Nautilus Documentation

July 30, 2014

Christophe Cossou

Contents

1   TODO

- More details on `grain_tunneling_diffusion` flag need to be added by someone who understand it

- More details on `modify_rate_flag` flag need to be added by someone who understand it

- More details on `conservation_type` flag need to be added by someone who understand it.

- idem for `is_absorption`

- Reference for cosmic ray ionization rate standard value?

- What is the reference UV flux? Paper describing it?

- X ionization rate set to 0, is it normal?

2   Starting with Nautilus

Whether you are a developer or a user, you do not have the same expectations about this manual.

[§ 2] explains the basics, mainly how to get the repository, how to compile the simulation code, how to use the various tools. Be sure to read carefully the section about **Installation** [§ 2.3 on the next page].

[§ 4 on page 13] explains the chemical network, its format, and the various types of reactions available. A chemical network is given with the code, but you can use your own.

Input files are presented in [§ 2.5 on page 7], but the most important one, the only one you will modify daily (*parameters.in*) is dealt with in [§ 3 on page 8].

Output files are presented in [§ 5 on page 14]. Simulations information are displayed in *info.out* (see [§ 5.1 on page 14]).

[§ 6 on page 14] explain what are the Python scripts you can use to plot useful information (mainly abundances) of your simulations (single plot or comparison between several runs).

One last section [§ 7 on page 15] present technical specifications of the code and how to maintain it. For developers who wants more detailed explanations about the code, please refer to the Doxygen documentation, available in the Git repository at `http://nautilus.googlecode.com/git/html/index.html` or in the **html** repository of any local Git clone.

2.1   Generic informations

2.1.1   Python scripts help

From all the python script that come with the code, you can see all parameters and sometimes a few examples by typing:

```
script_name.py help
```

2.1.2   Comments in input files

For all input files, the comment character is " !" and can be either at the beginning of a line, or anywhere else. Meaningful character must be comprised inside the 80 first characters of a given line.

2.1.3   Main parameter file

parameters.in is the main parameter file, and is rewritten by the code itself each and every time you run the code.

2.2   Getting the Git repository

First, you need to get the Git repository from *Googlecode* before actually using the code.

I assume you only want to use it, without ever putting on the distant repository anything. This way, you don't even need to have an account and a password.

In the parent directory where you want to put a **nautilus** folder containing the Git repository, type:

```
git clone https://code.google.com/p/nautilus/
```

Be sure to run once in a while the command to get distant *updates* if they exists:

```
git pull
```

2.3   Installation

Once you have the Git repository, I advice you to run the script *configure_nautilus.py* :

```
configure_nautilus.py
```

This script will add useful shortcuts in the *.bash_profile*. When updating your Git repository, the bash commands will also be updated automatically into the *.bash_profile*.

⚠ Nautilus is a file browser in GNU/Linux (Gnome) environments. Thus, You must be *very* careful. If you use:

```
nautilus
```

in a terminal, you will not run the nautilus code, but rather the file browser (which you do not want, especially in a remote server, I assure you).

The improvements brought by this script are:

- You can go to the Nautilus folder by typing:

  ```
  cd $nautilus
  ```

- You can launch the code, the output code and the rates code by these three commands, regardless of the folder you currently are:

  ```
  nautilus_code
  nautilus_outputs
  nautilus_rates
  nautilus_major_reactions
  ```

- You have now useful information about Git repositories directly into the Bash prompt.

- all scripts in the folder **scripts/** can be run from any folder. This include:

  ```
  nautilus-clean.sh # Cleaning a simulation folder
  nautilus-plot-abundances.py # plot abundances of a simulation
  nautilus-compare-abundances.py # compare two simulations or more.
  ```

Remark :   All of this will have little interest for you if you don't use 'Bash' as the interpreter for you terminal.
   You can also check the options of this script. Especially, you can undo the installation.

2.4   Compilation

The script *Makefile.py* allow you to compile the code (see [§ 2.1.1 on the previous page] for infos about Python scripts). The default compiler is *gfortran*.
   If you want to just compile the code[1], type:

```
Makefile.py
```

Remark :   Errors are in ***.log** files associated with the module incriminated. Warnings are stored in a generic file *compilation.log*.

---

[1]By default, all the binaries are compiled

⚠ By default, all warnings issued by *ODEPACK* routines are masked (to increase readability, since nobody can modify this thing from scratch). An option **opkd** can force their display.

If you want to compile *only* one binary, use respectively those command:

```
Makefile.py nautilus
Makefile.py output
Makefile.py rates
Makefile.py major
```

One particular option exist:

```
Makefile.py test
```

This option **test** use compilation options that allow comparison of the code between different versions and is to be used for *compare_simulations.py* (see [§ 7.2 on page 16]).

Remark : Mainly, this option avoid too hard optimization that can result in slightly different results when the code evolve (not because the maths changes but because of optimization only).

⚠ If you change compilation options, pay attention to use the option **force** to ensure all the modules will be compiled using these options.

2.5 Input files

All input files have the same ***.in** extension. An example simulation, containing all necessary input files is provided in the sub-folder **example_simulation**.

The main parameter file is *parameters.in* (see [§ 3 on the next page]).

abundances.in give initial abundances for a set of species. Default minimum values are applied to the species not present in this file (this value is set by the parameter *minimum_initial_abundance* in *parameters.in* (see [§ 3 on the following page]).

element.in gives information about prime elements (base elements used to construct molecules) existing in the simulation (name and mass in Atomic mass unit).

There are 2 parameter files listing all reactions in a given phase (gas or grain):

- *gas_reactions.in*

- *grain_reactions.in*

and 2 parameter files for species present in a given phase (gas or grain) reactions:

- *gas_species.in*

- *grain_species.in*

Remark : Species in *gas_species.in*, *grain_species.in* are not necessarily species from one phase. These are species *involved* in a given phase reaction. But adsorption or desorption transform grain species into gas species, and vice versa.

activation_energies.in provide activation energies for some endothermic reactions.

surface_parameters.in provide information about various energies and parameters for diffusion and movements on the grain surface.

2.6 Usefull tools

2.6.1 Cleaning a simulation folder

The script *nautilus-clean.sh* helps you delete all output files to have a clean simulation folder, with input only

To clean the current working directory, launch:

```
nautilus-clean.sh
```

2.6.2 Installation script

The script *configure_nautilus.py* will configure the bash profile to add several shortcuts (You only have to run the command once). More details are available in [§ 2.3 on page 6].

3 Parameter file : parameters.in

For generic informations, see [§ 2.1.2 on page 5].

parameters.in has the particularity to be re-written each time you launch a Nautilus simulation. This ensure several things :

- Parameters can be input in random ways, the code will sort them by categories

- New parameters, with default values will be added, to ensure retro-compatibility.

- A ***.bak** file is created before overwriting **parameters.in**, just in case it erases something important (in comments for instance).

3.1 Automatic test before computation

In *parameters.in*:

```
preliminary_test = 1
```

When set to 1, the parameter *preliminary_test* will allow you to test thoroughly the chemical network and print information in the file *info.out*.

> **Remark :** This parameter should be set to 0 only in the case of intensive campaign of simulations using the very same network, to avoid wasting computation time doing the sames tests. But in any other cases, I advice you to leave it activated, because it only take around 1 second at the beginning of the simulation.

The tests currently made are:

- Check that grain species judging from their indexes are indeed grain species

- Check that all species have production AND destruction reactions (error if none, warning if only one)

- Check that each reaction is balanced in prime element and in electric charge

- Display a warning for each reaction having alpha=0 (first parameter for reaction rate formula)

- Check that $T_{\min} < T_{\max}$ for each reaction

- For reaction with the same ID:

  - Check that they have the same reactants and products
  - Check that temperature ranges do not overlap.

- Check that each gas neutral species have a grain equivalent (excluding **GRAIN0** and **XH**).

- Check that each gas neutral species has an adsorption reaction (ITYPE=99) (excluding **GRAIN0** and **XH**).

- Check that each grain species has at least one reaction of each of the following types: 15, 16, 66, 67 (desorption reactions).

- Check that all index ranges associated with a reaction type join themselves to cover all the index range of all reactions.

3.2   Simulation parameters

```
start_time =  1.000E+00
```

In years, the first output time of the simulation (all simulation starts from $T = 0$).

```
stop_time =  1.000E+01
```

End of the simulation in years (and also the last output time)

```
nb_outputs =    3
```

Total number of outputs (including *start_time* and *stop_time*). This number will be used when *output_type* is **log** or **linear**.

```
output_type = log
```

Define the type of output you want. Possible values are **linear**, **log**, and **table**.

- **linear** : The spacing between the different output times will be linear

- **log** : The different output times will be log-spaced.

- **table** : The different output times are read from the file *structure_evolution.dat*. The parameter *nb_outputs* is then completely ignored.

```
relative_tolerance =  1.000E-04
```

Relative tolerance of the solver

```
minimum_initial_abundance =  1.000E-40
```

default minimum initial fraction abundance applied to species whose abundance is not specified in *abundances.in*.

3.3   Time evolution of the physical structure

```
is_structure_evolution = 1
```

If set, the physical structure we define will evolve with time. This evolution will be read from the file *structure_evolution.dat* that must exist in the simulation folder.
   This file will have the following format:

```
! time      log(Av)     log(n)      log(T)
! (Myr)   log(mag)  log(cm-3)  log(K)
0.000e+00 -1.231e+00 1.813e+00 1.698e+00
2.360e-01 -1.233e+00 1.758e+00 1.712e+00
```

We define respectively time, visual extinction, gas density and gas temperature.
   Optionally, one can add a 5-th column to define also grain temperature:

```
! time      log(Av)     log(n)      log(Tg)      log(Td)
! (Myr)   log(mag)  log(cm-3)  log(K)      log(K)
0.000e+00 -1.231e+00 1.813e+00 1.698e+00 1.500e+00
2.360e-01 -1.233e+00 1.758e+00 1.712e+00 1.510e+00
```

If so, the parameter *grain_temperature_type* must be set to:

```
grain_temperature_type = table
```

⚠ See [§ 3.4 on the next page] for incompatibilities

3.4   1D simulations

You can make simulation in 1D:

```
structure_type = 1D_disk_z
```

Then you must define the total number of spatial points:

```
1D_sample = 10
```

and the altitude of the farthest point (in AU):

```
z_max =  1.000E+01
```

Spatial points will then be linearly and equally spaced.

You have to define the structure in a data file *1D_evolution.dat* where you define gas density, gas temperature, visual extinction and diffusion coefficient. For instance:

```
! Distance [AU] ; Gas density [part/cm^3] ; Gas Temperature [K] ;
! Visual Extinction [mag] ; Diffusion coefficient [cm^2/s]
0.          150.          10.          15.          1e14
2.          150.          10.          15.          1e14
4.          300.          15.          20.          1e14
6.          150.          10.          15.          1e14
```

The spatial points in the data file doesn't necessarily need to match the one in the code. We will interpolate linearly to reconstruct the values at the spatial points defined in the code. One can simply put extremal values and the code will create the one in-between.

The dust temperature is ruled by *grain_temperature_type* (see [§ 3.5] for more details), even if **table** is not allowed here since 1D and time evolution of the structure are incompatible.

⚠ 1D simulations are not compatible with time evolution read from a data file (see [§ 3.3 on the previous page]). You must choose one or the other.

3.5   Grain temperature

You have four ways of defining grain temperature in the code. The parameter *grain_temperature_type* can have the following values:

fixed The grain temperature is fixed throughout the simulation. *initial_dust_temperature* defines this fixed value ;

gas The grain temperature is equal to the gas temperature, no matter what.

computed The grain temperature is calculated following an energy equilibrium with the gas in the structure

table The grain temperature is interpolated from the 5-th column of the *structure_evolution.dat* file. *is_structure_evolution* must be set to 1.

3.6   Switches

To activate (or not) accretion on dust grains and grain surface reactions:

```
is_grain_reactions = 1
```

To activate (or not) the self-shielding of H_2 and CO, related to visual extinction:

```
is_absorption = 1
```

To activate (or not) grain tunneling diffusion and choose the type of grain tunneling diffusion:

```
grain_tunneling_diffusion = 0
```

Different types are:

0 : Thermal for H, H_2

1 : Quantum tunneling diffusion rate $[\mathrm{s}^{-1}]$ [Watson, 1976]

2 : Quantum tunneling diffusion rate $[\mathrm{s}^{-1}]$ [Hasegawa et al., 1992]

3 : Choose fastest

To choose whether or not we can modify some rates:

```
modify_rate_flag = 1
```

Different types are:

-1 : H+H only

1 : modify H

2 : modify H,H_2

3 : modify all

To choose whether or not we can modify some abundances (this doesn't work in 1D, because of diffusion !):

```
conservation_type = 0
```

Different types are:

0 : Only electrons conserved

1 : element #1 conserved

2 : element #1 and #2 conserved

n : element #1...#n conserved

3.7 Gas parameters

```
initial_gas_density =  2.000E+04
```

initial gas density in particule/cm^3. Sometimes, there can be a confusion between two things. First, the gas density in particule per volume, and second, the proton density. We have:

$$n_{\mathrm{proton}} = n(\mathrm{H}) + 2 \times n(\mathrm{H}_2) \sim 2 \times n(\mathrm{H}_2)$$
$$n_{\mathrm{particule}} = n(\mathrm{H}) + \times n(\mathrm{H}_2) \sim n(\mathrm{H}_2)$$

We then have, assuming that the major reservoir for H is H_2:

$$n_{\mathrm{proton}} \simeq 2 \times n_{\mathrm{particule}} \tag{3.1}$$

Here, the gas density is $n_{\mathrm{particule}}$.

```
initial_gas_temperature =  1.000E+01
```

initial gas temperature in K

```
initial_visual_extinction =  1.500E+01
```

initial visual extinction in magnitude

```
cr_ionisation_rate =  1.300E-17
```

cosmic ray ionization rate in s^{-1}. A standard value is $1.3 \cdot 10^{-17}$ (TODO ref ?).

```
x_ionisation_rate =  0.000E+00
```

Ionisation rate due to X-rays in s^{-1}.

```
uv_flux =  1.000E+00
```

Scale factor for the UV flux, in unit of the reference flux. By choosing 1, you will use the nominal value.

3.8   Grain parameters

`initial_dust_temperature =  1.000E+01`

initial dust temperature in K, used when *grain_temperature_type* is **fixed**.

`initial_dtg_mass_ratio =  1.000E-02`

Total mass of dust divided by total mass of gas (dimensionless).

`sticking_coeff_neutral =  1.000E+00`

sticking coefficient for neutral species

`sticking_coeff_positive =  0.000E+00`

sticking coefficient for positive species

`sticking_coeff_negative =  0.000E+00`

sticking coefficient for negative species

`grain_density =  3.000E+00`

mass density of grain material in g/cm^3

`grain_radius =  1.000E-05`

grain radius in cm.

`diffusion_barrier_thickness =  1.000E-08`

Thickness of the barrier in cm that a surface species need to cross while undergoing quantum tunneling to diffuse from one surface site to another. This is used in the formalism [Hasegawa et al., 1992, see equation 10 (parameter a)].

`surface_site_density =  1.500E+15`

density of sites at the surface of the grains in $number/cm^2$.

`diff_binding_ratio =  5.000E-01`

Ratio (adimensioned) used to compute the DIFFUSION_BARRIER from the BINDING_ENERGY if not known

`chemical_barrier_thickness =  1.000E-08`

Parameter (in cm) used to compute the probability for a surface reaction with activation energy to occur through quantum tunneling. This is the thickness of the energy barrier [Hasegawa et al., 1992, See equation 6].

`cr_peak_grain_temp =  7.000E+01`

Peak grain temperature in K when struck by a cosmic ray.

`cr_peak_duration =  1.000E-05`

duration [s] of peak grain temperature

`Fe_ionisation_rate =  3.000E-14`

(cosmic) Fe-ion–grain encounter $[s^{-1}grain^{-1}]$ for 0.1 micron grain. For cosmic photo desorptions, only Fe-ions are efficient to heat grains.

`vib_to_dissip_freq_ratio =  1.000E-02`

(dimensionless) For the RRK (Rice Ramsperger-Kessel) desorption mechanism. Ratio of the vibration frequency (proper energy of a species when it is created on a grain) to the dissipation frequency (energy needed by the molecule to be evaporated from the grain surface). This ratio help to determine if a species evaporate after its formation on the grain surface. Since the dissipation frequency is usually unknown, this ratio is a free parameter. A common value is 1%.

4 Chemical network

4.1 Reaction files

The files concerned are : *gas_reactions.in*, *grain_reactions.in* and *activation_energies.in*

A typical reaction line is:

```
1  !    Reactants                    ->      Products
                                         A          B          C
      xxxxxxxxxxxxxxxxxxxxx ITYPE Tmin   Tmax formula ID xxxxx
2  C+        CH2OH              -> CH2+      HCO
                                      7.500E-10 -5.000E-01  0.000E+00 0.00e
      +00 0.00e+00   NA  4     10    280  3  6098 1  1
```

The reaction displayed here is:

$$C^+ + CH_2OH \rightarrow CH_2^+ + HCO$$

The temperature range is $T \in [10; 280]$ K. The type of reaction is 4. The ID of the reaction is 6098. The formula used to compute reaction rate is 3, with the 3 parameters $A = 7.5 \cdot 10^{-10}$, $B = -0.5$ and $C = 0$. Other columns are ignored, as the "xxx" emphasize in the legend line associated.

Each species name is encoded with 11 characters.

All reaction files have the same format. Depending on the evolution of the code, the number of reactants (`MAX_REACTANTS`) or products (`MAX_PRODUCTS`) may vary (increase), so theses files must be modified to take that into account.

The following global variables (in the source code) are here to tell to the code that theses number have changed.

```
1  MAX_REACTANTS = 3 !< The maximum number of reactants for one reaction.
2  MAX_PRODUCTS = 5 !< The maximum number of products for one reaction.
3  MAX_COMPOUNDS = MAX_REACTANTS + MAX_PRODUCTS !< Total maximum number of compounds for
        one reaction (reactants + products)
```

⚠️ Pay attention to the fact that some things might need manual modifications in the code. If this number change, `get_jacobian(N, T, Y, J, IAN, JAN, PDJ)` must be actualized, since each reactant and product has its own variable, a new one must be created for the new column possible.

4.2 Reaction types

Chemical reactions can be of several types.

Here is the list:

0 Gas phase reactions with GRAINS

1 Photodissociation/ionisation with cosmic rays (CR)

2 Gas phase photodissociations/ionisations by secondary UV photons generated by CR

3 Gas phase photodissociations/ionisations by UV

4-8 Bimolecular gas phase reactions - several possible formula

10-11 H_2 formation on the grains when `IS_GRAIN_REACTIONS=0`

> **Remark** : Only one reaction for each of the two types (10 and 11).

14 Grain surface reactions

15 Thermal evaporation

16 Cosmic-ray evaporation

17-18 Photodissociations by Cosmic rays on grain surfaces

19-20 Photodissociations by UV photons on grain surfaces

21 Grain surface reactions

66 Photodesorption by external UV

67 Photodesorption by UV induced by cosmic rays

98 storage of H_2S under a refractory form

99 Adsorption on grains

5   Outputs

⚠ All output files have the same ***.out** extension.

Output files are:

- *info.out*: Various information about the simulation (see [§ 5.1] for more details)

- *species.out*: The list of species and their corresponding index

- *elemental_abundances.out*: The prime elements abundances and mass at the beginning of the simulation. A ***.tmp** version display the same infos, but at the last output.

- *abundances.*.out*: In binary format (unformatted), abundances of all species, each file for a different output time. *nautilus_outputs* read theses files to generate ASCII files (see [§ 5.2] for more details).

- *rates.*.out*: In binary format (unformatted), rates of all reactions, each file for a different output time. *nautilus_rates* and *nautilus_major_reactions* analyse theses files.

- *abundances.tmp*: The abundances of all species at the last output in ASCII.

5.1   Informations : info.out

In the file *info.out*, the ID reference of the current version of nautilus, used to run the simulation is printed, as well as other useful information about the state of Nautilus and how the simulation was executed.

Information and warnings about coherence of the chemical network are also printed here.

5.2   Abundances

Files are named *abundances.000001.out* and so on, for each output time. Outputs are stored in binary format. Binary format is not detailed here. If you want details about variables in there, please refer to the Fortran routine **write_current_output** in the file **input_output.f90**

To get ASCII files from the binaries, one must compile the designed program (in the Git repository):

```
Makefile.py output
```

Then, in your simulation folder, type:

```
nautilus_outputs
```

(this assume you have an alias, but absolute path also work)

This program will generate **.ab* files in a sub-folder **ab** of the simulation folder. If in 1D, a file space.ab will store the spatial points. Indeed, each species file will now have one column for time, and one column per spatial point.

The program will also generate **.struct* files in a sub-folder **struct** of the simulation folder, one file per spatial point.

6   Graphic display

Python script were created to help the user display useful information about their simulations.

6.1   Plot abundances

The script *nautilus-plot-abundances.py* allow you to display the time evolution of the abundances of one or more species:

```
nautilus-plot-abundances.py species=CO,H20
```

You can zoom in a given period of time (**tmin**, **tmax** or both):

```
nautilus-plot-abundances.py species=CO,H20 tmin=1e4 tmax=1e6
```

Even if you can modify and store the result in the graphic windows displayed, a default version is automatically written in **abundances.pdf**

Check the other options and detailled examples via:

```
nautilus-plot-abundances.py help
```

6.2   Compare abundances

The script *nautilus-compare-abundances.py* allow you to display the time evolution of the abundances of one or more species for all sub-folders of the current working directory, assuming each one is a simulation:

```
nautilus-compare-abundances.py species=CO,H20
```

If there is a lot of sub-folder, you can select those you want by:

```
nautilus-compare-abundances.py species=CO,H20 dir=simu1,simu2
```

All option existing for *nautilus-plot-abundances.py* applies here.

Even if you can modify and store the result in the graphic windows displayed, a default version is automatically written in **compare_abundances.pdf**

Check the other options and detailed examples via:

```
nautilus-compare-abundances.py help
```

7   For developpers

Constants and global variables are defined in the module **global_variable.f90**. **nautilus_main.f90** contains all the main routines.

The four main programs are:

- *nautilus* in the source file **nautilus.f90** (compilation: `Makefile.py`)

- *nautilus_outputs* in the source file **nautilus_outputs.f90** (compilation: `Makefile.py output`)

- *nautilus_rates* in the source file **nautilus_rates.f90** (compilation: `Makefile.py rates`).

- *nautilus_major_reactions* in the source file **nautilus_major_reactions.f90** (compilation: `Makefile.py major`).

One last program exist to do some unitary tests on **Nautilus** (see [§ 7.1 on the following page] for more details).

A lot of routines are handled by pointers. This allow us to change easily from one routine to another in function of the parameters. Thus, some routine names (in **call**) might not exists *as is*. They are defined in **global_variable.f90**. For each pointer, an interface is defined that constrains the architecture of each subroutine he can point to.

7.1   Unitary tests

A **fortran** program *unitary_tests* (**unitary_tests.f90**) was specifically designed to test some routines of the code separately. This is the main reason why a **nautilus_main.f90** file was created, because we need to access these routines separately from the *nautilus* program.

To run the unitary tests, use the Python script designed for that (he will compile the source code too):

```
unitary_tests.py
```

The code will display some information and ask you what test you want to display graphically (and generate the corresponding **.pdf**:

```
0 : av_interpolation
1 : density_interpolation
2 : gas_temperature_interpolation
3 : grain_temperature_interpolation
4 : test_av_read
5 : test_density_read
6 : test_gas_temperature_read
What test do you want to display? (0-6 ;
'all' treat them all ; 'l' display list again)
```

The principle of this code is to do some tests, store the results in data files, generate Gnuplot script files associated. To get the plots, you only have to generate it by:

```
gnuplot script_name.gnuplot
```

All files generated by the program are stored in the "tests" sub-folder. You can generate the .pdf manually if you are familiar with Gnuplot, and of course, view them separately.

> **Remark** : It's up to you to write new routines in **unitary_tests.f90** and mimic the way I wrote the previous one to tests new functionality of the code.
>   Please keep in mind that you need to add a `call new_routine()` in the main program, just before `contains` to ensure your routine is executed.

7.2   Check before commit

You can have two types of modifications in the code, those that modify the outputs, and those that do not.

I created a Python script *compare_simulations.py* that can help you ensure the code do not change the results between two versions. It's up to you to check your modifications when the outputs are different though.

The basic use of this script is as follow (in the code main directory):

```
Makefile.py test && compare_simulation.py
```

If you want to update the "reference" version of the code, because outputs changed and you want a new reference, type:

```
compare_simulation.py rev=HEAD
```

rev can accept any commit reference, **HEAD**, **HEAD^** and a hashtag will work.

7.3   How to write documentation with Doxygen

7.3.1   General informations

Doxygen comments generally have a marker and the description. The character to declare the marker is @, but one need to start the commented line by > to declare there is something here.

If this is an inline comment, with the code on the left, comments are like this:

```
the code !< the doxygen description
```

just to say that the description refers to the code on the left.

To continue a description on another comment line, just double the comment character:

```
!> @brief I describe something
!! on several lines.
```

> **Remark** : This will not make two lines in the generated documentation though. This is only a way to avoid never ending lines with thousands of characters.

7.3.2 For a module

Start the module file with:

```
1  !*********************************************************************************
2  ! MODULE: Module Name
3  !*********************************************************************************
4  !
5  !> @author
6  !> Module Author Name and Affiliation
7  !
8  ! DESCRIPTION:
9  !> @brief Brief description of what can be done in this module.
10 !! This description can be on several lines.
11 !! \n\n Do not forget the symbol "\n" to create a new line.
12 !
13 !*********************************************************************************
```

7.3.3 For a subroutine

Before the definition of the routine, add the following text:

```
1  !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  !> @author
3  !> Routine Author Name and Affiliation.
4  !
5  ! DESCRIPTION:
6  !> @brief Brief description of routine.
7  !! Flow method (rate of change of position) used by integrator.
8  !! Compute \f$ \frac{d\lambda}{dt} , \frac{d\phi}{dt}, \frac{dz}{dt} \f$ }
9  !
10 !%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

In the rest of the routine, do not forget to add comments for input and outputs of the routine:

```
1  implicit none
2
3  ! Inputs
4  real(double_precision), intent(in) :: delta_t !<[in] description
5
6  ! Outputs
7  integer, intent(out) :: istate !<[out] Description of the variable
8  !! that can be continued on another line.
9
10 ! Inputs/Outputs
11 real(double_precision), intent(inout) :: time !< [in,out] description
12 !! \n Continuation line.
13
14 ! Locals
15 real(double_precision) :: t !< The local time, starting from 0 to delta_t
```

7.4 How to generate Doxygen documentation

To generate **Doxygen** documentation, type the following command in the root Git repository:

```
doxygen doxygen.conf 2>doxygen.log
```

doxygen.conf is the parameter file of *Doxygen*. Errors will be redirected in the file **doxygen.log**.

⚠ The documentation (in the folder **html/**) is synchronized with the server repository. Thus, you may generate documentation only after major changes in the *Doxygen* documentation, to avoid too many changes for too few improvements.

7.5   Bash profile

The file *.nautilus_profile* in the Git repository contains Bash commands to give user friendly tools to use *Nautilus* in command line. By running *configure_nautilus.py*, a call to this file will be done in the *.bash_profile*. See [§ 2.3 on page 6] for more details.

Bibliography

T. I. Hasegawa, E. Herbst, and C. M. Leung. Models of gas-grain chemistry in dense interstellar clouds with complex organic molecules. ApJS, 82:167–195, September 1992. doi: 10.1086/191713.

W. D. Watson. Interstellar molecule reactions. *Reviews of Modern Physics*, 48:513–552, October 1976. doi: 10.1103/RevModPhys.48.513.

Index