

FastChem 2.0

User and Reference Guide

Daniel Kitzmann

Joachim Stock

Contents

1	Introduction to FastChem	4
1.1	Overview	4
1.2	Licence	4
1.3	About this guide	5
I	Installation of FastChem and pyFastChem	6
2	Installation	7
2.1	Prerequisites	7
2.2	Obtaining the code	7
2.2.1	PyBind11 Library	7
2.2.2	Supported compilers	8
2.3	Configuration and compilation of FastChem with CMake	8
2.3.1	Notes on MacOS	9
2.3.2	Notes on Windows	9
II	Running FastChem and pyFastChem	10
3	Standard FastChem input and output files	12
3.1	Element abundance file	12
3.2	Species file	14
3.3	Basic element data file (optional)	16
3.4	FastChem parameter file (optional)	17
3.5	Output files	18
4	The FastChem C++ stand-alone executable	20
4.1	Starting the FastChem executable	20
4.2	Config file	20
5	Running pyFastChem	23
5.1	Provided Python examples	23
5.2	Detailed steps for running FastChem with pyFastChem	23

III	Detailed C++ object class and Python module description	26
6	FastChem class	27
6.1	Some comments on coding conventions	27
6.2	FastChem object class	28
6.3	FastChem constants	28
6.4	FastChem constructor	29
6.5	Input and output structures	30
6.5.1	Input structure	30
6.5.2	Output structure	30
6.6	Methods of the fastchem::FastChem object class	32
7	pyFastChem: The Python module of FastChem	35
7.1	The pyFastChem module	35
7.2	pyFastChem constants	35
7.3	pyFastChem constructor	36
7.4	pyFastChem input and output structures	37
7.5	pyFastChem functions	39
	Bibliography	41

1 Introduction to FastChem

1.1 Overview

FastChem is an open-source computer program that can calculate the gas phase chemical equilibrium composition of general systems for a given temperature pressure and element abundances.

FastChem is an equilibrium chemistry model that can calculate the gas phase chemical composition of general systems. It uses a semi-analytical approach to solve the non-linear system of mass action law equations, which results in a massive increase in computational performance over other approaches like Gibbs minimisation. The general concept and the original version 1.0 is described by [Stock et al. \(2018\)](#) (Paper I). Version 1.0, however, is restricted to systems that are dominated by hydrogen and helium and required an additional iteration to account for the pressure of the system. The current version 2.0 can now be applied to arbitrary element compositions. This version will be described by [Stock et al. \(2020\)](#), from here on referred to as Paper II.

FastChem has already been applied to numerous different systems, from brown dwarfs ([Kitzmann et al., 2020](#)), to mini-Neptunes, hot-Jupiters ([Bourrier et al., 2020](#)), to ultra-hot Jupiters ([Hoeijmakers et al., 2019](#)). It is directly coupled to the retrieval model **Helios-r2** ([Kitzmann et al., 2020](#)), to the general atmospheric model **HELIOS** ([Malik et al., 2019](#)), and the non-equilibrium chemistry **VULCAN** ([Tsai et al., 2018](#)), all of which are available under <https://github.com/exoclime>.

1.2 Licence

FastChem is released under the GNU Public Licence (GPL) 3.0. That means, it can be freely copied, edited, and re-distributed. If the code is re-distributed it has to be released under at least a GPL 3.0 licence as well. The full licence of **FastChem** can be found in the repository (LICENSE file) or under <https://www.gnu.org/licenses/gpl-3.0.html>.

This user guide is released under the Creative Commons Licence (CC BY SA). Licensees may copy and distribute the work and make derivative works based on it only if they give the authors the credits by providing a reference to this guide and the corresponding GitHub repository. Licensees may also distribute derivative works only under a license identical to ("not more restrictive than") the license that governs the original work.

The **FastChem** repository also contains an additional open source code, the PyBind11 library (<https://github.com/pybind/pybind11>) that converts C++ code into a module callable from Python. This library is licensed under the BSD licence, see <https://github.com/pybind/pybind11/blob/master/LICENSE> for details.

1.3 About this guide

This guide provides basic information on the **FastChem** code and how to use it. It is structured into three different parts:

- Part **I** contains the basic descriptions on how to obtain and compile **FastChem** and its Python module **pyFastChem**.
- Part **II** describes on how to run **FastChem** as a stand-alone application as well as through its Python interface **pyFastChem**.
- Part **III** is a more in-depth description of the code itself. It provides information on the interface methods and variables used by **FastChem** and **pyFastChem**.

Part I

Installation of FastChem and pyFastChem

2 Installation

2.1 Prerequisites

FastChem is written in C++. It uses features of the C++11 standard and, therefore, requires a compiler that implements this standard. We also provide an optional Python interface, allowing **FastChem** to be called directly from within a Python script. The interface is based on the Python package **PyBind11** that is included as a submodule within **FastChem**.

The complete list of prerequisites for a basic installation is:

- a C++ compiler (e.g. **g++** or **Clang** on MacOS)
- **CMake**, at least version 3.10

The C++ compiler will be detected by the **CMake** script when it generates the makefiles. For some of its optional components **FastChem** will need:

- an **OpenMP** library (to run **FastChem** in parallel)
- a Python interpreter (for the Python interface)

2.2 Obtaining the code

FastChem is hosted on the Exoclime GitHub page: <https://github.com/exoclime/fastchem>. If **git** is available on a computer, the repository can be simply cloned with

```
git clone --recursive https://github.com/exoclime/fastchem
```

Note that the **recursive** option is required to download **FastChem**, as well as its external submodules that are not part of the main repository. Alternatively, the source code can also be downloaded in a zip file directly from the GitHub website.

2.2.1 PyBind11 Library

For its Python interface, **FastChem** requires the **PyBind11** library that translates the Python calls into C++. While **PyBind11** can in theory be installed via **pip**, **conda**, or **brew** (on MacOS), **CMake** isn't always able to properly locate the installed library.

Therefore, we chose to include **PyBind11** as a submodule in the **FastChem** repository. **CMake** will take header files and Python scripts provided by the submodule to create the **PyFastChem** module. No separate compilation or installation of **PyBind11** is required.

2.2.2 Supported compilers

The compilation of **FastChem** has been tested on a variety of different compilers and platforms. In particular, it was verified that **FastChem** can be compiled with:

- GCC 7.5 or newer
- Clang 12.0 (including Apple's Clang 12.0)

Since **FastChem** just uses plain C++ without any external library, any compiler that supports the C++11 standard should be able to compile the code successfully.

2.3 Configuration and compilation of **FastChem** with **CMake**

Before **FastChem** can be compiled, **CMake** is required to configure the compilation files, locate libraries, and write the makefiles that will perform the actual compilations. If required libraries are missing, **CMake** will report a corresponding error message. In this case, the missing libraries or compilers need to be installed before the configuration can be completed.

To run the **CMake** configuration, first create the **build** folder inside the **FastChem** source code folder and switch to the folder:

```
mkdir build
cd build
```

For a basic installation, within the folder run **CMake**¹:

```
cmake ..
```

If the Python interface should be installed as well, run

```
cmake -DUSE_PYTHON=ON ..
```

CMake will also try to locate an **OpenMP** library to allow **FastChem** to be run in parallel. If it cannot detect the library, only the single-core version of **FastChem** will be compiled. If **FastChem** is to be run on MacOS, using **OpenMP** might be difficult since Apple's **Clang** compiler does not directly support **OpenMP**, even if the corresponding library has been installed. It might be possible, though, to install an alternative compiler, for example **g++**, that supports the use of **OpenMP**.

After **CMake** successfully configured the compilation files, **FastChem** can be compiled by running:

```
make
```

Upon successful compilation, the executable **fastchem** should be present in the main **FastChem** folder. If the optional Python interface is used, **FastChem** will be automatically compiled twice because the Python requires different compiler options.

¹Remember the `..` after the `cmake` command

2.3.1 Notes on MacOS

FastChem can be compiled and run on MacOS, but requires some libraries and apps that are not installed by default. This especially includes **CMake**. In order to compile **FastChem** on MacOS, the prerequisites listed above need to be installed. This can be easily achieved by, for example, using **brew**.

In a standard installation of MacOS, no compiler is available. The Apple version of the Clang compiler can be installed through Xcode and the command line tools by running

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

in the terminal.

Alternatives (e.g. **g++**) to the default Clang shipped with MacOS can also be installed via **brew**. However, **CMake** is not always able to detect these compilers and will still use Clang. This also applies to the optional **OpenMP** library that allows **FastChem** to be run in parallel. The Clang compiler does not directly support the library, even if it has been installed via **brew**.

If the Python interface of **FastChem** is used, a corresponding Python 3 installation is also required. By default, MacOS ships only with an outdated Python 2 version that cannot be used for **FastChem**. A more up-to-date version can also be installed by, for example, **brew**. However, one has to make sure that the **python3** executable and things like **pip3** (to install other required Python modules) actually link to that version. An alternative way to install and manage different versions of Python without interference from MacOS' internal Python version is **pyenv**, which can be found under <https://github.com/pyenv/pyenv>.

2.3.2 Notes on Windows

While in theory **FastChem** could be run on Windows if meeting all the prerequisites, we have never tested the compilation and execution of **FastChem** on such a system. In principle, this should be possible under a virtual Linux environment, such as **cygwin**, or with the Windows Subsystem for Linux (WSL) shipped with the newer versions of Windows 10. However, due to the lack of a Windows system, we are unable to test this and, therefore, officially at least we cannot support **FastChem** running on Windows.

Part II

Running FastChem and pyFastChem

In this part, we describe how **FastChem** can be run directly via the included C++ stand-alone version or via Python scripts. In chapter 3, we also describe the format of the input files with the thermochemical and element abundance data. A complete overview of all available functions and output from the **FastChem** code can be found in part III for both, the C++ object class and its Python interface.

3 Standard FastChem input and output files

Besides optional other parameter files that are used within the C++ stand-alone version or the Python version, **FastChem** requires two special input files, one for the element abundances and a second describing the mass action law constant parametrisations. Both are described in the following.

3.1 Element abundance file

This file should contain the element abundances for all chemical elements that are used in **FastChem**. The location of this file is usually supplied either within a separate parameter file or directly in the constructor of the **FastChem** object class.

A note on element abundances It is important to note that there are two different ways to define an element abundance. Both variants, denoted by x_j and ϵ_j , are related via:

$$\epsilon_j = 10^{x_j - 12} \quad (3.1)$$

or

$$x_j = \log(\epsilon_j) + 12. \quad (3.2)$$

In the x_i version, widely used in the astronomical literature, hydrogen has a value of 12 for solar element abundances, such that its ϵ_i is unity.

In its input file, **FastChem** uses the x_j notation, also employed in the usual standard abundance compilations (e.g. [Asplund et al. \(2009\)](#)). For example, in the x_j notation, the solar element abundance for oxygen is $x_O = 8.69$, whereas its value for ϵ_O would be 0.00048978.

Internally, **FastChem** converts the x_j from the input file to the computationally more appropriate ϵ_j . This also refers to all methods of the **FastChem** object class that are used to interact with the element abundances: these will **always** refer to ϵ_j .

Thus, if one wants to change the oxygen element abundance in the input file (which refers to x_j) to ten times its solar value, one would need to use a value of $x_O = 1 + 8.69 = 9.69$. If one, on the other hand, uses one of the internal **FastChem** methods to change element abundances on the fly, one would need to set it to a value of $\epsilon_O = 10 \cdot 0.00048978 = 0.0048978$.

File structure The element abundance file should have the following structure to be readable by FastChem:

```
#The solar chemical composition Asplund et al 2009:
e- 13.1139
Al 6.45
Ar 6.40
C 8.43
Ca 6.34
Cl 5.50
Co 4.99
Cr 5.64
Cu 4.19
F 4.56
Fe 7.50
Ge 3.65
H 12.00
He 10.93
K 5.03
Mg 7.60
Mn 5.43
N 7.83
Na 6.24
Ne 7.93
Ni 6.22
O 8.69
P 5.41
S 7.12
Si 7.51
Ti 4.95
V 3.93
Zn 4.56
```

The first line is always a header line that provides important information for the user and is ignored by **FastChem**. All subsequent lines contain each the symbol for an element and its element abundance. Molecules that contain elements not present in this file are ignored. The element abundance for the electron has an arbitrary value. It is only present in the file to inform **FastChem** that the electrons (and thus ions) should be included in the chemistry calculations. Its element abundance ϵ_e will internally be set to 0 because its number density is determined by charge balance. The elements are not required to be in any particular order.

Standard files Together with **FastChem**, we provide two different element abundance files, located in the `input/` folder. The file `element_abundances_solar.dat` provides the solar element abundances for species that are at least as abundant as germanium. This set of element abundances is used as our standard input file and is based on [Asplund et al. \(2009\)](#).

As an alternative version, we also include an additional file `element_abundances_solar_ext.dat` that includes more elements, up to uranium. This file can be used for the extended set of ion species described in the next section. These element abundances are also based on the [Asplund et al. \(2009\)](#) compilation.

3.2 Species file

Another important input is the thermochemical data for all molecules and ions. This includes in particular their stoichiometric information as well as a parametrisation for their mass action constants. As described in the first **FastChem** publication (Stock et al., 2018), we use the natural logarithm of the dimensionless mass action constant of species i

$$\ln \bar{K}_i(T) = -\frac{\Delta_r G_i^\ominus(T)}{RT}, \quad (3.3)$$

where $G_i^\ominus(T)$ is the Gibbs free energy of dissociation. For **FastChem**, these mass action constants are fitted with the expression

$$\ln \bar{K}_i(T) = \frac{a_0}{T} + a_1 \ln T + b_0 + b_1 T + b_2 T^2, \quad (3.4)$$

where a_0 , a_1 , b_0 , b_1 , and b_2 are the fit coefficients.

It is in principle possible to use your own parametrisation. For that, you need to edit the source code that performs the calculation of the mass action constants, located in the source file `mass_action_constant.cpp`.

File structure For **FastChem**, the species information file should have the following structure:

```
#logK = a1/T + a2 ln T + a3 + a4 T + a5 T^2 for FastChem:
#include elements with eps >= eps_Ge
#fit coefficients calculated from indicated data source.
Al1Cl1 Aluminum_Chloride : Al 1 Cl 1 # Chase, M. et al., JANAF tables, 1998.
    6.01726e+04   -9.82181e-01   -5.80778e+00   1.65774e-04   -6.11197e-09

Al1Cl1F1 Aluminum_Chloride_Fluoride : Al 1 Cl 1 F 1 # Chase, M. et al., JANAF tables, 1998.
    1.22295e+05   -1.60844e+00   -1.43675e+01   3.72486e-04   -1.98493e-08

Al1Cl1F2 Aluminum_Chloride_Fluoride : Al 1 Cl 1 F 2 # Chase, M. et al., JANAF tables, 1998.
    1.93126e+05   -1.90100e+00   -3.00531e+01   6.68640e-04   -3.72957e-08
```

The first three lines of the file are treated as header lines and discarded when reading in the file.

The data for each species consists of two lines, while different species are separated by a blank line. The first line starts with the species' sum formula. In the standard **FastChem** files, we use the modified Hill notation for the formulas. Isomeric species would in principle have the same formula in the Hill notation. For example, the two species HCN and HNC would both be referred to as **C1H1N1**. To distinguish the two in the standard set of **FastChem**, underscores are used, such that **C1H1N1_1** refers to HCN, while **C1H1N1_2** represents HNC. The use of the Hill notation is not a requirement. In a custom version of the species file, a different chemical notation could be used.

The sum formula is followed by an optional name for the species. This name should be contained within a single string. Thus, if the species name is a compound noun, the separating white

spaces should be replaced by other characters, for example underscores `_` as shown above (e.g. `Aluminum_Chloride` instead of `Aluminum Chloride`).

After a separator `:`, **FastChem** expects the stoichiometric information of the species, i.e. the elements and their stoichiometric coefficients. The elements need to be present in the element abundance file as well, otherwise the species will be discarded. They don't need to be in any specific order. The stoichiometric information is followed by an optional reference for the data. If a reference is used, a separator `#` is required between the stoichiometry and the reference.

The second line contains the fit coefficients for the mass action constants. **FastChem** will read in as many coefficients as it can find in that line but for its own parametrisation in Eq. (3.4) it will only use the first five.

Standard files Together with **FastChem**, we provide two different species files, located in the `input/` folder. The file `logK.dat` provides the standard set, discussed in [Stock et al. \(2018\)](#). This includes species for all elements at least as abundant as germanium.

As an alternative version, we also provide an additional file `logK_ext.dat` that includes more ions for elements up to uranium. The data for this file is discussed in [Hoeijmakers et al. \(2019\)](#).

3.3 Basic element data file (optional)

In addition to the element abundances, **FastChem** also needs to have additional basic data for the elements, such as their atomic weight to calculate the molecular weights of molecules, for example. For most elements up to uranium, this data is hard-coded in a standard set located in the header file `chemical_element_data.h`. If you want to change this standard set by removing or adding elements or add isotopes, you can change it directly in the header file and re-compile **FastChem**. Alternatively, **FastChem** also has the option to read an external file with the required information.

File structure The optional file has the following, simple structure:

e-	Electron	5.4857990907e-4
H	Hydrogen	1.008
He	Helium	4.002602
Li	Lithium	6.94
Be	Beryllium	9.0121831
B	Boron	10.81
C	Carbon	12.011
N	Nitrogen	14.007
O	Oxygen	15.999
F	Fluorine	18.998403163
Ne	Neon	20.1797
Na	Sodium	22.98976928
Mg	Magnesium	24.305
Al	Aluminium	26.9815385
Si	Silicon	28.085
P	Phosphorus	30.973761998
S	Sulfur	32.06
Cl	Chlorine	35.45
Ar	Argon	39.948
K	Potassium	39.0983
Ca	Calcium	40.078
Sc	Scandium	44.955908
Ti	Titanium	47.867
V	Vanadium	50.9415
Cr	Chromium	51.9961
Mn	Manganese	54.938044
Fe	Iron	55.845
Co	Cobalt	58.933194
Ni	Nickel	58.6934
Cu	Copper	63.546
Zn	Zinc	65.38
Ga	Gallium	69.723
Ge	Germanium	72.630
As	Arsenic	74.921595
Se	Selenium	78.971
Br	Bromine	79.904

It contains three columns, where the first one lists the elements' symbols, the second their names, and the third their atomic weights.

3.4 FastChem parameter file (optional)

FastChem is able to load a specific parameter file when one of its instances is created through the object class constructor. This parameter file includes the most important parameters and options used within **FastChem**. All of these quantities can also be changed during runtime by using the appropriate methods listed in Sect. 6.6 for the C++ object class and Sect. 7.5 for the Python module. Using the parameter file in principle allows changing these options and parameters outside of the code and, therefore, does not require the code to be recompiled.

File structure The optional parameter file has the following structure:

```
#element abundance file
input/element_abundances_solar.dat

#species data file
input/logK.dat

#accuracy of chemistry iteration
1.0e-4

#max error of Newton's method
1.0e-4

#max number of chemistry iterations
1000

#max number of Newton iterations
2000

#max number of Nelder-Mead iterations
2000

#max number of bisection iterations
2000

#element data file (optional)
input/basic_element_data.dat
```

The first two entries are the locations of the element abundance and species data files. The next two parameters determine the convergence criteria of the chemistry and Newton's method iterations. The next four parameters set the maximum numbers of iterations for the different numerical methods employed within **FastChem**. The last parameter is optional and does not need to be present in the file. It contains the path to the file for an alternative basic element data file. If this parameter is not present, **FastChem** will use the standard set that is directly located in the **FastChem** source code (see previous section).

3.5 Output files

The C++ stand-alone version will produce two output files: a detailed chemistry output and a monitor file with diagnostic information. The file names of both files can be chosen in the config file discussed in the previous section.

Chemistry output The chemistry output is organised in columns. The first line of the file is a header that describes the content of each column.

The first and second column contain the pressure in bar and the temperature in K, respectively. The third column lists the total number density of all atoms i , i.e. $n_{\text{tot}} = \sum_i (n_i + \sum_j n_j \nu_{ij})$, summed over their atomic number densities, as well as the ones contained in all other molecules/ions j . This is usually only a diagnostic quantity and rarely used in other applications.

The fourth column is the number density of the gas in units of cm^{-3} , derived from the ideal gas law. This is followed by a column of mean molecular weights of the mixture of species in units of the unified atomic mass unit. For all practical purposes, this can also be converted into units of g/mol.

All subsequent columns contain the number densities (in cm^{-3}) or the mixing ratios of all species, depending on the choice of output made in the config file. By default, elements will be placed in the beginning, followed by molecules and ions. Note that in its species data files, **FastChem** employs the modified Hill notation as used in the JANAF thermochemical tables ([Chase, 1986](#)) for the formulas of all non-element species. If, for example, you are looking for the abundance of carbon dioxide, you need to locate the C1O2 column rather than CO2, whereas NH3 would be listed as H3N1.

Monitor file The monitor output file is a **very** important diagnostic output that provides crucial details on the outcome of the chemistry calculations. You should further investigate any chemistry calculations that shows problems in this file. It is, therefore, advisable to check this file after each calculation to verify that everything went fine. The first line of the file is a header that describes the content of each column.

The monitor output is organised in columns, where the first column contains a simple integer that refers to index of the input temperature-pressure structure. The second column lists the number of chemistry iterations that were required to solve the system. If the number corresponds to the maximum number of allowed iteration steps, then this points to potential convergence issues.

The next columns contain information on the convergence of the chemistry and on the status of overall element conservation. If the chemistry did converge properly **ok** will be listed as output, whereas **fail** is used when the chemistry failed to converge in the maximum allowed number of steps. The same keywords are used for the element conservation status: **ok** if all elements were conserved, **fail** if any element was not conserved.

The next four columns contain basic chemistry output, that is also found in the chemistry output file: the pressure, temperature, total element density, gas number density, and mean molecular weight.

All remaining columns list the status of the element conservation for each element separately. The same keywords as for the overall element conservation status are used again in these columns. For

the electrons, this status refers to the charge balance rather than element conservation.

4 The FastChem C++ stand-alone executable

The **FastChem** object class is designed to be easily coupled to other models. In addition to the object class itself, we also provide a stand-alone executable that can call the module with some simple input scripts. This stand-alone version, however, only provides a very basic functionality, such as reading in a specific temperature-pressure profile that **FastChem** will be run for. The stand-alone version does, for example, not provide more advanced capabilities, such as looping over different metallicity values or C/O ratios. If you intend to use **FastChem** for such purposes, you need to adapt the code that calls **FastChem**.

The source code that is responsible for calling the actual **FastChem** chemistry is located in the folder `model_main/`. It is split across three different files: `model_main/model_main.cpp`, the actual main program, `model_main/read_config.h` for reading in the config file, and `model_main/save_output.h` for managing the output. Thus, if you want to add another parameter to the config file, you would need to edit `model_main/read_config.h`, while changes to the format of the output files can be made in `model_main/save_output.h`. Changing the contents of these files obviously require a re-compilation of the code.

4.1 Starting the FastChem executable

Following a successful configuration and compilation via **CMake**, the **FastChem** executable `fastchem` should be present in the root directory. The executable is started via

```
./fastchem input/config.input
```

where the second argument is the location of the config file that is explained in the next section. **FastChem** will read in a pre-defined temperature-pressure structures, the location of which is also specified in the config file. After a successful calculation, **FastChem** will produce two output files with a detailed chemistry output and one with diagnostic output. The location of these files is also contained in the config file and its contents are discussed in Sect. 3.5.

4.2 Config file

The config file that **FastChem** will read in at the beginning contains all important parameters and file locations necessary to initialise the chemistry and to perform the calculations. The numerical methods that these parameters refer to are described in Paper II. An example of such an input file is located in the input folder: `input/config.input`. While this config file allows to set the most important **FastChem** parameters, some more advanced ones are not contained in this file and can only be set by invoking special **FastChem** functions during runtime. This, in particular, refers to the use of the optional scaling factors as described in the appendix of Paper II. More information on activating these scaling factors can be found in the description of the object class in Sect. 6.6.

The config file used for the C++ stand-alone executable has the following structure:

```
#Atmospheric profile input file
input/kelt_9b.dat

#Chemistry output file
output/chemistry.dat

#Monitor output file
output/monitor.dat

#FastChem console verbose level (1 - 4); 1 = almost silent, 4 = detailed console output
1

#Output mixing ratios (MR) or particle number densities (ND, default)
MR

#Element abundance file
input/element_abundances_solar.dat

#Species data file
input/logK.dat

#Accuracy of chemistry iteration
1.0e-4

#Max error of Newton's method
1.0e-4

#Max number of chemistry iterations
80000

#Max number of Newton iterations
20000

#Max number of Nelder-Mead iterations
20000

#Max number of bisection iterations
20000
```

It contains the required parameters in the following order:

- location of the file with the temperature-pressure structure the chemistry should be calculated for
- desired location and file name for the chemistry output
- desired location and file name for the diagnostic output
- verbose level, where a level of 1 is almost silent and 4 produces a lot of diagnostic output on the terminal. Increase this level if you encounter issues to identify the source of the problems.
- the output format for the species' abundances. By default, **FastChem** will use number densities

in units of cm^{-3} . If you use the keyword **MR**, mixing ratios will be used instead. Any keyword other than **MR** will result in the default option of using number densities.

- location of the file with the element abundances, see Sect. 3 for details
- location of the file with the thermochemical and stoichiometric data for all species, see Sect. 3 for details
- relative accuracy of the chemistry iterations, used as convergence criterion
- convergence criterion for Newton's method
- maximum number of chemistry iterations
- maximum number of Newton method iterations
- maximum number of iterations for the Nelder-Mead method
- maximum number bisection method iterations

5 Running pyFastChem

In addition to the C++ executable, we provide several Python scripts that can run the **FastChem** code through its Python interface **pyFastChem**. The sample scripts can be found within the `python/` folder. These sample scripts show different use cases and can be used as a basis for your own **FastChem** Python scripts.

5.1 Provided Python examples

Currently, we provide the following examples:

`fastchem.py`

Runs a simple **FastChem** calculation on a temperature-pressure structure defined within the script, writes output files, and creates a plot with selected species.

`fastchem_c_o.py`

Runs a **FastChem** calculation on a temperature-pressure structure defined within the script and for a range of different C/O ratios. It will write output files, and create a plot with selected species.

`fastchem_metallicity.py`

Runs a **FastChem** calculation on a temperature-pressure structure defined within the script and for a range of different metallicity factors. It will write output files, and create a plot with selected species.

Note that the scripts should be executed from the **FastChem** root directory rather than from within the `python` directory since all file paths in the scripts are given relative to the root directory. These files can be used as templates to create your own special Python scripts to run **pyFastChem**. The following section provides some details on the steps required to run **FastChem** from within Python. A more detailed overview of all the methods and variables available within **pyFastChem** can be found in Chapter 7.

5.2 Detailed steps for running FastChem with pyFastChem

As a first step, we need to import the **pyFastChem** module:

```
import pyfastchem
```

This will import the module compiled by PyBind11. Next, we have to create a **FastChem** object (here named `fastchem`) with the constructor `pyfastchem.FastChem` provided by `pyFastChem`:

```
fastchem = pyfastchem.FastChem('input/element_abundances.dat', 'input/logK.dat', 1)
```

The constructor requires three different arguments: the location of the element abundance file, the location of the file with the equilibrium constants, and the verbose level. Alternatively, a **FastChem** object can also be created via

```
fastchem = pyfastchem.FastChem('input/parameters.dat', 1)
```

where the first argument is the location of the parameter file and the second one the initial verbose level. The latter one will later be replaced by the corresponding value read in from the parameter file. The structure of this parameter file is discussed in Section 3.4.

Creating a **FastChem** object with the first method will set internal parameters to their default values. The maximum number of chemistry iterations will be 1000, the number of Newton, bisection and Nelder-Mead method iterations is 3000, and the accuracy of the of Newton method and the chemistry iterations is set to 10^{-4} . All of these values can, however, be adjusted during runtime by using the methods listed in Section 7.5.

Next, we need to create the input and output structures used by `pyFastChem`:

```
input_data = pyfastchem.FastChemInput()
output_data = pyfastchem.FastChemOutput()
```

Details on these structures can be found in Section 7.4. The input structure basically contains the temperature (in K) and pressure (in bar) arrays that the chemistry should be calculated for. They can be set, for example, by:

```
input_data.temperature = temperature
input_data.pressure = pressure
```

where `temperature` and `pressure` are standard Python lists or NumPy arrays. Both arrays need to have the same length. With the input structure properly set up, we can now run the actual **FastChem** calculation by calling the `calcDensities` method:

```
fastchem_flag = fastchem.calcDensities(input_data, output_data)
```

This method returns an integer flag that describes the overall outcome of the calculation. A

description of the different flags can be found in Section 7.2. After calling the `calcDensity` method, the output structure will be filled with the corresponding output data. For example, `output_data.number_densities` will contain the number densities of the chemical species. This is a 2D list, where the first dimension refers to the temperature and pressure input arrays and the second dimension refers to the different chemical species. The list can be easily converted into a NumPy array via:

```
number_densities = np.array(output_data.number_densities)
```

The Python directory of the **FastChem** repository also contains two functions that save the output into files, identical to those from the C++ version. They can be called by:

```
saveChemistryOutput('output/test.dat', temperature, pressure, output_data, fastchem)
saveMonitorOutput('output/monitor_test.dat', temperature, pressure, output_data, fastchem)
```

Here, the first parameter is the file name, followed by the input temperature and pressure arrays, and the `fastchem` object. A detailed description of the output files can be found in Section 3.5. For the Python version, the chemistry output file lists the species' mixing ratios rather than their number densities. By default, the output file will contain the data for all species used in the chemistry. To create an output with only a selected set of species, we provide an alternative version of the `saveChemistryOutput` function that takes an additional parameter:

```
saveChemistryOutput('output/test.dat', temperature, pressure, output_data, fastchem, chem_species)
```

The additional parameter `chem_species` is an array (or list) of strings that contains the formulas of the species that should appear in the output file. The formulas must be identical to those from the input equilibrium constants file.

Part III

Detailed C++ object class and Python module description

6 FastChem class

FastChem has been written in an object oriented way, split across several different object classes. The entire source code of **FastChem** is contained in the folder `fastchem_src/`. For including **FastChem** in another C++ project, only adding the main fastchem header file `fastchem.h` is required. All **FastChem** code is encapsulated in its own namespace called `fastchem` to avoid clashing with other libraries.

6.1 Some comments on coding conventions

The entire **FastChem** code has been programmed using specific conventions that make it easy to recognise and differentiate class, method and variable names.

Class and structure names are always capitalised, for example

```
class FastChem{...
```

or

```
struct Molecule{...
```

If the name is a compound noun, each noun is capitalised separately, e.g.

```
struct ChemicalElementData{...
```

No separators like `_` are used for class or structure names.

Class methods and functions in general always start with a lowercase letter. If the name is a compound noun, the start of every other noun is capitalised and no separator is used, for example:

```
FastChem.setElementAbundances(...)
```

or

```
Molecule.calcMassActionConstant(...)
```

Variable names are always written in lowercase and compound nouns are separated by a `_`. For example:

```
FastChem.element_calculation_order
```

or

```
Element.molecule_list
```

The only exceptions are global constants. They contain only capitalised letters, e.g.:

```
constexpr unsigned int FASTCHEM_UNKNOWN_SPECIES
```

or

```
constexpr double CONST_AMU
```

6.2 FastChem object class

The entire **FastChem** model is encapsulated in an object class called **FastChem** that is defined in the header file `fastchem.h`. The object class is programmed as a template that be used in either `double` or `long double` precision. When creating an object of this class by calling a corresponding constructor, one therefore has to specify which of the two versions should be used:

```
FastChem<long double> fastchem("model_parameter_file.dat", 1);
```

```
FastChem<double> fastchem("model_parameter_file.dat", 1);
```

The `long double` version has a slightly higher computational overhead and larger memory requirements than the `double` one. On the other hand, it offers a higher numerical precision which is especially important when dealing with chemical systems where the mass action constants and number densities can vary by many orders of magnitudes. We strongly suggest to always use the `long double` version despite the additional computational overhead. In fact, in our experience the increased numerical precision of `long double` can effectively lead to a smaller number of iterations.

6.3 FastChem constants

The **FastChem** namespace `fastchem` contains a number of constants that are all defined in the file `fastchem_constants.h`.

This includes the constant `constexpr unsigned int fastchem::FASTCHEM_UNKNOWN_SPECIES` that is returned by some **FastChem** methods when a chemical species is not found.

The chemistry calculation will return several output flags that are also defined in this file. This includes the following constants:

```
constexpr unsigned int fastchem::FASTCHEM_SUCCESS
```

Indicates that the calculation has been successful, i.e. that the chemistry iterations converged.

```
constexpr unsigned int fastchem::FASTCHEM_NO_CONVERGENCE
```

Indicates that the calculation was not successful, i.e. that the chemistry did not converge within the allowed maximum number of iterations steps given in the config file or set manually via `FastChem.setMaxChemistryIter` (see Sect. 6.6). One way to solve such a problem is to increase the maximum number of iteration steps.

```
constexpr unsigned int fastchem::FASTCHEM_INITIALIZATION_FAILED
```

Indicates that something went wrong during reading one of the input files. To find the source of the problem, one can set the verbose level in the config file or manually via `FastChem.setVerboseLevel` (see Sect. 6.6) to a higher value and look at the terminal output.

```
constexpr unsigned int fastchem::FASTCHEM_IS_BUSY
```

The chemistry calculations of `FastChem` can only be called once for each object class instance. Attempting to start a new calculation while another is still running, for example via OpenMP, will result in `FastChem` returning this flag.

```
constexpr unsigned int fastchem::FASTCHEM_WRONG_INPUT_VALUES
```

`FastChem` returns this flag if some input values are wrong. Currently, this refers to the temperature and pressure vectors in the input structure not having the same size (see Sect. 6.5.1 for details on the input structure).

In addition to these flags, `fastchem_constants.h` also includes a constant string vector `const std::vector<std::string> fastchem::FASTCHEM_MSG` that contains string expressions for each of these flags. Using this vector with any of the aforementioned flags `fastchem::FASTCHEM_MSG[flag]` returns a string with a description of the corresponding flag's meaning. For example, `fastchem::FASTCHEM_MSG[fastchem::FASTCHEM_NO_CONVERGENCE]` will return the string "convergence failed".

6.4 FastChem constructor

Since `FastChem` is written as an object class, an instance of that class (i.e. an object) needs to be created before `FastChem` can be used. This is done by calling the constructor of the `FastChem` class. There are two primary ways to call the constructor and create an object.

```
FastChem(const std::string& model_parameter_file, const unsigned int verbose_level_init)
```

This constructor requires two parameters: the location of the parameter file, described in Sect. 3.4, as well as the initial verbose, i.e. the amount of debug output in the terminal window. All main options and parameters will be read from the parameter file, but can be changed later by using the appropriate methods described in Sect. 6.6.

```
FastChem(const std::string &element_abundances_file, const std::string &species_data_file, const ←
unsigned int verbose_level)
```

This constructor requires three parameters: the locations of the element abundance and species data files, as well as the verbose level. All other options and parameters within **FastChem** will be set to their default values but can be later changed by using the appropriate methods described in Sect. 6.6. The default maximum number of chemistry iterations is 1000, the number of Newton, bisection and Nelder-Mead method iterations is 3000, and the default accuracy of the of Newton method and the chemistry iterations is set to 10^{-4} .

A third way to create a **FastChem** object is to make a copy of an existing one. **FastChem** contains an internal copy constructor that manages the copy of all the object class' data structures. Assuming that `fastchem_a` is a valid object instance of the **FastChem** class, a second object, say `fastchem_b`, can simply be created by using

```
fastchem::FastChem fastchem_b(fastchem_a);
```

In this example, `fastchem_b` is a direct copy of `fastchem_a`, i.e. all parameters, options, and species & element data structures are identical. After the creation of `fastchem_b`, both objects can be used independently from each other and can even be run at the same time.

6.5 Input and output structures

When the chemistry calculation of **FastChem**, `FastChem.calcDensities(FastChemInput, FastChemOutput)` is called, input and output structures are required. Their definitions can be found in the source file `fastchem_src/input_output_struct.h`

6.5.1 Input structure

The input structure is defined as follows:

```
struct FastChemInput
{
    std::vector<double> temperature;
    std::vector<double> pressure;
};
```

It contains vectors for the temperatures (in K) and pressures (in bar) that the chemical composition should be calculated for. Both vectors need to have the same length. Otherwise, `FastChem.calcDensities` will return the constant `fastchem::FASTCHEM_WRONG_INPUT_VALUES`. Note that even if you want to run the chemistry for only a single temperature and pressure point, you still need to provide the input in vectorial form.

6.5.2 Output structure

The outout structure is defined as

```
struct FastChemOutput
{
    std::vector<std::vector<double>> number_densities;
    std::vector<double> total_element_density;
    std::vector<double> mean_molecular_weight;

    std::vector<std::vector<unsigned int>> element_conserved;
    std::vector<unsigned int> nb_chemistry_iterations;
    std::vector<unsigned int> fastchem_flag;
};
```

It has the following variables:

`std::vector<std::vector<double>> number_densities`

The two-dimensional vector contains the number densities in cm^{-3} of all species (elements, molecules, ions). The first dimension refers to the temperature-pressure grid and has the same size as the temperature and pressure vectors of the input structure. The second dimension refers to the number of species and has a length of `FastChem.getSpeciesNumber()` (see Sect. 6.6).

`std::vector<double> total_element_density`

Contains the total number density of all atoms i , i.e. $n_{\text{tot}} = \sum_i (n_i + \sum_j n_j \nu_{ij})$, summed over their atomic number densities, as well as the ones contained in all other molecules/ions j . This quantity is usually only a diagnostic output and not relevant for other calculations. The dimension of the vector is equal to that of the input temperature and pressure vectors.

`std::vector<double> mean_molecular_weight`

Contains the mean molecular weight of the mixture in units of the unified atomic mass unit. For all practical purposes, this can also be converted into units of g/mol. The dimension of the vector is equal to that of the input temperature and pressure vectors.

`std::vector<std::vector<unsigned int>> element_conserved`

The two-dimensional vector contains information on the state of element conservation. A value of 0 indicates that element conservation is violated, whereas a value of 1 means that the element has been conserved. The first dimension refers to the temperature-pressure grid and has the same size as the temperature and pressure vectors of the input structure. The second dimension refers to the number of elements and has a length of `FastChem.getElementNumber()` (see Sect. 6.6).

`std::vector<unsigned int> nb_chemistry_iterations`

Contains the number of chemistry iterations that were required to solve the system for each temperature-pressure point. The dimension of the vector is equal to that of the input temperature and pressure vectors.

`std::vector<unsigned int> fastchem_flag`

Contains flags that give information on potential issues of the chemistry calculation for each temperature-pressure point. The set of potential values is stated in Sect. 6.3. A string message for each corresponding flag can also be obtained from the constant `fastchem::FASTCHEM_MSG` vector of strings, via `fastchem::FASTCHEM_MSG[flag]`. The dimension of the vector is equal to that of the input temperature and pressure vectors.

The vectors of the output structure don't need to be pre-allocated. This will be done internally within `FastChem` when running the chemistry calculations. If the vectors already contain data, their contents will be overwritten.

6.6 Methods of the `fastchem::FastChem` object class

`unsigned int` `FastChem.calcDensities(FastChemInput input, FastChemOutput output)`

Starts a chemistry calculation with the provided `FastChemInput` and `FastChemOutput` structs. Returns an `unsigned int` that represents the highest value from the flag vector within the `FastChemOutput` struct.

`unsigned int` `FastChem.getSpeciesNumber()`

Returns the total number of chemical species (atoms, ions, molecules) as `unsigned int`

`unsigned int` `FastChem.getElementNumber()`

Returns the total number of elements as `unsigned int`

`unsigned int` `FastChem.getMoleculeNumber()`

Returns the total number of molecules and ions (anything other than elements) as `unsigned int`

`std::string` `FastChem.getSpeciesName(unsigned int species_index)`

Returns the name of a chemical species with index `species_index` as `std::string`; returns empty string if species does not exist

`std::string` `FastChem.getSpeciesSymbol(unsigned int species_index)`

Returns the symbol of an element or the formula of a molecule/ion with index `species_index` as `std::string`; returns empty string if species does not exist

`unsigned int` `FastChem.getSpeciesIndex(std::string symbol)`

Returns the index of a species (element/molecule/ion) with symbol/formula `symbol` as `unsigned int`; returns the constant `fastchem::FASTCHEM_UNKOWN_SPECIES` if species does not exist

`double` `FastChem.getElementAbundance(unsigned int species_index)`

Returns the abundance of an element with index `species_index` as `double`; returns 0 if element does not exist


```
std::vector<double> FastChem.getElementAbundance()
```

Returns the abundances of all elements as a vector of `double`; vector has a length of `FastChem.getElementNumber()`

```
double FastChem.getSpeciesMolecularWeight(const unsigned int species_index)
```

Returns the molecular weight of a species with index `species_index` as `double`; returns 0 if species does not exist; for an element this refers to the atomic weight

```
void setElementAbundances(std::vector<double> abundances)
```

Sets the abundances of all elements; the abundances are supplied as `std::vector<double>`, where the vector has to have a size of `FastChem.getElementNumber()`; if this is not the case, `FastChem` will print an error message and leave the element abundances unchanged

```
void FastChem.setVerboseLevel(unsigned int level)
```

Sets the verbose level of `FastChem`, i.e. the amount of text output in the terminal. A value of 0 will result in `FastChem` being almost silent, whereas a value of 4 would provide a lot of debug output. A value larger than 4 will be interpreted as 4. This value will overwrite the one from the `FastChem` config file.

```
void FastChem.setMaxChemistryIter(unsigned int nb_steps)
```

Sets the maximum number of internal chemistry iterations, provided by `unsigned int nb_steps`. This value will overwrite the one from the `FastChem` config file.

```
void FastChem.setMaxNewtonIter(unsigned int nb_steps)
```

Sets the maximum number of internal Newton iterations, provided by `unsigned int nb_steps`. This value will overwrite the one from the `FastChem` config file.

```
void FastChem.setMaxBisectionIter(unsigned int nb_steps)
```

Sets the maximum number of internal bisection iterations, provided by `unsigned int nb_steps`. This value will overwrite the one from the `FastChem` config file.

```
void FastChem.setMaxNelderMeadIter(unsigned int nb_steps)
```

Sets the maximum number of internal iterations of the Nelder-Mead, provided by `unsigned int nb_steps`. This value will overwrite the one from the `FastChem` config file.

```
void FastChem.setChemistryAccuracy(double accuracy)
```

Sets the desired accuracy of the chemistry convergence criterion, provided by `double accuracy`. This value will overwrite the one from the `FastChem` config file.

```
void FastChem.setNewtonAccuracy(double accuracy)
```

Sets the desired accuracy of the Newton's method convergence criterion, provided by `double accuracy`. This value will overwrite the one from the `FastChem` config file.

```
void FastChem.useScalingFactor(bool use)
```

Toggles the use of the internal scaling factor. The default value in `FastChem` is `false`. The use of the scaling factor is currently not accessible from the config files but can only be activated by this function.

7 pyFastChem: The Python module of FastChem

By using the library PyBind11, FastChem can be called directly within Python. This requires the compilation of FastChem's Python wrapper that is located in the file `python/fastchem_python_wrapper.cpp`. As described in Sect. 2.3 this will be done automatically when `cmake` is configured with the corresponding option. `pyFastChem` currently only links to the `long double` C++ version of FastChem.

If the configuration and compilation is successful, a module file should be present in the `python/` folder that contains the Python module which acts as a wrapper between Python and the C++ version of FastChem. The file should be named `pyfastchem.cpython-xxxx`, where `xxxx` will be a combination of your Python version and operating system. A description of the module is given in Sect. 7.1. Besides the `pyFastChem` module, we also provide several example Python scripts that show how to call FastChem from within Python for several different scenarios. We discuss the examples in Sect. ??.

7.1 The pyFastChem module

The `pyFastChem` module provides access to the `FastChem` object class as well as additional constants used within FastChem. They are essentially identical to their C++ counterparts discussed in Sect. 6. To include the `pyFastChem` module in your Python project, just import it using

```
import pyfastchem
```

This provides access to the `FastChem` object class as well as the input and output structures and additional pre-defined constants used by FastChem.

7.2 pyFastChem constants

The `pyFastChem` module contains a number of pre-defined constants. This includes the constant `pyfastchem.FASTCHEM_UNKNOWN_SPECIES` of type `int` that is returned by some `pyFastChem` methods when a chemical species is not found.

The chemistry calculation can also return several output flags of type `int` defined at constants in the `pyFastChem` module:

`pyfastchem.FASTCHEM_SUCCESS`

Indicates that the calculation has been successful, i.e. that the chemistry iterations converged.

`pyfastchem.FASTCHEM_NO_CONVERGENCE`

Indicates that the calculation was not successful, i.e. that the chemistry did not converge within the allowed maximum number of iterations steps given in the config file or set manually via

`FastChem.setMaxChemistryIter` (see Sect. 6.6). One way to solve such a problem is to increase the maximum number of iteration steps.

`pyfastchem.FASTCHEM_INITIALIZATION_FAILED`

Indicates that something went wrong during reading one of the input files. To find the source of the problem, one can set the verbose level in the config file or manually via `FastChem.setVerboseLevel` (see Sect. 6.6) to a higher value and look at the terminal output.

`pyfastchem.FASTCHEM_IS_BUSY`

The chemistry calculations of `FastChem` can only be called once for each object class instance. Attempting to start a new calculation while another is still running will result in `FastChem` returning this flag.

`pyfastchem.FASTCHEM_WRONG_INPUT_VALUES`

`FastChem` returns this flag if some input values are wrong. Currently, this refers to the temperature and pressure vectors in the input structure not having the same size (see Sect. 6.5.1 for details on the input structure).

In addition to these flags, the `pyFastChem` module also includes a constant string array `pyfastchem.FASTCHEM_MSG` that contains string expressions for each of these flags. Using this array with any of the aforementioned flags `pyfastchem.FASTCHEM_MSG[flag]` returns a string with a description of the corresponding flag's meaning. For example, `pyfastchem.FASTCHEM_MSG[pyfastchem.FASTCHEM_NO_CONVERGENCE]` will return the string "convergence failed".

7.3 pyFastChem constructor

Since `FastChem` is written as an object class, an instance of that class (i.e. an object) needs to be created before `FastChem` can be used. This is done by calling the constructor of the `FastChem` class that is contained within the `pyFastChem` module. There are two main ways to call the constructor and create an object.

`pyfastchem.FastChem(str element_abundance_file, str equilibrium_constants_file, int verbose_level)`

The constructor requires three different arguments: the location of the element abundance file, the location of the file with the equilibrium constants, and the verbose level, i.e. the amount of debug output in the terminal window. Creating a `FastChem` object with this constructor will set internal parameters to their default values. The maximum number of chemistry iterations will be 1000, the number of Newton, bisection and Nelder-Mead method iterations is 3000, and the accuracy of the of Newton method and the chemistry iterations is set to 10^{-4} . All of these values can be adjusted during runtime by using the methods listed in Section 7.5.

```
pyfastchem.FastChem(str parameter_file, int initial_verbose_level)
```

The constructor requires two different arguments: the location of the parameter file and the initial verbose level. The latter one will be replaced by the corresponding value read in from the parameter file. The structure of this parameter file is discussed in Section 3.4. All of parameter values read in from the file can also be adjusted during runtime by using the methods listed in Section 7.5.

7.4 pyFastChem input and output structures

Running a FastChem chemistry calculation requires input and output data structures, resembling those of the C++ version (see Sect. 6.5.1).

Input structure An input structure, in the example here called `input_data`, can be defined from the `pyFastChem` module in the following way:

```
input_data = pyfastchem.FastChemInput()
```

The input structure contains the following variables:

temperature

An array of `float` numbers that describe the temperature in K.

pressure

An array of `float` numbers that describe the pressure in bar.

Both arrays need to have the same length. The `PyBind11` library allows normal Python lists or NumPy arrays to be used here. For example, a NumPy array for the pressure could be defined using NumPy's `logspace` function:

```
input_data.pressure = np.logspace(-6, 1, num=1000)
```

Both input variables need to be an array-type variable, even if only a single temperature-pressure point is going to be calculated.

Output structure The output structure from the `pyFastChem` module, in the example here called `output_data`, can be defined in the following way:

```
output_data = pyfastchem.FastChemOutput()
```

It has the following variables:

number_densities

The two-dimensional array contains the number densities in cm^{-3} of all species (elements, molecules, ions) as `float` numbers. The first dimension refers to the temperature-pressure grid and has the same size as the temperature and pressure arrays of the input structure. The second dimension refers to the number of species and has a length of `FastChem.getSpeciesNumber()` (see Sect. 6.6).

total_element_density

One-dimensional array of **float** numbers that contains the total number density of all atoms i , i.e. $n_{\text{tot}} = \sum_i (n_i + \sum_j n_j \nu_{ij})$, summed over their atomic number densities, as well as the ones contained in all other molecules/ions j . This quantity is usually only a diagnostic output and not relevant for other calculations. The dimension of the array is equal to that of the input temperature and pressure vectors.

mean_molecular_weight

One-dimensional array of **float** numbers. Contains the mean molecular weight of the mixture in units of the unified atomic mass unit. For all practical purposes, this can also be converted into units of g/mol. The dimension of the array is equal to that of the input temperature and pressure vectors.

element_conserved

The two-dimensional array of **int** numbers contains information on the state of element conservation. A value of 0 indicates that element conservation is not fulfilled, whereas a value of 1 means that the element has been conserved. The first dimension refers to the temperature-pressure grid and has the same size as the temperature and pressure vectors of the input structure. The second dimension refers to the number of elements and has a length of `FastChem.getElementNumber()` (see Sect. 6.6).

nb_chemistry_iterations

One-dimensional array of **int** numbers. Contains the number of chemistry iterations that were required to solve the system for each temperature-pressure point. The dimension of the array is equal to that of the input temperature and pressure vectors.

fastchem_flag

One-dimensional array of **int** numbers. Contains flags that give information on potential issues of the chemistry calculation for each temperature-pressure point. The set of potential values is stated in Sect. 7.2. A string message for each corresponding flag can also be obtained from the constant `pyfastchem.FASTCHEM_MSG` vector of strings, via `pyfastchem.FASTCHEM_MSG[flag]`. The dimension of the array is equal to that of the input temperature and pressure vectors.

The arrays of the output structure don't need to be pre-allocated. This will be done internally within **FastChem** when running the chemistry calculations. If the arrays already contain data, their contents will be overwritten. The arrays from the output structure can also be easily converted to more practical NumPy arrays by using, for example:

```
number_densities = np.array(output_data.number_densities)
```

7.5 pyFastChem functions

The `pyFastChem` object returned from `pyfastchem.FastChem()` has several methods that allow to interact with `FastChem`. These methods are equivalent to those of the C++ object class discussed in Sect. 6.6.

int `calcDensities(pyfastchem.FastChemInput() input, pyfastchemFastChemOutput() output)`
Starts a chemistry calculation with the provided `pyfastchem.FastChemInput()` and `pyfastchem.FastChemOutput()` structures. Returns an **int** value that represents the highest value from the flag vector within the `pyfastchem.FastChemOutput()` structure.

int `getSpeciesNumber()`
Returns the total number of chemical species (atoms, ions, molecules) as **int** value.

int `getElementNumber()`
Returns the total number of elements as **int** value.

int `getMoleculeNumber()`
Returns the total number of molecules and ions (anything other than elements) as **int** value.

str `getSpeciesName(int species_index)`
Returns the name of a chemical species with **int** index¹ `species_index` as **str**; returns empty string if species does not exist.

str `getSpeciesSymbol(int species_index)`
Returns the symbol of an element or the formula of a molecule/ion with **int** index¹ `species_index` as **str**; returns empty string if species does not exist

int `getSpeciesIndex(str symbol)`
Returns the index of a species (element/molecule/ion) with **str** symbol/formula `symbol` as **int**; returns the constant `pyfastchem.FASTCHEM_UNKOWN_SPECIES` if species does not exist.

float `getElementAbundance(int species_index)`
Returns the abundance of an element with **int** index¹ `species_index` as **float**; returns 0 if the element does not exist

float `[] getElementAbundance()`
Returns the abundances of all elements as an array of **float** values; array has a length of `getElementNumber()`.

¹In the C++ class, an **unsigned int** is required here. Since this data type doesn't exist in Python, `PyBind11` will convert the supplied integer value to its unsigned integer version for C++. Even though the parameter is defined as an **int** value for Python, only positive numbers, including 0, are accepted as valid input. Using a negative value will result in an error message from `PyBind11`.

`float` FastChem.getSpeciesMolecularWeight(`int` species_index)

Returns the molecular weight of a species with `int` index¹ species_index as `float`; returns 0 if species does not exist; for an element this refers to the atomic weight.

`setElementAbundances(float [] abundances)`

Sets the abundances of all elements; the abundances are supplied as an array of `float` values, where the array has to have a size of `getElementNumber()`; if this is not the case, **FastChem** will print an error message and leave the element abundances unchanged

`setVerboseLevel(int level)`

Sets the verbose level of **FastChem**, i.e. the amount of text output in the terminal. A value of 0 will result in **FastChem** being almost silent, whereas a value of 4 would provide a lot of debug output. A value larger than 4 will be interpreted as 4. This value will overwrite the one from the **FastChem** config file.

`setMaxChemistryIter(int nb_steps)`

Sets the maximum number of internal chemistry iterations, provided by the `int`¹ variable nb_steps. This value will overwrite the one from the **FastChem** parameter file.

`setMaxNewtonIter(int nb_steps)`

Sets the maximum number of internal Newton iterations, provided by the `int`¹ variable nb_steps. This value will overwrite the one from the **FastChem** config file.

`setMaxBisectionIter(int nb_steps)`

Sets the maximum number of internal bisection iterations, provided by the `int`¹ variable nb_steps. This value will overwrite the one from the **FastChem** config file.

`setMaxNelderMeadIter(int nb_steps)`

Sets the maximum number of internal iterations of the Nelder-Mead, provided by the `int`¹ variable nb_steps. This value will overwrite the one from the **FastChem** config file.

`setChemistryAccuracy(float accuracy)`

Sets the desired accuracy of the chemistry convergence criterion, provided by `float` variable accuracy. This value will overwrite the one from the **FastChem** config file.

`setNewtonAccuracy(float accuracy)`

Sets the desired accuracy of the Newton's method convergence criterion, provided by `float` variable accuracy. This value will overwrite the one from the **FastChem** config file.

`useScalingFactor(bool use)`

Toggles the use of the internal scaling factor. The default value in **FastChem** is `false`. The use of the scaling factor is currently not accessible from the config files but can only be activated by this function.

Bibliography

Asplund, M., Grevesse, N., Sauval, A. J., & Scott, P. 2009, ARA&A, 47, 481

Bourrier, V., Kitzmann, D., Kuntzer, T., et al. 2020, A&A, 637, A36

Chase, M. W. 1986, JANAF thermochemical tables

Hoeijmakers, H. J., Ehrenreich, D., Kitzmann, D., et al. 2019, A&A, 627, A165

Kitzmann, D., Heng, K., Oreshenko, M., et al. 2020, ApJ, 890, 174

Malik, M., Kitzmann, D., Mendonça, J. M., et al. 2019, AJ, 157, 170

Stock, J. W., Kitzmann, D., & Patzer, A. B. C. 2020, in prep

Stock, J. W., Kitzmann, D., Patzer, A. B. C., & Sedlmayr, E. 2018, MNRAS, 479, 865

Tsai, S.-M., Kitzmann, D., Lyons, J. R., et al. 2018, ApJ, 862, 31