TRIQS: A toolbox for research on interacting quantum systems

Olivier Parcollet, Michel Ferrero, Thomas Ayral, Hartmut Hafermann, Igor Krivenko, Laura Messio, Priyanka Seth

TRIQS: A Toolbox for Research on Interacting Quantum Systems

Olivier Parcollet[a,*], Michel Ferrero[b], Thomas Ayral[a,b], Hartmut Hafermann[a], Igor Krivenko[c], Laura Messio[d,a], Priyanka Seth[b]

[a] *Institut de Physique Théorique (IPhT), CEA, CNRS, 91191 Gif-sur-Yvette, France*
[b] *Centre de Physique Théorique, Ecole Polytechnique, CNRS, 91128 Palaiseau Cedex, France*
[c] *I. Institut für Theoretische Physik, Uni. Hamburg, Jungiusstraße 9, 20355 Hamburg, Germany*
[d] *LPTMC, UMR 7600 CNRS, Université Pierre et Marie Curie, 75252 Paris, France*

Abstract

We present the TRIQS library, a Toolbox for Research on Interacting Quantum Systems. It is an open-source, computational physics library providing a framework for the quick development of applications in the field of many-body quantum physics, and in particular, strongly-correlated electronic systems. It supplies components to develop codes in a modern, concise and efficient way: e.g. Green's function containers, a generic Monte Carlo class, and simple interfaces to HDF5. TRIQS is a `C++`/`Python` library that can be used from either language. It is distributed under the GNU General Public License (GPLv3). State-of-the-art applications based on the library, such as modern quantum many-body solvers and interfaces between density-functional-theory codes and dynamical mean-field theory (DMFT) codes are distributed along with it.

PROGRAM SUMMARY

Program Title: TRIQS
Project homepage: http://ipht.cea.fr/triqs
Catalogue identifier: –
Journal Reference: –
Operating system: Unix, Linux, OSX
Programming language: `C++`/`Python`
Computers: any architecture with suitable compilers including PCs and clusters
RAM: Highly problem-dependent
Distribution format: GitHub, downloadable as zip
Licensing provisions: GNU General Public License (GPLv3)
Classification: 4.4, 4.6, 4.8, 4.12, 5, 6.2, 6.5, 7.3, 7.7, 20
PACS: 71.10.-w,71.27.+a,71.10.Fd,71.30.+h

*Corresponding author.
E-mail address: olivier.parcollet@cea.fr
Email addresses: `olivier.parcollet@cea.fr` (Olivier Parcollet), `michel.ferrero@polytechnique.edu` (Michel Ferrero), `thomas.ayral@polytechnique.edu` (Thomas Ayral), `hartmut.hafermann@cea.fr` (Hartmut Hafermann), `ikrivenk@physnet.uni-hamburg.de` (Igor Krivenko), `messio@lptmc.jussieu.fr` (Laura Messio), `priyanka.seth@polytechnique.edu` (Priyanka Seth)

Keywords: Many-body physics, Strongly-correlated systems, DMFT, Monte Carlo, ab initio calculations, C++, Python
External routines/libraries: `cmake`, `mpi`, `boost`, `FFTW`, `GMP`, `BLAS`, `LAPACK`, `HDF5`, `NumPy`, `SciPy`, `h5py`, `mpi4py`, `mako`.
Nature of problem:
Need for a modern programming framework to quickly write simple, efficient and higher-level code applicable to the studies of strongly-correlated electron systems.
Solution method:
We present a `C++/Python` open-source computational library that provides high-level abstractions for common objects and various tools in the field of quantum many-body physics, thus forming a framework for developing applications. *Running time:* Tests take less than a minute. Otherwise highly problem dependent (from minutes to several days).

1. Introduction

In this paper, we present the 1.2 release of the TRIQS project (Toolbox for Research in Interacting Quantum Systems), a free software library written in `Python` and `C++` for the implementation of algorithms in quantum many-body physics. TRIQS is distributed under the GNU General Public License (GPLv3).

Strongly-correlated quantum systems are a central challenge for theoretical condensed matter physics with a wide range of remarkable phenomena such as metal-insulator transitions, high-temperature superconductivity and magnetism. In the last two decades, tremendous progress has been made in the field of algorithms for the quantum many-body problem, both in refining existing techniques and in developing new systematic approximations and algorithms. Methods to address the quantum many-body problem include dynamical mean-field theory (DMFT) [1, 2] and its cluster [3] or diagrammatic extensions [4, 5] or the density matrix renormalization group (DMRG) [6]. DMFT methods can also now be combined with more traditional electronic structure methods such as density functional theory (DFT) leading to *ab initio* realistic computational techniques for strongly-correlated materials [2]. Several collaborative software development efforts have made some of these theoretical developments largely accessible, e.g. Refs. [7, 8, 9].

The purpose of the TRIQS project is to provide a modern framework of basic building blocks in `C++` and `Python`. This is needed for the rapid implementation of a broad spectrum of methods. Applications range from simple interactive phenomenological analysis in `Python` to high-performance quantum impurity solvers in `C++`. At this stage, TRIQS focuses primarily on, but is not limited to, solid-state physics computations, diagrammatic approximations and methods of the DMFT family (DMFT, clusters and underlying quantum impurity solvers).

A particular emphasis is placed on the documentation, in particular in providing short code examples that can be reused immediately (in `Python` and `C++`). Full documentation of the project is available online: `http://ipht.cea.fr/triqs`.

Several applications have already been built with the TRIQS library, and some of them are publicly distributed. Let us mention a state-of-the-art implementation of the hybridisation-expansion quantum impurity solver CTHYB (`http://ipht.cea.fr/triqs/applications/cthyb/`) and the DFT TOOLS project which provides an interface between DMFT and DFT packages such as WIEN2K for realistic computations for strongly-correlated materials (`http://ipht.cea.fr/triqs/applications/dft_tools/`). Since

these applications are not part of the library itself and involve a different set of authors, they will be presented in separate publications. However, they are distributed along with the TRIQS library under GPL license and are available for download on GitHub (`https://github.com/triqs`).

The TRIQS project uses professional code development methods to achieve the best possible quality for the library and the applications, including: *i) version control* using `git`; *ii) systematic code review* by the main TRIQS developers; *iii) test-driven development*: features of the library are first designed with a series of test cases. When the implementation is completed, they become the non-regression tests executed during the installation process.

This paper is organised as follows: we start in Sec. 2 with the main motivations for the project. In Sec. 3, we outline the structure of the TRIQS project. Sec. 4 summarizes our citation policy. In Sec. 5, we discuss the prerequisites to efficient usage of TRIQS and Sec. 6 describes the portability of the library. In Sec. 7, we provide two illustrating examples to give a flavour of the possibilities offered by the library: we show that TRIQS makes it possible to write a DMFT self-consistency loop in one page of `Python`, and, in another example, how equations can be coded efficiently in `C++`. In Sec. 8 we review the most important library components. In Appendix A we present the implementation of a fully working, MPI-parallelized, modern continuous-time quantum Monte Carlo algorithm (the so-called CT-INT algorithm [10, 11]) in about 200 lines. This example illustrates how TRIQS allows one to design a complex, yet short, readable and extensible code.

2. Motivations

The implementation of modern algorithms for quantum many-body systems raises several practical challenges.

Complexity: Theoretical methods and algorithms are becoming increasingly complex (e.g. quantum Monte Carlo [10, 12, 13, 11] and dual boson [14] methods). They are hence more difficult to implement, debug and maintain. This is especially true for realistic computations with methods of the DMFT family, where one has to handle not only the complexity of the many-body problem but also the various aspects (orbitals, lattices) of real materials, which usually requires a well-organised team effort.

Versatility/Agility: Algorithms are changing and improving rapidly, sometimes by orders of magnitude for some problems [15]. This can lead to a possibly quick obsolescence of a given implementation. To address new physics problems requires regular and substantial modifications of existing implementations. Moreover, there are numerous possibilities for new algorithms which need to be tested quickly.

Performance: Modern algorithms are still quite demanding on resources, e.g. quantum Monte Carlo methods. Hence, the performance of the codes is critical and a simple implementation in a high-level language is usually not sufficient in practice.

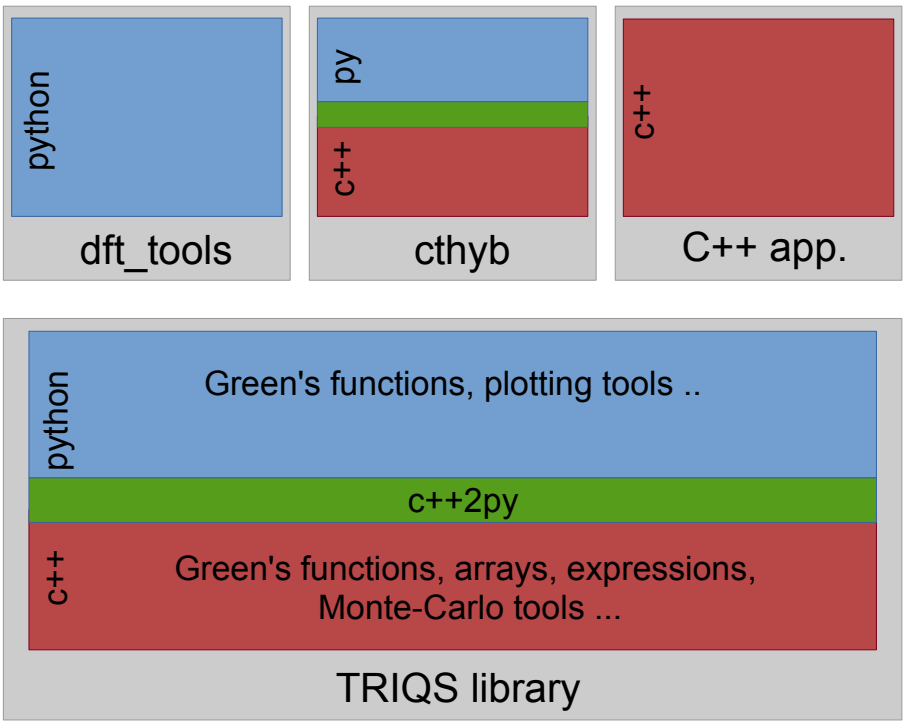
Reproducibility: The central role and the growing complexity of the algorithms in our field reinforces the need for reproducibility, which is central to any scientific endeavour. Therefore, the results obtained by a numerical computation should in principle be published systematically along with the code that produced them, in order to allow others to reproduce, falsify or improve on them. This requires codes to be readable (i.e. written to be read by other people than their author) and relatively quick to produce.

To address these challenges, one needs readable, clear and simple implementations with reusable components provided through high-level abstractions. We emphasize that this is *not* in contradiction with the requirement of high performance. The combination of a higher level of abstraction with high performance is achieved using modern programming techniques (e.g. generic programming). The purpose of the TRIQS project is to find and efficiently implement the relevant abstractions, basic components and algorithms for our domain.

3. Structure

The TRIQS framework is depicted in Fig. 1: the core library is at the root (bottom) consisting of basic building blocks, which are used in a series of applications (top). The applications can be in pure python (*e.g.* DFT TOOLS), in `C++` with a `Python` interface (*e.g.* CTHYB), or even in pure `C++`. The subject of this paper is the core library.

Figure 1: Structure of the TRIQS project



The components of the TRIQS library can be used both in `Python` and in `C++`: `C++` brings the performance needed for applications where speed is critical (like many-body solvers) and the type safety of a compiled language. On the other hand, `Python` is typically used as a higher-level interface for data analysis, investigation of phenomenological approaches, and tasks related to reproducibility. Most objects of the library are written

in C++ and exposed to Python using a specially designed tool described in Sec. 8.8. As a result, TRIQS can be used together with all the modern scientific tools of the Python community, in particular with IPython notebooks [16] which are recommended for an optimal interactive usage of the library in Python.

4. Citation policy

We kindly request that the present paper be cited in any published work using the TRIQS library directly (e.g. for data analysis) or indirectly (e.g. through TRIQS based applications). In the latter case, this citation should be added to the citations already requested by the application. This helps the TRIQS developers to better keep track of projects using the library and provides them guidance for future developments.

5. Programming Requirements

TRIQS can be used *at different levels of expertise*, starting from basic Python interactive usage to development of cutting-edge mixed Python/C++ high-performance and massively parallel codes, and in pure C++.

Most objects, in particular the Green's functions, have a rich Python interface, allowing one to easily plot and manipulate them. For example, simple operations such as value assignment, inversion or output to and input from HDF5 files are all one-line operations, as shown in the examples below.

At the C++ level, the required knowledge to make efficient use of the library is minimised. The systematic usage of *modern* C++ (C++11 and C++14) very often lead to simpler syntax than old C++. The library often favours a "functional style" programming and the simplest possible constructions for the C++ user. To fully exploit the capabilities of the library, some understanding of the basic notions of generic programming, such as concepts and templates, is helpful, but not required. More traditional object-oriented notions of C++ such as inheritance or dynamical polymorphism (virtual functions) are not necessary to use the TRIQS library.

6. Portability

TRIQS is written in *modern* C++, i.e. using the C++11 ISO standard. The motivation for this choice is twofold: first, we encourage the users of the library to benefit from the new features of C++, in particular those which produce much simpler code (e.g. `auto`, `for auto` loop or lambdas). Second, it dramatically reduces the cost of implementing and maintaining the library itself, since many of the new C++ features are designed to facilitate the use of the metaprogramming techniques needed to implement high-level, high-performance libraries.

As a result, TRIQS requires a C++11 *standard-compliant compiler*. The documentation provides an updated list of tested compilers. When it is available, we recommend using a C++14 compiler for development, in particular to get simpler error messages.

At the Python level, we use the 2.7 versions of Python. Support for Python 3 is planned for later releases. We use the binary hierarchical data format (HDF5) to guarantee portability of user generated data in binary form.

7. TRIQS in two examples

Here we illustrate the use of the library for two typical tasks encountered in many-body physics; this should give a flavour of the possibilities offered by the library. The first example is a complete DMFT computation implemented in `Python`, using a continuous-time quantum Monte Carlo solution of the impurity model (which is presented in Appendix A). The second example illustrates the manipulation of Green's functions in `C++` within TRIQS.

7.1. A DMFT computation in one page of IPython

This example requires the CT-INT tutorial application to be installed. The installation procedure is described in Sec. 9.2.

Fig. 2 shows a screenshot of an `IPython` notebook implementing a DMFT self-consistency loop. The essential steps are to load the solver module, set parameters and an initial guess for the Green's function and to loop over the DMFT iterations. The solver module, for which performance is critical, is written in `C++` but used from `Python`. This notebook is available in the sources of the `ctint_tutorial` application (in the `examples` subdirectory) along with the corresponding python script that is suitable for parallel execution.

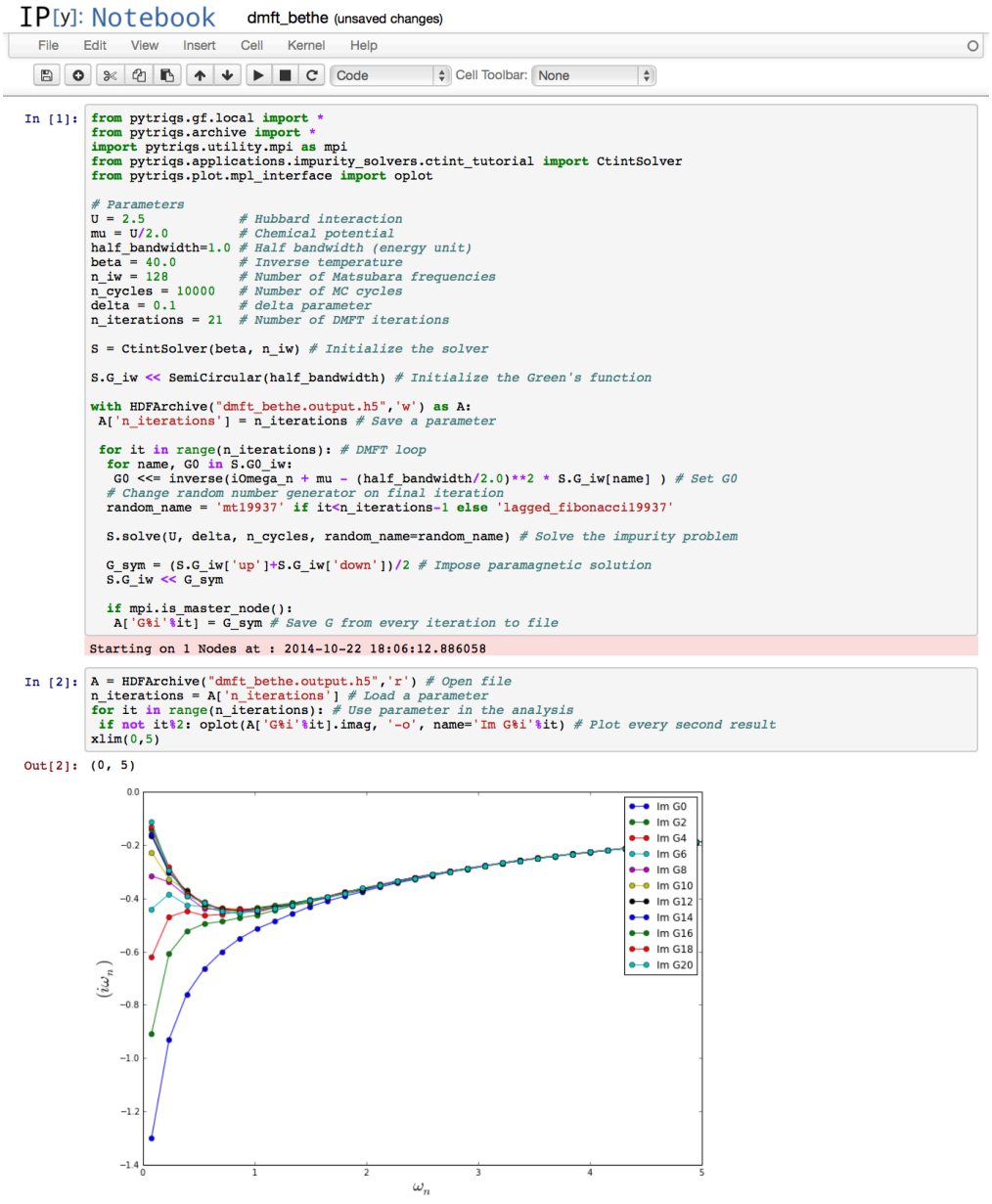
The `Python` framework is *highly flexible*. In this example, we exchange the random number generator in the final DMFT iteration to consolidate the result. We could also turn on additional measurements, or implement more sophisticated stopping criteria for the loop. Using the `IPython` notebook, results can be plotted and analysed interactively. As a minimal example, we load the HDF5 archive from the disk and plot the imaginary part of the Green's functions in the second cell of the notebook. Parameters used in the calculation can easily be saved to file and retrieved for data analysis. On a parallel machine, the first part of the script is executed in Python (without the notebook), on multiple cores, while the analysis can still be done in the IPython notebook.

Within this framework, DMFT can readily be explored and practised by non-experts. The major part of the calculation is of course performed by the solver module. In this example, we have used the interaction-expansion continuous-time quantum Monte Carlo (CT-INT) solver. We can easily switch to a more sophisticated hybridisation-expansion continuous-time quantum Monte Carlo algorithm (CT-HYB) solver by loading the appropriate solver module instead. The complete listing of the `C++` implementation of the solver module is given and explained in Appendix A, as a more detailed illustration of the library's features.

7.2. Easy manipulation of Green's functions in C++

Here, we show how to compute in `C++` a hybridisation function $\Delta(\tau)$ in imaginary time given the bare dispersion of a two-dimensional square lattice with nearest neighbour hopping, at chemical potential μ. This is a typical task which is usually performed at

Figure 2: Screenshot of an IPython notebook executing a DMFT loop using the CT-INT solver.



the beginning of a DMFT calculation. The necessary steps are the following:

$$G_0(i\omega_n) = \frac{1}{N}\sum_{\mathbf{k}} \frac{1}{i\omega_n + \mu - 2(\cos k_x + \cos k_y)} \tag{1}$$

$$\Delta(i\omega_n) = i\omega_n + \mu - G_0^{-1}(i\omega_n) \tag{2}$$

$$\Delta(\tau) = \frac{1}{\beta}\sum_n \Delta(i\omega_n) e^{-i\omega_n \tau} \tag{3}$$

The sum over $\mathbf{k} = (k_x, k_y)$ is taken over the Brillouin zone, ω_n is a fermionic Matsubara frequency and β is the inverse temperature. Using the library, these equations are implemented as follows:

Listing 1 Computing the hybridisation

```
#include <triqs/gfs.hpp>                                                          1
using namespace triqs::gfs;                                                       2
using namespace triqs::lattice;                                                   3
int main() {                                                                      4
                                                                                  5
 double beta = 10, mu = 0;                                                        6
 int n_freq = 100, n_pts = 100;                                                   7
                                                                                  8
 // Green's function on Matsubara frequencies, 1x1 matrix-valued.                 9
 auto Delta_iw = gf<imfreq>{{beta, Fermion, n_freq}, {1, 1}};                     10
 auto Gloc     = gf<imfreq>{{beta, Fermion, n_freq}, {1, 1}};                     11
                                                                                  12
 // Green's function in imaginary time, 1x1 matrix-valued.                        13
 auto Delta_tau = gf<imtime>{{beta, Fermion, 2 * n_freq + 1}, {1, 1}};            14
                                                                                  15
 auto bz = brillouin_zone{bravais_lattice{{ {1, 0}, {0, 1} }}};                   16
 auto bz_mesh = regular_bz_mesh{bz, n_pts};                                       17
                                                                                  18
 triqs::clef::placeholder<1> k_;                                                  19
 triqs::clef::placeholder<2> iw_;                                                 20
                                                                                  21
 // The actual equations                                                          22
 Gloc(iw_) << sum(1/(iw_ + mu - 2*(cos(k_[0]) + cos(k_[1]))), k_ = bz_mesh)       23
              /bz_mesh.size();                                  // (1)            24
 Delta_iw(iw_) << iw_ + mu - 1 / Gloc(iw_);                    // (2)            25
 Delta_tau() = inverse_fourier(Delta_iw);                      // (3)            26
                                                                                  27
 // Write the hybridization to an HDF5 archive                                    28
 auto file = triqs::h5::file("Delta.h5", H5F_ACC_TRUNC);                          29
 h5_write(file, "Delta_tau", Delta_tau);                                          30
 h5_write(file, "Delta_iw", Delta_iw);                                            31
}                                                                                 32
```

In the implementation of equations (1) and (2) (lines 23–25), we use a compact syntax for the assignment to the Green's function container provided by the TRIQS library (the CLEF library, Sec. 8.5). By definition, this is equivalent to assigning the evaluation of the expression on the right-hand side to the data points of the "Green's function"[1] `Delta_iw` on the left and for all Matsubara frequencies in its mesh. `iw_` is a *placeholder*, i.e. a dummy variable standing for all points in the Green's function's mesh.

In line 23, the formal expression made of `iw_` and `k_` is summed over the values of `k_`, and assigned to the function for each `iw_`. Note that no copy is made by this statement, the computation is inlined by the compiler, as if it was written manually. This technique is more concise than writing a for-loop on each variable, reduces the risk of errors and simultaneously increases the readability. Moreover, such techniques come with *no performance penalty* (as indicated by our tests on several standard compilers).

In line, we assign the inverse Fourier transform of $\Delta(\omega)$ to $\Delta(\tau)$ [Eq. (3)]. Note that the high-frequency expansion is part of the Green's function container and so is *automatically* computed in lines 23 and 25 (see Sec. 8.2 for details). It is used to properly treat the discontinuity in the Fourier transformation in line 26.

Finally, the Green's functions are stored in an HDF5 file with a simple interface, in a portable manner. The storage conventions are detailed in the reference documentation.

[1] Here we refer the Green's function containers and objects representing hybridisation functions or functions with the same signature simply as "Green's functions".

This example is interesting for two reasons: firstly, it shows that the TRIQS library performs a lot of low-lying operations. There is no need to reimplement them and the user can concentrate on the physics; secondly, it shows that one can write quite complex operations concisely, which is necessary in order to write readable codes.

8. Library components

In this section, we provide an overview of the TRIQS library components. We illustrate them either with small examples or in the CT-INT impurity solver example presented in Appendix A. The description is neither meant to be complete nor exhaustive; the online reference documentation (http://ipht.cea.fr/triqs) will fill the gaps.

8.1. Multidimensional arrays (C++)

TRIQS provides its own multidimensional arrays, with an emphasis on flexibility, performance and the `Python` interface. It is a fundamental building block for higher-level containers, such as Green's functions. Listing 2 below illustrates some of their features.

Listing 2 Array / matrix example

```
#include <triqs/arrays.hpp>
using namespace triqs::arrays;
int main() {
 auto a = matrix<double>(2, 2);                 // Declare a 2x2 matrix of double
 auto b = array<double, 3>(5, 2, 2);            // Declare a 5x2x2 array of double
 auto c = array<double, 2> {{1,2,3}, {4,5,6}}; // 2x3 array, with initialization

 triqs::clef::placeholder<0> i_;
 triqs::clef::placeholder<1> j_;
 triqs::clef::placeholder<2> k_;

 // Assignment of values using CLEF
 a(i_, j_) << i_ + j_;
 b(i_, j_, k_) << i_ * a(k_, j_);

 std::cout << "a = " << a << std::endl; // Printing

 matrix<double> i = inverse(a); // Inverse using LAPACK
 double d = determinant(a);     // Determinant using LAPACK

 auto ac = a;                  // Make a copy (the container is a regular type)
 ac = a * a + 2 * ac;          // Basic operations (uses BLAS for matrix product)
 b(0, range(), range()) = ac; // Assign ac into partial view of b

 // Writing the array into an HDF5 file.
 auto f = triqs::h5::file("a_file.h5", H5F_ACC_TRUNC);
 h5_write(f, "a", a);

 auto m = max_element(abs(b)); // maximum of the absolute value of the array.

 // A more "functional" example: compute the norm sum_{i,j} |A_{ij}|
 auto lambda = [](double r, double x) { return r + std::abs(x); };
 auto norm = fold(lambda)(a, 0);
}
```

The library provides three types of containers: *array* (for multidimensional arrays), *matrix* and *vector* with the following main characteristics:

- **Regular-type semantics**: Just like `std::vector`, these containers have regular-type semantics.
- **Views**: Each container has a corresponding view type (e.g. `array_view`) to e.g. work on slices and partial views.
- **CLEF**: The containers are compatible with CLEF (Sec. 8.5 and Listing 2) for fast assignment techniques.
- `Python` **interface**: These containers can be easily converted to and from `Python` NumPy arrays.
- **Interface to HDF5**: See Sec. 8.6 and Listing 2.
- **Arithmetics**: Arithmetic operations are implemented using expression templates for optimal performance.
- **BLAS/LAPACK**: A BLAS/LAPACK interface for matrices and vectors is provided for the most common operations.
- **STL compatible iterators**: The containers and views can be traversed using such iterators, or with simple `foreach` constructs.
- *Optionally*, a (slower) debug mode checks for out-of-range operations.

8.2. Green's functions (C++ and Python)

The library provides a special set of containers that allow one to store and manipulate the various Green's functions used in the quantum many-body problem and its algorithms. They are defined on meshes for various domains, they are tensor-, matrix-, or scalar-valued and can be block-diagonal.

Domains currently implemented include real and imaginary frequencies, real and imaginary times, Legendre polynomials, and Brillouin zones. Multiple variable Green's functions are also part of the library, but are restricted to `C++14` mode only and are of alpha quality in release 1.2. We will not discuss them in this paper.

Green's functions optionally include a description of their high-frequency behaviour in terms of their moments. Storing this information is important for several operations (e.g. Fourier transformation, frequency summation) where the high-frequency behaviour needs to be treated explicitly. Being part of the object, the singularity is consistently recomputed in all arithmetic operations so that the user need not work out the high-frequency asymptotics. Listing 3 illustrates some basic usage of Green's functions, while a `Python` example has been given above (Fig. 2).

Listing 3 Green's function example

```
#include <triqs/gfs.hpp>
using namespace triqs;
using namespace triqs::gfs;
using namespace triqs::lattice;
int main() {
```

```
 double beta = 10;
 int n_freq = 1000;
 clef::placeholder<0> iw_;
 clef::placeholder<1> k_;
 // Construction of a 1x1 matrix-valued fermionic gf on Matsubara frequencies.
 auto g_iw = gf<imfreq>{{beta, Fermion, n_freq}, {1, 1}};
 // Automatic placeholder evaluation
 g_iw(iw_) << 1 / (iw_ + 2);
 // Inverse Fourier transform to imaginary time
 auto g_tau = gf<imtime>{{beta, Fermion, 2 * n_freq + 1}, {1, 1}};
 g_tau() = inverse_fourier(g_iw); // Fills a full view of g_tau with FFT result
 // Create a block Green's function composed of three blocks,
 // labeled a,b,c and made of copies of the g_iw functions.
 auto G_iw = make_block_gf({"a", "b", "c"}, {g_iw, g_iw, g_iw});
 // A multivariable gf: G(k,omega)
 auto bz = brillouin_zone{bravais_lattice{{{1, 0}, {0, 1}}}};
 auto g_k_iw = gf<cartesian_product<brillouin_zone, imfreq>>{
     {{bz, 100}, {beta, Fermion, n_freq}}, {1, 1}};
 g_k_iw(k_, iw_) << 1 / (iw_ - 2 * (cos(k_(0)) + cos(k_(1))) - 1 / (iw_ + 2));
 // Writing the Green's functions into an HDF5 file.
 auto f = h5::file("file_g_k_iw.h5", H5F_ACC_TRUNC);
 h5_write(f, "g_k_iw", g_k_iw);
 h5_write(f, "g_iw", g_iw);
 h5_write(f, "g_tau", g_tau);
 h5_write(f, "block_gf", G_iw);
}
```

The main characteristics of Green's functions are:

- **Arithmetics**: Just like arrays, Green's functions implement arithmetic operations using **expression templates**.
- **Quick assignment**: The class uses the CLEF component of the TRIQS library for quick assignment (see Sec. 8.5 and Listing 3).
- `Python` **interface**: The Green's functions are easily shared between `Python` and `C++`, see Sec. 8.8, and can thus be used in conjunction with the `Python` visualisation tools.
- **Fourier transforms**: TRIQS provides a simple interface to fast Fourier transforms (FFTW). For Green's functions the information about the high-frequency behaviour is used to avoid numerical instabilities.
- **Interface to HDF5**.

8.3. Monte Carlo tools (C++)

The TRIQS library provides several classes for writing Metropolis-like (quantum) Monte Carlo algorithms. In addition to some basic analysis tools, like binning or jackknife, the library mainly contains the `mc_generic` class that implements the Metropolis algorithm (choose a move, try the move, compute Metropolis ratio, reject or accept, etc.) in terms of completely generic *moves* (configuration updates) and *measurements*.

In practice, one just needs to implement the moves and measurements. The only requirement is that they must model their respective *concepts* [2]. For example, the concept of a move is given by Listing 4. Note in particular that they do not require inheritance or virtual functions, which makes them particularly simple to use.

Listing 4 Concept of a Monte Carlo move

```
struct my_monte_carlo_move {
  // propose a change in the configuration and return the Metropolis ratio
  double attempt();
  // the move has been accepted: modify configuration
  double accept();
  // the move has been rejected: undo configuration changes
  void reject();
};
```

A concrete usage of the class is shown in the CT-INT solver example (Appendix A). The class is particularly convenient for complex Monte Carlo algorithms with several moves: the moves are isolated from the implementation of the Metropolis algorithm itself and each move can be implemented independently.

The Monte Carlo is of course automatically MPI-enabled. Furthermore, random number generators can easily be changed dynamically to ensure there is no subtle correlation effect.

Listing 5 illustrates a basic application of the tools for statistical analysis on a correlated random series. Let us assume we have two long vectors `V1` and `V2` storing (possibly correlated) samples of the random variables X and Y and that we wish to compute estimates of $\langle X \rangle$ and $\langle X \rangle / \langle Y \rangle$, together with the corresponding error bars. In both cases, the correlation between samples has to be removed using a *binning* procedure. This being done, the first computation is quite straightforward, while the second one further requires a jackknife procedure to remove the bias introduced by the nonlinearity. In TRIQS, all these operations are performed by the following code snippet, using a little library similar to e.g. `ALPS/alea` [7]:

Listing 5 Statistics: error analysis

```
//fill observable with the series
observable<double> X, Y;
for(auto const & x : V1) X << x; //V1: a vector of statistical samples
for(auto const & y : V2) Y << y; //V2: a vector of statistical samples

std::cout << "<X> is approximately " << average_and_error(X) << std::endl;
std::cout << "<X>/<Y> is approx.  " << average_and_error(X/Y) << std::endl;
```

`X<<x` fills the observable `X` (a stack of the samples) with the values `x` of `V1`. `average_and_error(X)` computes an estimate of $\langle X \rangle$ and of the error $\Delta\langle X \rangle$, while `average_and_error(X/Y)` computes an estimate of $\langle X \rangle / \langle Y \rangle$ and of $\Delta\langle X \rangle / \langle Y \rangle$.

8.4. Determinant manipulations (C++)

The manipulation of determinants is central to many Monte Carlo approaches to fermionic problems, see e.g. [10, 12, 13, 11]. Several cases can be abstracted from

[2] In the sense of C++ concepts.

the following mathematical problem. Let us consider a function $F(x, y)$ taking real or complex values (the type of the arguments x and y is arbitrary) and the square matrix M defined by

$$M_{ij} = F(x_i, y_j), \tag{4}$$

for two sets of parameters $\{x_i\}$ and $\{y_j\}$ of equal length. The problem consists in quickly updating M and its inverse M^{-1} following successive insertions and removals of one or two lines (labelled by x_i) and columns (labelled by y_j) using the Sherman-Morrison and Woodbury formulas [17, 18].

This generic algorithm is implemented in the TRIQS `det_manip` class, using BLAS Level 2 [19, 20] internally. The class provides a simple API, in order to make these manipulations as straightforward and efficient as possible.

For optimal efficiency within a Monte Carlo framework, the modifications to the matrices can be done in two steps: a first step which only returns the determinant ratio between the matrix before (M) and after the modification (M'), i.e. $\xi = \det M' / \det M$ (which is generally used in the acceptance rate of a Metropolis move) and a second step which updates the matrix and its inverse. This computationally more expensive step is usually done only if the Monte Carlo move is accepted. An example of this class is employed is the CT-INT solver discussed in Appendix A.

8.5. CLEF (C++)

CLEF (Compile-time Lazy Expressions and Functions) is a component of TRIQS which allows one to write expressions with placeholders and functions, and to write quick assignments. For example, the following – quite involved – equation

$$\chi^{0\,\sigma\sigma'}_{\nu\nu'\omega} = \beta(g^{0\,\sigma}_{\nu} g^{0\,\sigma'}_{\nu'} \delta_{\omega} - g^{0\,\sigma}_{\nu} g^{0\,\sigma}_{\nu+\omega} \delta_{\nu\nu'} \delta_{\sigma\sigma'}) \tag{5}$$

can be coded as quickly as (variables with underscores denote placeholders)

```
chi0(s_, sp_)(nu_, nup_, om_) <<
      beta * (g[s_](nu_) * g[sp_](nup_) * kronecker(om_))
    - beta * (g[s_](nu_) * g[s_](nu_ + om_)
                         * kronecker(nu_, nup_) * kronecker(s_, sp_));
```

This writing is clearly much simpler and less error-prone than a more conventional five-fold nested `for`-loop. At the same time, these expressions are inlined and optimised by the compiler, as if the code were written manually. The library also automatically optimises the memory traversal (the order of `for` loops) for performance based on the actual memory layout of the container `chi0`.

The CLEF expressions are very similar to `C++` lambdas, except that their variables are found by name (the placeholder) instead of a positional argument (in calling a lambda). This is much more convenient for complex codes.

The precise definition of the automatic assignment is as follows. Any code of the form (e.g. with three placeholders):

```
A(i_,j_,x_) << expression;
```

where `expression` is an expression involving placeholders[3] is rewritten by the compiler as follows:

[3] For a precise list of what is allowed in expressions, the reader is referred to the reference documentation.

```
triqs_auto_assign(
  A, [](auto& i,auto& j,auto& x) {
           return eval(expression, i_=i, j_=j, x_=x);
       }
);
```

where `triqs_auto_assign` is a free function defined by the container A, which fills the container with the result of the evaluation of the lambda, and `eval` evaluates the expression (`eval` is a function and is part of CLEF). The precise details of this operation, such as the memory traversal order, are encoded in this function. The CLEF quick assignment mechanism can therefore easily and efficiently be extended to any object of the library. The library provides adaptors to allow standard mathematical functions such as `cos` or `abs` and `std::vector` to be used in expressions. User-defined functions and class methods can conveniently be made compatible with the CLEF quick assignment through macros.

8.6. HDF5 (C++ and Python)

HDF5 is a standard, portable and compact file format, see http://www.hdfgroup.org. Almost all objects in the TRIQS library (including arrays or Green's functions) can be stored in and retrieved from HDF5 files, from `C++` and/or `Python`, with a simple and uniform interface. For example, in `C++`:

```
auto a = array<double, 2> {{1,2,3}, {4,5,6}}; // some data
{
 auto f = h5::file("data.h5", 'w');           // open the file
 h5_write(f, 'a', a);                         // write to the file
}                                             // closes the file
```

or, the corresponding code in `Python`:

```
a = numpy.array([[1,2,3],[4,5,6]])
with HDFArchive("data.h5", 'w') as f:
  f['a'] = a
```

In `Python`, the HDFArchive behaves in a similarly as a `dict`. Therefore, one can reload a complex object (e.g. a block Green's function) in a single command in a script. An example can be seen in Listing 1.

An HDF5 file can be seen as a tree whose leaves are "basic" objects (multidimensional rectangular arrays, double, integer, strings, ...). More complex objects are usually decomposed by the library into a subtree of smaller objects, which are stored in an HDF5 subgroup. For example, a block diagonal Green's function (of type *BlockGf*) is stored with subgroups containing the Green's functions it is made of; a Green's function is stored as a subgroup containing the array of data, the mesh, and possibly the high frequency singularity. This *format*, i.e. the precise conventions for the names and types of the small objects and the storage order of the data in the arrays, is described in the reference documentation. The HDF5 files can be read *without the TRIQS library* from `C`, `C++`, Fortran, `Python` codes, the HDF5 command line tools and with any tool supporting this format. This enables publishing data and facilitates sharing them across different groups and platforms. The HDF5 format is indeed widely used, e.g. by the ALPS project [7].

Note also that the HDF5 files written from `C++` or `Python` have exactly the same format. Hence one can straightforwardly load some Green's functions in `Python` that have been computed and written using a `C++` code, or vice-versa.

8.7. Second-quantized operators (C++ and Python)

The theories of strongly-correlated electron systems often use a language of second-quantized operators to formulate the problems under consideration. The model Hamiltonians as well as the observables of interest are routinely written as polynomials of fermionic operators $c^\dagger$ and c.

The TRIQS library implements a `C++` template class `many_body_operator`, which abstracts the notion of a second-quantized operator. The purpose of this class is to make expressions for second-quantized operators *written in the* `C++` *or* `Python` *code as close as possible to their analytical counterparts*. In order to pursue this goal, the class implements the standard operator algebra. The library stores the expression in normal order, so it performs automatically basic simplications, for example when an expression vanishes. Any operator can be constructed as a polynomial of the elementary operators carrying an arbitrary number of integer/string indices (defined at compile time). The coefficients of the polynomials may be real, complex or of a user-defined numeric type in advanced use scenarios.

There is also a `Python` version of the same class (called `Operator`), specialised for the case with real coefficients and the fermionic operators with two indices (this particular choice is made for compatibility with the Green's function component). Anyone writing a TRIQS-based many-body solver may benefit from this class. For example, the user of the solver could define a model Hamiltonian in a `Python` script and subsequently pass it to the solver:

```
from pytriqs.operators.operators import Operator, n, c_dag, c
# Spin operators
Sp = c_dag("up",0)*c("dn",0)          # S_+
Sm = c_dag("dn",0)*c("up",0)          # S_-
Sz = 0.5*(n("up",0) - n("dn",0))      # S_z
S2 = Sz*Sz + (Sp*Sm + Sm*Sp)/2        # S^2
# The Hamiltonian of a half-filled Hubbard atom: four equivalent forms
U = 1.0
H1 = -U/2*(n("up",0) + n("dn",0)) + U*n("up",0)*n("dn",0)
H2 = U*(n("up",0) - 0.5)*(n("dn",0) - 0.5) - U/4
H3 = -2.0*U*Sz*Sz
H4 = -2.0/3.0*U*S2
print H1, '\n', H2, '\n', H3, '\n', H4
# All four forms are indeed equivalent
print (H1-H2).is_zero() and (H2-H3).is_zero() and (H3-H4).is_zero()
```

8.8. C++/Python wrapping tool

The tool that glues together the `C++` components to `Python` is a crucial piece of the TRIQS project. Indeed the `C++`/`Python` architecture of the project is very demanding in this aspect: we need to expose diverse components from `C++` to `Python`. These range from simple functions to complex objects with methods, overloaded arithmetic operators, the interface to HDF5, and so on. The tool must be very flexible, while being as simple as possible to use in the most common cases.

The TRIQS library proposes such a tool in version 1.2. From a simple `Python`-written *description* of the classes and functions to expose to `Python`, it generates the necessary `C` wrapping code to build the `Python` module. Utilities are also included to actually compile and setup the modules with `cmake`.

In most cases, the process can be fully automatised, using a second tool based on the `Clang` library, which parses the `C++` code using `libClang` and retrieves the description of the classes and functions along with their documentation. As an example, the

automatically produced description files for the CT-INT algorithm is provided in the Appendix.

In more complex cases, some information can be added manually to the class description, for example the fact that the object forms an algebra over the doubles. In such a case, by adding *a single line* to the description file, the tool automatically generates all the necessary operators for the algebra structure in `Python` by calling their `C++` counterparts.

As a consequence, this tool also allows the TRIQS user to *write* `C++` *code directly within the* `IPython` *notebook and use it immediately*, using a so-called "magic cell command", in `IPython` terminology. This is illustrated in Fig. 3. In this case, the command

Figure 3: Using C++ directly within the IPython notebook

```
In [1]: %reload_ext pytriqs.magic

In [2]: %%triqs
        #include <string>
        std::string hello(){ return "Hello, world!"; }

In [3]: print hello()
        Hello, world!

In [4]: type(hello())
Out[4]: str
```

`%%triqs` extracts the prototype of the `C++` `hello()` function, writes, compiles and loads the `Python` module to be used in the next cell. TRIQS objects, along with STL containers (e.g. vector, tuple), can be used as function arguments or return values.

Using this feature one can tinker with `C++` codes directly inside a `Python` environment, without having to set up a `C++` project. It is suitable for debugging, quick testing, or executing short `C++` code. For longer codes, it is better to set up a `Python`/`C++` project along the lines shown for the CT-INT in the Appendix. Note that this feature is experimental in release 1.2 and currently limited to a single `C++` function per cell (even though generalisation is quite straightforward).

9. Getting started

Detailed information on installation can be found on the TRIQS website and current issues and updates are available on GitHub.

9.1. Obtaining TRIQS

The TRIQS source code is available publicly and can be obtained by cloning the repository on the GitHub website https://github.com/TRIQS/triqs. As the TRIQS project is continuously evolving, we recommend that users always obtain TRIQS from GitHub. Fixes to possible issues are also applied to the GitHub source.

9.2. Installation

Installing TRIQS is straightforward. We use the `cmake` tool to configure, build and test the library. Assuming that all dependencies have been installed (refer to the online documentation), the library is simply installed by issuing the following commands at the shell prompt:

```
$ git clone https://github.com/TRIQS/triqs.git src
$ mkdir build_triqs && cd build_triqs
$ cmake ../src
$ make
$ make test
$ make install
```

By default, the installation directory `INSTALL_DIR` will be located inside the build directory. Further installation instructions and help on installing the dependencies can be found in the online documentation.

9.3. Usage

There are different ways of using TRIQS. In the following, we assume that the location of the `INSTALL_DIR/bin` folder is in the search path. We recommend starting with one of the interactive `IPython` notebook examples provided with this paper (see below). The interactive `IPython` notebook is started using the command

```
$ ipytriqs_notebook
```

which will open the browser and allow one to open an existing or a new notebook. Providing a notebook name as an argument will open the notebook directly.

The `IPython` example in Fig. 2 uses the CT-INT solver of Appendix A, which is shipped as a separate application. Installing external applications is straightforward. The CT-INT application, for example, is installed as follows:

```
$ git clone https://github.com/TRIQS/ctint_tutorial.git src_ctint
$ mkdir build_ctint && cd build_ctint
$ cmake -DTRIQS_PATH=INSTALL_DIR_ABSOLUTE_PATH ../src_ctint
$ make
$ make test
$ make install
```

where `INSTALL_DIR_ABSOLUTE_PATH` is the (absolute) path to the TRIQS installation directory. The application will be installed into the `applications` subdirectory in this TRIQS installation directory. Assuming that `INSTALL_DIR_ABSOLUTE_PATH/bin` is in the UNIX search path, one can then execute the example notebook in Fig. 2. To this end, navigate to the `examples` directory of the `ctint_tutorial` application sources and issue the following command:

```
$ ipytriqs_notebook dmft_bethe.ipynb
```

This will load the notebook inside a browser. Individual cells can be executed by pressing [Shift+ENTER] (refer to the IPython notebook documentation). The same directory contains a Python script to execute the same DMFT loop from the command line, which is another mode to use TRIQS that is better suited for long computations on a parallel machine. It can be executed by typing

```
$ pytriqs dmft_bethe.py
```

or in parallel by running, e.g.,

```
$ mpirun -np 4 pytriqs dmft_bethe.py
```

These commands produce a file dmft_bethe.output.h5. To plot the Green's function from the final iteration, we can launch ipytriqs and type:

```
$ ipytriqs
...
In [1]: from pytriqs.archive import *

In [2]: from pytriqs.gf.local import *

In [3]: from pytriqs.plot.mpl_interface import oplot, plt

In [4]: A = HDFArchive("dmft_bethe.output.h5","r")

In [5]: oplot(A["G20"].imag, "-o", name="Im G20")

In [6]: plt.show()
```

As a starting point for developing an external application, we provide a minimal skeleton application called hello_world. It can be installed in the same way as the CT-INT solver. The C++ examples of this paper, various IPython notebooks and the hello_world are provided in a dedicated GitHub repository https://github.com/TRIQS/tutorials.git.

10. Contributing

TRIQS is an open source project and we encourage feedback and contributions from the user community to the library and the publication of applications based on it. Issues should be reported exclusively via the GitHub web site at https://github.com/TRIQS/triqs/issues. For contributions, we recommend to use the *pull request* system on the GitHub web site. Before any major contribution, we recommend to coordinate with the main TRIQS developers.

11. Summary

We have presented the TRIQS library, a Toolbox for Research on Interacting Quantum Systems. This open-source computational physics library provides a framework

for the quick development of applications in the field of many-body quantum physics. Several applications have been built on this library already. They are available at https://github.com/TRIQS and will be described in other publications.

12. Acknowledgements

The TRIQS project is supported by the ERC Grant No. 278472–*MottMetals*. We acknowledge contributions to the library and feedbacks from M. Aichhorn, A. Antipov, L. Boehnke, L. Pourovskii, as well as feedback from our user community. I.K. acknowledges support from Deutsche Forschungsgemeinschaft via Project SFB 668-A3. P.S. acknowledges support from ERC Grant No. 617196–*CorrelMat*.

Appendix A. A sample application: Interaction expansion continuous-time quantum Monte Carlo algorithm

In this Appendix, we present the implementation of a simple interaction expansion continuous-time quantum Monte Carlo algorithm (CT-INT). We have used this solver in the DMFT `IPython` example in Fig. 2. We first briefly recall the formalism of the CT-INT algorithm before discussing the code.

Appendix A.1. Formalism

We consider the following single-orbital impurity action

$$S = -\sum_{\sigma} \int_0^\beta \int_0^\beta d\tau d\tau' \bar{d}_\sigma(\tau) \tilde{G}^{-1}_{0\sigma}(\tau - \tau') d_\sigma(\tau') + \int_0^\beta d\tau \mathcal{H}_{\rm int}(\tau), \tag{A.1}$$

whose interaction term is a slightly modified Hubbard term

$$\mathcal{H}_{int} = \frac{U}{2} \sum_{s=\uparrow,\downarrow} \left(\hat{n}_\uparrow - \alpha^{s\uparrow}\right)\left(\hat{n}_\downarrow - \alpha^{s\downarrow}\right) \tag{A.2}$$

with

$$\alpha^{s\sigma} = \frac{1}{2} + (2\delta_{s\sigma} - 1)\delta. \tag{A.3}$$

Here δ is a free small parameter which reduces the sign problem and $\delta_{s\sigma}$ is a Kronecker symbol. This rewriting of the interaction term results in a shift of the chemical potential (absorbed in the bare Green's function $\tilde{G}_0$): $\tilde{\mu} = \mu - \frac{U}{2}$. The α's only appear in the interaction term.

The CT-INT algorithm consists in expanding the partition function $Z = \int \mathcal{D}[\bar{d}, d] e^{-S}$ in powers of $\mathcal{H}_{int}$. One obtains:

$$Z = Z_0 \sum_{k=0}^{\infty} \frac{(-U)^k}{k!} \int_0^\beta d\tau_1 \dots d\tau_k \frac{1}{2^k} \times$$
$$\times \sum_{s_1 \dots s_k = \uparrow,\downarrow} \left\langle T_\tau (n_\uparrow(\tau_1) - \alpha^{s_1\uparrow}) \cdots (n_\uparrow(\tau_k) - \alpha^{s_k\uparrow})(n_\downarrow(\tau_1) - \alpha^{s_1\downarrow}) \cdots (n_\downarrow(\tau_k) - \alpha^{s_k\downarrow}) \right\rangle_0 . \tag{A.4}$$

where T_τ is the time ordering operator. In the original CT-INT algorithm proposed in Ref. [10], $\alpha^{\uparrow\uparrow} = 1 - \alpha^{\uparrow\downarrow} = \alpha$ and $\alpha^{\downarrow\uparrow} = \alpha^{\downarrow\downarrow} = 0$. This choice can be shown to eliminate the sign problem for the half-filled single band Anderson model. Here we sum over the indices to make the formulation slightly more symmetric. It has the advantage that the non-interacting Green's function $\tilde{G}$ does not explicitly depend on α's. Note that in the case of all $\alpha^{s_i\sigma}$ being the same, the sum over s_i produces a factor 2^k, which cancels the 2^k in the denominator. The non-interacting Green's function has no off-diagonal up/down terms, so that the average factorises into product of two correlation functions for each spin. Let us furthermore introduce time ordering by replacing the integrals over the complete time intervals into a product of time-ordered integrals,

$$\int_0^\beta d\tau_1 \dots d\tau_k \left\langle \prod_{i=1}^{k} \prod_{\sigma} (n_\sigma(\tau_i) - \alpha^{s_i\sigma}) \right\rangle_0 = k! \int_0^\beta d\tau_1 \int_0^{\tau_1} d\tau_2 \dots \int_0^{\tau_{k-1}} d\tau_k \times$$
$$\times \left\langle (n_\uparrow(\tau_1) - \alpha^{s_1\uparrow}) \cdots (n_\uparrow(\tau_k) - \alpha^{s_k\uparrow}) \right\rangle_0 \left\langle (n_\downarrow(\tau_1) - \alpha^{s_1\downarrow}) \cdots (n_\downarrow(\tau_k) - \alpha^{s_k\downarrow}) \right\rangle_0 . \quad \text{(A.5)}$$

Using Wick's theorem and the usual definition for the Green's function

$$G_0^\sigma(\tau) = -\left\langle T_\tau d_\sigma(\tau) \bar{d}_\sigma(0) \right\rangle_0 , \quad \text{(A.6)}$$

the averages can be represented by determinants. We hence arrive at

$$Z = Z_0 \sum_{k=0}^{\infty} \int_> d\tau_1 \dots d\tau_k \sum_{s_1 \dots s_k} \frac{(-U)^k}{2^k} \det D_k^\uparrow \det D_k^\downarrow . \quad \text{(A.7)}$$

The determinants explicitly read

$$D_k^\sigma = \begin{bmatrix} \bar{G}_0^{s_1\sigma}(0^-) & \bar{G}_0^{s_1\sigma}(\tau_1 - \tau_2) & \dots & \bar{G}_0^{s_1\sigma}(\tau_1 - \tau_k) \\ \bar{G}_0^{s_2\sigma}(\tau_2 - \tau_1) & \bar{G}_0^{s_2\sigma}(0^-) \dots & \bar{G}_0^{s_2\sigma}(\tau_2 - \tau_k) & \\ \dots & \dots & \dots & \dots \\ \bar{G}_0^{s_k\sigma}(\tau_k - \tau_1) & \bar{G}_0^{s_k\sigma}(\tau_k - \tau_2) & \dots & \bar{G}_0^{s_k\sigma}(0^-) \end{bmatrix} , \quad \text{(A.8)}$$

where we have defined the Green's function $\bar{G}_0^{s\sigma}$ as

$$\bar{G}^{s_1\sigma}(\tau_1 - \tau_2) = \begin{cases} \tilde{G}_0^\sigma(0^-) - \alpha^{s_1\sigma} & \tau_1 = \tau_2 \\ \tilde{G}_0^\sigma(\tau_1 - \tau_2) & \tau_1 \neq \tau_2 \end{cases} . \quad \text{(A.9)}$$

We can sample the partition function (A.7) by defining a Monte Carlo configuration as $\mathcal{C} \coloneqq \{\{\tau_1, s_1\}, \dots, \{\tau_k, s_k\}\}$ and the Monte Carlo weight of a configuration according to $\omega(\mathcal{C}) = |(-U/2)^k \det D_k^\uparrow D_k^\downarrow|$. The Metropolis acceptance rate for an insertion of a vertex is

$$A_{x,y} = \min \left[1, \frac{-\beta U}{k+1} \frac{\det D_{k+1}^\uparrow D_{k+1}^\downarrow}{\det D_k^\uparrow D_k^\downarrow}\right], \quad \text{(A.10)}$$

while for a removal, it is

$$A_{x,y} = \min \left[1, \frac{-k}{\beta U} \frac{\det D_{k-1}^\uparrow D_{k-1}^\downarrow}{\det D_k^\uparrow D_k^\downarrow}\right]. \quad \text{(A.11)}$$

The Green's function can be calculated as

$$G_\sigma(\tau) = -\frac{1}{\beta}\frac{\delta \ln Z}{\delta \Delta^\sigma(-\tau)}. \tag{A.12}$$

Carrying out the functional derivative and Fourier transforming yields

$$G^\sigma(i\omega_n) = \tilde{G}_0^\sigma(i\omega_n) - \frac{1}{\beta}(\tilde{G}_0^\sigma(i\omega_n))^2 \sum_{\mathcal{C}}^{\mathrm{MC}} \sum_{ij} [D_k^\sigma]^{-1}_{ij} e^{i\omega_n(\tau_i - \tau_j)} \mathrm{sign}[\omega(\mathcal{C})]\omega(\mathcal{C}). \tag{A.13}$$

Separating the Monte Carlo weight, we need to accumulate

$$M^\sigma(i\omega_n) \equiv -\frac{1}{Z\beta} \sum_{\mathcal{C}}^{\mathrm{MC}} \sum_{ij} [D_k^\sigma]^{-1}_{ij} e^{i\omega_n(\tau_i - \tau_j)} \times \mathrm{sign}[\omega(\mathcal{C})], \tag{A.14}$$

$$Z = \sum_{\mathcal{C}}^{\mathrm{MC}} \mathrm{sign}[\omega(\mathcal{C})], \tag{A.15}$$

From M, we can compute the Green's function as follows [13]:

$$G^\sigma(i\omega_n) = \tilde{G}_0^\sigma(i\omega_n) + \tilde{G}_0^\sigma(i\omega_n) M^\sigma(i\omega_n) \tilde{G}_0^\sigma(i\omega_n). \tag{A.16}$$

Appendix A.2. Implementation

As an example of an application of the library, we discuss here the complete code listing of a fully working, parallelized implementation of the weak-coupling CTQMC algorithm described above. How this code can be used in an actual computation is illustrated in the DMFT example of Sec. 7.1. Through the use of the various components of the library, including `gf`, `mc_tools`, `det_manip` and `CLEF`, the full implementation takes about 200 lines; it comes with a `Python` interface. Note that this simple implementation can easily be extended: further measurements and moves may be added, or it may be generalised to multi-orbital case or to a retarded interaction.

We divide the code into several listings that we discuss briefly. The purpose is to give an illustration of the possibilities of the TRIQS library without entering into all the details. We start with the main header file (Listing 6) of the code. It mainly defines and provides access to the Green's functions that are used in the code, in particular in the main member function `solve`.

Listing 6 CT-INT: the header file

```
#include <triqs/gfs.hpp>
#include <boost/mpi.hpp>

// ------------ The main class of the solver -----------------------

using namespace triqs::gfs;
enum spin {up, down};

class ctint_solver {

 block_gf<imfreq> g0_iw, g0tilde_iw, g_iw, M_iw;
 block_gf<imtime> g0tilde_tau;
 double double_occ, percent_done_, beta;
 int n_matsubara, n_times_slices;
```

```
15
16   public:
17
18   // Accessors of the class
19   block_gf_view<imfreq> G0_iw() { return g0_iw; }
20   block_gf_view<imtime> G0_tau() { return g0tilde_tau; }
21   block_gf_view<imfreq> G_iw() { return g_iw; }
22
23   ctint_solver(double beta_, int n_iw = 1024, int n_tau = 100001);
24
25   // The method that runs the qmc
26   void solve(double U, double delta,
27              int n_cycles, int length_cycle = 50, int n_warmup_cycles = 5000,
28              std::string random_name = "",
29              int max_time = -1);
30
31 };
```

Listing 7 defines the Monte Carlo configurations through a simple vector of determinants A.8 (instances of the `det_manip` class). They contain all the necessary information to completely determine a configuration $\mathcal{C} \coloneqq \{\{\tau_1, s_1\}, \ldots, \{\tau_k, s_k\}\}$. The determinants are constructed from a function object `g0bar_tau`, also declared in this listing, that is used to fill the elements of the matrix A.8.

Listing 7 CT-INT: define the configurations

```
1  // --------------- The QMC configuration ----------------
2
3  // Argument type of g0bar
4  struct arg_t {
5   double tau;   // The imaginary time
6   int s;        // The auxiliary spin
7  };
8
9  // The function that appears in the calculation of the determinant
10 struct g0bar_tau {
11  gf<imtime> const &gt;
12  double beta, delta;
13  int s;
14
15  double operator()(arg_t const &x, arg_t const &y) const {
16   if ((x.tau == y.tau)) { // G_\sigma(0^-) - \alpha(\sigma s)
17    return 1.0 + gt[0](0, 0) - (0.5 + (2 * (s == x.s ? 1 : 0) - 1) * delta);
18   }
19   auto x_y = x.tau - y.tau;
20   bool b = (x_y >= 0);
21   if (!b) x_y += beta;
22   double res = gt[closest_mesh_pt(x_y)](0, 0);
23   return (b ? res : -res); // take into account antiperiodicity
24  }
25 };
26
27 // The Monte Carlo configuration
28 struct configuration {
29  // M-matrices for up and down
30  std::vector<triqs::det_manip::det_manip<g0bar_tau>> Mmatrices;
31
32  int perturbation_order() const { return Mmatrices[up].size(); }
33
34  configuration(block_gf<imtime> &g0tilde_tau, double beta, double delta) {
35   // Initialize the M-matrices. 100 is the initial matrix size
36   for (auto spin : {up, down})
37    Mmatrices.emplace_back(g0bar_tau{g0tilde_tau[spin], beta, delta, spin}, 100);
38  }
39 };
```

Now that the configuration is declared, the next step is to define the Monte Carlo *moves* that are going to act on this configuration. In Listing 8, two moves are im-

plemented: the insertion of an interaction vertex at a random imaginary time and the removal of a randomly chosen vertex. They are described by classes that must model the concept of a Monte Carlo move. In other words they must have the three members `attempt`, `accept`, `reject`. The `attempt` method tries a modification of the configuration and returns a Metropolis acceptance ratio (e.g. for the insertion this ratio is given by A.10). The Monte Carlo class will use this ratio to decide wether to accept or reject the proposed configuration and then call `accept` or `reject` accordingly. Note that for efficiency reasons the update of the determinants is done in two steps: in the `attempt` method only the ratio of the new to the old determinant is computed (via `try_insert`). The actual update of the full inverse matrix is performed only if the move is accepted (see the `complete_operation` call in `accept`).

Listing 8 CT-INT: define the moves

```
// ------------ QMC move : inserting a vertex ------------------

struct move_insert {
 configuration *config;
 triqs::mc_tools::random_generator &rng;
 double beta, U;

 double attempt() { // Insert an interaction vertex at time tau with aux spin s
  double tau = rng(beta);
  int s = rng(2);
  auto k = config->perturbation_order();
  auto det_ratio = config->Mmatrices[up].try_insert(k, k, {tau, s}, {tau, s}) *
                   config->Mmatrices[down].try_insert(k, k, {tau, s}, {tau, s});
  return -beta * U / (k + 1) * det_ratio; // The Metropolis ratio
 }

 double accept() {
  for (auto &d : config->Mmatrices) d.complete_operation(); // Finish insertion
  return 1.0;
 }

 void reject() {}
};

// ------------ QMC move : deleting a vertex ------------------

struct move_remove {
 configuration *config;
 triqs::mc_tools::random_generator &rng;
 double beta, U;

 double attempt() {
  auto k = config->perturbation_order();
  if (k <= 0) return 0; // Config is empty, trying to remove makes no sense
  int p = rng(k);       // Choose one of the operators for removal
  auto det_ratio = config->Mmatrices[up].try_remove(p, p) *
                   config->Mmatrices[down].try_remove(p, p);
  return -k / (beta * U) * det_ratio; // The Metropolis ratio
 }

 double accept() {
  for (auto &d : config->Mmatrices) d.complete_operation();
  return 1.0;
 }

 void reject() {} // Nothing to do
};
```

The measurement of the Green's function is shown in Listing 9. It is a simple transcription of Eq. A.15. Again, the measurements are described by classes that obey the

concept of a Monte Carlo measurement: they have a method `accumulate` which is called during the Monte Carlo chain and accumulates data, and a `collect_results` method that is called at the very end of the calculation. Typically the `collect_results` MPI-reduces the results from several cores in a parallelized calculation. Note that `std14::plus` in lines 34 and 35 is the `C++14` version of `std::plus`, which does not require a type, and which is provided by TRIQS for backward compatibility to `C++11`.

Listing 9 CT-INT: define the measures

```
1  //  -------------- QMC measurement ----------------
2
3  struct measure_M {
4
5   configuration const *config; // Pointer to the MC configuration
6   block_gf<imfreq> &Mw;        // reference to M-matrix
7   double beta, Z = 0;
8
9   measure_M(configuration const *config_, block_gf<imfreq> &Mw_, double beta_)
10     : config(config_), Mw(Mw_), beta(beta_) {
11    Mw() = 0;
12  }
13
14  void accumulate(double sign) {
15   Z += sign;
16
17   for (auto spin : {up, down}) {
18
19    // A lambda to measure the M-matrix in frequency
20    auto lambda = [this, spin, sign](arg_t const &x, arg_t const &y, double M) {
21     auto coeff = std::exp(-1_j * M_PI * (x.tau - y.tau) / beta);
22     auto fact = coeff * coeff;
23     for (auto const &om : this->Mw[spin].mesh()) {
24      this->Mw[spin][om](0, 0) += sign * M * coeff;
25      coeff *= fact;
26     }
27    };
28
29    foreach(config->Mmatrices[spin], lambda);
30   }
31  }
32
33  void collect_results(boost::mpi::communicator const &c) {
34   boost::mpi::all_reduce(c, Mw, Mw, std14::plus<>());
35   boost::mpi::all_reduce(c, Z, Z, std14::plus<>());
36   Mw = Mw / (-Z * beta);
37  }
38 };
```

The above components are put together in the main solver body shown in Listing 10. The first part is the constructor that only defines the dimension of the Green's functions. The second part is the `solve` method that actually runs the Monte Carlo simulation. It first constructs the Fourier transform $\tilde{G}_0(\tau)$ of the non-interacting Green's function given by the user (it is stored in `g0_iw`). Then a Monte Carlo simulation is created by adding the relevant moves and measures. This is done via the `add_move` and `add_measure` methods. Note that both the moves and the measurements are constructed with a reference to the Monte Carlo configuration `config`. The simulation is launched with `start` and final results are collected at the end of the simulation with `collect_results`. In line 50, we finally compute the actual Green's function through a compact CLEF expression that implements (A.15).

Listing 10 CT-INT: the main solver body

```
// ------------ The main class of the solver ------------------------

ctint_solver::ctint_solver(double beta_, int n_iw, int n_tau) : beta(beta_) {

 g0_iw =
     make_block_gf({"up", "down"}, gf<imfreq>{{beta, Fermion, n_iw}, {1, 1}});
 g0tilde_tau =
     make_block_gf({"up", "down"}, gf<imtime>{{beta, Fermion, n_tau}, {1, 1}});
 g0tilde_iw = g0_iw;
 g_iw = g0_iw;
 M_iw = g0_iw;
}

// The method that runs the qmc
void ctint_solver::solve(double U, double delta, int n_cycles, int length_cycle,
                         int n_warmup_cycles, std::string random_name,
                         int max_time) {

 boost::mpi::communicator world;
 triqs::clef::placeholder<0> spin_;
 triqs::clef::placeholder<1> om_;

 for (auto spin : {up, down}) { // Apply shift to g0_iw and Fourier transform
  g0tilde_iw[spin](om_) << 1.0 / (1.0 / g0_iw[spin](om_) - U / 2);
  g0tilde_tau()[spin] = triqs::gfs::inverse_fourier(g0tilde_iw[spin]);
 }

 // Rank-specific variables
 int verbosity = (world.rank() == 0 ? 3 : 0);
 int random_seed = 34788 + 928374 * world.rank();

 // Construct a Monte Carlo loop
 triqs::mc_tools::mc_generic<double> CTQMC(n_cycles, length_cycle,
                                           n_warmup_cycles, random_name,
                                           random_seed, verbosity);

 // Prepare the configuration
 auto config = configuration{g0tilde_tau, beta, delta};

 // Register moves and measurements
 CTQMC.add_move(move_insert{&config, CTQMC.rng(), beta, U}, "insertion");
 CTQMC.add_move(move_remove{&config, CTQMC.rng(), beta, U}, "removal");
 CTQMC.add_measure(measure_M{&config, M_iw, beta}, "M measurement");

 // Run and collect results
 CTQMC.start(1.0, triqs::utility::clock_callback(max_time));
 CTQMC.collect_results(world);

 // Compute the Green function from Mw
 g_iw[spin_](om_) << g0tilde_iw[spin_](om_) + g0tilde_iw[spin_](om_) *
                                              M_iw[spin_](om_) *
                                              g0tilde_iw[spin_](om_);
}
```

The listings above give a complete implementation of the CT-INT algorithm in C++: the `ctint_solver` class is ready to be used from within other C++ programs. It is however convenient to control calculations on the Python level. As discussed above, TRIQS provides a tool to easily *expose* C++ to Python. Starting from a descriptor written in Python (shown in Listing 11), it automatically generates a C wrapping code that constructs the Python modules for the `ctint_solver`. The descriptor basically lists the C++ elements that need to be exposed to Python. In most cases, this descriptor can be generated automatically by a small analysing tool provided with TRIQS. Here the script 11 has been generated automatically using this tool.

Listing 11 CT-INT: Python wrapper descriptor

```
# Generated automatically using the command :
```

```
# c++2py.py ../c++/ctint.hpp -p -m pytriqs.applications.impurity_solvers.ctint_tutorial -o ctint_tutorial
from wrap_generator import *

# The module
module = module_(full_name = "pytriqs.applications.impurity_solvers.ctint_tutorial", doc = "")

# All the triqs C++/Python modules
module.use_module('gf')

# Add here all includes beyond what is automatically included by the triqs modules
module.add_include("../c++/ctint.hpp")

# Add here anything to add in the C++ code at the start, e.g. namespace using
module.add_preamble("""
using namespace triqs::gfs;
""")

# The class ctint_solver
c = class_(
        py_type = "CtintSolver",  # name of the python class
        c_type = "ctint_solver",   # name of the C++ class
)

c.add_constructor("""(double beta_, int n_iw = 1024, int n_tau = 100001)""",
                  doc = """ """)

c.add_method("""void solve (double U, double delta, int n_cycles, int length_cycle = 50, int n_warmup_cycles = 5000, std::string random_name = "", int max_time = -1)""",
             doc = """ """)

c.add_property(name = "G0_iw",
               getter = cfunction("block_gf_view<imfreq> G0_iw ()"),
               doc = """ """)

c.add_property(name = "G0_tau",
               getter = cfunction("block_gf_view<imtime> G0_tau ()"),
               doc = """ """)

c.add_property(name = "G_iw",
               getter = cfunction("block_gf_view<imfreq> G_iw ()"),
               doc = """ """)

module.add_class(c)

module.generate_code()
```

After the code has been compiled and installed a new `Python` module is available in `pytriqs.applications.impurity_solvers.ctint_tutorial`. The solver can then be used as illustrated in Fig. 2. As this example shows, `C++` members like `g0_iw` can directly be initialised from a `Python` script and the `solve` method is also accessible. Controlling the solver, or any other `C++` code directly from `Python` makes it very easy to change parameters, plot results, build flexible control structures around it, etc., without the need to recompile the codes.

References

[1] A. Georges, G. Kotliar, W. Krauth, M. J. Rozenberg, Dynamical mean-field theory of strongly correlated fermion systems and the limit of infinite dimensions, Rev. Mod. Phys. 68 (1) (1996) 13. doi:10.1103/RevModPhys.68.13.
URL http://dx.doi.org/10.1103/RevModPhys.68.13

[2] G. Kotliar, S. Y. Savrasov, K. Haule, V. S. Oudovenko, O. Parcollet, C. A. Marianetti, Electronic structure calculations with dynamical mean-field theory, Rev. Mod. Phys. 78 (2006) 865–951. doi:10.1103/RevModPhys.78.865.
URL http://link.aps.org/doi/10.1103/RevModPhys.78.865
[3] T. Maier, M. Jarrell, T. Pruschke, M. H. Hettler, Quantum cluster theories, Rev. Mod. Phys. 77 (2005) 1027–1080. doi:10.1103/RevModPhys.77.1027.
URL http://link.aps.org/doi/10.1103/RevModPhys.77.1027
[4] A. Toschi, A. A. Katanin, K. Held, Dynamical vertex approximation: A step beyond dynamical mean-field theory, Physical Review B (Condensed Matter and Materials Physics) 75 (4) (2007) 045118. doi:10.1103/PhysRevB.75.045118.
URL http://link.aps.org/abstract/PRB/v75/e045118
[5] A. N. Rubtsov, M. I. Katsnelson, A. I. Lichtenstein, Dual fermion approach to nonlocal correlations in the hubbard model, Phys. Rev. B 77 (2008) 033101. doi:10.1103/PhysRevB.77.033101.
URL http://link.aps.org/doi/10.1103/PhysRevB.77.033101
[6] U. Schollwöck, The density-matrix renormalization group, Rev. Mod. Phys. 77 (2005) 259–315. doi:10.1103/RevModPhys.77.259.
URL http://link.aps.org/doi/10.1103/RevModPhys.77.259
[7] B. Bauer, L. D. Carr, H. G. Evertz, A. Feiguin, J. Freire, S. Fuchs, L. Gamper, J. Gukelberger, E. Gull, S. Guertler, A. Hehn, R. Igarashi, S. V. Isakov, D. Koop, P. N. Ma, P. Mates, H. Matsuo, O. Parcollet, G. Pawlowski, J. D. Picon, L. Pollet, E. Santos, V. W. Scarola, U. Schollwöck, C. Silva, B. Surer, S. Todo, S. Trebst, M. Troyer, M. L. Wall, P. Werner, S. Wessel, The alps project release 2.0: open source software for strongly correlated systems, Journal of Statistical Mechanics: Theory and Experiment 2011 (05) (2011) P05001.
[8] L. Huang, Y. Wang, Z. Y. Meng, L. Du, P. Werner, X. Dai, iqist: An open source continuous-time quantum monte carlo impurity solver toolkitarXiv:1409.7573.
URL http://arxiv.org/abs/1409.7573
[9] M. Stoudenmire, S. White. [link].
URL http://itensor.org
[10] A. N. Rubtsov, V. V. Savkin, A. I. Lichtenstein, Continuous-time quantum monte carlo method for fermions, Phys. Rev. B 72 (3) (2005) 035122. doi:10.1103/PhysRevB.72.035122.
[11] E. Gull, A. J. Millis, A. I. Lichtenstein, A. N. Rubtsov, M. Troyer, P. Werner, Continuous-time monte carlo methods for quantum impurity models, Rev. Mod. Phys. 83 (2) (2011) 349–404. doi:10.1103/RevModPhys.83.349.
[12] P. Werner, A. Comanac, L. de' Medici, et al., Continuous-time solver for quantum impurity models, Phys. Rev. Lett. 97 (7) (2006) 076405. doi:10.1103/PhysRevLett.97.076405.
[13] E. Gull, P. Werner, O. Parcollet, M. Troyer, Continuous-time auxiliary-field monte carlo for quantum impurity models, EPL (Europhysics Letters) 82 (5) (2008) 57003.
URL http://stacks.iop.org/0295-5075/82/i=5/a=57003
[14] E. G. C. P. van Loon, A. I. Lichtenstein, M. I. Katsnelson, O. Parcollet, H. Hafermann, Beyond extended dynamical mean-field theory: Dual boson approach to the two-dimensional extended hubbard model, Phys. Rev. B 90 (2014) 235135. doi:10.1103/PhysRevB.90.235135.
URL http://link.aps.org/doi/10.1103/PhysRevB.90.235135
[15] A. M. Läuchli, P. Werner, Krylov implementation of the hybridization expansion impurity solver and application to 5-orbital models, Phys. Rev. B 80 (2009) 235117. doi:10.1103/PhysRevB.80.235117.
URL http://link.aps.org/doi/10.1103/PhysRevB.80.235117
[16] F. Pérez, B. E. Granger, IPython: a system for interactive scientific computing, Computing in Science and Engineering 9 (3) (2007) 21–29. doi:10.1109/MCSE.2007.53.
URL http://ipython.org
[17] Abstracts of papers, The Annals of Mathematical Statistics 20 (4) (1949) 620–624. doi:10.1214/aoms/1177729959.
URL http://dx.doi.org/10.1214/aoms/1177729959
[18] J. Sherman, W. J. Morrison, Adjustment of an inverse matrix corresponding to a change in one element of a given matrix, The Annals of Mathematical Statistics 21 (1) (1950) 124–127. doi:10.1214/aoms/1177729893.
URL http://dx.doi.org/10.1214/aoms/1177729893
[19] C. L. Lawson, R. J. Hanson, D. R. Kincaid, F. T. Krogh, Algorithm 539: Basic Linear Algebra Subprograms for Fortran usage [F1], ACM Transactions on Mathematical Software 5 (3) (1979) 324–325.
[20] L. S. Blackford, J. Demmel, I. Duff, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet,

R. C. Whaley, An updated set of basic linear algebra subprograms (blas), ACM Trans. Math. Softw. 28 (2) (2002) 135–151. doi:10.1145/567806.567807.