

TREXIO: A File Format and Library for Quantum Chemistry

Evgeny Posenitskiy,^{1,2} Vijay Gopal Chilkuri,^{1,3} Abdallah Ammar,¹ Michał Hapka,⁴ Katarzyna Pernal,⁵ Ravindra Shinde,⁶ Edgar Josué Landinez Borda,⁶ Claudia Filippi,⁶ Kosuke Nakano,^{7,8} Otto Kohulák,^{8,1} Sandro Sorella,⁸ Pablo de Oliveira Castro,⁹ William Jalby,⁹ Pablo López Ríos,¹⁰ Ali Alavi,¹⁰ and Anthony Scemama^{1, a)}

¹⁾ *Laboratoire de Chimie et Physique Quantiques (LCPQ), Université de Toulouse (UPS) and CNRS, Toulouse, France*

²⁾ *Qubit Pharmaceuticals, Incubateur Paris Biotech Santé, 24 Rue du Faubourg Saint Jacques, 75014 Paris, France*

³⁾ *Aix Marseille Univ, CNRS, Centrale Marseille, ISM2, Marseille, France*

⁴⁾ *Faculty of Chemistry, University of Warsaw, ul. L. Pasteura 1, 02-093 Warsaw, Poland*

⁵⁾ *Institute of Physics, Lodz University of Technology, ul. Wolczanska 217/221, 93-005 Lodz, Poland*

⁶⁾ *MESA+ Institute for Nanotechnology, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands*

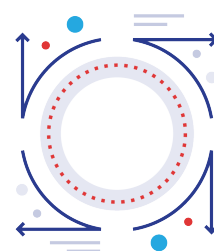
⁷⁾ *Research and Services Division of Materials Data and Integrated System, National Institute for Materials Science (NIMS), Tsukuba, Ibaraki 305-0047, Japan*

⁸⁾ *International School for Advanced Studies (SISSA), Via Bonomea 265, 34136, Trieste, Italy*

⁹⁾ *Université Paris-Saclay, UVSQ, LI-PaRAD*

¹⁰⁾ *Max Planck Institute for Solid State Research, Heisenbergstrasse 1, 70569 Stuttgart, Germany*

TREXIO is an open-source file format and library developed for the storage and manipulation of data produced by quantum chemistry calculations. It is designed with the goal of providing a reliable and efficient method of storing and exchanging wave function parameters and matrix elements, making it an important tool for researchers in the field of quantum chemistry. In this work, we present an overview of the TREXIO file format and library. The library consists of a front-end implemented in the C programming language and two different back-ends: a text back-end and a binary back-end utilizing the HDF5 library which enables fast read and write operations. It is compatible with a variety of platforms and has interfaces for the Fortran, Python, and OCaml programming languages. In addition, a suite of tools has been developed to facilitate the use of the TREXIO format and library, including converters for popular quantum chemistry codes and utilities for validating and manipulating data stored in TREXIO files. The simplicity, versatility, and ease of use of TREXIO make it a valuable resource for researchers working with quantum chemistry data.



Keywords: quantum chemistry, data, interoperability

I. INTRODUCTION

Quantum chemistry relies on quantum mechanics to explain and predict the properties and behaviors of atoms, molecules, and materials. Although density functional theory (DFT) is one of the most widely used approaches thanks to its excellent ratio between computational cost and accuracy, another important tool is wave function theory (WFT), which describes the behavior of a quantum system in terms of its wave function. In order to perform WFT calculations, it is necessary to manipulate a large number of parameters, such as the expansion coefficients of the wave function and the matrix elements of the Hamiltonian operator. These parameters are typically numerous and difficult to handle, making it important to have a robust and efficient method for storing and accessing them.

Reproducible research remains a challenging topic, despite recent advances such as the introduction of the

FAIR (findable, accessible, interoperable, reusable) data principles.¹ A key aspect of reproducibility is software interoperability, which refers to the ability of different programs to work together and exchange information, allowing different systems to communicate and exchange data in order to function as a cohesive whole. Interoperable software is prevalent nowadays and is a key component of the Unix philosophy.² In Unix shells, the most straightforward application of software interoperability is made through the use of the *pipe* operator, where the output of a program is the input of another program. Similarly, shell scripts are created through the composition of smaller programs, exchanging data through files or pipes.

A major challenge of reproducible research is the uniformity of input/output (I/O) data within a particular research domain. The Unix philosophy recommends the use of text files because they are architecture-independent, readable in any language, and can be read as a stream, which is useful for making programs communicate over a network. However, storing data in a text format can result in large file sizes and conversion from ASCII to binary format can be computationally expen-

^{a)} Electronic mail: scemama@irsamc.ups-tlse.fr

sive for large data sets. To address this concern, domain-specific binary formats have been developed, such as the Joint Photographic Experts Group (JPEG) format³ for digital images and the Moving Picture Experts Group (MPEG) format⁴ for videos. These binary formats are utilized through standardized application programming interfaces (API).

In the field of wave function theory such a standard format and API is still lacking, and the purpose of the TREXIO library presented in this article is to fill this gap. This paper is organized as follows: firstly, a brief overview of the related work is presented. Secondly, the TREXIO format for the electronic wave functions is introduced together with some details concerning the internal representation and the associated API. Finally, some applications are demonstrated with a major focus on the interoperability achieved within the TREX Center of Excellence in Exascale Computing⁵ due to the use of the TREXIO format.

II. RELATED WORK

It is worth mentioning that there have been several efforts to unify the data formats within different subdomains of quantum chemistry. Probably one of the earliest works in this direction was the definition of the Crystallographic Information File (CIF) for establishing databases of crystal structures.⁶ A few years later, the Chemical Markup Language (CML)^{7,8} was introduced. It is a format based on the Extensible Markup Language (XML) which is used to describe chemical data: molecules, chemical properties, reactions, spectra, materials, *etc.* With formats like CIF or CML, the burden of following a standard is placed on the code *writing* the data. As a consequence, any tool that can read the format will be able to interpret the data without needing to understand the specific code that was used to produce it. This means that data can be easily shared and reused across different programs, and new tools can be developed to work with the format without needing to know anything about the code used to produce the data.

Recently, the `cclib` Python package⁹, originally developed for performing computational chemistry calculations, has accumulated several internal converters capable of parsing and transforming the output of different programs into the internal representation called `ccData`. A similar approach has been taken by the developers of `IOData`¹⁰, who have implemented converters and parsers for commonly used programs and their output files. However, there is currently no unified data representation or API that can be integrated into quantum chemistry codes to improve interoperability. Consequently, each time a given program modifies its input/output formatting, the `IOData` package must be adapted accordingly and promptly, which poses an additional challenge for maintainers. More recently, consolidated efforts have given rise to `QCSchema`¹¹, which provides an API-like access

to data generated by existing quantum chemistry codes, thereby addressing the issue of dependence on the output file's formatting style. In this case, the responsibility for adhering to conventions falls on the code *reading* the data, as it must be aware of the conventions chosen by the code that generated the data. With the Electronic Structure Common Data Format (ESCDF)¹² and its associated library, codes that write data can supply metadata to assist codes that read data in comprehending the organization of the data in the files. Hence, ESCDF aims to provide low-level tools and flexibility to facilitate the exchange of large datasets between codes with high-performance I/O. While this greatly reduces the difficulty of understanding conventions for developers reading the data, they may still need to apply different conversions depending on the code that generated the data. Consequently, implementing support for ESCDF may require more effort on the part of code developers compared to using a standardized format such as CML.

Another popular format for storing quantum chemistry data is the Gaussian¹³ `fchk` format. While it is a proprietary format specific to the Gaussian software package, its compatibility with several other software programs has contributed to its extensive utilization. However, the format's proprietary and closed-source nature prevents external developers from improving the format, leaving enhancements and compatibility updates solely in the hands of Gaussian developers.

Recently, the `mwfn`¹⁴ format was introduced with the primary goal of enhancing the existing solutions such as `wfn`,¹³ `wfx`,¹⁵ and `Molden`¹⁶ formats, which were designed to store parameters of molecular orbitals and atomic basis sets in view of reconstructing the one-particle density matrix. Although `mwfn` is an improvement on these other formats, it does not allow the user to store enough information for a wave function coming from a configuration interaction (CI) or coupled cluster (CC) calculation.

For post-Hartree-Fock calculations, the `FCIDUMP` format¹⁷ has become a *de facto* standard because of its simplicity. It is a text-based format that only contains minimal information for building the second-quantized Hamiltonian, namely the one- and two-electron integrals in the basis of molecular orbitals (MO), the number of electrons and information about the spin state and orbital symmetries. The nuclear coordinates and basis set are not saved in `FCIDUMP` files. The text format makes its adoption extremely simple, but it has a very high impact on the performance since `FCIDUMP` files are usually large. Although very practical, the use of the `FCIDUMP` format has other important limitations than efficiency. Once a program has computed a post-Hartree-Fock wave function using an `FCIDUMP` file as an input, the parameters of the basis set and the molecular orbitals may have been lost unless they were stored in a separate file in another format. Although configuration interaction or coupled cluster calculations can be performed using `FCIDUMP` files, this format is too limited to be used for quantum Monte Carlo (QMC) calculations, which require *all* the

wave function parameters.

The Q5Cost^{18–20} initiative was one of the first attempts aiming at standardizing the WFT data by introducing both a format and the API to interact with it. With Q5Cost, it was possible to store all the wave function parameters of CI expansions together with the basis set, molecular orbitals, and even electron repulsion integrals. The Q5Cost library was relying on the Hierarchical Data Format version 5 (HDF5)²¹ to provide efficient I/O and keep the data well organized in the file. Nevertheless, Q5Cost had some severe drawbacks. First, Q5Cost was written in Fortran which made its use tedious in other programming languages such as C++ or Python. In addition, to be able to interpret a Q5Cost file, it was often necessary to know which code had generated it. Indeed, most WFT codes have different conventions in terms of normalization of the basis functions, ordering of the atomic orbitals, *etc.*, and no conversion into a unique internal representation was imposed by the library. So the burden of understanding conventions was still on the shoulders of the readers of the files. Finally, Q5Cost had important technical limitations: the Q5Cost library was intended to be used as a compiled Fortran module (a so-called `.mod` file), that depended on the compiled Fortran modules provided by the HDF5 library. As the format of the compiled Fortran modules is specific to the compiler vendor and even to the version of the compiler, the Q5Cost library could not be simply linked as an external library to any code. Using the Q5Cost library in a Fortran code imposed that the user’s code was compiled with the same Fortran compiler as the one that was used to compile both the HDF5 Fortran modules and the Q5Cost library. This contamination of dependencies could lead to some important impact on the performance of the user’s code, and the only solution to solve that problem was to compile many different versions of the HDF5 Fortran interface and Q5Cost library with multiple compilers and compiler versions.

The TREXIO initiative, heavily influenced by the Q5Cost project, aims to propose a standard format and library for wave function calculations. This initiative seeks to leverage the strengths of the Q5Cost project and learn from its design flaws that hindered its widespread adoption. One of the key improvements we aim to achieve is to shift the effort of adopting a format and conventions to the side of the code writing the data. This way, the files will be easily readable without any prior knowledge by any code, similar to CML or JPEG.

III. THE TREXIO FORMAT

The TREXIO format (version 2.3.0) is designed to store all the necessary information to represent a wave function, including: the number of up- and down-spin electrons, nuclear coordinates and charges, basis set and effective core potential (ECP) parameters, atomic and molecular orbital parameters, Slater determinants and

CI coefficients, configuration state function (CSF) definitions, and metadata related to the description of excited states. It is also capable of storing data required for the computation of the wave function, such as one- and two-electron integrals, numerical integration grids used in DFT calculations, and one- and two-particle reduced density matrices.

One notable feature of TREXIO is that it is self-contained, meaning that all the parameters needed to recreate the wave function are explicitly stored within the file, eliminating the need for external databases. For example, instead of storing the name of a basis set (such as `cc-pVDZ`), the actual basis set parameters used in the calculation are stored. All data are stored in atomic units for simplicity.

The data in TREXIO are organized into *groups*, each containing multiple *attributes* defined by their *type* and *dimensions*. Each attribute within a group corresponds to a single scalar or array variable in a code. In what follows, the notation `<group>.<attribute>` will be used to identify an attribute within a group. For example, `nucleus.charge` refers to the `charge` attribute in the `nucleus` group. It is an array of type `float` with dimension `nucleus.num`, the attribute describing the number of nuclei. For simplicity, the singular form is always used for the names of groups and attributes.

A. Data types

So that TREXIO can be used in any language, we use a limited number of data types. It is important to keep in mind that these types are abstract in the sense that they are defined independently of their implementation, and are not tied to any specific representation on a computer. The main data types are `int` for integers, `float` for floating-point values, and `str` for character strings. The real and imaginary parts of complex numbers are stored separately as `floats`. To minimize the risk of integer overflow and accuracy loss, numerical data types are stored using 64-bit representations by default. However, in specific cases where integers are bounded (such as orbital indices in four-index integrals), the smallest possible representation is used to reduce the file size. The API presented in the next section handles any necessary type conversions.

There are also two types derived from `int`: `dim` and `index`. `dim` is used for dimensioning variables, which are positive integers used to specify the dimensions of an array. In the previous example, `nucleus.num` is a dimensioning variable that specifies the dimensions of the `nucleus.charge` array. `index` is used for integers that correspond to array indices, because some languages (such as C or Python) use zero-based indexing, while others (such as Fortran) use one-based indexing by default. For convenience, values of the `index` type are shifted by one when TREXIO is used in one-based languages to be consistent with the semantics of the language.

Arrays can be stored in either dense or sparse formats. If the sparse format is selected, the data is stored in coordinate format. For example, the element $\mathbf{A}(i, j, k, l)$ is stored as a quadruplet of integers (i, j, k, l) along with the corresponding value. Typically, one- and two-dimensional arrays are stored as dense arrays, while arrays with higher dimensions are stored in sparse format.

B. Stored data

In this section, we provide a comprehensive overview of the data that can be stored in TRESPIO files. A complete list of the groups and attributes is available as supplementary information or in the documentation of the library. In both resources, multi-dimensional arrays are expressed in column-major order, meaning that elements of the same column are stored contiguously.

1. Metadata

In order to facilitate the archiving of TRESPIO files in open-data repositories, users have the option to store metadata in the `metadata` group. This includes the names of the codes that were used to create the file, a list of authors, and a textual description. This allows for more information about the file to be easily accessible and transparent.

2. System information

The chemical system consists of nuclei and electrons, where the nuclei are considered as fixed point charges with Cartesian coordinates. The wave function is stored in the spin-free formalism,²² and therefore, it is necessary to explicitly store the number of spin-up (N_\uparrow) and spin-down (N_\downarrow) electrons. These numbers correspond to the normalization of the spin-up and spin-down single-particle reduced density matrices.

Certain calculations, such as DFT calculations, require the use of a numerical integration grid. The `grid` group provides information for storing grids, inspired by the data required by the `numgrid` software.^{23,24}

To keep things simple, TRESPIO can only store a single wave function per file. When working with excited states, it is often the case that multiple states only differ in their CI coefficients, while other parameters (such as geometry, basis set, molecular orbitals, etc.) are the same. To facilitate the storage of multiple states, TRESPIO provides the option to store all the data needed to describe one state in a main file, along with the names of additional TRESPIO files that contain only the state-specific parameters.

3. Basis set

In the `basis` group, the atomic basis set is defined as a list of shells. Each shell i is centered at a center A_i , has a specified angular momentum l_i , and a radial function R_i . The radial function is a linear combination of $N_{\text{prim } i}$ primitive functions, which can be Slater type orbitals (STO, $p = 1$) or Gaussian type orbitals (GTO, $p = 2$). These primitive functions are parameterized by exponents γ_{ki} and coefficients a_{ki} :

$$R_i(\mathbf{r}) = \mathcal{N}_i |\mathbf{r} - \mathbf{R}_{A_i}|^{n_i} \sum_{k=1}^{N_{\text{prim } i}} a_{ki} f_{ki}(\gamma_{ki}, p) e^{-\gamma_{ki} |\mathbf{r} - \mathbf{R}_{A_i}|^p}. \quad (1)$$

Different codes have different normalization practices, so it is necessary to store normalization factors in the TRESPIO file to ensure that it is self-contained and does not rely on the client program having the ability to compute overlap integrals. Some codes assume that the contraction coefficients are applied to *normalized* linear combinations of primitives, so a normalization constant f_{ki} for each primitive must also be stored. Some codes assume that the functions R_i are normalized, requiring the computation of an additional normalization factor, \mathcal{N}_i .

4. Atomic orbitals

The `ao` group in TRESPIO contains information related to the expansion of the shells in the basis set into atomic orbitals (AOs). For example, a p -shell is expanded into three AOs: p_x , p_y , and p_z . AOs are defined as follows:

$$\chi_i(\mathbf{r}) = \mathcal{N}'_i P_{\eta(i)}(\mathbf{r}) R_{s(i)}(\mathbf{r}) \quad (2)$$

where i is the atomic orbital index, P refers to either polynomials or spherical harmonics, and $s(i)$ specifies the shell on which the AO is expanded.

$\eta(i)$ denotes the chosen angular function. The AOs can be expressed using real spherical harmonics or polynomials in Cartesian coordinates. In the case of real spherical harmonics, the AOs are ordered as $0, +1, -1, +2, -2, \dots, +m, -m$. In the case of polynomials, the canonical (or alphabetical) ordering is used,

$$\begin{aligned} p &: p_x, p_y, p_z \\ d &: d_{xx}, d_{xy}, d_{xz}, d_{yy}, d_{yz}, d_{zz} \\ f &: f_{xxx}, f_{xxy}, f_{xxz}, f_{xyy}, f_{xyz}, f_{xzz}, f_{yyy}, f_{yyz}, f_{yzz}, f_{zzz} \\ &\vdots \end{aligned}$$

Note that for p orbitals in real spherical harmonics, the ordering is $0, +1, -1$ which corresponds to p_z, p_x, p_y .

\mathcal{N}'_i is a normalization factor that allows for different normalization coefficients within a single shell, as in the GAMESS²⁵ convention where each individual function is unit-normalized. Using GAMESS convention, the normalization factor of the shell \mathcal{N}_d (Eq. 1) in the `basis`

group is appropriate for instance for the d_z^2 function (i.e. $\mathcal{N}_d \equiv \mathcal{N}_{z^2}$) but not for the d_{xy} AO, so the correction factor \mathcal{N}'_i for d_{xy} in the `ao` groups is the ratio $\frac{\mathcal{N}_{xy}}{\mathcal{N}_{z^2}}$.

5. Effective core potentials

An effective core potential (ECP) V_A^{ECP} can be used to replace the core electrons of atom A. It can be expressed as:²⁶

$$V_A^{\text{ECP}} = V_{A\ell_{\max}+1} + \sum_{\ell=0}^{\ell_{\max}} \delta V_{A\ell} \sum_{m=-\ell}^{\ell} |Y_{\ell m}\rangle \langle Y_{\ell m}| \quad (3)$$

The first term in this equation is attributed to the local channel, while the remaining terms correspond to non-local channel projections. ℓ_{\max} refers to the maximum angular momentum in the non-local component of the ECP. The functions $\delta V_{A\ell}$ and $V_{A\ell_{\max}+1}$ are parameterized as:

$$\delta V_{A\ell}(\mathbf{r}) = \sum_{q=1}^{N_{q\ell}} \beta_{Aq\ell} |\mathbf{r} - \mathbf{R}_A|^{n_{Aq\ell}} e^{-\alpha_{Aq\ell} |\mathbf{r} - \mathbf{R}_A|^2}$$

$$V_{A\ell_{\max}+1}(\mathbf{r}) = -\frac{Z_{\text{eff}}}{|\mathbf{r} - \mathbf{R}_A|} + \delta V_{A\ell_{\max}+1}(\mathbf{r}) \quad (4)$$

where Z_{eff} is the effective nuclear charge of the center. All the parameters can be stored in the `ecp` group.

6. Molecular orbitals

The `mo` group is devoted to the storage of the molecular orbitals (MOs). MO coefficients are stored in a two-dimensional array, with additional information such as symmetries or occupation numbers stored in separate arrays. It is also possible to store the spin to enable the description of unrestricted Hartree-Fock or unrestricted Kohn-Sham determinants.

7. Hamiltonian matrix elements

One-electron integrals can be stored in the AO and MO bases in the groups `ao_1e_int` and `mo_1e_int`, respectively. Similarly, two-electron integrals can be stored in the AO and MO bases in the groups `ao_2e_int` and `mo_2e_int`, respectively. One-electron integrals are stored as two-dimensional arrays, while two-electron integrals are stored in a sparse format, with a quadruplet of indices and the corresponding value stored for each non-zero integral. The order of the indices follows Dirac's bra-ket notation.

It is also possible to store a low-rank representation of the two-electron integrals, obtained via a Cholesky decomposition.

8. CI expansion

The wave function Ψ can be represented as a combination of Slater determinants D_I :

$$|\Psi\rangle = \sum_I C_I |D_I\rangle \quad (5)$$

In the `determinant` group of a TREXIO file, the definition of these Slater determinants, as well as the configuration interaction (CI) expansion coefficients, can be stored. Each Slater determinant is represented as a Waller-Hartree double determinant,²⁷ i.e. the product of a determinant with \uparrow -spin electrons and a determinant with \downarrow -spin electrons. To enable the storage of arbitrary CI expansions and to reduce the storage size, the determinants are stored as pairs of binary strings: one for the \uparrow spin sector and one for the \downarrow spin. Each binary string has a length equal to the number of MOs, with the i -th bit set to one if and only if the i -th MO is included in the determinant. As the creation of these binary strings may be tedious, we provide some helper functions to transform lists of orbital indices into binary strings. If the orbital indices are not in increasing order, a reordering is made and the user is informed if a change of sign is needed in the corresponding CI coefficient.

Alternatively, the wave function may be expanded in a basis of configuration state functions (CSFs),

$$|\Psi\rangle = \sum_I \tilde{C}_I |\psi_I\rangle. \quad (6)$$

where each CSF ψ_I is a linear combination of Slater determinants. The `csf` group allows for the storage of the CSF expansion coefficients, as well as the matrix $\langle D_I | \psi_J \rangle$ in a sparse format. This enables the projection of the CSFs onto the basis of Slater determinants.

9. Amplitudes

The wave function may also be expressed in terms of the action of the cluster operator \hat{T} :

$$\hat{T} = \hat{T}_1 + \hat{T}_2 + \hat{T}_3 + \dots \quad (7)$$

on a reference wave function Ψ , where \hat{T}_1 is the single excitation operator,

$$\hat{T}_1 = \sum_{ia} t_i^a \hat{a}_a^\dagger \hat{a}_i, \quad (8)$$

\hat{T}_2 is the double excitation operator,

$$\hat{T}_2 = \frac{1}{4} \sum_{ijab} t_{ij}^{ab} \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_j \hat{a}_i, \quad (9)$$

etc. Indices i, j, a and b denote molecular orbital indices.

Wave functions obtained with perturbation theory or configuration interaction are of the form:

$$|\Phi\rangle = \hat{T}|\Psi\rangle \quad (10)$$

and coupled-cluster wave functions are of the form:

$$|\Phi\rangle = e^{\hat{T}}|\Psi\rangle \quad (11)$$

The reference wave function Ψ is stored using the `determinant` and/or `csf` groups, and the amplitudes are stored using the `amplitude` group. The attributes with the `exp` suffix correspond to exponentialized operators.

10. Reduced density matrices

The reduced density matrices, stored in the `rdm` group, are defined in the basis of molecular orbitals.

The \uparrow -spin and \downarrow -spin components of the one-body density matrix are given by

$$\gamma_{ij}^{\uparrow} = \langle \Psi | \hat{a}_{i\alpha}^{\dagger} \hat{a}_{j\alpha} | \Psi \rangle \quad (12)$$

$$\gamma_{ij}^{\downarrow} = \langle \Psi | \hat{a}_{i\beta}^{\dagger} \hat{a}_{j\beta} | \Psi \rangle \quad (13)$$

and the spin-summed two-body density matrix is

$$\gamma_{ij} = \gamma_{ij}^{\uparrow} + \gamma_{ij}^{\downarrow} \quad (14)$$

The $\uparrow\uparrow$, $\downarrow\downarrow$, and $\uparrow\downarrow$ components of the two-body density matrix are given by

$$\Gamma_{ijkl}^{\uparrow\uparrow} = \langle \Psi | \hat{a}_{k\alpha}^{\dagger} \hat{a}_{l\alpha}^{\dagger} \hat{a}_{j\alpha} \hat{a}_{i\alpha} | \Psi \rangle \quad (15)$$

$$\Gamma_{ijkl}^{\downarrow\downarrow} = \langle \Psi | \hat{a}_{k\beta}^{\dagger} \hat{a}_{l\beta}^{\dagger} \hat{a}_{j\beta} \hat{a}_{i\beta} | \Psi \rangle \quad (16)$$

$$\Gamma_{ijkl}^{\uparrow\downarrow} = \langle \Psi | \hat{a}_{k\alpha}^{\dagger} \hat{a}_{l\beta}^{\dagger} \hat{a}_{j\beta} \hat{a}_{i\alpha} | \Psi \rangle + \langle \Psi | \hat{a}_{l\alpha}^{\dagger} \hat{a}_{k\beta}^{\dagger} \hat{a}_{i\beta} \hat{a}_{j\alpha} | \Psi \rangle, \quad (17)$$

and the spin-summed one-body density matrix is

$$\Gamma_{ijkl} = \Gamma_{ijkl}^{\uparrow\uparrow} + \Gamma_{ijkl}^{\downarrow\downarrow} + \Gamma_{ijkl}^{\uparrow\downarrow}. \quad (18)$$

11. Correlation factors

Explicit correlation factors can be introduced in the wave function, such as in QMC, F_{12} , or transcorrelated methods.

In the current version of the library, it is possible to store two different types of Jastrow factors. The Jastrow factor is an N -electron function which multiplies the reference wave function expansion: $\Psi = \Phi \times \exp(J)$, where

$$J(\mathbf{r}, \mathbf{R}) = J_{\text{eN}}(\mathbf{r}, \mathbf{R}) + J_{\text{ee}}(\mathbf{r}) + J_{\text{eeN}}(\mathbf{r}, \mathbf{R}). \quad (19)$$

In the following, we use the notations $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ and $R_{i\alpha} = |\mathbf{r}_i - \mathbf{R}_{\alpha}|$, where indices i and j correspond to electrons and α to nuclei.

The first form of Jastrow factor is the one used in the CHAMP²⁸ program.²⁹ J_{eN} contains electron-nucleus terms:

$$J_{\text{eN}}(\mathbf{r}, \mathbf{R}) = \sum_{i=1}^{N_{\text{elec}}} \sum_{\alpha=1}^{N_{\text{nuc1}}} \left[\frac{a_{1,\alpha} f_{\alpha}(R_{i\alpha})}{1 + a_{2,\alpha} f_{\alpha}(R_{i\alpha})} + \sum_{p=2}^{N_{\text{ord}}^a} a_{p+1,\alpha} [f_{\alpha}(R_{i\alpha})]^p - J_{\text{eN}}^{\infty} \right] \quad (20)$$

J_{ee} contains electron-electron terms:

$$J_{\text{ee}}(\mathbf{r}) = \sum_{i=1}^{N_{\text{elec}}} \sum_{j=1}^{i-1} \left[\frac{\frac{1}{2}(1 + \delta_{ij}^{\uparrow\downarrow}) b_1 f_{\text{ee}}(r_{ij})}{1 + b_2 f_{\text{ee}}(r_{ij})} + \sum_{p=2}^{N_{\text{ord}}^b} b_{p+1} [f_{\text{ee}}(r_{ij})]^p - J_{\text{ee},ij}^{\infty} \right] \quad (21)$$

where $\delta_{ij}^{\uparrow\downarrow}$ is zero when the electrons i and j have the same spin, and one otherwise. J_{eeN} contains electron-electron-nucleus terms:

$$J_{\text{eeN}}(\mathbf{r}, \mathbf{R}) = \sum_{\alpha=1}^{N_{\text{nuc1}}} \sum_{i=1}^{N_{\text{elec}}} \sum_{j=1}^{i-1} \sum_{p=2}^{N_{\text{ord}}} \sum_{k=0}^{p-1} \sum_{l=0}^{p-k-2\delta_{k,0}} c_{lkp\alpha} [g_{\text{ee}}(r_{ij})]^k \left[[g_{\alpha}(R_{i\alpha})]^l + [g_{\alpha}(R_{j\alpha})]^l \right] [g_{\alpha}(R_{i\alpha}) g_{\alpha}(R_{j\alpha})]^{(p-k-l)/2}, \quad (22)$$

$c_{lkp\alpha}$ being non-zero only when $p - k - l$ is even. The terms J_{ee}^{∞} and J_{eN}^{∞} are shifts to ensure that J_{ee} and J_{eN} have an asymptotic value of zero. f and g are scaling functions defined as

$$f_{\alpha}(r) = \frac{1 - e^{-\kappa_{\alpha} r}}{\kappa_{\alpha}} \quad \text{and} \quad g_{\alpha}(r) = e^{-\kappa_{\alpha} r}, \quad (23)$$

and the possible presence of an index α indicates that the scaling coefficient κ depends on the atom α .

The second form of Jastrow factor is the μ Jastrow factor³⁰

$$J_{\text{ee}}(\mathbf{r}) = \sum_{i=1}^{N_{\text{elec}}} \sum_{j=1}^{i-1} r_{ij} (1 - \text{erf}(\mu r_{ij})) - \frac{1}{\mu\sqrt{\pi}} e^{-(\mu r_{ij})^2}. \quad (24)$$

It is a single parameter correlation factor that has been recently introduced in the context of transcorrelated methods. It imposes the electron-electron cusp and is built such that the leading order in $1/r_{12}$ of the effective two-electron potential reproduces the long-range interaction of the range-separated density functional theory. An envelope function has then been introduced to cancel out the Jastrow effects between two electrons when at least one electron is close to a nucleus, and standard one-body terms were also introduced to avoid the expansion of the one-body density.

As there exist multiple forms of Jastrow factors in the literature, contributions to extend this section are welcome.

12. QMC data

We also provide in the `qmc` group some information specific to QMC calculations. In QMC methods, the wave function is evaluated at points in the $3N$ -dimensional space, where N is the number of electrons. It might be convenient to store the coordinates of points together with the wave function, and to store the value of the wave function and the local energy $\hat{H}\Psi(\mathbf{r})/\Psi(\mathbf{r})$ evaluated at these points, for example, to check that different codes give the same values.

IV. THE TREXIO LIBRARY

The TREXIO library is written in the C language, and is licensed under the open-source 3-clause BSD license to allow for use in all types of quantum chemistry software, whether commercial or not.

The design of the library is divided into two main sections: the front-end and the back-end. The front-end serves as the interface between users and the library, while the back-end acts as the interface between the library and the physical storage.

A. The front-end

By using the TREXIO library, users can store and extract data in a consistent and organized manner. The library provides a user-friendly API, including functions for reading, writing, and checking for the existence of data. The functions follow the pattern `trexio_[has|read|write]_<group>_<attribute>`, where the group and attribute specify the particular data being accessed. It also includes an error handling mechanism, in which each function call returns an exit code of type `trexio_exit_code`, explaining the type of error. This can be used to catch exceptions and improve debugging in the upstream user application. Figures 1 and 2 show examples of usage of the TREXIO library in C and Python, respectively.

To ensure the consistency of the data, the attributes can only be written if all the other attributes on which they explicitly depend have been written. For example, as the `nucleus.coord` array is dimensioned by the number of nuclei `nucleus.num`, the `nucleus.coord` attribute can only be written after `nucleus.num`. However, the library is not aware of non-explicit dependencies, such as the relation between the electron repulsion integrals (ERIs) and MO coefficients. A complete control of the consistency of the data is therefore impossible, so the attributes were chosen to be by default *immutable*. By only allowing data to be written only once, the risk of modifying data in a way that creates inconsistencies is reduced. For example, if the ERIs have already been written, it would be inconsistent to later modify the MO

```

1 #include <stdio.h>
2 #include <trexio.h>
3
4 int main() {
5     int num = 3; // Number of atoms
6     double coord[][3] = {
7         // xyz coordinates in atomic units
8         { 0. , 0. , -0.24962655,
9         { 0. , 2.70519714, 1.85136466,
10        { 0. , -2.70519714, 1.85136466 };
11
12    trexio_exit_code rc;
13
14    // Open the TREXIO file
15    trexio_t* f = trexio_open("water.trexio",
16                            'w', TREXIO_HDF5, &rc);
17    if (rc != TREXIO_SUCCESS) {
18        fprintf(stderr, "Error: %s\n",
19                trexio_string_of_error(rc));
20        return -1;
21    }
22
23    // Write the number of nuclei
24    rc = trexio_write_nucleus_num(f, num);
25    if (rc != TREXIO_SUCCESS) {
26        fprintf(stderr, "Error: %s\n",
27                trexio_string_of_error(rc));
28        return -1;
29    }
30
31    // Write the nuclear coordinates
32    rc = trexio_write_nucleus_coord(f, &coord[0][0]);
33    if (rc != TREXIO_SUCCESS) {
34        fprintf(stderr, "Error: %s\n",
35                trexio_string_of_error(rc));
36        return -1;
37    }
38
39    // Close the TREXIO file
40    rc = trexio_close(f);
41    if (rc != TREXIO_SUCCESS) {
42        fprintf(stderr, "Error: %s\n",
43                trexio_string_of_error(rc));
44        return -1;
45    }
46    return 0;
47 }

```

FIG. 1. C code writing the nuclear coordinates of a water molecule in a TREXIO file, with error handling.

```

1 import trexio
2 coord = [ # xyz coordinates in atomic units
3     [0. , 0. , -0.24962655],
4     [0. , 2.70519714, 1.85136466],
5     [0. , -2.70519714, 1.85136466]
6 ]
7 # The Python API calls can raise `trexio.Error`
8 # exceptions to be handled via try/except clauses
9 # in the user application
10 with trexio.File("water.trexio", 'w',
11                back_end=trexio.TREXIO_HDF5) as f:
12     trexio.write_nucleus_num(f, len(coord))
13     trexio.write_nucleus_coord(f, coord)

```

FIG. 2. Python code writing the nuclear coordinates of a water molecule in a TREXIO file.

coefficients. To allow for flexibility, the library also allows for the use of an *unsafe* mode, in which data can be overwritten. However, this mode carries the risk of producing inconsistent files, and the `metadata` group's `unsafe` attribute is set to 1 to indicate that the file has potentially been modified in a dangerous way. This attribute can be manually reset to 0 if the user is confident

that the modifications made are safe.

B. The back-end

At present, TREXIO supports two back-ends: one relying only on the C standard library to produce plain text files (the so-called text back-end), and one relying on the HDF5 library.

With the text back-end, the TREXIO “file” is a directory containing multiple text files, one for each group. This back end is intended to be used in development environments, as it gives access to the user to standard tools such as `diff` and `grep`. In addition, text files are better adapted than binary files for version control systems such as Git, so this format can be also used for storing reference data for unit tests.

HDF5 is a binary file format and library for storing and managing large amounts of data in a hierarchical structure. It allows users to manipulate data in a way similar to how files and directories are manipulated within the file system. The HDF5 library provides optimal performance through its memory mapping mechanism and supports advanced features such as serial and parallel I/O, chunking, and compression filters. However, HDF5 files are in binary format, which requires additional tools such as `h5dump` to view them in a human-readable format. HDF5 is widely used in scientific and engineering applications, and is known for its high performance and ability to handle large data sets efficiently.

The TREXIO HDF5 back-end is the recommended choice for production environments, as it provides high I/O performance. Furthermore, all data is stored in a single file, making it especially suitable for parallel file systems like Lustre. These file systems are optimized for large, sequential I/O operations and are not well-suited for small, random I/O operations. When multiple small files are used, the file system may become overwhelmed with metadata operations like creating, deleting, or modifying files, which can adversely affect performance.

In a benchmarking program designed to compare the two back-ends of the library, the HDF5 back-end was found to be significantly faster than the text back-end. The program wrote a wave function made up of 100 million Slater determinants and measured the time taken to write the Slater determinants and CI coefficients. The HDF5 back-end achieved a speed of 10.4×10^6 Slater determinants per second and a data transfer rate of 406 MB/s, while the text back-end had a speed of 1.1×10^6 determinants per second and a transfer rate of 69 MB/s. These results were obtained on a DELL 960 GB mix-use solid-state drive (SSD). The HDF5 back-end was able to achieve a performance level close to the peak performance of the SSD, while the text back-end’s performance was limited by the speed of the CPU for performing binary to ASCII conversions.

In addition to the HDF5 and text back-ends, it is also possible to introduce new back-ends to the library. For

example, a back-end could be created to support object storage systems, such as those used in cloud-based applications³¹ or for archiving in open data repositories. To use a new back-end, only a minor modification is required in the code using TREXIO: the correct back-end argument needs to be passed to the `trexio_open` function (see Figures 1 and 2).

C. Supported languages

One of the main benefits of using C as the interface for a library is that it is easy to use from other programming languages. Many programming languages, such as Python or Julia, provide built-in support for calling C functions, which means that it is relatively straightforward to write a wrapper that allows a library written in C to be called from another language. In general, libraries with a C interface are the easiest to use from other programming languages, because C is widely supported and has a simple, stable application binary interface (ABI). Other languages, such as Fortran and C++, may have more complex ABIs and may require more work to interface with them.

TREXIO has been employed in codes developed in various programming languages, including C, C++, Fortran, Python, OCaml, and Julia. While Julia is designed to enable the use of C functions without the need for additional manual interfacing, the TREXIO C header file was automatically integrated into Julia programs using the `CBindings.jl` package.³² In contrast, specific bindings have been provided for Fortran, Python, and OCaml to simplify the user experience.

In particular, the binding for Fortran is not distributed as multiple compiled Fortran module files (`.mod`), but instead as a single Fortran source file (`.F90`). The distribution of the source file instead of the compiled module has multiple benefits. It ensures that the TREXIO module is always compiled with the same compiler as the client code, avoiding the compatibility problem of `.mod` files between different compiler versions and vendors. The single-file model requires very few changes in the build system of the user’s codes, and it facilitates the search for the interface of a particular function. In addition, advanced text editors can parse the TREXIO interface to propose interactive auto-completion of the TREXIO function names to the developers.

Finally, the Python module, partly generated with SWIG³³ and fully compatible with NumPy,³⁴ allows Python users to interact with the library in a more intuitive and user-friendly way. Using the Python interface is likely the easiest way to begin using TREXIO and understanding its features. In order to help users get started with TREXIO and understand its functionality, tutorials in Jupyter notebooks are available on GitHub (<https://github.com/TREX-CoE/trexio-tutorials>), and can be executed via the Binder platform.

D. Source code generation and documentation

Source code generation is a valuable technique that can significantly improve the efficiency and consistency of software development. By using templates to generate code automatically, developers can avoid manual coding and reduce the risk of errors or inconsistencies. This approach is particularly useful when a large number of functions follow similar patterns, as in the case of the TREXIO library, where functions are named according to the pattern `trexio_[has|read|write]_<group>_<attribute>`. By generating these functions from the format specification using templates, the developers can ensure that the resulting code follows a consistent structure and is free from errors or inconsistencies.

The description of the format is written in a text file in the Org format.³⁵ Org is a structured plain text format, containing information expressed in a lightweight markup language similar to the popular Markdown language.³⁶ While Org was introduced as a mode of the GNU Emacs text editor, its basic functionalities have been implemented in most text editors such as Vim, Atom or VS Code.

There are multiple benefits in using the Org format. The first benefit is that the Org syntax is easy to learn and allows for the insertion of equations in L^AT_EX syntax. Additionally, Org files can be easily converted to Hyper-Text Markup Language (HTML) or Portable Document Format (PDF) for generating documentation. The second benefit is that GNU Emacs is a programmable text editor and code blocks in Org files can be executed interactively, similar to Jupyter notebooks. These code blocks can also manipulate data defined in tables and this feature is used to automatically transform tables describing groups and attributes in the documentation into a JavaScript Object Notation (JSON) file.^{37,38} This JSON file is then used by a Python script to generate the needed functions in C language, as well as header files and some files required for the Fortran, Python, and OCaml interfaces.

With this approach, contributions to the development of the TREXIO library can be made simply by adding new tables to the Org file, which can be submitted as *pull requests* on the project's GitHub repository (<https://github.com/trex-coe/trexio>). Overall, this process allows for a more efficient and consistent development process and enables contributions from a wider range of individuals, regardless of their programming skills.

E. Availability and reliability

The TREXIO library is designed to be portable and easy to install on a wide range of systems. It follows the C99 standard to ensure compatibility with older systems, and can be configured with either the GNU Autotools or the CMake build systems. The only external dependency

is the HDF5 library, which is widely available on HPC platforms and as packages on major Linux distributions. Note that it is possible to disable the HDF5 back-end at configuration time, allowing TREXIO to operate only with the text back-end and have zero external dependencies. This can be useful for users who may not be able to install HDF5 on certain systems.

TREXIO is distributed as a tarball containing the source code, generated code, documentation, and Fortran interface. It is also available as a binary `.deb` package for Debian-based Linux distributions and as packages for Guix³⁹, Spack⁴⁰ and Conda.⁴¹ The Python module can be found in the PyPI repository, the OCaml binding is available in the official OPAM repository, and the `.deb` packages are already available in Ubuntu 23.04.

To ensure the reliability and quality of the TREXIO library, we have adopted standard continuous integration and deployment practices. For example, we use unit tests that are executed automatically using GitHub actions whenever modifications are made to the codebase. These tests cover a wide range of functionalities and help to identify any potential issues or bugs in the code. Additionally, the TREXIO library is regularly used by the authors of the present paper, and as such, it is continuously tested and validated in the context of ongoing research activities.

TREXIO was built, tested and installed successfully on 20 different architectures supported by the Debian build farm. Furthermore, we ensure that the quality of our code meets the requirements of the CERT coding standards,⁴² and we use the `cppcheck`⁴³ tool to validate the quality of our code. These measures demonstrate our commitment to ensuring that the TREXIO library is a reliable and trustworthy tool.

F. Open-Source Governance and Sustainability Strategies

Our approach to the development and governance of the TREXIO library follows the standard design of open-source projects, which typically involve a collaborative effort from a community of contributors. The TREX European Center of Excellence initiated the project and proposed the first functional version of the software. However, we consider this to be just the starting point for a larger community effort.

As an open-source project, we encourage contributions from anyone interested in the development of the library. This includes not only contributions to the codebase but also contributions to the documentation, testing, and other aspects of the project. We believe that this collaborative approach is the key to the success of any open-source project.

Regarding governance, we have a small group of maintainers who oversee the development of the project, review and merge contributions, and ensure the quality of the code. However, we strive to make the development process as transparent and open as possible, and

we encourage contributions from anyone interested in the project.

Overall, our strategy for the governance and development of the TREXIO library follows the standard design of open-source projects, which emphasizes collaboration and transparency. We believe that this approach, combined with our commitment to seeking and securing funding for the continued development and maintenance of TREXIO, will ensure the long-term success and usefulness of the library to the quantum chemistry community.

V. EXAMPLES OF APPLICATIONS

The open-source Python package `trexio.tools`⁴⁴ has been created to enhance the use of the TREXIO library and corresponding data format. It includes converters for transforming output files from codes such as Gaussian, GAMESS,²⁵ or PySCF⁴⁵ into TREXIO files. However, in the future, it would be preferable if the developers of these codes were to offer the option to export data in TREXIO format in order to maintain numerical precision and ensure consistency in the stored data. In addition, the package includes utilities to convert certain data blocks from TREXIO files into FCIDUMP or Molden formats. It also has a feature that validates the consistency of a wave function by numerically calculating overlap integrals on a grid and comparing them to the overlap matrix stored in the file. This helps to confirm that all basis set parameters are consistent with the conventions of the original program.

TREXIO is currently used to exchange wave function parameters between the selected CI code Quantum Package⁴⁶ and the QMC code CHAMP.²⁸ The QMC codes QMC=Chem⁴⁷ and TurboRVB⁴⁸ are also able to read TREXIO files, allowing for comparison of the three QMC codes using the same wave function. TREXIO is also used to transfer integrals between Quantum Package and the FCIQMC code NECI,⁴⁹ and to read density matrices produced by Quantum Package in GammCor⁵⁰ for symmetry-adapted perturbation theory (SAPT)⁵¹ molecular interaction calculations with near-full CI density matrices.⁵² In addition, the recent development of a code for calculating electron repulsion integrals using Slater-type orbitals⁵³ now produces TREXIO files, enabling FCIQMC calculations using Slater-type orbitals with NECI and similar selected CI calculations with Quantum Package, which can then be used as trial wave functions for QMC calculations.

VI. CONCLUSION

The TREXIO format and library offer a convenient and flexible way to store and exchange quantum chemistry data. Its open-source nature allows for easy integration into various software applications and its compatibility with multiple programming languages makes it

accessible to a wide range of users. The use of the HDF5 library as the default back-end ensures efficient storage and retrieval of data, while the option to disable HDF5 and use the text back-end allows for zero external dependencies. The development of TREXIO has been driven by the need to facilitate collaboration and reproducibility in quantum chemistry research, and its adoption in various codes and projects is a testament to its usefulness in achieving these goals. We would like to emphasize that the TREXIO library is a work in progress, and we are committed to expanding its scope and functionality in future releases. Our immediate priorities include supporting periodic boundary conditions and other basis sets such as grids, and plane waves. Overall, the TREXIO format and library is a valuable resource for the quantum chemistry community and its continued development and adoption will surely benefit the field.

ACKNOWLEDGMENTS

The authors would like to thank Susi Lehtola for providing valuable feedback on an earlier version of this manuscript. This work was supported by the European Centre of Excellence in Exascale Computing TREX — Targeting Real Chemical Accuracy at the Exascale. Hence, the name of the software is *TREX Input/Output* (TREXIO). This project has received funding from the European Union’s Horizon 2020 — Research and Innovation program — under grant agreement no. 952165. A CC-BY 4.0 (<https://creativecommons.org/licenses/by/4.0/>) public copyright license has been applied by the authors to the present document and will be applied to all subsequent versions up to the Author Accepted Manuscript arising from this submission, in accordance with the grant’s open access conditions.

¹M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne, J. Bouwman, A. J. Brookes, T. Clark, M. Crosas, I. Dillo, O. Dumon, S. Edmunds, C. T. Evelo, R. Finkers, A. Gonzalez-Beltran, A. J. G. Gray, P. Groth, C. Goble, J. S. Grethe, J. Heringa, P. A. C. ’t Hoen, R. Hooft, T. Kuhn, R. Kok, J. Kok, S. J. Lusher, M. E. Martone, A. Mons, A. L. Packer, B. Persson, P. Rocca-Serra, M. Roos, R. van Schaik, S.-A. Sansone, E. Schultes, T. Sengstag, T. Slater, G. Strawn, M. A. Swertz, M. Thompson, J. van der Lei, E. van Mulligen, J. Velterop, A. Waagmeester, P. Wittenburg, K. Wolstencroft, J. Zhao, and B. Mons, *Sci. Data* **3**, 1 (2016).

²M. D. McIlroy, E. N. Pinson, and B. A. Tague, *Bell System Technical Journal* **57**, 1899 (1978).

³“JPEG image compression standard,” (2022), online; accessed 22. Dec. 2022.

⁴“Standards – MPEG,” (2022), online; accessed 22. Dec. 2022.

⁵“Targeting Real Chemical accuracy at the EXascale,” (2023), [Online; accessed 25. Jan. 2023].

⁶S. R. Hall, F. H. Allen, and I. D. Brown, *Acta Crystallogr., Sect. A: Found. Crystallogr.* **47**, 655 (1991).

⁷P. Murray-Rust and H. S. Rzepa, *J. Chem. Inf. Comput. Sci.* **39**, 928 (1999).

⁸P. Murray-Rust and H. S. Rzepa, *J. Cheminf.* **3**, 1 (2011).

⁹N. M. O’boyle, A. L. Tenderholt, and K. M. Langner, *J. Comput. Chem.* **29**, 839 (2008).

- ¹⁰T. Verstraelen, W. Adams, L. Pujal, A. Tehrani, B. D. Kelly, L. Macaya, F. Meng, M. Richer, R. Hernández-Esparza, X. D. Yang, M. Chan, T. D. Kim, M. Cools-Ceuppens, V. Chuiko, E. Vöhringer-Martinez, P. W. Ayers, and F. Heidar-Zadeh, *J. Comput. Chem.* **42**, 458 (2021).
- ¹¹D. G. A. Smith, A. T. Lolincio, Z. L. Glick, J. Lee, A. Ale-naizan, T. A. Barnes, C. H. Borca, R. Di Remigio, D. L. Dotson, S. Ehlert, A. G. Heide, M. F. Herbst, J. Hermann, C. B. Hicks, J. T. Horton, A. G. Hurtado, P. Kraus, H. Kruse, S. J. R. Lee, J. P. Misiewicz, L. N. Naden, F. Ramezanghorbani, M. Scheurer, J. B. Schriber, A. C. Simmonett, J. Steinmetzer, J. R. Wagner, L. Ward, M. Welborn, D. Altarawy, J. Anwar, J. D. Chodera, A. Dreuw, H. J. Kulik, F. Liu, T. J. Martínez, D. A. Matthews, H. F. Schaefer, J. Šponer, J. M. Turney, L.-P. Wang, N. De Silva, R. A. King, J. F. Stanton, M. S. Gordon, T. L. Windus, C. D. Sherrill, and L. A. Burns, *J. Chem. Phys.* **155**, 204801 (2021).
- ¹²M. J. T. Oliveira, N. Papior, Y. Pouillon, V. Blum, E. Artacho, D. Caliste, F. Corsetti, S. de Gironcoli, A. M. Elena, A. García, V. M. García-Suárez, L. Genovese, W. P. Huhn, G. Huhs, S. Kokott, E. Küçükbenli, A. H. Larsen, A. Lazzaro, I. V. Lebedeva, Y. Li, D. López-Durán, P. López-Tarifa, M. Lüders, M. A. L. Marques, J. Minar, S. Mohr, A. A. Mostofi, A. O’Cais, M. C. Payne, T. Ruh, D. G. A. Smith, J. M. Soler, D. A. Strubbe, N. Tancogne-Dejean, D. Tildesley, M. Torrent, and V. W.-z. Yu, *J. Chem. Phys.* **153**, 024117 (2020).
- ¹³M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, J. R. Cheeseman, G. Scalmani, V. Barone, G. A. Petersson, H. Nakatsuji, X. Li, M. Caricato, A. V. Marenich, J. Bloino, B. G. Janesko, R. Gomperts, B. Mennucci, H. P. Hratchian, J. V. Ortiz, A. F. Izmaylov, J. L. Sonnenberg, D. Williams-Young, F. Ding, F. Lipparini, F. Egidi, J. Goings, B. Peng, A. Petrone, T. Henderson, D. Ranasinghe, V. G. Zakrzewski, J. Gao, N. Rega, G. Zheng, W. Liang, M. Hada, M. Ehara, K. Toyota, R. Fukuda, J. Hasegawa, M. Ishida, T. Nakajima, Y. Honda, O. Kitao, H. Nakai, T. Vreven, K. Throssell, J. A. Montgomery, Jr., J. E. Peralta, F. Ogliaro, M. J. Bearpark, J. J. Heyd, E. N. Brothers, K. N. Kudin, V. N. Staroverov, T. A. Keith, R. Kobayashi, J. Normand, K. Raghavachari, A. P. Rendell, J. C. Burant, S. S. Iyengar, J. Tomasi, M. Cossi, J. M. Millam, M. Klene, C. Adamo, R. Cammi, J. W. Ochterski, R. L. Martin, K. Morokuma, O. Farkas, J. B. Foresman, and D. J. Fox, “Gaussian[®]16,” (2016), Gaussian Inc. Wallingford CT.
- ¹⁴T. Lu and Q. Chen, *ChemRxiv* (2022), 10.26434/chemrxiv-2021-lt04f-v6.
- ¹⁵T. A. Keith, “*wfx* Format Specification,” (2014), [Online; accessed 27. Dec. 2022].
- ¹⁶G. Schaftenaar and J. H. Noordik, *J. Comput.-Aided Mol. Des.* **14**, 123 (2000).
- ¹⁷P. J. Knowles and N. C. Handy, *Comput. Phys. Commun.* **54**, 75 (1989).
- ¹⁸S. Borini, A. Monari, E. Rossi, A. Tajti, C. Angeli, G. L. Bendazzoli, R. Cimiraglia, A. Emerson, S. Evangelisti, D. Maynau, J. Sanchez-Marin, and P. G. Szalay, *J. Chem. Inf. Model.* **47**, 1271 (2007).
- ¹⁹A. Scemama, A. Monari, C. Angeli, S. Borini, S. Evangelisti, and E. Rossi, in *International Conference on Computational Science and Its Applications (ICCSA 2008)*, Lecture Notes in Computer Science, Vol. 5072, edited by Gervasi, O. Murgante, and B (SPRINGER-VERLAG, Perugia, Italy, 2008) pp. 1094–1107.
- ²⁰E. Rossi, S. Evangelisti, A. Laganà, A. Monari, S. Rampino, M. Verdicchio, K. K. Baldrige, G. L. Bendazzoli, S. Borini, R. Cimiraglia, C. Angeli, P. Kallay, H. P. Lüthi, K. Ruud, J. Sanchez-Marin, A. Scemama, P. G. Szalay, and A. Tajti, *J. Comput. Chem.* **35**, 611 (2014).
- ²¹Q. Koziol, D. Robinson, and U. O. of Science, “Hdf5,” (2018).
- ²²R. Pauncz, *Int. J. Quantum Chem.* **35**, 717 (1989).
- ²³R. Bast, “Numgrid: Numerical integration grid for molecules,” (2021).
- ²⁴J. Burkardt, “SphereLebedev_rule: Quadrature rules for the unit sphere,” (2010).
- ²⁵G. M. J. Barca, C. Bertoni, L. Carrington, D. Datta, N. De Silva, J. E. Deustua, D. G. Fedorov, J. R. Gour, A. O. Gunina, E. Guidez, T. Harville, S. Irle, J. Ivanic, K. Kowalski, S. S. Leang, H. Li, W. Li, J. J. Lutz, I. Magoulas, J. Mato, V. Mironov, H. Nakata, B. Q. Pham, P. Piecuch, D. Poole, S. R. Pruitt, A. P. Rendell, L. B. Roskop, K. Ruedenberg, T. Sattasathuchana, M. W. Schmidt, J. Shen, L. Slipchenko, M. Sosonkina, V. Sundriyal, A. Tiwari, J. L. Galvez Vallejo, B. Westheimer, M. Wloch, P. Xu, F. Zahariev, and M. S. Gordon, *J. Chem. Phys.* **152**, 154102 (2020).
- ²⁶J. R. Trail and R. J. Needs, *J. Chem. Phys.* **146**, 204107 (2017).
- ²⁷R. Pauncz, *Int. J. Quantum Chem.* **25**, 707 (1984).
- ²⁸C. J. Umrigar and C. Filippi, “Cornell-Holland ab-initio materials package (CHAMP),” (2023).
- ²⁹A. D. Güçlü, G. S. Jeon, C. J. Umrigar, and J. K. Jain, *Phys. Rev. B* **72**, 205327 (2005).
- ³⁰E. Giner, *J. Chem. Phys.* **154**, 084119 (2021).
- ³¹J. Liu, Q. Koziol, G. F. Butler, N. Fortner, M. Chaarawi, H. Tang, S. Byna, G. K. Lockwood, R. Cheema, K. A. Kallback-Rose, D. Hazen, and M. Prabhat, in *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)* (IEEE, 2018) pp. 24–34.
- ³²K. Rutkowski, “Cbindings,” <https://github.com/analytech-solutions/CBinding.jl> (2023).
- ³³D. M. Beazley, in *TCLTK’96: Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4* (USENIX Association, 1996) p. 15.
- ³⁴C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, *Nature* **585**, 357 (2020).
- ³⁵E. Schulte, D. Davison, T. Dye, and C. Dominik, *J. Stat. Soft.* **46**, 1 (2012).
- ³⁶S. Leonard, “The text/markdown Media Type,” RFC 7763 (2016).
- ³⁷T. Bray, “The JavaScript Object Notation (JSON) Data Interchange Format,” RFC 8259 (2017).
- ³⁸F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, in *WWW ’16: Proceedings of the 25th International Conference on World Wide Web* (International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 2016) pp. 263–273.
- ³⁹L. Courtès, in *European Lisp Symposium* (Madrid, Spain, 2013).
- ⁴⁰T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, in *SC ’15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Association for Computing Machinery, New York, NY, USA, 2015) pp. 1–12.
- ⁴¹conda-forge community, “The conda-forge Project: Community-based Software Distribution Built on the conda Package Format and Ecosystem,” (2015).
- ⁴²R. C. Seacord, *The CERT® C Coding Standard, Second Edition: 98 Rules for Developing Safe, Reliable, and Secure Systems* (Addison-Wesley Professional, Boston, MA, USA, 2014).
- ⁴³“Cpcheck - A tool for static C/C++ code analysis,” (2023), [Online; accessed 23. Mar. 2023].
- ⁴⁴“Trexio tools,” (2022).
- ⁴⁵Q. Sun, X. Zhang, S. Banerjee, P. Bao, M. Barbry, N. S. Blunt, N. A. Bogdanov, G. H. Booth, J. Chen, Z.-H. Cui, J. J. Eriksen, Y. Gao, S. Guo, J. Hermann, M. R. Hermes, K. Koh, P. Koval, S. Lehtola, Z. Li, J. Liu, N. Mardirossian, J. D. McClain, M. Motta, B. Mussard, H. Q. Pham, A. Pulkin, W. Purwanto, P. J. Robinson, E. Ronca, E. R. Sayfutyarova, M. Scheurer, H. F. Schurkus, J. E. T. Smith, C. Sun, S.-N. Sun, S. Upadhyay, L. K. Wagner, X. Wang, A. White, J. D. Whitfield, M. J. Williamson, S. Wouters, J. Yang, J. M. Yu, T. Zhu, T. C. Berkel-

- bach, S. Sharma, A. Yu. Sokolov, and G. K.-L. Chan, *J. Chem. Phys.* **153**, 024109 (2020).
- ⁴⁶Y. Garniron, T. Applencourt, K. Gasperich, A. Benali, A. Ferté, J. Paquier, B. Pradines, R. Assaraf, P. Reinhardt, J. Toulouse, P. Barbaresco, N. Renon, G. David, J.-P. Malrieu, M. Vêril, M. Caffarel, P.-F. Loos, E. Giner, and A. Scemama, *J. Chem. Theory Comput.* **15**, 3591 (2019).
- ⁴⁷A. Scemama, M. Caffarel, E. Oseret, and W. Jalby, *J. Comput. Chem.* **34**, 938 (2013).
- ⁴⁸K. Nakano, C. Attaccalite, M. Barborini, L. Capriotti, M. Casula, E. Coccia, M. Dagrada, C. Genovese, Y. Luo, G. Mazzola, A. Zen, and S. Sorella, *J. Chem. Phys.* **152**, 204121 (2020).
- ⁴⁹K. Guthrie, R. J. Anderson, N. S. Blunt, N. A. Bogdanov, D. Cleland, N. Dattani, W. Dobrutz, K. Ghanem, P. Jeszenszki, N. Liebermann, G. L. Manni, A. Y. Lozovoi, H. Luo, D. Ma, F. Merz, C. Overy, M. Rampp, P. K. Samanta, L. R. Schwarz, J. J. Shepherd, S. D. Smart, E. Vitale, O. Weser, G. H. Booth, and A. Alavi, *J. Chem. Phys.* **153**, 034107 (2020).
- ⁵⁰K. Pernal, M. Hapka, M. Przybytek, M. Modrzejewski, and A. Sokół, “Gammcor code,” .
- ⁵¹B. Jeziorski, R. Moszyński, and K. Szalewicz, *Chem. Rev.* **94**, 1887 (1994).
- ⁵²M. Hapka, M. Przybytek, and K. Pernal, *J. Chem. Theory Comput.* **17**, 5538 (2021).
- ⁵³M. Caffarel, *J. Chem. Phys.* **151**, 064101 (2019).

Appendix A: Table of stored data

TABLE I: List of all the data that can be stored in TREXIO files. The name of the group is written in the first line of each block. Multi-dimensional arrays are expressed in column-major order, meaning that elements of the same column are stored contiguously.

Attribute	Type	Dimensions	Description
metadata			
code_num	dim		Number of codes used to produce the file
code	str	(metadata.code_num)	Names of the codes used
author_num	dim		Number of authors of the file
author	str	(metadata.author_num)	Names of the authors of the file
package_version	str		TREXIO version used to produce the file
description	str		Text describing the content of file
unsafe	int		1: true, 0: false
nucleus			
num	dim		Number of nuclei
charge	float	(nucleus.num)	Charges of the nuclei
coord	float	(3,nucleus.num)	Coordinates of the atoms
label	str	(nucleus.num)	Atom labels
point_group	str		Symmetry point group
repulsion	float		Nuclear repulsion energy
grid			
description	str		Details about the used quadratures can go here
rad_precision	float		Radial precision parameter
num	dim		Number of grid points
max_ang_num	int		Maximum number of angular grid points
min_ang_num	int		Minimum number of angular grid points
coord	float	(grid.num)	Discretized coordinate space
weight	float	(grid.num)	Grid weights according to a given partitioning (e.g. Becke)
ang_num	dim		Number of angular integration points
ang_coord	float	(grid.ang_num)	Discretized angular space
ang_weight	float	(grid.ang_num)	Angular grid weights
rad_num	dim		Number of radial integration points
rad_coord	float	(grid.rad_num)	Discretized radial space
rad_weight	float	(grid.rad_num)	Radial grid weights
electron			
num	dim		Number of electrons
up_num	int		Number of \uparrow -spin electrons
dn_num	int		Number of \downarrow -spin electrons
state			
num	dim		Number of states (including the ground state)
id	int		Index of the current state (0 is ground state)
current_label	str		Label of the current state
label	str	(state.num)	Labels of all states
file_name	str	(state.num)	Names of the TREXIO files linked to the current one
basis			
type	str		Type of basis set: “Gaussian” or “Slater”
prim_num	dim		Total number of primitives
shell_num	dim		Total number of shells
nucleus_index	index	(basis.shell_num)	One-to-one correspondence between shells and atomic indices
shell_ang_mom	int	(basis.shell_num)	One-to-one correspondence between shells and angular momenta
shell_factor	float	(basis.shell_num)	Normalization factor of each shell (N_s)
r_power	int	(basis.shell_num)	Power to which r is raised (n_s)
shell_index	index	(basis.prim_num)	One-to-one correspondence between primitives and shell index
exponent	float	(basis.prim_num)	Exponents of the primitives (γ_{ks})
coefficient	float	(basis.prim_num)	Coefficients of the primitives (a_{ks})
prim_factor	float	(basis.prim_num)	Normalization coefficients for the primitives (f_{ks})
e_cut	float		Energy cut-off for plane-wave calculations
ecp			
max_ang_mom_plus_1	int	(nucleus.num)	$\ell_{\max} + 1$, in the removed core orbitals
z_core	int	(nucleus.num)	Number of core electrons to remove per atom
num	dim		Total number of ECP functions for all atoms and all values of ℓ
ang_mom	int	(ecp.num)	One-to-one correspondence between ECP items and ℓ
nucleus_index	index	(ecp.num)	One-to-one correspondence between ECP items and the atom index
exponent	float	(ecp.num)	$\alpha_{Aq\ell}$ all ECP exponents
coefficient	float	(ecp.num)	$\beta_{Aq\ell}$ all ECP coefficients

power	int	(ecp.num)	$n_{Aq\ell}$ all ECP powers
ao			
cartesian	int		1: true, 0: false
num	dim		Total number of atomic orbitals
shell	index	(ao.num)	Basis set shell for each AO
normalization	float	(ao.num)	Normalization factors N'
ao.1e.int			
overlap	float	(ao.num,ao.num)	$\langle p q \rangle$
kinetic	float	(ao.num,ao.num)	$\langle p \hat{T}_e q \rangle$
potential_n_e	float	(ao.num,ao.num)	$\langle p \hat{V}_{ne} q \rangle$
ecp	float	(ao.num,ao.num)	$\langle p \hat{V}_{ecp} q \rangle$
core_hamiltonian	float	(ao.num,ao.num)	$\langle p \hat{h} q \rangle$
overlap_im	float	(ao.num,ao.num)	$\langle p q \rangle$ (imaginary part)
kinetic_im	float	(ao.num,ao.num)	$\langle p \hat{T}_e q \rangle$ (imaginary part)
potential_n_e_im	float	(ao.num,ao.num)	$\langle p \hat{V}_{ne} q \rangle$ (imaginary part)
ecp_im	float	(ao.num,ao.num)	$\langle p \hat{V}_{ECP} q \rangle$ (imaginary part)
core_hamiltonian_im	float	(ao.num,ao.num)	$\langle p \hat{h} q \rangle$ (imaginary part)
ao.2e.int			
eri	float sparse	(ao.num,ao.num,ao.num,ao.num)	Electron repulsion integrals
eri_lr	float sparse	(ao.num,ao.num,ao.num,ao.num)	Long-range electron repulsion integrals
eri_cholesky_num	dim		Number of Cholesky vectors for ERI
eri_cholesky	float sparse	(ao.num,ao.num,ao.2e.int.eri_cholesky_num)	Cholesky decomposition of the ERI
eri_lr_cholesky_num	dim		Number of Cholesky vectors for long range ERI
eri_lr_cholesky	float sparse	(ao.num,ao.num,ao.2e.int.eri_lr_cholesky_num)	Cholesky decomposition of the long range ERI
mo			
type	str		String to identify the set of MOs (HF, Natural, Local, CASSCF, etc)
num	dim		Number of MOs
coefficient	float	(ao.num,mo.num)	MO coefficients
coefficient_im	float	(ao.num,mo.num)	MO coefficients (imaginary part)
class	str	(mo.num)	Choose among: Core, Inactive, Active, Virtual, Deleted
symmetry	str	(mo.num)	Symmetry in the point group
occupation	float	(mo.num)	Occupation number
energy	float	(mo.num)	For canonical MOs, corresponding eigenvalue
spin	int	(mo.num)	For UHF wave functions, 0 is α and 1 is β
mo.1e.int			
overlap	float	(mo.num,mo.num)	$\langle i j \rangle$
kinetic	float	(mo.num,mo.num)	$\langle i \hat{T}_e j \rangle$
potential_n_e	float	(mo.num,mo.num)	$\langle i \hat{V}_{ne} j \rangle$
ecp	float	(mo.num,mo.num)	$\langle i \hat{V}_{ECP} j \rangle$
core_hamiltonian	float	(mo.num,mo.num)	$\langle i \hat{h} j \rangle$
overlap_im	float	(mo.num,mo.num)	$\langle i j \rangle$ (imaginary part)
kinetic_im	float	(mo.num,mo.num)	$\langle i \hat{T}_e j \rangle$ (imaginary part)
potential_n_e_im	float	(mo.num,mo.num)	$\langle i \hat{V}_{ne} j \rangle$ (imaginary part)
ecp_im	float	(mo.num,mo.num)	$\langle i \hat{V}_{ECP} j \rangle$ (imaginary part)
core_hamiltonian_im	float	(mo.num,mo.num)	$\langle i \hat{h} j \rangle$ (imaginary part)
mo.2e.ints			
eri	float sparse	(mo.num,mo.num,mo.num,mo.num)	Electron repulsion integrals
eri_lr	float sparse	(mo.num,mo.num,mo.num,mo.num)	Long-range electron repulsion integrals
eri_cholesky_num	dim		Number of Cholesky vectors for ERI
eri_cholesky	float sparse	(mo.num,mo.num,mo.2e.int.eri_cholesky_num)	Cholesky decomposition of the ERI
eri_lr_cholesky_num	dim		Number of Cholesky vectors for long range ERI
eri_lr_cholesky	float sparse	(mo.num,mo.num,mo.2e.int.eri_lr_cholesky_num)	Cholesky decomposition of the long range ERI
determinant			
num	dim readonly		Number of determinants
list	int special	(determinant.num)	List of determinants as integer bit fields
coefficient	float buffered	(determinant.num)	Coefficients of the determinants from the CI expansion
csf			
num	dim readonly		Number of CSFs
coefficient	float buffered	(csf.num)	Coefficients C_I of the CSF expansion
det_coefficient	float sparse	(determinant.num,csf.num)	Projection on the determinant basis
amplitude			
single	float sparse	(mo.num,mo.num)	Single excitation amplitudes
single_exp	float sparse	(mo.num,mo.num)	Exponentialized single excitation amplitudes
double	float sparse	(mo.num,mo.num,mo.num,mo.num)	Double excitation amplitudes
double_exp	float sparse	(mo.num,mo.num,mo.num,mo.num)	Exponentialized double excitation amplitudes
triple	float sparse	(mo.num,mo.num,mo.num,mo.num,mo.num,mo.num)	Triple excitation amplitudes
triple_exp	float sparse	(mo.num,mo.num,mo.num,mo.num,mo.num,mo.num)	Exponentialized triple excitation amplitudes
quadruple	float sparse	(mo.num,mo.num,mo.num,mo.num,mo.num,mo.num,mo.num,mo.num)	Quadruple excitation amplitudes
quadruple_exp	float sparse	(mo.num,mo.num,mo.num,mo.num,mo.num,mo.num,mo.num,mo.num)	Exponentialized quadruple excitation amplitudes
rdm			
1e	float	(mo.num,mo.num)	One body density matrix
1e_up	float	(mo.num,mo.num)	\uparrow -spin component of the one body density matrix
1e_dn	float	(mo.num,mo.num)	\downarrow -spin component of the one body density matrix
2e	float sparse	(mo.num,mo.num,mo.num,mo.num)	Two-body reduced density matrix (spin trace)
2e_upup	float sparse	(mo.num,mo.num,mo.num,mo.num)	$\uparrow\uparrow$ component of the two-body reduced density matrix
2e_dndn	float sparse	(mo.num,mo.num,mo.num,mo.num)	$\downarrow\downarrow$ component of the two-body reduced density matrix
2e_updn	float sparse	(mo.num,mo.num,mo.num,mo.num)	$\uparrow\downarrow$ component of the two-body reduced density matrix
2e_cholesky_num	dim		Number of Cholesky vectors
2e_cholesky	float sparse	(mo.num,mo.num,rdm.2e_cholesky_num)	Cholesky decomposition of the two-body RDM (spin trace)

2e_upup_cholesky_num	dim		Number of Cholesky vectors $\uparrow\uparrow$
2e_upup_cholesky	float	sparse (mo.num,mo.num,rdm.2e_upup_cholesky_num)	Cholesky decomposition of the two-body RDM ($\uparrow\uparrow$)
2e_dndn_cholesky_num	dim		Number of Cholesky vectors $\downarrow\downarrow$
2e_dndn_cholesky	float	sparse (mo.num,mo.num,rdm.2e_dndn_cholesky_num)	Cholesky decomposition of the two-body RDM ($\downarrow\downarrow$)
2e_updn_cholesky_num	dim		Number of Cholesky vectors $\uparrow\downarrow$
2e_updn_cholesky	float	sparse (mo.num,mo.num,rdm.2e_updn_cholesky_num)	Cholesky decomposition of the two-body RDM ($\uparrow\downarrow$)
<hr/>			
jastrow			
type	string		Type of Jastrow factor: CHAMP or Mu
ee_num	dim		Number of electron-electron parameters
en_num	dim		Number of electron-nucleus parameters, per nucleus
een_num	dim		Number of electron-electron-nucleus parameters, per nucleus
ee	float	(jastrow.ee_num)	Electron-electron parameters
en	float	(jastrow.en_num)	Electron-nucleus parameters
een	float	(jastrow.een_num)	Electron-electron-nucleus parameters
en_nucleus	index	(jastrow.en_num)	Nucleus relative to the eN parameter
een_nucleus	index	(jastrow.een_num)	Nucleus relative to the een parameter
ee_scaling	float		κ value in CHAMP Jastrow for electron-electron distances
en_scaling	float	(nucleus.num)	κ value in CHAMP Jastrow for electron-nucleus distances
<hr/>			
qmc			
num	dim		Number of 3N-dimensional points
point	float	(3,electron.num,qmc.num)	3N-dimensional points
psi	float	(qmc.num)	Wave function evaluated at the points
e_loc	float	(qmc.num)	Local energy evaluated at the points