type="Position", &
                                  & scalar_space="Real", &
                                  & matrix_shape_type="Opt", &
                                  & Nb=Molecule_1%Nb, &
                                  & w=Molecule_1%w, &
                                  & m=Molecule_1%m)

  CALL MolecCav_Construct_Operator(Operator=N_ho_molecule_1, &
                                  & operator_type="Nb_photons", &
                                  & scalar_space="Real", &
                                  & matrix_shape_type="Opt", &
                                  & Nb=Molecule_1%Nb, &
                                  & w=Molecule_1%w, &
                                  & m=Molecule_1%m)
CALL MolecCav_Action_Total_Hamiltonian_1D(Result_total_WF, Matter_hamiltonianSystem_WF, &
                                         & H_ho_cavity_mode_1, x_ho_cavity_mode_1, &
                                         & Matter_dipolar_moment, System_WF, &
                                         & Cavity_mode_1%lambda, Cavity_mode_1%w)
```

Figure 12: Application program

2.11 The compilation

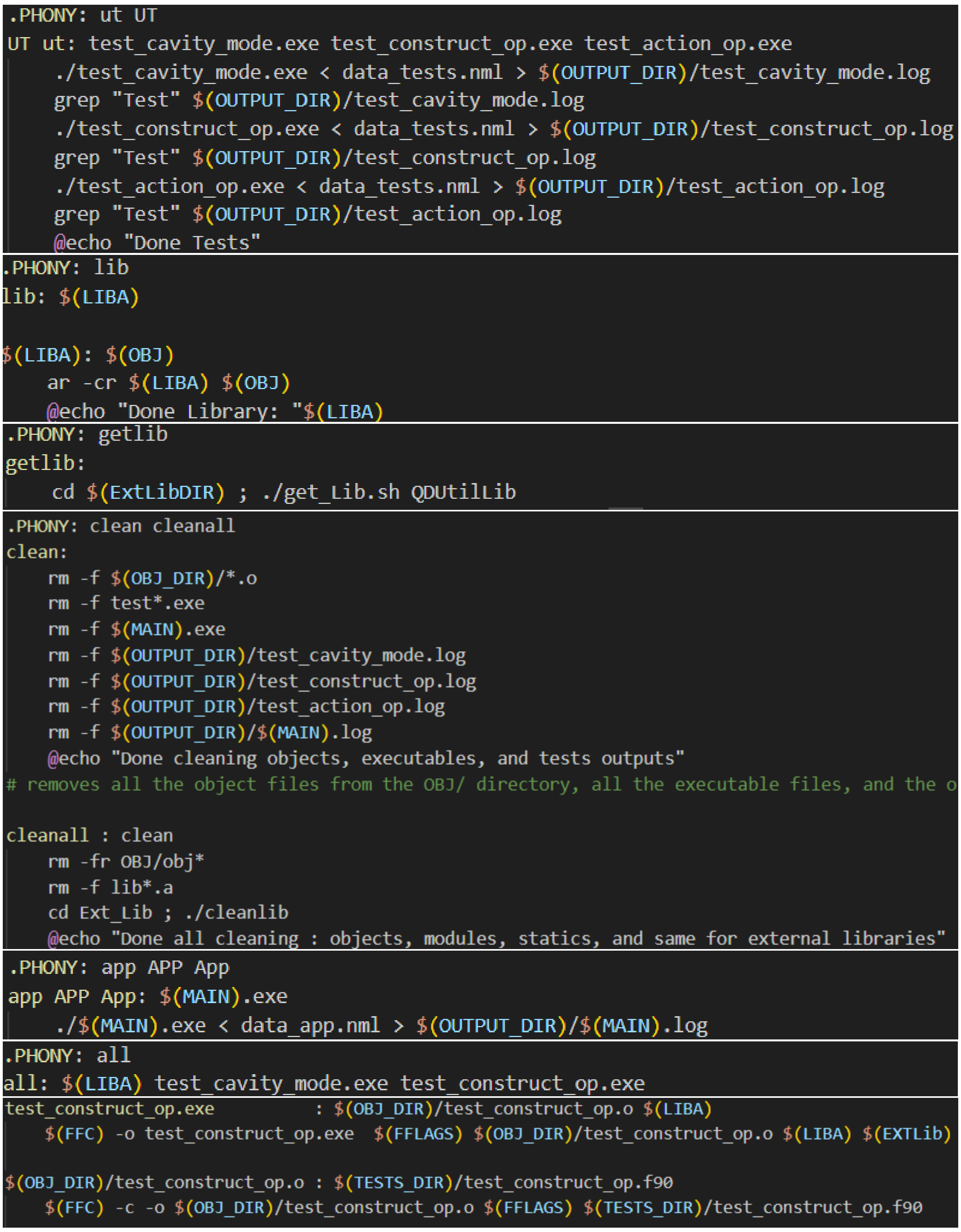
As mentioned before, two ways have been designed to ease the manipulation of the library : the makefile and the run.sh shell script. Both can be used according to the user's preferences and despite some small differences in the features, use the same commands and perform the same actions. These commands are illustrated as writtent in the makefile in Fig. 13 and are the following :

- **getlib** : Executes the get_Lib.sh script from the QDUtilLib external library directory. It will check if the installation of this library is needed and, if it is, will download and install it from its GitHub repository [6].
- **lib** : Compiles the external library's modules and build the .a static library file if needed, then does the same for the MolecCav library.
- **all** : Performs the same actions as **lib** in addition also to compile the source files of the tests and link them with the static library file into executables.
- **ut** : Performs the same actions as **all** in addition also to execute the tests, using the indicated data file and direct the output into the corresponding .log output file. Then it grabs the final sentence of these files for each test and display them on the screen. In both makefile and run.sh, it is the default command if no arguments are passed when they are called.

- **app** : Performs the same actions as **all** in addition also to compile the application source file, link it with the library into an executable and execute it using the indicated data file and direct the output into the corresponding .log output file.
- **clean** : Deletes all the object, executable and output files from MolecCav (not those of the external libraries).
- **cleanall** : Performs the same cleaning as **clean** in addition also to delete the .mod and static library files, and run the cleanlib shell scrip from the external QDUtilLib library that takes the same actions upon its files.

2.11.1 The makefile

This file is structured into several sections, that are the definition of some useful variable name to manage the different files, the definition of the commands, and the definition of the dependencies between the files. The main difference with the run.sh script is the latter section, which make it possible, every time a file is needed, to check the last modification date of this file and to build it again if and only if it is older than the files it depends on. That allows not to compile the whole library every time a command is called. The Fig. 13 illustrates how the commands are written in this file.

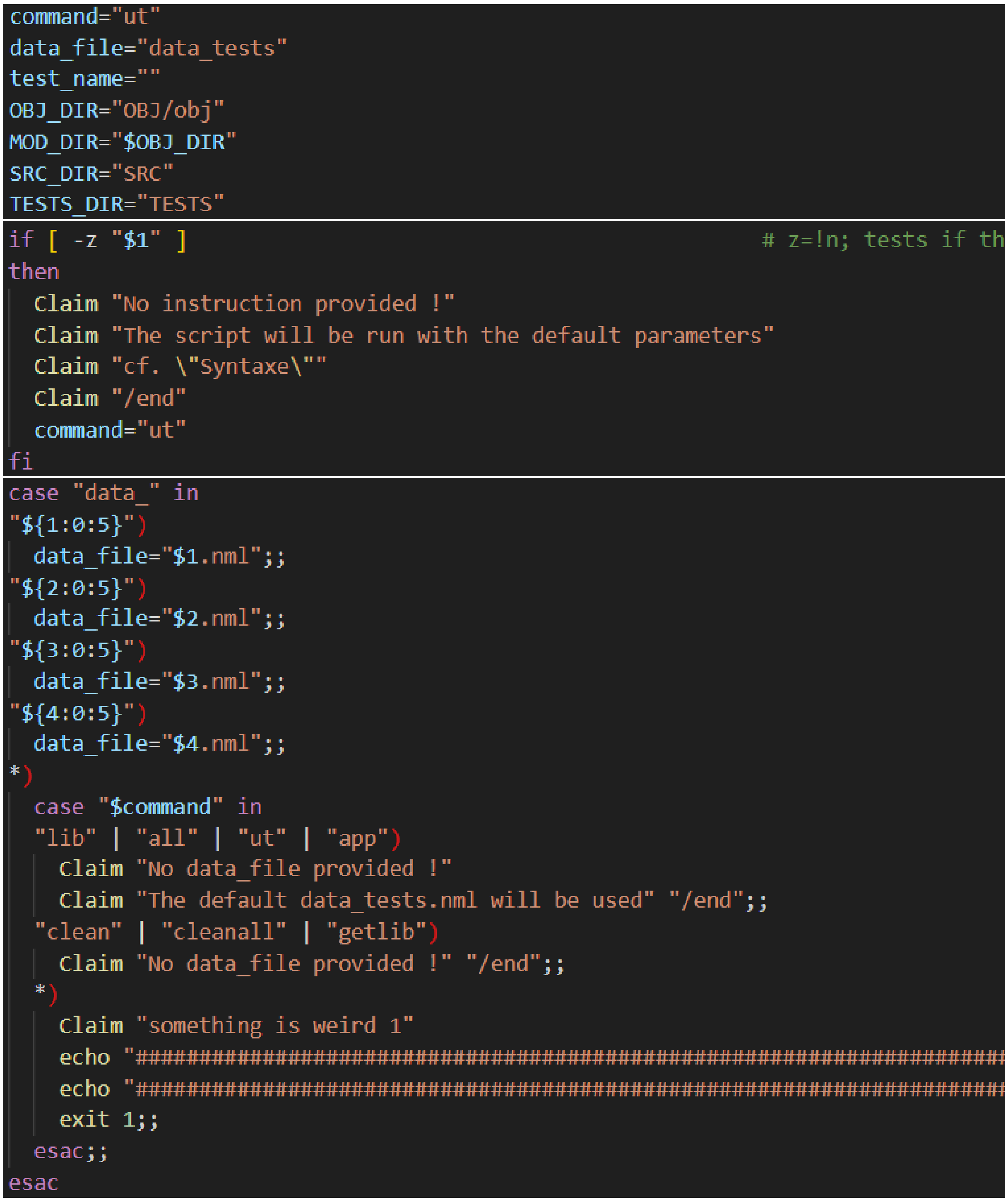


```
.PHONY: ut UT
UT ut: test_cavity_mode.exe test_construct_op.exe test_action_op.exe
    ./test_cavity_mode.exe < data_tests.nml > $(OUTPUT_DIR)/test_cavity_mode.log
    grep "Test" $(OUTPUT_DIR)/test_cavity_mode.log
    ./test_construct_op.exe < data_tests.nml > $(OUTPUT_DIR)/test_construct_op.log
    grep "Test" $(OUTPUT_DIR)/test_construct_op.log
    ./test_action_op.exe < data_tests.nml > $(OUTPUT_DIR)/test_action_op.log
    grep "Test" $(OUTPUT_DIR)/test_action_op.log
    @echo "Done Tests"
.PHONY: lib
lib: $(LIBA)

$(LIBA): $(OBJ)
    ar -cr $(LIBA) $(OBJ)
    @echo "Done Library: "$(LIBA)
.PHONY: getlib
getlib:
    cd $(ExtLibDIR) ; ./get_Lib.sh QDUtilLib
.PHONY: clean cleanall
clean:
    rm -f $(OBJ_DIR)/*.o
    rm -f test*.exe
    rm -f $(MAIN).exe
    rm -f $(OUTPUT_DIR)/test_cavity_mode.log
    rm -f $(OUTPUT_DIR)/test_construct_op.log
    rm -f $(OUTPUT_DIR)/test_action_op.log
    rm -f $(OUTPUT_DIR)/$(MAIN).log
    @echo "Done cleaning objects, executables, and tests outputs"
# removes all the object files from the OBJ/ directory, all the executable files, and the o

cleanall : clean
    rm -fr OBJ/obj*
    rm -f lib*.a
    cd Ext_Lib ; ./cleanlib
    @echo "Done all cleaning : objects, modules, statics, and same for external libraries"
.PHONY: app APP App
app APP App: $(MAIN).exe
    ./$(MAIN).exe < data_app.nml > $(OUTPUT_DIR)/$(MAIN).log
.PHONY: all
all: $(LIBA) test_cavity_mode.exe test_construct_op.exe
test_construct_op.exe          : $(OBJ_DIR)/test_construct_op.o $(LIBA)
    $(FFC) -o test_construct_op.exe  $(FFLAGS) $(OBJ_DIR)/test_construct_op.o $(LIBA) $(EXTLib)

$(OBJ_DIR)/test_construct_op.o : $(TESTS_DIR)/test_construct_op.f90
    $(FFC) -c -o $(OBJ_DIR)/test_construct_op.o $(FFLAGS) $(TESTS_DIR)/test_construct_op.f90
```

Figure 13: Makefile

2.11.2 The run.sh script

The sections that structure this file are the same except for the definition of some functions at the beginning. Contrary to the makefile, this script does not check the creation dates of the files before it builds them. Therefore all the library is built again each command, except for the external library that is managed by its own scripts. However it may offer a little more flexibility in some particular situations. It makes it possible for instance to choose a different OBJ/obj file to store the object files, choose a different data file than the default one whatever program is executing, or choose to execute only one test to compile and execute instead of all. The Fig. 14 illustrates how are recovered the arguments from command-line.



```
command="ut"
data_file="data_tests"
test_name=""
OBJ_DIR="OBJ/obj"
MOD_DIR="$OBJ_DIR"
SRC_DIR="SRC"
TESTS_DIR="TESTS"
if [ -z "$1" ]                                       # z=!n; tests if th
then
  Claim "No instruction provided !"
  Claim "The script will be run with the default parameters"
  Claim "cf. \"Syntaxe\""
  Claim "/end"
  command="ut"
fi
case "data_" in
"${1:0:5}")
  data_file="$1.nml";;
"${2:0:5}")
  data_file="$2.nml";;
"${3:0:5}")
  data_file="$3.nml";;
"${4:0:5}")
  data_file="$4.nml";;
*)
  case "$command" in
  "lib" | "all" | "ut" | "app")
    Claim "No data_file provided !"
    Claim "The default data_tests.nml will be used" "/end";;
  "clean" | "cleanall" | "getlib")
    Claim "No data_file provided !" "/end";;
  *)
    Claim "something is weird 1"
    echo "####################################################################
    echo "####################################################################
    exit 1;;
  esac;;
esac
```

Figure 14: run.sh script

3 Acknowlegments

- David LAUVERGNAT

4 Biblography

[1] *Jaynes–Cummings model*. en. Page Version ID: 1256540074. Nov. 2024. URL: https://en.wikipedia.org/w/index.php?title=Jaynes%E2%80%93Cummings_model&oldid=1256540074 (visited on 01/16/2025).

[2] Thomas Schnappinger and Markus Kowalewski. "Nonadiabatic Wave Packet Dynamics with Ab Initio Cavity-Born-Oppenheimer Potential Energy Surfaces". In: *Journal of Chemical Theory and Computation* 19.2 (Jan. 2023). Publisher: American Chemical Society, pp. 460–471. ISSN: 1549-9618. DOI: 10.1021/acs.jctc.2c01154. URL: https://doi.org/10.1021/acs.jctc.2c01154 (visited on 01/08/2025).

[3] David Lauvergnat, Peter Felker, Yohann Scribano, David M. Benoit, and Zlatko Bačić. "H2, HD, and D2 in the small cage of structure II clathrate hydrate: Vibrational frequency shifts from fully coupled quantum six-dimensional calculations of the vibration-translation-rotation eigenstates". In: *The Journal of Chemical Physics* 150.15 (Apr. 2019), p. 154303. ISSN: 0021-9606. DOI: 10.1063/1.5090573. URL: https://doi.org/10.1063/1.5090573 (visited on 01/08/2025).

[4] matheosgd. *matheosgd/MolecCav*. original-date: 2024-11-02T15:59:58Z. Jan. 2025. URL: https://github.com/matheosgd/MolecCav (visited on 01/16/2025).

[5] Eduarda Sangiogo Gil, David Lauvergnat, and Federica Agostini. "Exact factorization of the photon–electron–nuclear wavefunction: Formulation and coupled-trajectory dynamics". In: *The Journal of Chemical Physics* 161.8 (Aug. 2024). Publisher: American Institute of Physics. DOI: 10.1063/5.0224779. URL: https://hal.science/hal-04730124 (visited on 01/08/2025).

[6] lauvergn. *lauvergn/QDUtilLib*. original-date: 2022-12-21T16:31:41Z. Nov. 2024. URL: https://github.com/lauvergn/QDUtilLib (visited on 01/18/2025).