29-09-2021

D2.2 – Report on the final release of the I/O library

Version V1.0

GA no 952165

Dissemination Level

- ☒ PU: Public
- ☐ PP: Restricted to other programme participants (including the Commission)
- ☐ RE: Restricted to a group specified by the consortium (including the Commission)
- ☐ CO: Confidential, only for members of the consortium (including the Commission)

Document Information

Project Title	Targeting Real Chemical accuracy at the EXascale
Project Acronym	TREX
Grant Agreement No	952165
Instrument	Call: H2020-INFRAEDI-2019-1
Topic	INFRAEDI-05-2020 Centres of Excellence in EXascale computing
Start Date of Project	01-10-2020
Duration of Project	36 Months
Project Website	https://trex-coe.eu/

Deliverable Number	D2.2
Deliverable title	D2.2 – Report on the final release of the I/O library, as seen in GA
Due Date	M12 – 31-09-2021 (from GA)
Actual Submission Date	29-09-2021

Work Package	WP2 – Code modularization and interfacing
Lead Author (Org)	Anthony Scemama (Centre National de la Recherche Scientifique (CNRS))
Contributing Author(s) (Org)	Evgeny Posenitskiy (CNRS)
Reviewers (Org)	Axel Auweter (Megware computer vertrieb und service GmbH (Megware)), Dirk Pleiter (Kungliga Tekniska högskolan (KTH)), Jan Beerens (Universiteit Twente (UT))
Version	V1.0
Dissemination level	PU
Nature	Report
Draft / final	Final
No. of pages including cover	20

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

Disclaimer

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

The content of this document does not represent the opinion of the European Union, and the European Union is not responsible for any use that might be made of such content.

Versioning

Version	Date	Authors	Notes
1.0	29-09-2021	Anthony Scemama (CNRS)	First Official Release

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

Abbreviations

AO	Atomic Orbital
API	Application Programming Interface
CNRS	Centre National de la Recherche Scientifique
CoE	Center of Excellence
ECP	Effective Core Potential
HDF5	Hierarchical Data Format
HTML	HyperText Markup Language
JSON	JavaScript Object Notation
KTH	Kungliga Tekniska högskolan
Megware	Megware computer vertrieb und service GmbH
MO	Molecular Orbital
TREX	Targeting REal chemical accuracy at the eXascale
TREXIO	TREX Input/Output
UT	Universiteit Twente
WP	Work Package

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

Table of Contents

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

1 Summary of the previous report

A report for the progress of this Work Package (WP) was made at month 6: *D2.1 - Report on a first alpha release of the I/O library, ready for WP4*.[1] We recall in this section the key points of the developed library.

The objective of the TREX Input/Output (TREXIO) library is to facilitate inter-operability between codes in the field of quantum chemistry, primarily focused on enabling the communication of data between the flagship codes of the Targeting REal chemical accuracy at the eXascale (TREX) Center of Excellence (CoE): NECI, GAMMCOR, QUANTUM PACKAGE, QMC=CHEM, CHAMP, TURBORVB. In the long term, we expect this library to be also adopted by codes outside of the CoE. Data is read from or written to files in which the electronic wave function, reduced density matrices, integrals, etc. are stored, and an Application Programming Interface (API) is provided to store and retrieve the data in the files. To maximize the portability of the library, the source code is written in the C99 language, and it is released under the BSD-3 clause license.

Hierarchical design

The data stored in the files is organized in different groups:

- Metadata
- Nucleus
- Atomic Orbital (AO)
- AO one-electron integrals
- Effective Core Potential (ECP)
- Electron
- Basis
- Molecular Orbital (MO)
- MO one-electron integrals

and the conventions for the data stored in each group are specified in the documentation (https://trex-coe.github.io/trexio/trex.html). The data groups are defined in a file, `trex.json`, from which the library is automatically generated by a script.

Multiple back ends are proposed to write the data to disk, transparently to the users. The first back end relies on version $\geq$1.8 of the Hierarchical Data Format (HDF5) library, and is expected to be the default choice for production. The second back end is based on simple text files. Its purpose is to reduce the dependencies by allowing the users to install the library even if they are not able to install HDF5. A second purpose of this back end is for situations where the TREXIO files need to be stored in a version control system, where the differences between binary files are difficult to deal with.

[1] https://cordis.europa.eu/project/id/952165/results

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

Data types

A TREXIO file is referenced with a variable of type `trexio_t`. For the data contained in the files, three types are considered: integers, floats and strings. Internally the data is always stored using 64 bits of precision for both floats and integers, but all the functions are available using 32-bit or 64-bit interfaces and the type conversions are done automatically by the library. The default interface is 32-bit for integers and 64-bit for floats. Endianness is taken care of by the HDF5 back end.

As TREXIO is intended to be interfaced with multiple programming languages, it is important to specify precisely the binary representation of numbers. In the C language, the definition of the standard types of integers (`short`, `long int`, *etc*) depends on the architecture on which the code is compiled, and such behavior is not desirable for files which may be used on different architectures. Hence, we use the integer types defined in the `<stdint.h>` header.

In addition, a data type is proposed for the type of back end (`back_end_t`) and for the exit codes of the functions (`trexio_exit_code`). These two types are just explicit redefinitions of `int32t`, and are present to make the interface more readable.

Naming conventions

All the function names of the library are built using simple rules, such that users easily guess the function names without reading the complete documentation. The names are always built as `trexio_[read|write|has]_[group]_[data]`: for example `trexio_read_nucleus_num` reads the variable `num` of group `nucleus`. To avoid confusion, the singular form is chosen for all the names of the groups or variables.

Error handling

All the functions return an exit code, of type `trexio_exit_code`. Upon success of the execution of the function, the exit code is `TREXIO_SUCCESS` (stored internally as `0`). Otherwise, another exit code is returned (see documentation for the complete list), and a textual description of the error can be obtained by passing the returned exit code to the `trexio_string_of_error` function. This gives the complete control on error handling to the users of the library: the library never aborts the calling program.

File locking

The "multiple reader single writer" locking model was implemented, to avoid the accidental production of corrupted files. It is implemented both at the thread level on the front end and at the file level on the back end.

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

2 Changes with respect to the alpha version

Safe functions

In the alpha version, the arrays passed as arguments were given as pointers, and their dimensions were obtained from data stored in the files. For instance, to read the array of nuclear charges, the users would call `trexio_read_nucleus_charge(trexio_file, charges)` where `trexio_file` is a pointer to the TREXIO file and `charges` is a pointer to an array of type `double`. This approach minimizes the number of arguments and makes the API simple, but there is no possibility to check that the arrays provided by the user are large enough to accept the data read from the file, and this can lead to segmentation faults or to a security vulnerability if users provide too small arrays as arguments. To solve this problem, we have introduced safe variants of the functions with an extra argument corresponding to the size of the array. It is now possible to call `trexio_read_safe_nucleus_charge(trexio_file, charges, nucleus_num)` where `nucleus_num` is the expected allocated size of the array. This allows the library to check that the dimension of the array is valid, assuming that the argument passed by the user is correct.

Documentation

The technical documentation was improved. The JavaScript Object Notation (JSON) file containing the definition of the data is now generated using the Emacs text editor from a file in Org syntax containing the documentation of the groups. This file is also used to produce the HyperText Markup Language (HTML) documentation.[2] As a consequence, the documentation is always consistent with the provided library, and the definition of new groups of data is even more user-friendly.

The documentation now contains examples and tutorials to help the adoption of the library by newcomers. Examples are provided in Python, C and Fortran. An additional GitHub repository called `trexio-tutorials`[3] has been created in order to host TREXIO tutorials in the Jupyter notebook format. This approach also facilitates creation of an isolated virtual environment in the cloud (e.g. on Binder[4]). The produced environment can then be used be explore and execute the Jupyter notebooks online without the need to install any software on the user machine. A demo notebook can be launched using this link.

[2] https://trex-coe.github.io/trexio/
[3] https://github.com/TREX-CoE/trexio-tutorials/
[4] https://mybinder.org/

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

3 New features

String handling

String types are now implemented. Special care was needed for these data types as strings are handled differently in C and Fortran, and are well known to be problematic for the Fortran-C interoperability. In Fortran, string sizes are fixed at allocation whereas in C, the size of a string is determined by counting the number of characters until a null terminating character is found. As a consequence, several functions were written to make the proper conversions, such that the user experience is the same in both languages.

Index types

C uses zero-based indexing for arrays while Fortran uses one-based indexing. In other words, the first index of an array is 0 in C and 1 in Fortran. This is problematic for integer arrays which contain data corresponding to indices of other arrays. For example, the `nucleus_index` array of the `basis` group refers to the index of the first shell of each atom in the array of shells. These values are expected to be consistent with the conventions of the language, shifted by one when used in Fortran with respect to C. To solve this issue, a new data type (`index`) was added in the JSON description file for integer variables referring specifically to array indices and index updates are handled by the library.

Python interface

The Python interface to the C library was one of our priorities, as the AiiDA software used in WP 4 is written in Python. In addition, it enables the possibility to create a set of simple tools to manage TREXIO files. For example, we have written a script that computes numerically the overlap matrix in the basis of atomic orbitals, and compare it with the stored overlap matrix. This tool is valuable to developers integrating TREXIO in their software as it can help checking that the conventions used for the definition of the basis set parameters are correct.

Another important aspect of having a Python interface is to enable the use of TREXIO files in Jupyter notebooks. This will have a large benefit for tutorials, teaching, prototyping and interacting with cloud services.

The Python interface was packaged to provide a simple installation procedures via the Pip package manager. To install TREXIO within Python, users simply need to run `pip install trexio`.

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

Continuous integration

The Autotools configuration scripts have been finalized. Automake is now used together with libtool to maximize the portability of the compilation of the library. The library can be configured in two modes. A "maintainer" mode for the TREX developers working on the source code of the generator or on the templates library. This mode requires Emacs to be present on the system to build the library and the documentation. The second mode, the "normal" mode, is dedicated to users of the library who only need to compile the source files and install the library on their system along with the documentation, without requiring to re-generate all the files generated in the "maintainer" mode. This last mode requires less dependencies than the "maintainer mode", and is the default mode provided in the tarballs of the distribution.

Continuous integration was set up with GitHub actions, testing the generation of the library and the compilation in "maintainer" mode, the creation of the distributed tarball and the proper functioning of the library. The library is now automatically tested for the latest x86 Ubuntu and MacOS operating systems. In addition, the library was tested for the Apple M1 ARM processor.

Practical Applications

A plug-in was written for the Quantum Package program to export the computed wave function into a TREXIO file. Several Python scripts[5] were also written to convert output files from the Gaussian and GAMESS codes into TREXIO format.

4 Future work

TREXIO

This first release enables the storage of single determinant wave functions (Hartree-Fock, Kohn-Sham, ...). In the next release, we will introduce the possibility to store multi-determinant wave functions. In addition, we will provide the possibility to store 4-index tensors such as the electron repulsion integrals and the two-body reduced density matrices. This will require the inclusion of new data structures able to handle efficiently sparse arrays, possibly using compression. Finally, we will provide new functions to enable the storage of quantities required for periodic systems.

An updated release of the I/O library is envisioned when submitting deliverable D2.6 *Report on final release of TREX platform with inter-operable flagship codes* at M36 (30/09/2023).

TREX codes

The next step ongoing in our project is the integration of TREXIO in the TREX flagship codes. This will be done using two different scenarios. The first one is to link TREXIO with the code, and call directly its functions. This approach will be used in the QMCKL library, QUANTUM PACKAGE and QMC=CHEM. In other codes such as the AiiDA plugins of the codes or TURBORVB, it will be easier to use the Python interface of TREXIO to produce the program-specific files.

[5] https://github.com/TREX-CoE/trexio_tools

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

A TREXIO Tutorial

This interactive Tutorial covers some basic use cases of the TREXIO library based on the Python API. At this point, it is assumed that the TREXIO Python package has been successfully installed on the user machine or in the virtual environment. If this is not the case, feel free to follow the installation guide.

Importing TREXIO

First of all, let's import the TREXIO package.

```
    try:
        import trexio
    except ImportError:
        raise Exception("Unable to import trexio. Please check that trexio is \
properly installed.")
```

If no error occurs, then it means that the TREXIO package has been successfully imported. Within the current import, TREXIO attributes can be accessed using the corresponding `trexio.attribute` notation. If you prefer to bound a shorter name to the imported module (as commonly done by the NumPy users with `import numpy as np`), this is also possible. To do so, replace `import trexio` with `import trexio as tr` for example. To learn more about importing modules, see the corresponding page of the Python documentation.

Creating a new TREXIO file

TREXIO currently supports two back ends for file I/O:

1. `TREXIO_HDF5`, which relies on extensive use of the HDF5 library and the associated binary file format. This back end is optimized for high performance but it requires HDF5 to be installed on the user machine.

2. `TREXIO_TEXT`, which relies on basic I/O operations that are available in the standard C library. This back end is not optimized for performance but it is supposed to work "out-of-the-box" since there are no external dependencies.

Armed with these new definitions, let's proceed with the tutorial. The first task is to create a TREXIO file called `benzene_demo.h5`. But first we have to remove the file if it exists in the current directory

```
    filename = 'benzene_demo.h5'

    import os
    try:
```

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

```
    os.remove(filename)
except:
    print(f"File {filename} does not exist.")
```

```
File benzene_demo.h5 does not exist.
```

We are now ready to create a new TREXIO file:

```
demo_file = trexio.File(filename, mode='w', back_end=trexio.TREXIO_HDF5)
```

This creates an instance of the `trexio.File` class, which we refer to as `demo_file` in this tutorial. You can check that the corresponding file called `benzene_demo.h5` exists in the root directory. It is now open for writing as indicated by the user-supplied argument `mode'w'`. The `file has been initiated using the TREXIO_HDF5` back end and will be accessed accordingly from now on. The information about back end is stored internally by TREXIO, which means that there is no need to specify it every time the I/O operation is performed. If the file named `benzene_demo.h5` already exists, then it is re-opened for writing (and not truncated to prevent data loss).

Writing data in the TREXIO file

Prior to any work with TREXIO library, we highly recommend users to read about TREXIO internal configuration, which explains the structure of the wavefunction file. The reason is that TREXIO API has a naming convention, which is based on the groups and variables names that are pre-defined by the developers. In this Tutorial, we will only cover contents of the `nucleus` group. Note that custom groups and variables can be added to the TREXIO API.

In this Tutorial, we consider benzene molecule (C_6H_6) as an example. Since benzene has 12 atoms, let's specify it in the previously created `demo_file`. In order to do so, one has to call `trexio.write_nucleus_num` function, which accepts an instance of the `trexio.File` class as a first argument and an `int` value corresponding to the number of nuclei as a second argument.

```
nucleus_num = 12
```

```
trexio.write_nucleus_num(demo_file, nucleus_num)
```

In fact, all API functions that contain `write_` prefix can be used in a similar way. Variables that contain `_num` suffix are important part of the TREXIO file because some of them define dimensions of arrays. For example, `nucleus_num` variable corresponds to the number of atoms, which will be internally used to write/read the `nucleus_coord` array of nuclear coordinates. In order for TREXIO files to be self-consistent, overwriting num-suffixed variables is currently disabled.

The number of atoms is not sufficient to define a molecule. Let's first create a list of nuclear charges, which correspond to benzene.

```
charges = [6., 6., 6., 6., 6., 6., 1., 1., 1., 1., 1., 1.]
```

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

According to the TREX configuration file, there is a `charge` attribute of the `nucleus` group, which has `float` type and `[nucleus_num]` dimension. The `charges` list defined above fits nicely in the description and can be written as follows

```
trexio.write_nucleus_charge(demo_file, charges)
```

Note: TREXIO function names only contain parts in singular form. This means that, both `write_nucleus_charges` and `write_nuclear_charges` are invalid API calls. These functions simply do not exist in the `trexio` Python package and the corresponding error message should appear.

Alternatively, one can provide a list of nuclear labels (chemical elements from the periodic table) that correspond to the aforementioned charges. There is a `label` attribute of the `nucleus` group, which has `str` type and `[nucleus_num]` dimension. Let's create a list of 12 strings, which correspond to 6 carbon and 6 hydrogen atoms:

```
labels = [
    'C',
    'C',
    'C',
    'C',
    'C',
    'C',
    'H',
    'H',
    'H',
    'H',
    'H',
    'H']
```

This can now be written using the corresponding `trexio.write_nucleus_label` function:

```
trexio.write_nucleus_label(demo_file, labels)
```

Two examples above demonstrate how to write arrays of numbers or strings in the file. TREXIO also supports I/O operations on single numerical or string attributes. In fact, in this Tutorial you have already written one numerical attribute: `nucleus_num`. Let's now write a string `'D6h'`, which indicates a point group of benzene molecule. According to the TREX configuration file, `point_group` is a `str` attribute of the `nucleus` group, thus it can be written in the `demo_file` as follows

```
point_group = 'D6h'

trexio.write_nucleus_point_group(demo_file, point_group)
```

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

Writing NumPy arrays (float or int types)

The aforementioned examples cover the majority of the currently implemented functionality related to writing data in the file. It is worth mentioning that I/O of numerical arrays in TREXIO Python API relies on extensive use of the NumPy package. This will be discussed in more details in the section about reading data. However, TREXIO `write_` functions that work with numerical arrays also accept `numpy.ndarray` objects. For example, consider a `coords` list of nuclear coordinates that correspond to benzene molecule

```
coords = [
    [0.00000000  ,  1.39250319 ,  0.00000000 ],
    [-1.20594314 ,  0.69625160 ,  0.00000000 ],
    [-1.20594314 , -0.69625160 ,  0.00000000 ],
    [0.00000000  , -1.39250319 ,  0.00000000 ],
    [1.20594314  , -0.69625160 ,  0.00000000 ],
    [1.20594314  ,  0.69625160 ,  0.00000000 ],
    [-2.14171677 ,  1.23652075 ,  0.00000000 ],
    [-2.14171677 , -1.23652075 ,  0.00000000 ],
    [0.00000000  , -2.47304151 ,  0.00000000 ],
    [2.14171677  , -1.23652075 ,  0.00000000 ],
    [2.14171677  ,  1.23652075 ,  0.00000000 ],
    [0.00000000  ,  2.47304151 ,  0.00000000 ],
    ]
```

Let's take advantage of using NumPy arrays with fixed precision for floating point numbers. But first, try to import the `numpy` package

```
try:
    import numpy as np
except ImportError:
    raise Exception("Unable to import numpy. Please check that numpy is \
properly installed.")
```

You can now convert the previously defined `coords` list into a numpy array with fixed `float64` type as follows

```
coords_np = np.array(coords, dtype=np.float64)
```

TREXIO functions that write numerical arrays accept both lists and numpy arrays as a second argument. That is, both `trexio.write_nucleus_coord(demo_file, coords)` and `trexio.write_nucleus_coord(demo_file, coords_np)` are valid API calls. Let's use the latter and see if it works

```
trexio.write_nucleus_coord(demo_file, coords_np)
```

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

Congratulations, you have just completed the `nucleus` section of the TREXIO file for benzene molecule! Note that TREXIO API is rather permissive and do not impose any strict ordering on the I/O operations. The only requirement is that dimensioning (`_num` suffixed) variables have to be written in the file **before** writing arrays that depend on these variables. For example, attempting to write `nucleus_charge` or `nucleus_coord` fails if `nucleus_num` has not been written.

TREXIO error handling

TREXIO Python API provides the `trexio.Error` class which simplifies exception handling in the Python scripts. This class wraps up TREXIO return codes and propagates them all the way from the C back end to the Python front end. Let's try to write a negative number of basis set shells `basis_num` in the TREXIO file.

```
try:
    trexio.write_basis_num(demo_file, -256)
except trexio.Error as e:
    print(f"TREXIO error message: {e.message}")

TREXIO error message: Invalid argument 2
```

The error message says **Invalid argument 2**, which indicates that the user-provided value -256 is not valid.

As mentioned before, `_num`-suffixed variables cannot be overwritten in the file. But what happens if you accidentally attempt to do so? Let's have a look at the `write_nucleus_num` function as an example:

```
try:
    trexio.write_nucleus_num(demo_file, 24)
except trexio.Error as e:
    print(f"TREXIO error message: {e.message}")

TREXIO error message: Attribute already exists
```

The API rightfully complains that the target attribute already exists and cannot be overwritten.

Alternatively, the aforementioned case can be handled using `trexio.has_nucleus_num` function as follows

```
if not trexio.has_nucleus_num:
    trexio.write_nucleus_num(demo_file, 24)
```

TREXIO functions with `has_` prefix return `True` if the corresponding variable exists and `False` otherwise.

What about writing arrays? Let's try to write an list of 48 nuclear indices instead of 12

```
indices = [i for i in range(nucleus_num*4)]
```

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

```
try:
    trexio.write_basis_nucleus_index(demo_file, indices)
except trexio.Error as e:
    print(f"TREXIO error message: {e.message}")

TREXIO error message: Access to memory beyond allocated
```

According to the TREX configuration file, the `nucleus_index` attribute of a `basis` group is supposed to have `[nucleus_num]` elements. In the example above, we have tried to write 4 times more elements, which might lead to memory and/or file corruption. Luckily, TREXIO internally checks the array dimensions and returns an error in case of inconsistency.

Closing the TREXIO file

It is good practice to close the TREXIO file at the end of the session. In fact, `trexio.File` class has a destructor, which normally takes care of that. However, if you intend to re-open the TREXIO file, it has to be closed explicitly before. This can be done using the `close` method, i.e.

```
demo_file.close()
```

Good! You are now ready to inspect the contents of the `benzene_demo.h5` file using the reading functionality of TREXIO.

Reading data from the TREXIO file

First, let's try to open an existing TREXIO file in read-only mode. This can be done by creating a new instance of the `trexio.File` class but this time with `mode'r'=` argument. Back end has to be specified as well.

```
demo_file_r = trexio.File(filename, mode='r', back_end=trexio.TREXIO_HDF5)
```

When reading data from the TREXIO file, the only required argument is a previously created instance of the `trexio.File` class. In our case, it is `demo_file_r`. TREXIO functions with `read_` prefix return the desired variable as an output. For example, `nucleus_num` value can be read from the file as follows

```
nucleus_num_r = trexio.read_nucleus_num(demo_file_r)

print(f"nucleus_num from {filename} file ---> {nucleus_num_r}")

nucleus_num from benzene_demo.h5 file ---> 12
```

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

The function call assigns `nucleus_num_r` to 12, which is consistent with the number of atoms in benzene that we wrote in the previous section.

All calls to functions that read data can be done in a very similar way. The key point here is the choice of the function name, which in turn defines the output format. Hopefully by now you got used to the TREXIO naming convention and the contents of the `nucleus` group. Which function would you call to read a `point_group` attribute of the `nucleus` group? What type does it return? See the answer below:

```
point_group_r = trexio.read_nucleus_point_group(demo_file_r)

print(f"nucleus_point_group from {filename} TREXIO file ---> {point_group_r}\n")
print(f"Is return type of read_nucleus_point_group a string? ---> {isinstance(point_
```

```
nucleus_point_group from benzene_demo.h5 TREXIO file ---> D6h

Is return type of read_nucleus_point_group a string? ---> True
```

The `trexio.read_nucleus_point_group` function call returns a string `D6h`, which is exactly what we provided in the previous section. Now, let's read nuclear charges and labels.

```
labels_r = trexio.read_nucleus_label(demo_file_r)

print(f"nucleus_label from {filename} file \n---> {labels_r}")
```

```
nucleus_label from benzene_demo.h5 file
---> ['C', 'C', 'C', 'C', 'C', 'C', 'H', 'H', 'H', 'H', 'H', 'H']
```

```
charges_r = trexio.read_nucleus_charge(demo_file_r)

print(f"nucleus_charge from {filename} file \n---> {charges_r}")
```

```
nucleus_charge from benzene_demo.h5 file
---> [6. 6. 6. 6. 6. 6. 1. 1. 1. 1. 1. 1.]
```

The values are consistent with each other and with the previously written data. Not bad. What about the format of the output?

```
print(f"nucleus_label return type: {type(labels_r)}")
```

```
nucleus_label return type: <class 'list'>
```

This makes sense, isn't it? We have written a `list` of nuclear labels and have received back a `list` of values from the file. What about nuclear charges?

```
print(f"nucleus_charge return type: {type(charges_r)}")
```

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

```
nucleus_charge return type: <class 'numpy.ndarray'>
```

It looks like the `trexio.read_nucleus_charge` function returns a `numpy.ndarray` even though we have provided a python-ic `list` to `trexio.write_nucleus_charge` in the previous section. Why is it so? As has been mentioned before, the TREXIO Python API internally relies on the use of the NumPy package to communicate arrays of `float`-like or `int`-like values. This prevents some memory leaks and grants additional flexibility to the API. What kind of flexibility? Check this out:

```
print(f"return dtype in NumPy notation: ---> {charges_r.dtype}")
```

```
return dtype in NumPy notation: ---> float64
```

It means that the default precision of the TREXIO output is double (`np.float64`) for arrays of floating point numbers like `nucleus_charge`. But what if you do not need this extra precision and would like to read nuclear charges in single (`np.float32`) or even reduced (e.g. =np.float16=) precision? TREXIO Python API provides an additional (optional) argument for this. This argument is called `dtype` and accepts one of the NumPy data types. For example,

```
charges_np = trexio.read_nucleus_charge(demo_file_r, dtype=np.float32)

print(f"return dtype in NumPy notation: ---> {charges_np.dtype}")
```

```
return dtype in NumPy notation: ---> float32
```

Reading multidimensional arrays

So far, we have only read flat 1D arrays. However, we have also written a 2D array of nuclear coordinates. Let's now read it back from the file:

```
coords_r = trexio.read_nucleus_coord(demo_file_r)

print(f"nucleus_coord from {filename} TREXIO file: \n{coords_r}")
```

```
nucleus_coord from benzene_demo.h5 TREXIO file:
[[ 0.          1.39250319  0.        ]
 [-1.20594314  0.6962516   0.        ]
 [-1.20594314 -0.6962516   0.        ]
 [ 0.         -1.39250319  0.        ]
 [ 1.20594314 -0.6962516   0.        ]
 [ 1.20594314  0.6962516   0.        ]
 [-2.14171677  1.23652075  0.        ]
 [-2.14171677 -1.23652075  0.        ]
 [ 0.         -2.47304151  0.        ]
 [ 2.14171677 -1.23652075  0.        ]
 [ 2.14171677  1.23652075  0.        ]
 [ 0.          2.47304151  0.        ]]
```

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

```
print(f"return shape: ---> {coords_r.shape}")

return shape: ---> (12, 3)
```

We can see that TREXIO returns a 2D array with 12 rows and 3 columns, which is consistent with the `nucleus_coord` dimensions `[nucleus_num, 3]`. What this means is that **by default TREXIO reshapes the output flat array into a multidimensional one** whenever applicable. This is done based on the shape specified in the TREX configuration file.

In some cases, it might be a good idea to explicitly check that the data exists in the file before reading it. This can be achieved using `has_`-suffixed functions of the API. For example,

```
if trexio.has_nucleus_coord(demo_file_r):
    coords_safer = trexio.read_nucleus_coord(demo_file_r)
```

Conclusion

In this Tutorial, you have created a TREXIO file using the HDF5 back end and have written the number of atoms, point group, nuclear charges, labels and coordinates, which correspond to benzene molecule. You have also learned how to read this data back from the TREXIO file and how to handle some TREXIO errors.

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.