

29-09-2023



D1.4 - Report on final release of open-source human-readable implementation of the libraries

Version 1.0

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 EX
Project Acronym	TREX
Grant Agreement No	952165
Instrument	Call: H2020-INFRAEDI-2019-1
Topic	INFRAEDI-05-2020 Centres of Excellence in
Start Date of Project	01-10-2020
Duration of Project	36 Months
Project Website	https://trex-coe.eu/
Deliverable Number	D1.4
Deliverable title	D1.4 – Report on final release of open-source human-readable implementation of the libraries, as seen in GA
Due Date	M36 – 30-09-2023 as seen in GA
Actual Submission Date	29-09-2023
Work Package	WP1 – Standard API for QMC kernels and
Lead Author (Org)	Anthony Scemama (Centre National de la (CNRS))
Contributing Author(s) (Org)	All beneficiaries
Reviewers (Org)	Mariella Ippolito (CINECA), William Jalby (U
Version	1.0
Dissemination level	Public
Nature	Report
Draft / final	Final
No. of pages including cover	33



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 and contribution history

Version	Date	Authors	Notes
1.0	29-09-2023	Anthony Scemama (CNRS)	First Official Release



Abbreviations

AO	Atomic Orbital
API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
CNRS	Centre National de la Recherche Scientifique
CoE	Center of Excellence
ECP	Effective Core Potentials
FFI	Foreign Function Interface
GPU	Graphical Processing Unit
HPC	High Performance Computing
HTML	HyperText Markup Language
LAPACK	Linear Algebra PACKage
MO	Molecular Orbital
PDF	Portable Document Format
QMCKI	the Quantum Monte Carlo kernel library
QMC	Quantum Monte Carlo
SCF	Self Consistent Field
TREX	Targeting REal chemical accuracy at the eXascale



Table of Contents

Document Information	ii
Disclaimer	iii
Versioning and contribution history	iv
Abbreviations	v
Table of Contents	vi
1 Executive summary	1
2 Introduction.....	1
3 Development Principles and Practices in QMCKl	3
3.1 Programming with Documentation in Mind.....	3
3.2 Cross-Language Compatibility and Interoperability.....	4
3.3 Packaging and External Dependencies	4
3.4 Utilization of GitHub Tools in the the Quantum Monte Carlo kernel library (QMCKl) Development Ecosystem	5
3.5 Licensing Strategy	6
4 Implementation details.....	6
4.1 Core Computational Kernels in QMCKl.....	6
4.2 Adaptive Algorithmic Strategies for Diverse Computational Scenarios	6
4.3 Upcoming Features in the Next Release of QMCKl	7
5 Using QMCKl	7
5.1 Building the library.....	7
5.2 Code examples	9
5.3 Tested architectures and software environments	10
6 Integration of the QMCKl Library in the TREX flagship codes	10
6.1 QMC=Chem	10
6.2 CHAMP	11
6.3 TurboRVB	12
6.4 Quantum Package	13
6.5 Extending the applicability of QMCKl beyond QMC frameworks	14
7 Conclusion	14
References	I



A	Examples	II
A.1	Complete example in Python	II
A.2	Complete example in C	IV
A.3	Complete example in Fortran	VIII



1 Executive summary

The QMCKl library represents a significant advancement in the field of quantum chemistry, particularly in Quantum Monte Carlo (QMC) simulations. Distributed under the 3-clause BSD license, the library aims for broad adoption across both academic and commercial platforms by offering a permissive open-source licensing model. The development methodology of QMCKl is rooted in literate programming principles, ensuring a seamless integration of documentation and code. The library's API is designed in C, ensuring cross-language compatibility and universal adaptability.

QMCKl offers specialized computational kernels optimized for diverse scenarios, including adaptive algorithm selection and specialized routines for matrix operations. The library is efficient in handling both small and large electron regimes and includes functions for evaluating atomic and molecular orbitals, and the Jastrow factor in the form used in CHAMP. An upcoming release aims to introduce higher-level kernels for direct local energy computation, further enhancing its computational efficiency.

The build system of QMCKl is flexible and user-friendly, supporting multiple configurations and language bindings, including options optimized for high-performance computing. It has undergone rigorous testing across various hardware architectures, operating systems, and compilers, ensuring its portability and applicability in diverse computational environments.

Furthermore, the library has been strategically integrated into the flagship codes QMC=Chem, CHAMP, TurboRVB, and Quantum Package. This integration enhances the computational capabilities of each software.

In summary, the QMCKl library serves as a comprehensive ecosystem for quantum chemistry computations, offering a blend of efficiency, versatility, and adaptability that is poised to drive future advancements in the field.

2 Introduction

The QMCKl library is dedicated to the implementation of the essential kernels inherent to Quantum Monte Carlo (QMC) techniques. Our methodology adheres to the principle of *separation of concerns*: in an ideal setting, researchers and method developers in scientific domains should not be burdened by the intricacies of system architecture. Conversely, IT specialists and research engineers in High Performance Computing (HPC) should direct their efforts towards the optimization of foundational mathematical libraries and system-level components. Work Package 1, or WP1, emphasizes the delineation of the Application Programming Interface (API), development of tests, and a pedagogical exposition of the primary algorithms of QMC methods. Within WP3, specialists in HPC utilize this repository as a guide for crafting optimized versions of the functions originally proposed in the pedagogical version of the library. Interested parties can access detailed documentation of the library at <https://trex-coe.github.io/qmckl>. Moreover, the source code can be found in the dedicated GitHub repository, available at <https://github.com/trex-coe/qmckl>.

In QMC simulations, the wave function Ψ is usually represented using a particular form involving



both Slater determinants D_I and a Jastrow correlation factor $\exp(J)$:

$$\Psi(\mathbf{r}_1, \dots, \mathbf{r}_{N_{\text{elec}}}) = \exp(J(\mathbf{r}_1, \dots, \mathbf{r}_{N_{\text{elec}}})) \sum_I c_I D_I(\mathbf{r}_1, \dots, \mathbf{r}_{N_{\text{elec}}}) \quad (1)$$

The Jastrow factor is a function with positive values, reflecting the correlations between electron-electron, electron-nucleus, and electron-electron-nucleus distances. The *Slater determinants*, fundamental to this expression, can be defined as the product of determinants corresponding to both \uparrow -spin and \downarrow -spin:

$$D_I(\mathbf{r}_1, \dots, \mathbf{r}_{N_{\text{elec}}}) = D_I^{\uparrow}(\mathbf{r}_1, \dots, \mathbf{r}_{N_{\text{elec}}^{\uparrow}}), D_I^{\downarrow}(\mathbf{r}_{N_{\text{elec}}^{\uparrow}+1}, \dots, \mathbf{r}_{N_{\text{elec}}}), \quad (2)$$

where the total number of electrons N_{elec} is represented by the summation of up-spin ($N_{\text{elec}}^{\uparrow}$) and down-spin ($N_{\text{elec}}^{\downarrow}$) electrons.

In the context of QMC simulations, determinants can be derived from the *Slater matrix*, composed of Molecular Orbitals (MOs) evaluated at the electronic positions. The specific determinant for \uparrow -spin can be described as:

$$D_I^{\uparrow}(\mathbf{r}_1, \dots, \mathbf{r}_{N_{\text{elec}}^{\uparrow}}) = \det(S_i^{\uparrow}) = \begin{vmatrix} \phi_{i(1)}(\mathbf{r}_1) & \dots & \phi_{i(N_{\text{elec}}^{\uparrow})}(\mathbf{r}_1) \\ \vdots & \ddots & \vdots \\ \phi_{i(1)}(\mathbf{r}_{N_{\text{elec}}^{\uparrow}}) & \dots & \phi_{i(N_{\text{elec}}^{\uparrow})}(\mathbf{r}_{N_{\text{elec}}^{\uparrow}}) \end{vmatrix} \quad (3)$$

Each Slater matrix is unique and distinguished by the indices $\{i(1), \dots, i(N_{\text{elec}}^{\uparrow})\}$ corresponding to the MOs involved.

The MOs, denoted as ϕ , are themselves expressed as linear combinations of atomic orbitals (Atomic Orbitals (AOs)), symbolized by χ :

$$\phi_j(\mathbf{r}) = \sum_k C_{kj} \chi_k(\mathbf{r}), \quad (4)$$

with each AO being a function centered on a nucleus, encompassing both radial and angular components.

The atomic orbitals (AOs) used in QMC simulations have both angular and radial components. The angular component is typically represented by spherical harmonics Y_{lm} or polynomial expressions. In contrast, the radial part is generally formulated as a linear combination of Gaussian functions:

$$\chi_k(\mathbf{r}) = (x - X_A)^a (y - Y_A)^b (z - Z_A)^c \sum_l d_{kl} \exp[-\gamma_{kl} |\mathbf{r} - \mathbf{R}_A|^2], \quad (5)$$

with $\mathbf{r} = (x, y, z)$ denoting the coordinates of an electron and $\mathbf{R}_A = (X_A, Y_A, Z_A)$ indicating the coordinates of the nucleus labeled A.

Within the QMC simulation, the evaluation of the wave function at the electron positions has to be done at every step of the Monte Carlo sampling. Along with this, the electron dynamics necessitate the computation of the wave function's gradient with respect to the electron coordinates. For the

calculation of the kinetic energy, the Laplacian of the wave function is required. The gradient and Laplacian of a Slater determinant can be determined through the adjugate of the Slater matrix:

$$\nabla_j D_i = \sum_k [\nabla S_i]_{jk} [\text{adj}(S_i)]_{kj}. \quad (6)$$

In the QMCKl library, these particular quantities involved in computing the wave function are considered pivotal kernels. Their implementation holds significant importance as they form the principal computational challenge in QMC simulations, thereby defining a critical aspect of the performance efficiency of the entire process.

3 Development Principles and Practices in QMCKl

The development of the QMCKl library is guided by a set of principles and practices that aim to ensure its robustness, adaptability, and ease of use. These guidelines are designed to address the diverse needs of both the quantum chemistry and high-performance computing communities. This section delves into the key aspects that shape the development landscape of QMCKl, ranging from documentation and language compatibility to dependency management, tooling, and licensing.

3.1 Programming with Documentation in Mind

The conventional approach to software development often involves writing the source code, scripts, and Makefiles, with comments interspersed to elucidate complex portions of the code. Typically, the documentation resides in a distinct directory and may not always be synchronized with the latest version of the software.

In contrast, our methodology adopts the principles of literate programming[1], which treats the software as a document of publishable quality. In this paradigm, the source files predominantly consist of textual explanations, mathematical equations, tables, and figures. The actual code lines serve as computer-readable translations of the concepts and algorithms detailed in the document.

The structure of this “document” mimics that of a conventional text, complete with sections, subsections, a table of contents, and a bibliography. Code segments are inserted where they logically fit within the narrative, rather than where a compiler would expect them. This dual-purpose source file serves both as a comprehensive document and as the basis for generating the executable binary.

This approach is especially beneficial for this project, which centers around the documentation of an API. The code implementation essentially serves as a compilable expression of the documented algorithms, facilitating their validation. This methodology ensures unwavering congruence between the documentation and the actual code.

For the development environment, we opted for org-mode[2, 3], a format compatible with any text editor. This choice allows for versatile documentation generation, with options to convert org-mode files into various formats like HyperText Markup Language (HTML) or Portable Document Format (PDF). The source code can be conveniently extracted using Emacs, invoked via command-line within the Makefile, and subsequently compiled. Additionally, Emacs enables interactive execution of code blocks, akin to Jupyter notebooks[4], but with the added benefit of multi-language support.

3.2 Cross-Language Compatibility and Interoperability

Within the Targeting REal chemical accuracy at the eXascale (TREX) Center of Excellence (CoE), Fortran remains the dominant programming language, supplemented by Bash and Python scripts. This trend extends beyond the CoE, with Fortran also being pivotal in projects like Casino and Amolqc. However, the community also frequently employs languages such as C and C++ (e.g., in QMCPack and QWalk), and Python is a popular choice for initial prototyping. Julia and Rust are emerging languages gaining traction[5, 6].

To ensure that our library is universally compatible, and also for the purpose of enhancing adaptability to unexplored architectures, a decision was made to furnish an API which aligns with the standards of the C programming language. This choice is motivated by the fact that all these languages offer a Foreign Function Interface (FFI) that allows for the invocation of C functions, particularly for system-level interactions.

The high-performance variants of QMCKl are crafted by HPC specialists in WP3. These versions are fine-tuned for specific hardware architectures, including x86 and ARM processors as well as Graphical Processing Unit (GPU) accelerators. Given the availability of efficient tools for leveraging processor-specific features[7] and GPU capabilities (e.g., SYCL[8], OneAPI, HIP, Cuda), it is anticipated that some of these optimized libraries will be developed in C++, aligning with our C-compatible API design.

For the pedagogical implementation of QMCKl, Fortran is selected as the language to encapsulate the core QMC algorithms. This choice is motivated by Fortran's widespread acceptance within the scientific community and its inherent readability, which caters to both QMC and HPC specialists. While the algorithms are natively written in Fortran, they are made accessible via the library's API as though they were C-based implementations. This is achieved through Fortran's `iso_c_binding` FFI, thereby ensuring a seamless interface that enhances the library's cross-language compatibility.

Upon successful compilation, the QMCKl library offers two types of output: a static library denoted as `libqmckl.a` and a shared library represented by `libqmckl.so`. To streamline the development process, all function prototypes are aggregated into a singular C header file, `qmckl.h`. This centralization simplifies the task of function invocation and ensures consistency across different parts of the project. Furthermore, for Fortran users, the library includes a specialized module, `qmckl_f.F90`, which furnishes interfaces to the C functions.

In addition to the native support for languages like Fortran, C, and C++, the QMCKl library also offers Python bindings to facilitate rapid prototyping. Recognizing Python's widespread adoption for exploratory coding and data analysis, these bindings serve as a convenient interface for developers and researchers. They allow for seamless integration with Python-based workflows, thereby expanding the library's applicability and ease of use. This feature aims to bridge the gap between high-level scripting and low-level computational routines, offering a flexible and efficient development environment.

3.3 Packaging and External Dependencies

Managing dependencies on external software components is a pivotal aspect that warrants meticulous attention in the development of the QMCKl library. The ease with which users can install and utilize the library is directly contingent upon the complexity of its dependency chain. Therefore, it is imperative to minimize the list of external dependencies to only those that are absolutely essential.

Such external libraries should be incorporated *solely* when their functionality is indispensable and when there is a high degree of confidence that users can readily access them.

The choice of a build system is a critical decision that impacts both developers and end-users. For the QMCKl library, we have opted for the GNU build system, Autotools[9], to handle configuration and installation scripts. While there are more contemporary build tools available, our selection was guided by two primary considerations: ease of use for the end-user and robust portability across diverse computing architectures. Autotools has a long-standing reputation for reliability, making it a prudent choice for a library.

Looking ahead, we aim to further simplify the installation process by offering pre-compiled packages for popular Linux distributions. Specifically, we plan to release .deb packages for Debian-based systems and .rpm packages for Red Hat-based distributions. Additionally, we intend to provide a package for the Spack package manager[10], a versatile tool commonly used in high-performance computing environments. To facilitate this future expansion, we have preemptively structured our build environment to make the packaging process straightforward. For instance, the use of Autotools inherently simplifies the integration with Spack.

For those opting to use pre-compiled versions of the QMCKl library, the essential dependencies are Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK). Additionally, the TREXIO library, developed under WP2, serves as an optional dependency that users may choose to integrate.

When it comes to compiling the library from source, the prerequisites extend to include a C and a Fortran compiler, along with GNU Make. This ensures that the library remains compatible with a broad range of development environments.

3.4 Utilization of GitHub Tools in the QMCKl Development Ecosystem

The contemporary software development ecosystem for the QMCKl library leverages a suite of tools provided by GitHub to streamline various aspects of the development workflow. Firstly, GitHub Pull Requests serve as the primary conduit for developers to submit code modifications, facilitating peer review and collaborative refinement. This ensures that all contributions are scrutinized for quality and consistency before integration into the main codebase.

Secondly, the library employs GitHub Actions as its Continuous Integration and Continuous Delivery (CI/CD) platform. This tool automates the testing of each new code submission for both the pedagogical version of the library and the highly optimized one. This comprehensive testing matrix ensures that the library remains robust and compatible across different computational environments.

Thirdly, GitHub Issues functions as a centralized repository for bug reports and feature requests. This tool enables both users and developers to document software anomalies or propose enhancements, thereby fostering an open channel for community-driven development.

Lastly, the library's documentation is hosted on GitHub Pages, a static site hosting service. This platform provides a user-friendly interface where comprehensive software documentation is made readily accessible, thereby aiding in the effective utilization of the QMCKl library.

3.5 Licensing Strategy

The QMCKl library is distributed under the terms of the 3-clause BSD license, a permissive open-source license. This licensing model has been strategically chosen to encourage widespread adoption across various quantum chemistry software platforms, irrespective of their commercial or non-commercial nature. The 3-clause BSD license grants users the freedom to use, modify, and distribute the library, while imposing minimal restrictions. Specifically, it allows for the incorporation of the library into proprietary software, thereby making it an attractive choice for commercial entities. At the same time, the license ensures that the original authors are credited and that the library's open-source character is preserved. This balanced approach aims to foster a broad user base, ranging from academic research projects to commercial software solutions, thereby maximizing the library's impact and utility in the field of quantum chemistry.

4 Implementation details

4.1 Core Computational Kernels in QMCKl

The QMCKl library is engineered to provide optimized implementations of the most computationally intensive kernels commonly encountered in QMC simulations. These kernels serve as the building blocks for more complex quantum chemistry calculations and are crucial for the efficiency and accuracy of QMC codes. The following is a list of key kernels that have been implemented:

- AOs with Gradients and Laplacian,
- MOs with Gradients and Laplacian,
- adjustment of the electron-nucleus cusp on the MOs
- Jastrow Factor in the form as used in CHAMP,
- adjugate of a Matrix, utilizing algorithms like Sherman-Morrison-Woodbury for large matrices and specialized routines for small matrices.

In the context of evaluating the non-local component of the Effective Core Potentials (ECP), it is often necessary to compute the AOs or MOs at additional points in space. For these specific scenarios, the gradients and Laplacian are generally not required. To enhance the computational efficiency, the QMCKl library includes functions that return only the values of AOs or MOs at these extra points.

4.2 Adaptive Algorithmic Strategies for Diverse Computational Scenarios

The QMCKl library is designed to cater to a wide range of computational needs, from systems with a limited number of electrons but a multitude of Slater determinants, to those featuring a single determinant with a large electron count. This diversity in user requirements necessitates a library architecture that is efficient across both small and large electron regimes. In scenarios with a high

electron count, the interactions tend to be localized, allowing for the incorporation of linear-scaling algorithms. However, these algorithms are suboptimal when dealing with smaller systems. To reconcile these contrasting needs, the library employs a dynamic algorithm selection mechanism. It assesses the computational context at runtime and chooses the most time-efficient algorithm for the task at hand. For instance, QMCKl incorporates specialized routines for the inversion of compact matrices and also offers Sherman-Morrison variants for enhanced computational efficiency.

4.3 Upcoming Features in the Next Release of QMCKl

The development roadmap for the QMCKl library includes significant enhancements aimed at further simplifying the user experience and optimizing computational efficiency. One of the most anticipated features in the upcoming release is the introduction of higher-level kernels that directly compute the local energy in QMC simulations.

Within QMC simulations, the local energy is a critical quantity that encapsulates various interactions and contributions, including kinetic and potential energies. Currently, users have to manually combine lower-level kernels, such as AOs, MOs, and Jastrow factors, to compute the local energy. The next release of QMCKl will feature higher-level kernels specifically designed to directly compute the local energy.

The introduction of higher-level kernels for direct local energy computation in the forthcoming release of QMCKl is poised to offer multiple advantages. Firstly, these kernels are designed to be user-friendly, abstracting away the underlying complexities and thereby facilitating easier integration into existing QMC workflows. Secondly, they are optimized for computational efficiency, streamlining the sequence of operations and eliminating redundant calculations to enhance performance. Lastly, the automation of local energy computation through these kernels is expected to minimize the risk of algorithmic errors, thereby offering a more streamlined and efficient approach for users.

5 Using QMCKl

5.1 Building the library

The Autotools scripts for the QMCKl library are designed to facilitate out-of-source builds. This feature is particularly advantageous as it allows for the creation of multiple build directories, each with distinct configurations. For instance, one can configure builds with or without the inclusion of HPC-optimized functions, or using different compilers. This out-of-source build capability is also highly recommended when generating RPM or Debian packages, and as such, has been incorporated into the library from its inception.

In addition to the above, we offer a variety of build options to cater to different needs. These include flags for generating Python bindings and building documentation. Furthermore, we have introduced a flag that enables the use of the external QMCKl-dgemm library, developed under WP3, for efficient small-matrix multiplications.

Below is a sequence of shell commands that outlines the steps to compile the reference version of the QMCKl library:


```
# Navigate to the root directory of QMCKl and set an environment variable for
# convenience
export QMCKL_PATH=$PWD

# Create a build directory specifically for the pedagogical version
mkdir -p _build/simple _install/simple
cd _build/simple

# Configure and compile the library
$QMCKL_PATH/configure --prefix=$QMCKL_PATH/_install/simple
make -j 4

# Verify the library's functionality
make -j 4 check

# Install the library into the designated directory
make install
```

Depending on the architecture and/or compiler used, it is possible to pass more options to configure to optimize the compilation, for example to build an optimized version of the library using the Intel compiler and with Python support, the following commands can be used:

```
cd $QMCKL_PATH

# Create a build directory for the optimized version
mkdir -p _build/fast _install/fast
cd _build_fast

# Configure and compile the library with additional flags for optimization
# and Python support
$QMCKL_PATH/configure --prefix=$QMCKL_PATH/_install/fast --enable-hpc \
--with-icc --with-ifort --enable-python
make -j 4

# Verify the library's functionality
make -j 4 check

# Install the library into the designated directory
make install
```

To specify which version of the library to use, the paths to the lib and include directories in either `$QMCKL_PATH/_install/simple` or `$QMCKL_PATH/_install/fast` can be set as environment variables:

```
MODE=fast # or simple
```

```
export C_INCLUDE_PATH=$C_INCLUDE_PATH:$QMCKL_PATH/_install/$MODE/include
export LIBRARY_PATH=$LIBRARY_PATH:$QMCKL_PATH/_install/$MODE/lib
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$QMCKL_PATH/_install/$MODE/lib
```

Finally, a C code can be compiled by simply appending the `-lqmckl` flag to the link arguments.

To further streamline the user experience, we provide configuration files for `pkg-config`, a command-line utility that simplifies the process of fetching compiler and linker flags for installed libraries. This can be set up as follows:

```
export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:$QMCKL_PATH/_install/$MODE/pkgconfig
```

The users' makefiles can now take advantage of QMCKl as:

```
CFLAGS=$(shell pkg-config --cflags qmckl)
LIB=$(shell pkg-config --libs qmckl)
```

For the users using CMake for their code, we provide a `FindQMCKL` file to help CMake detect the presence of QMCKl on the system.

5.2 Code examples

In appendix A, we provide a hands-on example to demonstrate the capabilities of the QMCKl library. We furnish code samples in C, Fortran, and Python, serving as exhaustive tutorials for effectively leveraging QMCKl. The focal point of this example is the numerical evaluation of the overlap matrix in the MO basis. Utilizing QMCKl, we approximate the MOs at discrete points of a regular three-dimensional grid to compute the overlap matrix S_{ij} as follows:

$$S_{ij} = \int \phi_i(\mathbf{r}) \phi_j(\mathbf{r}) d\mathbf{r} \approx \sum_k \phi_i(\mathbf{r}_k) \phi_j(\mathbf{r}_k) \delta\mathbf{r} \quad (7)$$

The code starts by reading a wave function from a TREXIO file. This is accomplished using the `qmckl_trexio_read` function, which populates a `qmckl_context` with the necessary wave function parameters. The context serves as the primary interface for interacting with the QMCKl library, encapsulating the state and configurations of the system. Then, the code retrieves various attributes such as the number of nuclei and coordinates, for setting up the integration grid.

The core of the example lies in the numerical computation of the overlap matrix. To achieve this, the grid points are then populated into the `qmckl_context` using the `qmckl_set_point` function. The MO values at these grid points are computed using the `qmckl_get_mo_basis_mo_value` function. These values are then used to calculate the overlap matrix through a matrix multiplication operation facilitated by the `qmckl_dgemm` function.

The code is also instrumented to measure the execution time for the MO value computation, providing an empirical assessment of the computational efficiency. Error handling is robustly implemented at each stage to ensure the reliability of the simulation.

5.3 Tested architectures and software environments

Portability stands as a cornerstone in the design philosophy of the QMCKl library. Recognizing the diverse computational environments in which the library may be deployed, we have instituted a rigorous testing regime that spans multiple architectures, operating systems, and compilers. Specifically, the library undergoes regular testing across a variety of hardware architectures, including Intel, AMD and ARM architectures. In terms of operating systems, the library is compatible with a range of platforms, most notably Linux and MacOS. This broad OS support not only extends the library’s applicability but also makes it a viable choice for both server-based and desktop computing environments.

Furthermore, we have taken steps to ensure that QMCKl is compiler-agnostic. It has been successfully compiled and tested using a multitude of compilers, such as GNU, Intel, Nvidia, LLVM, ARM, and IBM. This compiler diversity guarantees that users have the flexibility to choose a compiler that best suits their specific needs, be it in terms of performance optimization or compatibility.

As of the current version, we are pleased to report that all possible combinations of architectures, operating systems, and compilers not only successfully compile but also pass all designated tests. This achievement underscores our commitment to delivering a robust and portable library that can seamlessly integrate into a myriad of computational setups.

6 Integration of the QMCKl Library in the TREX flagship codes

In this section, we explain how the QMCKl library has been seamlessly integrated into QMC=Chem, CHAMP, TurboRVB, and Quantum Package. The integration of QMCKl into these software packages is not merely a plug-and-play operation but a strategic enhancement that brings about significant computational advantages. Each code utilizes different components of the QMCKl library, ranging from the evaluation of AOs and MOs to the computation of the Jastrow factor (Fig. 1). This multifaceted integration underscores the versatility and efficacy of the QMCKl library in addressing diverse computational needs.

6.1 QMC=Chem

The development of a second version of QMC=Chem is currently in full swing, and it introduces substantial structural and functional alterations aimed at enhancing both efficiency and flexibility. In this revamped architecture, specific kernels from the QMCKl library that are integrated into this new version of QMC=Chem:

- Evaluation of AOs at the positions of electrons
- Evaluation of MOs at the positions of electrons
- Cusp-Adjustment of the MOs
- Evaluation of the Jastrow Factor in the CHAMP-defined form

While the original functions in QMC=Chem were designed to operate in single precision for the sake of speed, the kernels in QMCKl are optimized to efficiently utilize double precision, thereby enhancing the overall accuracy.

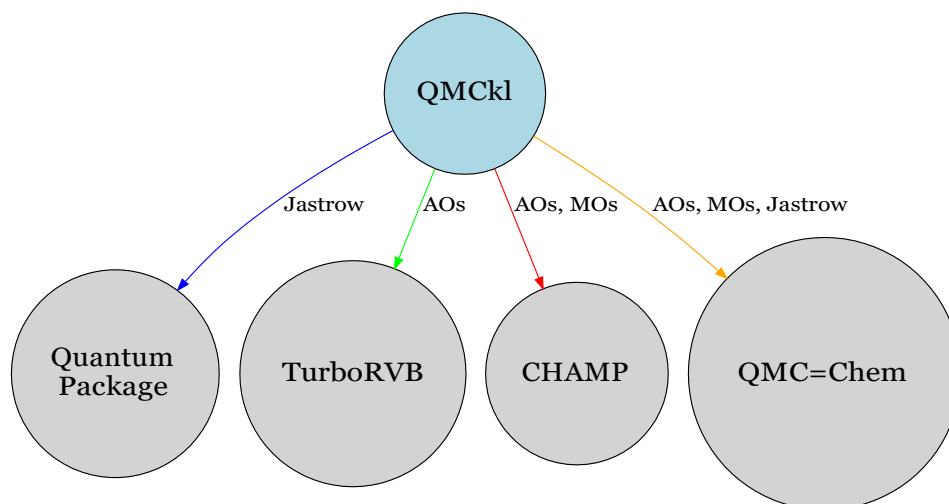


Figure 1: Summary of the usage of the QMCKl library across different software packages.

Furthermore, the adoption of the Jastrow factor evaluation, conforming to the format outlined in the CHAMP library, ensures full compatibility between the wave functions computed in both CHAMP and QMC=Chem. This feature is particularly advantageous as it allows for a seamless transition between the two computational codes, each optimized for specific tasks. For example, users can initiate a calculation in CHAMP to finetune the parameters of a complex Jastrow factor, a task where CHAMP excels. Then, they can effortlessly switch to QMC=Chem to execute large-scale multi-determinant calculations, a type of simulations for which QMC=Chem is particularly well optimized.

Another pivotal change is the exclusive reliance on the TREXIO file format for reading parameters that define the wave function. This strategic move serves multiple purposes: it streamlines the computational workflow, minimizes the risk of data inconsistency, and most importantly, enhances interoperability with other computational chemistry tools that also support the TREXIO format. Moreover, this approach ensures a high level of consistency between the wave function components computed in QMCKl and those computed in QMC=Chem. By relying on a unified file format, the system guarantees that the wave function parameters are identical across both computational environments, thereby eliminating discrepancies and enhancing the reliability of the results.

6.2 CHAMP

The QMCKl library has been seamlessly incorporated into the CHAMP software, serving as the computational engine for the evaluation of molecular orbitals, as well as their associated gradients and Laplacians. Beyond these core functionalities, the library also plays a crucial role in the computation of the non-local component of the effective core potentials. Specifically, QMCKl is employed for the evaluation of MOs at designated grid points.

This multi-faceted integration has led to remarkable gains in computational performance. As

evidenced in Fig. 3, the speed improvements are substantial, falling within the range of 40-45%. These metrics are based on the energy computation of a benzene molecule, utilizing a single determinant approach for the calculations.

It is worth noting that these performance gains are not the ceiling of what can be achieved. Anticipated future enhancements include the incorporation of the QMCKI library in the computation of the Jastrow factor, which is expected to further accelerate the computational process. Additionally, it should be highlighted that the performance metrics reported here are based solely on tests conducted using the pedagogical yet optimized version of the QMCKI library.

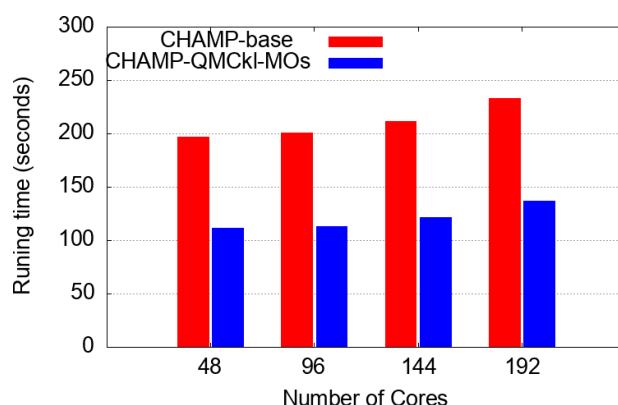


Figure 2: Graphical representation of the performance gains achieved in CHAMP through the native integration of the QMCKI library for molecular orbital computations. The data presented pertains to a QMC energy calculation for a benzene molecule executed on a JUWELS Skylake CPU.

6.3 TurboRVB

The TurboRVB software has recently undergone a significant upgrade through its integration with the QMCKI library, marking a pivotal step forward in its computational capabilities. This integration is particularly impactful in the evaluation of the determinant component.

While TurboRVB is equipped with its own internal wavefunction evaluator, the incorporation of QMCKI for the computation of AOs, along with their gradients and Laplacians, has yielded a noteworthy computational speed-up of approximately 20%. This acceleration is not limited to the AOs; the effective core potentials within TurboRVB also benefit from this integration, mirroring the enhancements observed in the CHAMP software.

Looking ahead, efforts are underway to extend the QMCKI integration to the computation of MOs, aiming to further accelerate the TurboRVB code. Given that TurboRVB is a GPU-accelerated software, plans are also in motion to integrate the GPU-optimized version of the QMCKI library, developed as part of WP3. This future integration is expected to unlock additional layers of computational efficiency, particularly in GPU-intensive tasks.

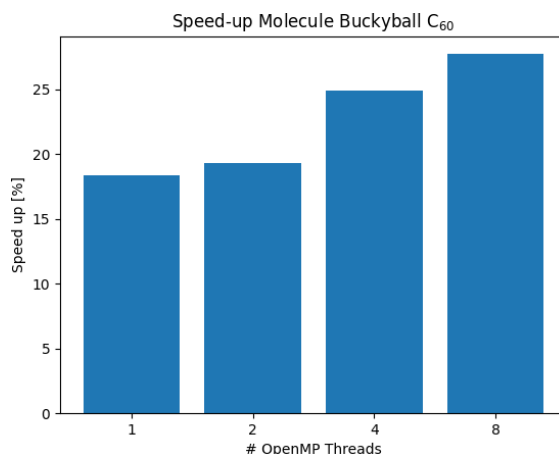


Figure 3: Graphical representation of the performance gains achieved in TurboRVB through the native integration of the QMCKI library for atomic orbital computations. The data presented pertains to a QMC energy calculation for a C₆₀ molecule. The speedup is calculated as $\frac{(\text{time without qmckl}) - (\text{time with qmckl})}{\text{time without qmckl}}$.

6.4 Quantum Package

Originally, Quantum Package was not expected to leverage QMCKI, but the library has proven to be an invaluable asset, particularly in the context of transcorrelated methods.

Transcorrelated methods represent a sophisticated class of quantum chemistry techniques to enhance the precision of electronic correlation calculations. These methods transcend the limitations of conventional wavefunction-based approaches by incorporating a Jastrow factor into the wavefunction. Essentially, transcorrelated methods reformulate the original Schrödinger equation through a similarity transformation that includes the Jastrow factor, yielding a new, *transcorrelated* Hamiltonian.

While the inclusion of the Jastrow factor substantially improves the accuracy of electronic structure calculations, it also introduces a layer of computational complexity. Specifically, the evaluation of integrals involving the Jastrow factor can become a bottleneck. For instance, the required integrals involve tensors with up to six indices, as opposed to the standard four indices in the Hamiltonian. Moreover, high-quality Jastrow factors necessitate numerical integration, as analytical solutions are generally unavailable.

The QMCKI library serves a dual purpose in this context. First, it facilitates the numerical computation of integrals that involve the Jastrow factor, in the sophisticated form used by CHAMP. Secondly, it allows for the optimization of Jastrow parameters within CHAMP, followed by the utilization of QMC=Chem to evaluate both the variational energy E_{var} and the transcorrelated energy E_{TC} within a single Variational Monte Carlo (VMC) sampling:

$$E_{\text{var}} = \frac{\langle \Psi e^J | \hat{H} | e^J \Psi \rangle}{\langle \Psi e^J | e^J \Psi \rangle} \quad (8)$$

$$E_{\text{TC}} = \frac{\langle \Psi | e^{-J} \hat{H} e^J | \Psi \rangle}{\langle \Psi | \Psi \rangle} \quad (9)$$

This dual evaluation serves as a cross-validation mechanism. The variational energy confirms that the Jastrow factor used in QMC=Chem via QMCKl aligns with the one optimized in CHAMP, while the transcorrelated energy verifies the accuracy of the numerical integration implemented in Quantum Package.

Once the Jastrow factor's consistency is confirmed, Quantum Package can proceed to further optimize the wavefunction. Initially, this involves the optimization of the MOs through a transcorrelated Self Consistent Field (SCF) procedure. The wavefunction can be further refined using the Transcorrelated Configuration Interaction using a Perturbative Selection made Iteratively (TC-CIPSI) algorithm.

Finally, the optimized wavefunction can be exported in the TREXIO format, facilitating its transfer to either QMC=Chem or CHAMP for more advanced Diffusion Monte Carlo (DMC) calculations.

6.5 Extending the applicability of QMCKl beyond QMC frameworks

After integrating the QMCKl library with Quantum Package, we noticed that the potential applicability of QMCKl extends far beyond the confines of QMC simulations. Specifically, we realized that the library's capabilities could be harnessed for a broader range of computational tasks, such as the efficient evaluation of AOs MOs on numerical grids.

This expanded utility is particularly relevant in the context of numerical integration techniques commonly employed in Density Functional Theory (DFT) and transcorrelated methods. For instance, the evaluation of AOs and MOs on a grid is a fundamental step in the numerical integration required for solving the Kohn-Sham equations in DFT.

Moreover, the QMCKl library proves to be invaluable for the three-dimensional visualization of various quantum mechanical properties. This includes not only the visualization of Molecular Orbitals (MOs) but also other functionals like the electron density and the Electron Localization Function (ELF).^[11]

By recognizing and leveraging the versatile capabilities of the QMCKl library, we have thus opened up new avenues for its application, extending its reach to encompass a wider array of computational chemistry methodologies.

7 Conclusion

In conclusion, the QMCKl library stands as a pivotal advancement in quantum chemistry computations, particularly in the context of QMC simulations. The library's licensing under the 3-clause BSD license not only ensures its accessibility but also promotes its widespread adoption across both academic and commercial platforms. The meticulous development principles, which include literate programming and cross-language compatibility, contribute to the library's robustness and adaptability.

The library is not just a collection of computational kernels; it is a comprehensive ecosystem designed for efficiency, versatility, and user-friendliness. Its upcoming release promises to further simplify computational workflows and reduce algorithmic errors by introducing higher-level kernels for direct local energy computation. The build system of QMCKl is designed with flexibility in mind, offering multiple configurations that cater to high-performance computing and providing bindings for various programming languages.

Moreover, the library's integration into the TREX flagship codes QMC=Chem, CHAMP, TurboRVB, and Quantum Package signifies its practical utility and adaptability. This integration is not superficial but deeply strategic, enhancing the computational capabilities of each software in unique ways.

Therefore, by centralizing the development effort into a single, versatile library, QMCKI paves the way for easy reusability and collaborative advancement, fortifying its role as an invaluable resource for the broader scientific community.



References

- [1] D. E. Knuth, *Literate Programming*. Cambridge, England, UK: Cambridge University Press, Mar 1992.
- [2] E. Schulte, D. Davison, T. Dye, and C. Dominik, "A Multi-Language Computing Environment for Literate Programming and Reproducible Research," *Journal of Statistical Software*, vol. 46, no. 1, pp. 1–24, Jan 2012. [Online]. Available: <https://doi.org/10.18637/jss.v046.i03>
- [3] "Org Mode," Jan 2021, [Online; accessed 10. Mar. 2021]. [Online]. Available: <https://orgmode.org>
- [4] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, "Jupyter notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds. IOS Press, 2016, pp. 87 – 90.
- [5] J. M. Perkel, "Why scientists are turning to Rust," *Nature*, vol. 588, pp. 185–186, Dec. 2020.
- [6] D. Poole, J. L. Galvez Vallejo, and M. S. Gordon, "A New Kid on the Block: Application of Julia to Hartree–Fock Calculations," *J. Chem. Theory Comput.*, vol. 16, no. 8, pp. 5006–5013, Aug 2020.
- [7] A. Pohl, B. Cosenza, M. A. Mesa, C. C. Chi, and B. Juurlink, "An evaluation of current simd programming models for c++," in *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, ser. WPMVP '16. New York, NY, USA: Association for Computing Machinery, 2016.
- [8] R. Keryell, R. Reyes, and L. Howes, *Khronos SYCL for OpenCL: a tutorial*. New York, NY, USA: Association for Computing Machinery, May 2015.
- [9] John Calcote, "Autotools, 2nd Edition," Mar 2021, [Online; accessed 11. Mar. 2021]. [Online]. Available: <https://www.penguinrandomhouse.ca/books/600402/autotools-2nd-edition-by-john-calcote/9781593279721>
- [10] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, *The Spack package manager: bringing order to HPC software chaos*. New York, NY, USA: Association for Computing Machinery, Nov 2015.
- [11] B. Silvi and A. Savin, "Classification of chemical bonds based on topological analysis of electron localization functions," *Nature*, vol. 371, no. 6499, pp. 683–686, Oct. 1994.

A Examples

A.1 Complete example in Python

```
import qmckl
import numpy as np
import time

def main(trexio_filename):
    rc = "QMCKL_SUCCESS"

    # Create a QMCKL context
    context = qmckl.context_create()

    # Read the TREXIO file into the context
    qmckl.trexio_read(context, trexio_filename)

    # Retrieve the number of nuclei
    nucl_num = qmckl.get_nucleus_num(context)

    # Retrieve the nuclear coordinates
    nucl_coord = qmckl.get_nucleus_coord(context, 'N', nucl_num*3)
    nucl_coord = np.reshape(nucl_coord, (3, nucl_num))

    # Retrieve the number of MOs
    mo_num = qmckl.get_mo_basis_mo_num(context)

    # Initialize grid points for the calculation
    nx = np.array([120, 120, 120])
    shift = np.array([5.0, 5.0, 5.0])
    point_num = np.prod(nx)

    rmin = np.array( list([ np.min(nucl_coord[:,a]) for a in range(3) ]) )
    rmax = np.array( list([ np.max(nucl_coord[:,a]) for a in range(3) ]) )

    linspace = [ None for i in range(3) ]
    step = [ None for i in range(3) ]
    for a in range(3):
        linspace[a], step[a] = np.linspace(rmin[a]-shift[a],
                                           rmax[a]+shift[a],
                                           num=nx[a],
                                           retstep=True)

    dr = step[0] * step[1] * step[2]

    point = []
    for x in linspace[0]:
        for y in linspace[1]:
            for z in linspace[2]:
```



```

        point += [ [x, y, z] ]

point = np.array(point)
point_num = len(point)
qmckl.set_point(context, 'N', point_num, np.reshape(point, (point_num*3)))

# Perform the actual calculation and measure the time taken
before = time.time()
mo_value = qmckl.get_mo_basis_mo_value(context, point_num*mo_num)
after = time.time()

mo_value = np.reshape( mo_value, (point_num, mo_num) ).T

print(f"Number of MOs: {mo_num}")
print(f"Number of grid points: {point_num}")
print(f"Execution time: {after - before} seconds")

# Compute the overlap matrix
overlap = mo_value @ mo_value.T * dr

# Print the overlap matrix
print(overlap)

qmckl.context_destroy(context)

if __name__ == "__main__":
    import sys
    trexio_filename = sys.argv[1]
    main(trexio_filename)

```

A.2 Complete example in C

```
#include <qmckl.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>

int main(int argc, char** argv) {
    // Define the TREXIO file to be read
    const char* trexio_filename = argv[1];
    qmckl_exit_code rc = QMCKL_SUCCESS;

    // Create a QMCKl context
    qmckl_context context = qmckl_context_create();

    // Read the TREXIO file into the context
    rc = qmckl_trexio_read(context, trexio_filename, strlen(trexio_filename));
    if (rc != QMCKL_SUCCESS) {
        fprintf(stderr, "Error reading TREXIO file:\n%s\n", qmckl_string_of_error(rc));
        exit(1);
    }

    // Retrieve the number of nuclei
    int64_t nucl_num;
    rc = qmckl_get_nucleus_num(context, &nucl_num);
    if (rc != QMCKL_SUCCESS) {
        fprintf(stderr, "Error getting nucl_num:\n%s\n", qmckl_string_of_error(rc));
        exit(1);
    }

    // Retrieve the nuclear coordinates
    double nucl_coord[nucl_num][3];
    rc = qmckl_get_nucleus_coord(context, 'N', &(nucl_coord[0][0]), nucl_num * 3);
    if (rc != QMCKL_SUCCESS) {
        fprintf(stderr, "Error getting nucl_coord:\n%s\n", qmckl_string_of_error(rc));
        exit(1);
    }

    // Retrieve the number of MOs
    int64_t mo_num;
    rc = qmckl_get_mo_basis_mo_num(context, &mo_num);
    if (rc != QMCKL_SUCCESS) {
        fprintf(stderr, "Error getting mo_num:\n%s\n", qmckl_string_of_error(rc));
        exit(1);
    }

    // Initialize grid points for the calculation
    size_t nx[3] = { 120, 120, 120 };
    double shift[3] = { 5., 5., 5. };
    int64_t point_num = nx[0] * nx[1] * nx[2];
}
```

```
double rmin[3] = { nucl_coord[0][0], nucl_coord[0][1], nucl_coord[0][2] } ;
double rmax[3] = { nucl_coord[0][0], nucl_coord[0][1], nucl_coord[0][2] } ;

for (size_t i=0 ; i<nucl_num ; ++i) {
    for (int j=0 ; j<3 ; ++j) {
        rmin[j] = nucl_coord[i][j] < rmin[j] ? nucl_coord[i][j] : rmin[j];
        rmax[j] = nucl_coord[i][j] > rmax[j] ? nucl_coord[i][j] : rmax[j];
    }
}

rmin[0] -= shift[0]; rmin[1] -= shift[1]; rmin[2] -= shift[2];
rmax[0] += shift[0]; rmax[1] += shift[1]; rmax[2] += shift[2];

double step[3];

double* linspace[3];
for (int i=0 ; i<3 ; ++i) {

    linspace[i] = (double*) calloc( sizeof(double), nx[i] );

    if (linspace[i] == NULL) {
        fprintf(stderr, "Allocation failed (linspace)\n");
        exit(1);
    }

    step[i] = (rmax[i] - rmin[i]) / ((double) (nx[i]-1));

    for (size_t j=0 ; j<nx[i] ; ++j) {
        linspace[i][j] = rmin[i] + j*step[i];
    }
}

double* point = (double*) calloc(sizeof(double), 3*point_num);

if (point == NULL) {
    fprintf(stderr, "Allocation failed (point)\n");
    exit(1);
}

size_t m = 0;
for (size_t i=0 ; i<nx[0] ; ++i) {
    for (size_t j=0 ; j<nx[1] ; ++j) {
        for (size_t k=0 ; k<nx[2] ; ++k) {

            point[m] = linspace[0][i];
            m++;

            point[m] = linspace[1][j];
            m++;
        }
    }
}
```

```

        point[m] = linspace[2][k];
        m++;

    }
}

rc = qmckl_set_point(context, 'N', point_num, point, (point_num*3));

if (rc != QMCKL_SUCCESS) {
    fprintf(stderr, "Error setting points:\n%s\n", qmckl_string_of_error(rc));
    exit(1);
}

// Perform the actual calculation and measure the time taken
long before, after;
struct timeval timecheck;
double* mo_value = (double*) calloc(sizeof(double), point_num * mo_num);
if (mo_value == NULL) {
    fprintf(stderr, "Allocation failed (mo_value)\n");
    exit(1);
}

gettimeofday(&timecheck, NULL);
before = (long)timecheck.tv_sec * 1000 + (long)timecheck.tv_usec / 1000;

rc = qmckl_get_mo_basis_mo_value(context, mo_value, point_num * mo_num);
if (rc != QMCKL_SUCCESS) {
    fprintf(stderr, "Error getting mo_value\n");
    exit(1);
}

gettimeofday(&timecheck, NULL);
after = (long)timecheck.tv_sec * 1000 + (long)timecheck.tv_usec / 1000;

printf("Number of MOs: %ld\n", mo_num);
printf("Number of grid points: %ld\n", point_num);
printf("Execution time : %f seconds\n", (after - before) * 1.e-3);

// Compute the overlap matrix
double dr = step[0] * step[1] * step[2];
double* overlap = (double*) malloc(mo_num * mo_num * sizeof(double));
rc = qmckl_dgemm(context, 'N', 'T', mo_num, mo_num, point_num, dr,
                mo_value, mo_num, mo_value, mo_num, 0.0,
                overlap, mo_num);

// Print the overlap matrix
for (size_t i = 0; i < mo_num; ++i) {
    printf("%4ld", i);
    for (size_t j = 0; j < mo_num; ++j) {

```

```
    printf(" %f", overlap[i * mo_num + j]);  
    }  
    printf("\n");  
}  
  
// Clean-up and exit  
free(overlap);  
  
rc = qmckl_context_destroy(context);  
if (rc != QMCKL_SUCCESS) {  
    fprintf(stderr, "Error destroying context\n");  
    exit(1);  
}  
  
return 0;  
}
```



A.3 Complete example in Fortran

```
#include <qmckl_f.F90>

program main
  use iso_c_binding
  use qmckl
  implicit none

  ! Declare variables
  integer :: argc
  character(128) :: trexio_filename, err_msg
  integer(qmckl_exit_code) :: rc
  integer(qmckl_context) :: context
  integer*8 :: nucl_num, mo_num, point_num
  double precision, allocatable :: nucl_coord(:, :)
  integer*8 :: nx(3)
  double precision, dimension(3) :: shift, step, rmin, rmax
  double precision, allocatable :: mo_value(:, :), overlap(:, :), point(:, :), linspace(:, :)
  double precision :: before, after, dr
  integer*8 :: i, j, k, m

  ! Initialize variables
  err_msg = ' '
  argc = command_argument_count()
  if (argc /= 1) then
    print *, "Usage: ./program <TREXIO filename>"
    stop -1
  end if
  call get_command_argument(1, trexio_filename)
  rc = QMCKL_SUCCESS

  ! Create a QMCKL context
  context = qmckl_context_create()

  ! Read the TREXIO file into the context
  rc = qmckl_trexio_read(context, trim(trexio_filename), len(trexio_filename)*1_8)
  if (rc /= QMCKL_SUCCESS) then
    call qmckl_string_of_error(rc, err_msg)
    write(*, *) "Error reading TREXIO file:", err_msg
    stop -1
  end if

  ! Retrieve the number of nuclei
  rc = qmckl_get_nucleus_num(context, nucl_num)
  if (rc /= QMCKL_SUCCESS) then
    call qmckl_string_of_error(rc, err_msg)
    write(*, *) "Error getting nucl_num:", err_msg
    stop -1
  end if
```

```

! Retrieve the nuclear coordinates
allocate(nucl_coord(3, nucl_num))
rc = qmckl_get_nucleus_coord(context, 'N', nucl_coord, nucl_num * 3_8)
if (rc /= QMCKL_SUCCESS) then
  call qmckl_string_of_error(rc, err_msg)
  write(*,*) "Error getting nucl_coord:", err_msg
  stop -1
end if

! Retrieve the number of MOs
rc = qmckl_get_mo_basis_mo_num(context, mo_num)
if (rc /= QMCKL_SUCCESS) then
  call qmckl_string_of_error(rc, err_msg)
  write(*,*) "Error getting mo_num:", err_msg
  stop -1
end if

! Initialize grid points for the calculation
nx = (/ 120, 120, 120 /)
shift = (/ 5.d0, 5.d0, 5.d0 /)
point_num = nx(1) * nx(2) * nx(3)

! Initialize rmin and rmax
rmin = nucl_coord(:,1)
rmax = nucl_coord(:,1)

! Update rmin and rmax based on nucl_coord
do i = 1, 3
  do j = 1, nucl_num
    rmin(i) = min(nucl_coord(i,j), rmin(i))
    rmax(i) = max(nucl_coord(i,j), rmax(i))
  end do
end do

! Apply shift
rmin = rmin - shift
rmax = rmax + shift

! Initialize linspace and step
allocate(linspace(3, maxval(nx)))

do i = 1, 3
  step(i) = (rmax(i) - rmin(i)) / real(nx(i) - 1, 8)
  do j = 1, nx(i)
    linspace(i, j) = rmin(i) + (j - 1) * step(i)
  end do
end do

! Initialize point array
allocate(point(3 * point_num))
m = 1
do i = 1, nx(1)

```

```

do j = 1, nx(2)
  do k = 1, nx(3)
    point(m) = linspace(1, i); m = m + 1
    point(m) = linspace(2, j); m = m + 1
    point(m) = linspace(3, k); m = m + 1
  end do
end do
end do

deallocate(linspace)

! Set points in QMCKL context
rc = qmckl_set_point(context, 'N', point_num, point, point_num * 3)
if (rc /= QMCKL_SUCCESS) then
  call qmckl_string_of_error(rc, err_msg)
  write(*,*) "Error setting point:", err_msg
  stop -1
end if

! Perform the actual calculation and measure the time taken
call cpu_time(before)
allocate(mo_value(point_num, mo_num))
rc = qmckl_get_mo_basis_mo_value(context, mo_value, point_num * mo_num)
if (rc /= QMCKL_SUCCESS) then
  call qmckl_string_of_error(rc, err_msg)
  write(*,*) "Error getting mo_value:", err_msg
  stop
end if
call cpu_time(after)

write(*,*) "Number of MOs:", mo_num
write(*,*) "Number of grid points:", point_num
write(*,*) "Execution time:", (after - before), "seconds"

! Compute the overlap matrix
dr = step(1) * step(2) * step(3)

allocate(overlap(mo_num, mo_num))
rc = qmckl_dgemm(context, 'N', 'T', mo_num, mo_num, point_num, dr, &
  mo_value, mo_num, mo_value, mo_num, 0.d0, overlap, mo_num)

! Print the overlap matrix
do i = 1, mo_num
  write(*, '(i4)', advance='no') i
  do j = 1, mo_num
    write(*, '(f8.4)', advance='no') overlap(i, j)
  end do
end do

```



```
    write(*,*)  
end do  
  
! Clean-up and exit  
deallocate(mo_value, overlap)  
rc = qmckl_context_destroy(context)  
if (rc /= QMCKL_SUCCESS) then  
    call qmckl_string_of_error(rc, err_msg)  
    write(*,*) "Error destroying context:", err_msg  
    stop -1  
end if  
  
end program main
```

