`adcc`: A versatile toolkit for rapid development of algebraic-diagrammatic construction methods

Michael F. Herbst[*], Maximilian Scheurer[†], Thomas Fransson[†,‡], Dirk R. Rehn[†], Andreas Dreuw[†]

Article Type

Software Focus

Abstract

ADC-connect (`adcc`) is a hybrid `python`/`C++` module for performing excited state calculations based on the algebraic-diagrammatic construction scheme for the polarisation propagator (ADC). Key design goal is to restrict `adcc` to this single purpose and facilitate connection to external packages, e.g., for obtaining the Hartree-Fock references, plotting spectra, or modelling solvents. Interfaces to four self-consistent field codes have already been implemented, namely `pyscf`, `psi4`, `molsturm`, and `veloxchem`. The computational workflow, including the numerical solvers, are implemented in `python`, whereas the working equations and other expensive expressions are done in `C++`. This equips `adcc` with adequate speed, making it a flexible toolkit for both rapid development of ADC-based computational spectroscopy methods as well as unusual computational workflows. This is demonstrated by three examples. Presently, ADC methods up to third order in perturbation theory are available in `adcc`, including the respective core-valence separation and spin-flip variants. Both restricted or unrestricted Hartree-Fock references can be employed.

[*]CERMICS, École des Ponts ParisTech, 6 & 8 avenue Blaise Pascal, 77455 Marne-la-Vallée, France; Inria Paris, 75589 Paris Cedex 12, France; Sorbonne Université, Institut des sciences du calcul et des données, ISCD, 75005 Paris, France

[†]Interdisciplinary Center for Scientific Computing, Heidelberg University, 69120 Heidelberg, Germany

[‡]Fysikum, Stockholm University, Albanova, 10691 Stockholm, Sweden

GRAPHICAL TABLE OF CONTENTS

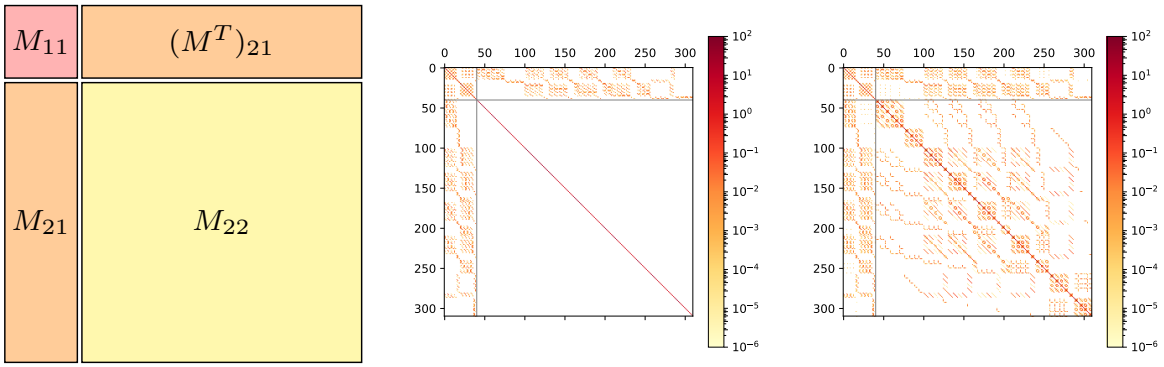
`adcc`: A versatile toolkit for research and teaching in computational spectroscopy based on the algebraic-diagrammatic construction scheme for the polarisation propagator (ADC).

INTRODUCTION

In recent years, high-level programming languages have attracted more and more attention from computational simulation frameworks. In the field of quantum chemistry, a multitude of packages have emerged, mostly employing `python` as scripting language of choice [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. The feature sets of such packages are steadily growing and can be compared to those of traditional quantum-chemical program packages. In addition to features usable in practice, `python`-driven packages have also paved the way for rapid prototyping and development of new methodologies, most notably through the `pyscf` and `psi4` programs [1, 2, 3]. Through individual components and libraries comprising these programs, a community-driven, open, and sustainable development can be guaranteed [1, 11]. The aforementioned codes are freely available and encourage code contributions from external users and developers. For improved performance, a combination of `python` with a compiled programming language, e.g., `C++`, is commonly employed. With this approach, computationally demanding routines are handed off to `C++`, while flexibility is maintained by exposing these routines to `python`. This exploits the strengths of each individual language. A more detailed discussion on this hybrid `python`/`C++` design can be found in the `psi4` and `psi4numpy` publications [1, 2]. Especially the high-level reference implementations and tutorial in `psi4numpy` demonstrate the enormous flexibility of a synergistic combination of `python` and `C++`.

A family of methods that has obtained little attention in the context of hybrid `python`/`C++` program design is the algebraic-diagrammatic construction scheme for the polarisation propa-

gator (ADC) [12, 13]. To this extent, we have developed "ADC-connect" (`adcc`), a `python`/`C++` package for carrying out ADC calculations as well as allowing for rapid development of ADC methods through a high-level `python` interface. Our package is a standalone toolkit that can be seamlessly connected to any quantum-chemical host program to perform ADC methods on top of its SCF results. Beyond the SCF, `adcc` also tries to employ as much existing software as possible for standard tasks such as tensor operations or visualisation, allowing us to focus solely on the implementation of ADC methods. In this manner, `adcc` is agnostic of the host program and other third-party codes of the `python` ecosystem. By keeping existing interfaces open, we or our users do not commit to a single software stack, but are provided full flexibility. A large set of ADC methods is already available in `adcc` and to date `psi4` [1, 2], `pyscf` [3], `molsturm` [4], and `veloxchem` [14] are fully supported as host programs.



(a) schematic representation (b) water ADC(2) matrix (c) water ADC(3) matrix

Figure 1: Structure of the ADC matrix. (a) shows a schematic representation indicating the singles block M_{11}, doubles block M_{22} and coupling block M_{21}, (b) and (c) depict the ADC(2) and ADC(3) matrix of water in an STO-3G [15] basis, respectively. The elements are coloured in a $\log_{10}$-scale.

The key equation for each ADC(n) model is the Hermitian eigenvalue problem

$$\mathbf{MX} = \mathbf{\Omega X}, \qquad \mathbf{X}^\dagger \mathbf{X} = \mathbf{I}, \tag{1}$$

where $\Omega_{nm} = \delta_{nm}\omega_n$ is the diagonal matrix of excitation energies and $\mathbf{M}$ is the so-called ADC matrix. The matrix $\mathbf{M}$ exhibits a block structure, shown in Figure 1a, where individual blocks are treated at different orders of perturbation theory. On top of this block

structure the individual blocks are sparse (Figure 1b and c) which is a direct consequence of the selection rules obtained from spin and permutational symmetry in the tensor contractions required for computing $\mathbf{M}$. Exploiting this sparsity when diagonalising the matrix (1) is a key step to make ADC tractable for relevant systems. In this regard, `adcc` follows the conventional approach [13, 16] to use contraction-based [4], iterative eigensolvers, such as the Jacobi-Davidson [17]. All tensor operations in the required ADC matrix-vector products are performed on block-sparse tensors. In this setting, the computational scaling of ADC(2) is given as $O(N^5)$ where N is the number of orbitals, whereas ADC(2)-x and ADC(3) scale as $O(N^6)$ [13]. These computationally demanding procedures are implemented in `C++` through the parallelised `libtensor` library [18] and made available to the `python` layer. In this manner, `adcc` achieves comparable performance as a `C++`-only implementation [16]. Still, all data is readily accessible from `python`, e.g., to perform post-processing in `numpy` [19] or `matplotlib` [20]. Consequently, our design enables a very flexible computational workflow and interactive usage in Jupyter notebooks or IPython shells [21, 22]. This will be demonstrated by several examples. Additionally, exposing linear algebra operations of `C++` tensor objects to the `python` layer made it possible to implement numerical procedures, e.g., iterative solver schemes completely in `python`. As such, `adcc` provides all the necessary building blocks for development of complex workflows or novel approaches of ADC that can be arbitrarily assembled and extended on the `python` layer. Together with the comprehensive documentation available online (`https://adc-connect.org`), the barrier between "users" and "developers" of `adcc` is minimal. `adcc` is freely available on `https://github.com/adc-connect/adcc`. All figures and tables of the paper can be reproduced using the scripts and details of the supporting information (SI-4).

The remainder of the paper is structured as follows: The next Section discusses design and structure of `adcc` with emphasis on the computational workflow and the integration with external packages. Examples how `adcc` can be used in practice are provided thereafter, followed by a short review of the currently supported feature set of `adcc`.

DESIGN AND KEY COMPONENTS

Design goals

With `adcc`, we aimed for a flexible library for rapid ADC method development that seamlessly integrates into the existing `python` ecosystem. Consequently, secondary goals for the design philosophy of the code arise which are briefly outlined in the following.

1. *Build on established software.* Instead of designing a complete quantum-chemical program, `adcc` focuses only on ADC methods and their numerical procedures. For most other aspects, `adcc` relies on established third-party packages, which allows to reuse many years of development effort and bug fixing. This makes `adcc` a light-weight package to be easily integrated in more complex computational workflows.

2. *Open interfaces for reproducible and sustainable science.* The SCF interface of `adcc` is deliberately kept simple for easiest connectability to an existing SCF codes. This has two important consequences: (1) It makes `adcc` sustainable and maintainable, because in case an SCF code became unavailable in the future, a replacement could be easily integrated. (2) Results can be verified and reproduced across different SCF implementations, reducing the chance of building on top of wrongful assumptions and introducing accidental dependencies.

3. *Good compromise between performance and code complexity.* Given the asymptotic scaling up to $O(N^6)$ of ADC methods, performance aspects cannot be ignored in a practically useful implementation. The total time required to achieve a scientific outcome in computational sciences, however, clearly goes beyond just the runtime of simulations. Much rather, the time needed to install, setup, and familiarise oneself with a framework also matters. In case additional features are to be added to the code, the time effort to understand the code base, to implement, debug, and test also plays a pivotal role. Unfortunately, achieving peak performance typically has a negative impact on readability and usability, such that a balance needs to be found. In `adcc` we have thus chosen a design, where the workflow and the numerical schemes are completely

controlled from high-level `python` code with only selected computationally demanding parts implemented in `C++`. This allows `adcc` users both to quickly experiment with workflows or numerical routines and to treat problems of practical relevance in a single toolkit.

4. *Low barriers between users and developers.* With our toolkit we not only want to allow for use of provided, predefined functionality, which already exists inside `adcc`. By providing detailed documentation and readable code, we want to encourage users to become active developers, e.g., by extending our workflows, beyond anything we as the `adcc` developers could ever imagine. It is our hope that such an open platform will lead to a community-driven improvement of `adcc` in particular and computational spectroscopy in general.

Demand-driven computation in `adcc`

The general flow and main computational tasks of an ADC calculation with `adcc` is shown in Figure 2 with classes or functions from `adcc` given in `teletype`. The term "host program" refers to the (third-party) program environment from which the ADC calculation has been started. In practice, this is the code that yields the Hartree-Fock SCF ground state and provides access to an integral library for obtaining the antisymmetrised electron-repulsion integral (ERI) tensor or operator integrals (e.g., electric dipole operator integrals). Steps 2–4 are setting up the necessary scaffold to import host-program-specific data into `adcc`. Afterwards, the main bookkeeping classes of `adcc` are constructed in steps 5–7, i.e., MP quantities, recurring parts of the ADC working equations (intermediates) [23], and a *lazy matrix* [4, 24] representation of the ADC matrix. Of note, this `AdcMatrix` class collects *all* MP results, intermediates, and expressions for the working equations of a particular ADC method. The ADC eigenvalue problem is then solved in step 8, and the results are wrapped by an `ExcitedStates` object, see the next Section for details.

With the workflow schematic of `adcc` in place, we will now explain how `adcc` efficiently implements this complex computational procedure. Here, the main challenge is that individual computational steps can explicitly or implicitly become mutually dependent. This not

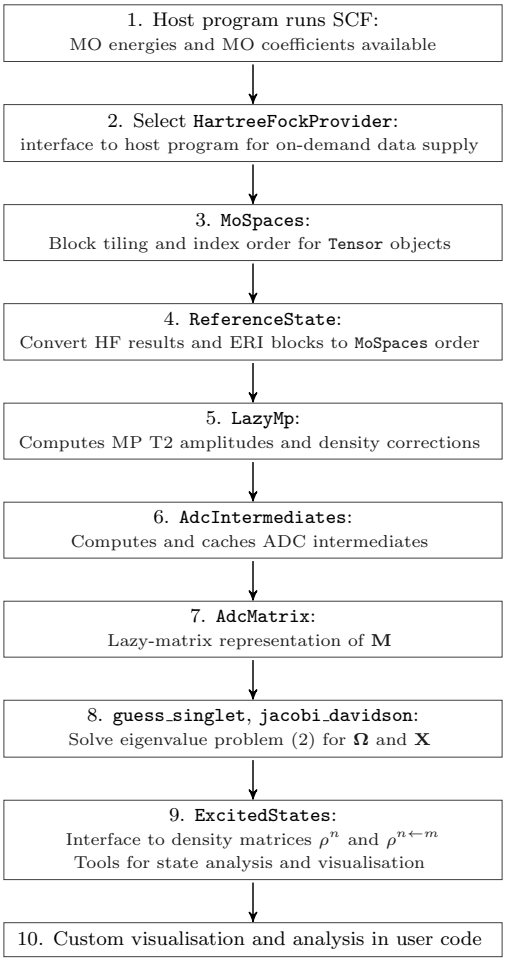


Figure 2: Schematic of an ADC calculation using `adcc`. Steps 2 to 9 take place inside `adcc`, whereas steps 1 and 10 are executed in the host program or user code. Functions or classes from the `adcc` library are marked with `teletype` font.

only applies to ADC, but also to related methodologies with similar computational stages. The ADC intermediates for example (step 6 in Figure 2) require the MP(2) T2 amplitudes. Therefore, step 6 obviously depends on step 5. In turn, a naive sequential implementation requires step 5 to know that the T2 amplitudes will be needed in step 6 in order to decide whether these quantities are to be computed or not.

One way to resolve this circular dependency is to compute every quantity at every step, regardless of whether it is used later or not. This approach, however, is rather inefficient. Another option is to inspect the computational parameters at the beginning of the `adcc` run and schedule computations for each step. While this "schedule" approach works reasonably

well for simple workflows, it inevitably leads to a combinatorial explosion in the required bookkeeping logic if the number of parameters increases. Already at the current stage, `adcc` has about 40 different code execution paths through the workflow in Figure 2 with varying amounts of work required at each step.

For this reason we have taken a different approach: Upon initialisation of data structures, such as `ReferenceState`, `LazyMp`, or `AdcIntermediates`, these objects are empty. For example, constructing `LazyMp` does not lead to the computation of the T2 amplitudes at this very instance. Only once the T2 amplitudes are needed for the first time, e.g., by computation of the ADC(2) intermediates, they are computed and cached in memory. Thus, the first demand for a specific quantity drives its computation. For this reason we have termed this strategy "demand-driven". This idea is heavily inspired from a concept called *lazy evaluation* in programming language theory [25], where any expression in the source code is only ever evaluated once its outcome is needed. In `adcc`, the caching check is typically an `if`-statement directly enwrapping the computation. This has two important advantages. Firstly, it makes the code responsible for computing one quantity self-contained, which lowers the code complexity. Secondly, it prevents schedule logic and computational algorithm to get out of sync. Thus, a major advantage of the demand-driven computation is that one simply cannot forget to request the computation of a particular quantity in a previous step or forget to remove this request. This decouples code entities and makes `adcc` automatically choose the path of least computational load.

To illustrate the demand-driven workflow in practice, Figure 3 shows the memory and time profile for a computation of ten singlet excited states of noradrenaline at the ADC(2)/6-31++G** level [26]. The time spent in selected classes and functions of `adcc` is indicated by the alternating yellow-green background, and the memory size of the executing `python` process as detected by the operating system is plotted in grey. Of note, the memory profile not only depends on the behaviour of `adcc`, but also on the memory management of the `python` interpreter and the C library (in this case `glibc`). The time axis is split into two segments with the first resolving the first six minutes of the calculation to higher detail. The lower part of Figure 3 shows two bars of orange and blue blocks, which highlight, respectively, the time spent inside `AdcMatrix` for computing the ADC working equations and the time

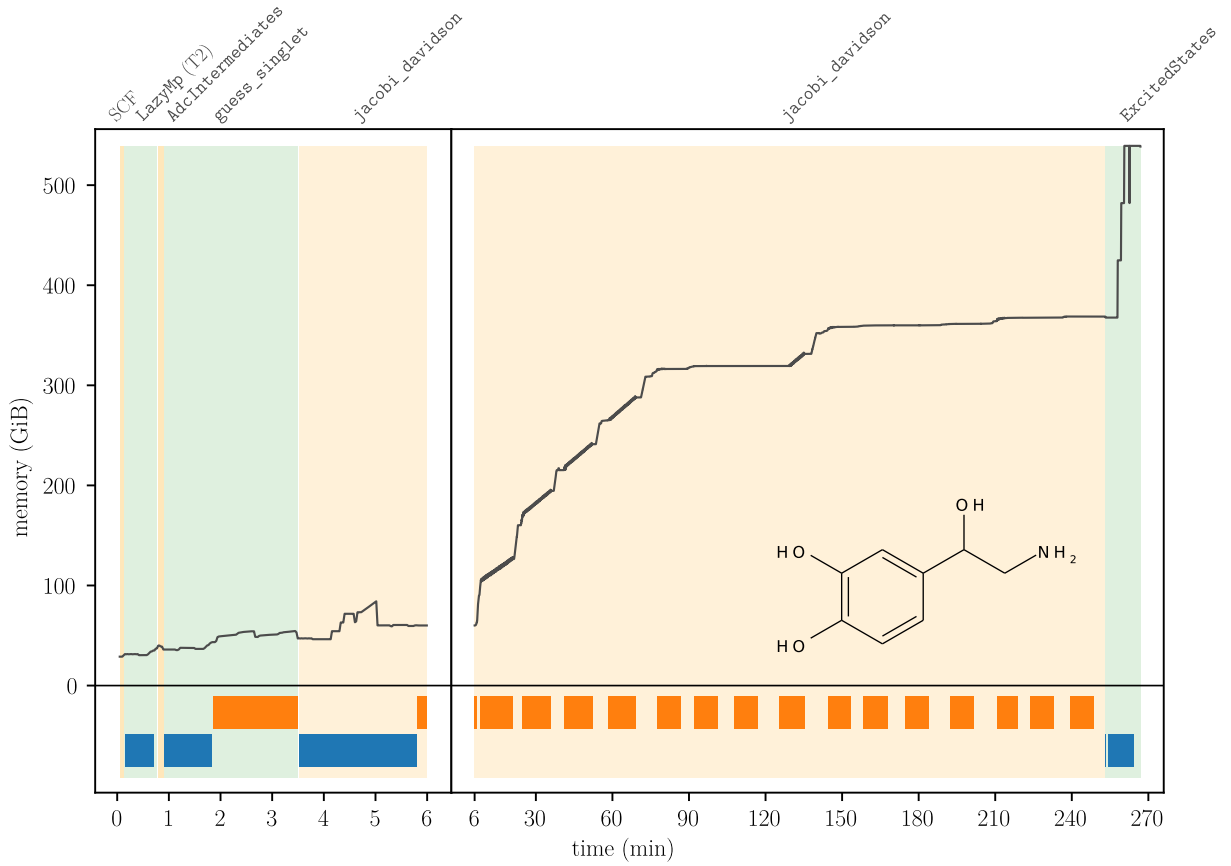


Figure 3: Memory and time profile of an ADC(2) calculation for the lowest 10 singlet excitations of noradrenaline in a 6-311++G** basis [26]. The green and yellow background indicate the time spent in functions or classes of `adcc`, which are labelled in `teletype` font. The orange and blue bars at the bottom indicate the activity of the `AdcMatrix` class and the time spent importing the electron-repulsion integrals, respectively. Details about the computational hardware can be found in the supporting information, Section SI-3.

needed for importing the ERI tensor.

One notices the ERI tensor not to be imported at once, but at four distinct times in the profile. Each time interval corresponds to computing one block of the ERI tensor in the host program via an AO-to-MO transformation followed by importing it into the `ReferenceState`. Through the demand-driven design of `adcc`, one can easily follow the order of the block import, summarised in Table 1. Note that the objects requiring import of a new ERI block often also depend on previously imported blocks that are already cached.

Table 1: Breakdown of the demand-driven import of ERI blocks for the noradrenaline calculation of Figure 3. The first demand of a block is indicated by the operations of the rightmost column. o refers to occupied orbitals and v to virtual orbitals. Multiple blocks in one row are imported sequentially.

time (min)	ERI block	←	first demand		
0	$\langle oo\|\|vv\rangle$	←	`LazyMp` T2	←	`AdcIntermediates`
1	$\langle ov\|\|ov\rangle$	←	`AdcMatrix.diagonal`	←	`guess_singlet`
3.5	$\langle ov\|\|vv\rangle$, $\langle oo\|\|ov\rangle$	←	`AdcMatrix.matvec`	←	`jacobi_davidson`
250	$\langle oo\|\|oo\rangle$, $\langle vv\|\|vv\rangle$	←	transition densities	←	`ExcitedStates`

The complete chain of imports and computations in Table 1 has actually been triggered by requesting only three things directly from `adcc`, namely (1) the ADC(2) guesses via `guess_singlet`, (2) the converged ADC(2) excitation vectors from `jacobi_davidson` and (3) the computation of the oscillator strengths for these states via `ExcitedStates`. The other computations including the import of the ERI tensor blocks were implicitly driven by this initial demand. If this demand is modified, e.g., by performing a computation employing the CVS, frozen-core (FC) or frozen-virtual (FV) approximations, the ERI blocks not required by the respective approximations, will never be computed in the host program during the ADC calculation. Still, dropped blocks can be requested via the `ReferenceState` in user code, meaning that any additional demand on top of the ADC calculation will be satisfied as well. This is a great advantage during debugging and in order to extend features of `adcc` in user code.

Typically the import of an ERI tensor block leads to an increased memory usage, since tensor data is generated with the AO-to-MO transformation in the host program. As discussed, in `adcc` the ERI import is automatically delayed for as long as possible. This implies that the allocation of the ERI tensor memory is delayed as well. As a result, the peak memory usage for the noradrenaline calculation in Figure 3 is only obtained at the very end of the calculation, namely during the property calculation, once the $\langle vv||vv\rangle$ block has been imported. This memory profile implies that `adcc` will run out of memory as late as possible.

In other words it will still finish intermediate work, from which the calculation could be restarted on a node with more RAM. Note that at the moment `adcc` does not yet implement any form of checkpointing, however.

Structure and mix of programming languages in `adcc`

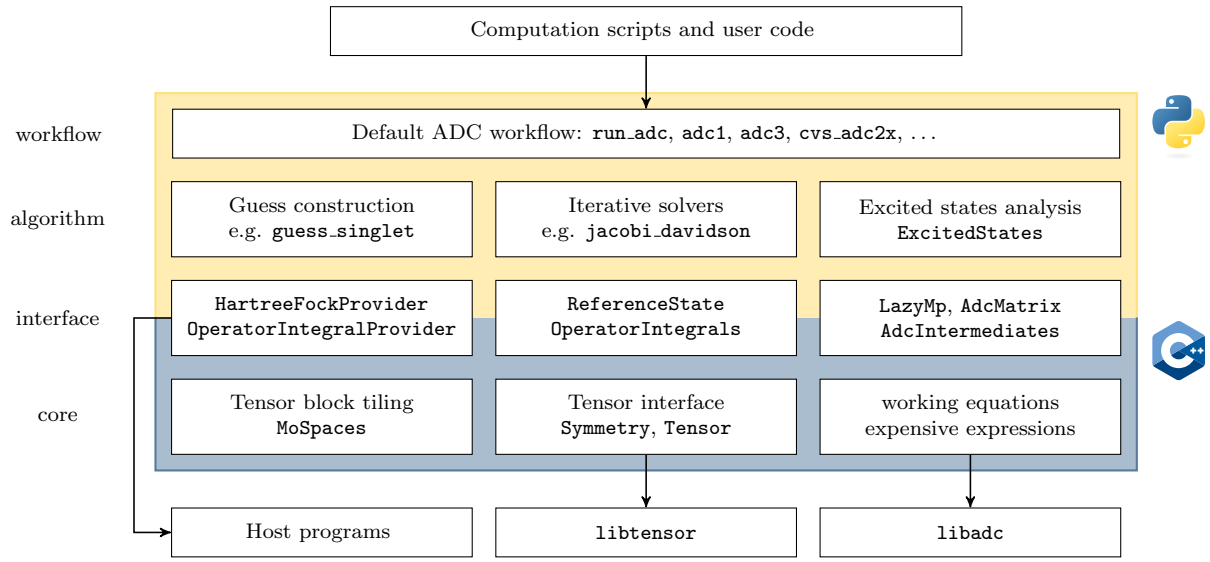


Figure 4: Code structure of `adcc` and interfaces to user scripts, external host programs or the `C++` libraries `libtensor` [18] and `libadc` [16]. Shown inside the yellow-blue box are the four layers of `adcc` with key classes and functions marked in `Teletype` font. The background of the box indicates the predominant programming language employed in the respective layer, yellow for `python` and blue for `C++`.

With the `adcc` workflow and the concept of demand-driven computations in place, we will now explain the actual structure and building blocks of `adcc` in detail, together with the hybrid `python-C++` programming approach.

Figure 4 shows the code structure of `adcc` along with interfaces to external codes or user scripts as indicated by arrows leaving or entering the `adcc` box. Inside the box, the colours indicate the predominant programming language used to provide the respective features. Going from top to bottom, the first layer is the *workflow layer*. It contains a default ADC workflow for setting up the `AdcMatrix` and subsequently calling the functions of the *algorithm*

layer to solve the ADC eigenvalue problem (1). Main ingredients of this second layer are the guess functions, the `python`-based implementations of the iterative solver algorithms and the `ExcitedStates` class to analyse the obtained results. Next, the *interface layer* contains the datastructures responsible for the ADC working equations and their requirements, including the `HartreeFockProvider` and `OperatorIntegralProvider` interface classes for each host program. As discussed, most computation in `adcc` happens in this layer, demanding the computation or import of prerequisites as needed. For performing these tasks, the interface layer accesses helper functionality from the bottommost part of `adcc`, the *core layer*. This layer contains the `MoSpaces` class for defining the tensor tilings as well as the `Tensor` and `Symmetry` classes, which expose slightly amended primitive tensor operations from the `libtensor` [18] block-sparse tensor library to `adcc`. It also contains the interface code to the `libadc` library, which implements the ADC working equations and other expensive expressions in `libtensor` syntax. Both these libraries are written in `C++`, such that implementing the remainder of the core layer in `C++` turned out to be most practical. The `libadc` library has been previously distributed as part of the `adcman` [16] module of the Q-Chem quantum chemistry package [27]. As part of the interface to `libadc`, the core layer further makes use of the hierarchical storage datastructures of the ctx library [28] for organising the data required for evaluating the ADC equations.

On the contrary, the interface layer, as indicated by the colour-coding in Figure 4, is partly written in `python` and partly in `C++`. The divide between these languages is not at all clear-cut: Much rather some functions and datastructures are implemented *using both languages*. Using Pybind11 [29] the respective `C++` functionality is exposed to `python`, where it can be accessed and extended without duplicating code. In this way, the interface to performance-critical portions of `adcc` in the core layer is implemented in `C++` and the remaining aspects may fully exploit the language features `python` has to offer. One example where this approach turned out to be very useful are the `HartreeFockProvider` interfaces, which on the one hand need to interact with a third-party host program, whose interfaces we as `adcc` developers cannot control, on the other hand it still needs to provide the generated data to the `libtensor` library. For the former aspect, a dynamic language like `python` is extremely handy, while for the latter, low-level memory access is required and thus a

language like C++ is much more suitable.

The algorithm layer itself is also an example for python code, which extends the C++ core. By proper Pybind11-wrapping of key classes, such as the MoSpaces, Tensor, and Symmetry classes, the algorithm layer can directly configure and access *all* the functionality of the core layer albeit being written in python. This includes raw tensor operations such as addition, multiplication or tensor contraction. Provided that the tensor tiling in MoSpaces has been setup appropriately, these operations automatically take symmetry into account and are parallelised. In this way all relevant numerical schemes of ADC, such as the iterative diagonalisation inside the jacobi_davidson, can be implemented purely in python. Certainly, a multitude of calls between python and C++ are required which are associated with a runtime overhead. Since the time spent for computing tensor operations in libadc, libtensor, and the core layer dominates the overall runtime, such that call time overhead as well as the general performance penalty of python are completely negligible. Compared to a C++-only implementation of the algorithm, however, flexibility is gained through python. This enables more involved or highly problem-specific numerical schemes, which can be developed, tested, and implemented with much reduced effort.

Keeping the implementation of iterative solvers and guesses aside, the algorithm layer also contains the ExcitedStates class. This class is returned to the user once the ADC calculation in adcc has finished and holds key results of the calculation such as the excitation vectors or the excitation energies. From these, state densities ρ^n, transition densities $\rho^{n \leftarrow m}$, and other properties can be computed on demand. The returned quantities are in fact Tensor objects, allowing the user to post-process them directly using the tensor operations of libtensor. Alternatively, these objects may be converted to and from dense numpy arrays to allow full integration with the usual numpy/scipy ecosystem [19, 30]. More details about customisable post-processing can be found in the example Section and on the adcc website. Some conventional analysis and visualisation techniques, such as the plotting of an empirically broadened excitation spectra are directly available by calling the plot_spectrum function of the ExcitedStates. These functions integrate well with existing python infrastructure, in this case matplotlib [20]. An example for the spectra plotting of ExcitedStates can be found in the examples. Such tools to quickly visualise results are key

for reducing the feedback loop when working with `adcc` from, e.g., a Jupyter notebook [22] or for educational purposes.

A user who is new to `adcc` will start to interact with the library mostly via the workflow layer, calling the predefined ADC procedure it exposes via the `run_adc python` function. More method-specific functions with slightly varying presets are available for individual ADC methods, for example `adc1`, `cvs_adc3`, ... These functions provide only limited capability for customisation, passing parameters such as the subspace size or the structure of the core space (for core-valence separation) to the rest of the library. For more advanced use cases, like the ones mentioned above, they may then leverage the `ExcitedStates` class or construct deviating ADC workflows building on the `python` primitives of the algorithm and interface layer. Simulation procedures resulting from this process *are* already developments (in `python` code), which could potentially be integrated back into the library in the future. In this way the original user has smoothly become a developer of `adcc`. Even for yet deeper modifications one may stay in high-level `python` code, being able to tinker with advanced aspects such as the setup of the ADC guesses or the numerical procedure to diagonalise the ADC matrix. Overall, it is our hope that the sketched structure of `adcc` allows to motivate users to become developers as well.

Related to the aspect of obtaining a sustainable base of users and developers is the question of a sustainable software stack. For this it is important to (1) build on top of software, which is already established and thus unlikely to disappear and (2) to stay sufficiently flexible to be able to swap components if this may still become necessary. We achieved this by building on top of two actively developed `C++` libraries, `libtensor` and `libadc` [18, 16], and abstracting from these libraries explicitly through the `Tensor` and `Symmetry` classes. In this way, clearly defined entry points from `adcc` to these third-party codes are defined, such that other tensor libraries could be supported in the future as well. Similarly, with respect to the host programs for supplying integral and SCF data, `adcc` supports multiple SCF codes out of the box. As will be explained in more details in the next Section, the SCF interface expected by the `HartreeFockProvider` has been designed to be easily fulfilled, such that support for further host programs can be added with ease.

HF interface and ERI data import

To start an ADC calculation, `adcc` requires two kinds of data from the host program. Firstly, the obtained SCF results, such as the molecular orbital energies, coefficients and occupation vector as well as the Fock matrix and the (antisymmetrised) ERI tensor. Secondly, metadata such as the SCF convergence threshold and whether or not a restricted SCF procedure was employed. Only optionally, if the computation of properties such as dipole moments and oscillator strengths is desired, `adcc` requires further the total molecular charge, the nuclear dipole moment and the electric dipole integrals in the atomic orbital basis. We will only review key design aspects about the interface between `adcc` and host programs in this section, the complete documentation can be found in the supporting information SI-1 and SI-2 or online under https://adc-connect.org/q/hostprograms.

At the moment data can be supplied to `adcc` in three ways. The most straightforward implementation only requires one to prepare a `python` dictionary, which supplies the above quantities as either primitive `python` data types or as `numpy` arrays. The antisymmetrisation of the ERI tensor may either be performed in `adcc` or the antisymmetrised tensor may be supplied directly. This dictionary is then passed to `run_adc` or any other method from the workflow layer to start an ADC calculation. For this type of interface the key focus was simplicity rather than efficiency. Therefore, `adcc` makes no attempt to exploit any kind of symmetries for the Fock matrix or the ERI tensor. Even for restricted SCF results, all four spin blocks of the Fock matrix need to be passed, including the $\alpha\beta$ and $\beta\alpha$ blocks, which are always zero. Naturally this leads to a huge memory overhead and thus allows this interface only to be used for small test calculations. The second option is a variant of the dictionary-based interface, where the data is supplied from an HDF5 file [31]. This version integrates more closely into the demand-driven workflow, such that `adcc` will only read those parts of the tensors from disk, which are needed.

The best performance for the data import, however, is achieved through an implementation of a `HartreeFockProvider` and a corresponding `OperatatorIntegralProvider` specific to the host program. As the names suggest, the former is responsible for all HF-related quantities and the latter for integrals such as the electric dipole operator integrals. Both classes

allow to integrate host-program-specific routines, e.g., for the AO-to-MO transformation, into the demand-driven workflow of `adcc`. This requires the definition of about 20 functions, of which most are trivial and only require the user to return plain data.

The integration with the import of the ERI tensor, however, is more involved and the design will be briefly discussed. Once the first demand towards an ERI tensor block triggers the import process inside the `ReferenceState` class, the tensor tiling and tensor symmetry is already known and can be used to deduce the minimally required subset of tensor elements to fully represent the complete block in question. A block-sparse tensor library such as `libtensor` only holds these elements in memory and consequently only these need to be filled with data by the host program. In the import code one thus only loops over symmetry-reduced chunks of the ERI tensor and requests the respective tensor data from the `HartreeFockProvider`. This is done by passing it an index range, some details about the memory alignment and a data pointer with the memory location to place the generated ERI elements into. If this memory pointer can be directly employed as output memory inside the AO-to-MO routine of the host program, the ERI tensor import operates without a single copy. The import is implemented inside the `C++` part of `ReferenceState` as well as the core layer of `adcc`. On this low level a link between `adcc` and a host program can be achieved directly by inheriting from a `C++` abstract base class. One may also implement a `HartreeFockProvider` in `python`, since we employ Pybind11 [29] to allow `python` classes to overwrite the `C++` base class. At this level `adcc` also uses `numpy` arrays to conveniently hide the details of the memory pointer without an additional copy. An implementation of a `HartreeFockProvider` from `python` thus has the full flexibility of `numpy` and `python` to interact between host program and `adcc`, but at no additional expense.

Since all three interface approaches are `python`-based, they can be mixed. This is useful for adding initial support of a new host program, since one can start with a rapid prototype using the dictionary-based approach. Based on this the functions of the `HartreeFockProvider` can be implemented step by step, verifying correctness along the way.

EXAMPLES

Comparison of ADC methods

```
8  mol = psi4.geometry("""
9      units au; symmetry c1
10     0 0 0 0; H 0 0 1.795239827225189; H 1.693194615993441 0 -0.599043184453037
11 """.replace(";", "\n"))
12 psi4.core.be_quiet()
13 psi4.set_options({'basis': "cc-pvtz", 'e_convergence': 1e-7})
14 _, wfn = psi4.energy('SCF', return_wfn=True)
15
16 # Run ADC(1) on 7 singlets
17 n_singlets = 7
18 adc_1 = adcc.adc1(wfn, n_singlets=n_singlets)
19
20 # Build guess for ADC(2) by appending a zero doubles part to each vector
21 adc2matrix = adcc.AdcMatrix("adc2", adc_1.ground_state)
22 def append_doubles_block(adc1_vector):
23     symmetries = adcc.guess_symmetries(adc2matrix,
24                                        spin_block_symmetrisation="symmetric")
25     return adcc.AmplitudeVector(adc1_vector["s"], adcc.Tensor(symmetries[1]))
26 guesses_1 = [append_doubles_block(exv) for exv in adc_1.excitation_vectors]
27
28 # Run ADC(2), ADC(2x) and ADC(3)
29 adc_2 =  adcc.adc2(adc2matrix, n_singlets=n_singlets, guesses=guesses_1)
30 adc_x = adcc.adc2x(adc_2.ground_state, n_singlets=n_singlets,
31                    guesses=adc_2.excitation_vectors)
32 adc_3 =  adcc.adc3(adc_x.ground_state, n_singlets=n_singlets,
33                    guesses=adc_x.excitation_vectors)
34
35 adc_1.plot_spectrum(label="ADC(1)")
36 adc_2.plot_spectrum(label="ADC(2)")
37 adc_x.plot_spectrum(label="ADC(2)-x")
38 adc_3.plot_spectrum(label="ADC(3)")
39 plt.legend()
40 plt.show()
```

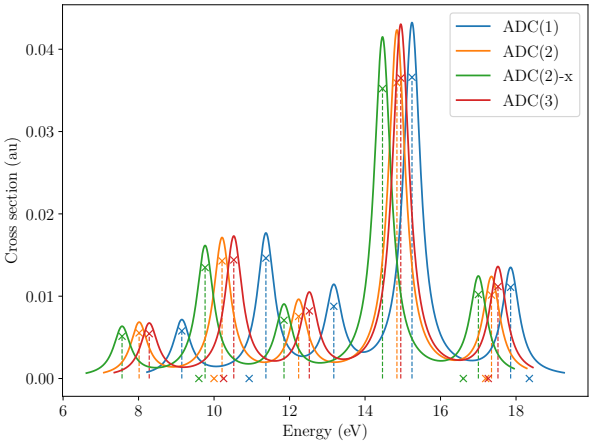


Figure 5: `python` script computing the seven lowest-energy singlet excited states of water in a cc-pVTZ basis [32] at ADC(1), ADC(2), ADC(2)-x and ADC(3) level and resulting excitation spectrum. The procedure uses the respective lower level of theory as a guess for the next computation. Spectra are broadened empirically with a Lorentzian with width parameter $\gamma = 0.01$ atomic units and shown in the same colour as the computed excitation energies and cross sections, which are marked by a cross.

A frequent task in benchmarking is to compare the result of multiple levels of theory on the same system. Using `adcc`, different ADC methods can be used upon the same system with concise `python` code, shown in Figure 5 alongside the resulting spectra. In lines 8 to 14, the script prepares a restricted Hartree-Fock reference of water in `psi4` using a cc-pVTZ basis [32]. On top of this the `adc1`, `adc2`, `adc2x` and `adc3` functions of `adcc` perform the respective ADC method on top. In each case, the `excitation_vectors` of the `ExcitedStates` object of the lower level of theory are used as guesses, which for the case of employing ADC(1) results in ADC(2) requires to append a zero doubles part of the appropriate singlet symmetry in lines 22 to 26. For starting the ADC calculation only the

first invocation in line 18 makes reference to the `wfn` object containing the HF reference. All other calculations start directly from a `LazyMp` `ground_state` or even an `AdcMatrix`, which allows to share and re-use previously computed quantities, such as the T2 amplitudes. In lines 35 to 40, the obtained excited states are broadened with a Lorentzian (width parameter $\gamma = 0.01$ atomic units) and plotted. For this the `plot_spectrum` function of `adcc` integrates with `matplotlib` placing the spectrum directly on a `matplotlib` figure. In this way the plot can be extended via the usual `matplotlib` functionality. In this example, we add a legend with `plt.legend()` and display the image shown on the right of Figure 5 with `plt.show()`.

The complete script with 40 lines including the code needed for the visualisation of the excited states spectra is very tractable and most lines of code are completely self-explanatory. Obtaining key quantities such as the state densities as `numpy` arrays allows to easily extend the analysis and, e.g., visualise density differences in `matplotlib`. Such direct access to key quantities greatly decreases the feedback loop between calculation and insight. Furthermore, the brevity of the code implies that it can be written spontaneously in an interactive IPython [21] shell or a Jupyter notebook [22] during a scientific discussion or a lecture. This provides a powerful hands-on technique for rapid ADC method development, debugging or interactive teaching sessions directly in the web browser.

Flexible selection of frozen MOs and CVS spaces

For the calculation of core-excited state, some flavour of the core-valence separation (CVS) approach is routinely employed to avoid the difficulty of considering states buried in a continuum of valence transitions [33, 34, 35, 36, 37]. Previous ADC implementation have, however, used a construction of the CVS space where all (core) MOs up to the last probed one is included in the CVS-ADC matrix. This leads to matrix dimensions which are larger than necessary, as we will now demonstrate using the `adcc` implementation, which enables the use of any CVS spaces, as well as the freezing of arbitrary occupied or unoccupied MOs. We consider the carbon X-ray absorption spectrum of 1,1-difluoroethene, with results illustrated in Fig. 6. This system has been investigated experimentally [38] and in theory [39], and possess significant shifts in transition energy due to the substitution of electronegative fluo-

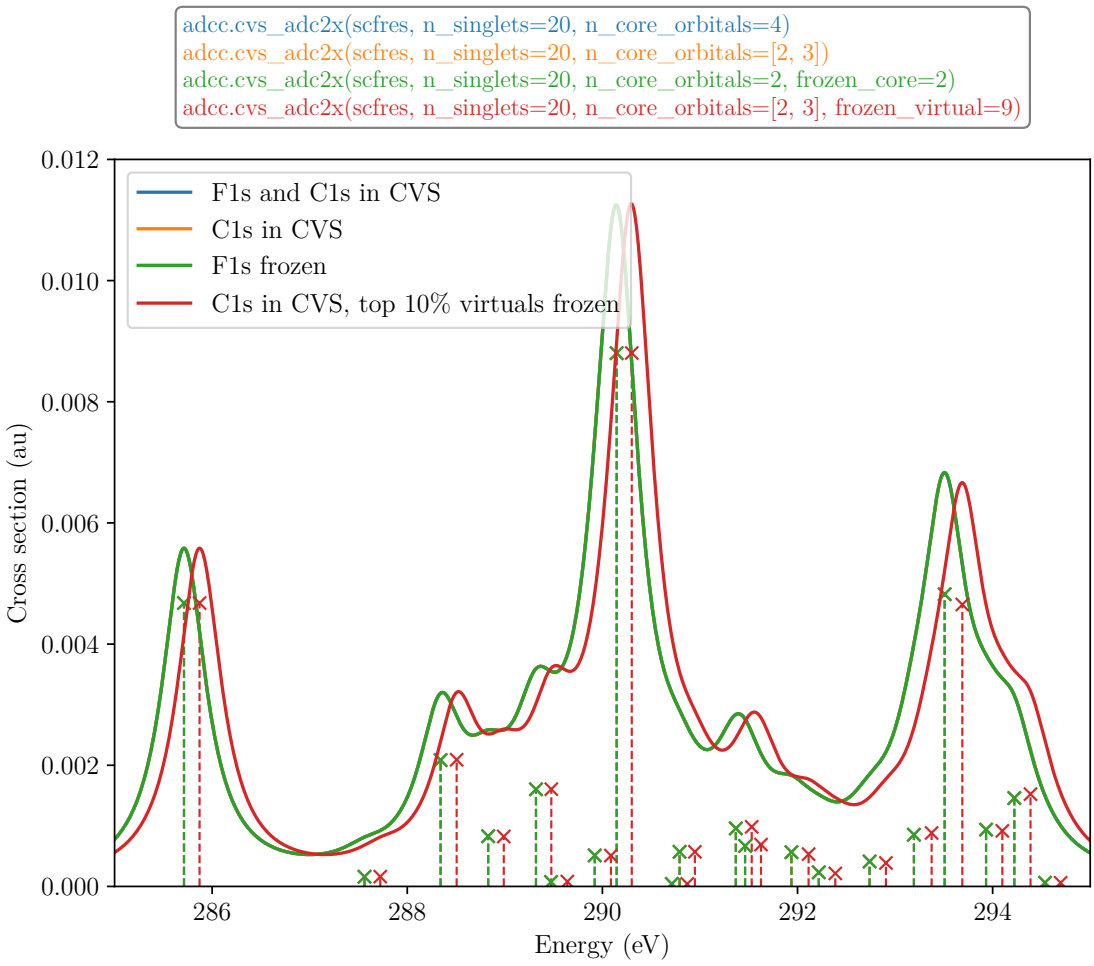


Figure 6: Carbon K-edge X-ray absorption spectrum of 1,1-difluoroethene, using different subspaces in CVS as well as by freezing core and/or virtual orbitals. Commands for calculating these spectra are shown above the Figure.

rine. The spectra have been calculated for a MP(2)/cc-pVTZ [32] optimised structure, [27] with excited states calculated using CVS-ADC(2)-x/6-311++G** [26]. This combination of ADC level and basis set has been noted to provide results in close agreement to experiment [35, 40]. Our results overestimates experimental measurement by $0.3 - 0.6$ eV [38], when scalar relativistic effects are accounted for.

We note that including only the two carbon core orbitals in the CVS space leads to identical results as including also the fluorines, at a lower computational cost. Alternatively, it is possible to freeze the fluorine core MOs and then select the lowest two MOs in the CVS space, which then again lead to identical results at a lower cost than including also the

fluorines. The fluorine core MOs can be left outside the CVS space or frozen with identical results as they are spatially and energetically separated from the carbon core electrons, but this would not be the case if, e.g., the L-edge of heavier atoms are considered. Finally, we illustrate the use of a carbon-specific CVS space together with freezing the 10% highest virtuals (here, 9 virtuals), which leads to an increase in transition energy of 0.16 ± 0.01 eV due to some lack of relaxation. Such a freezing of virtuals, as well as choosing CVS spaces focused on a single, chemically unique core orbital, can be employed to obtain additional lowering of computational cost. Care must be taken if any of these approaches are applied.

Solvent shift of nile red

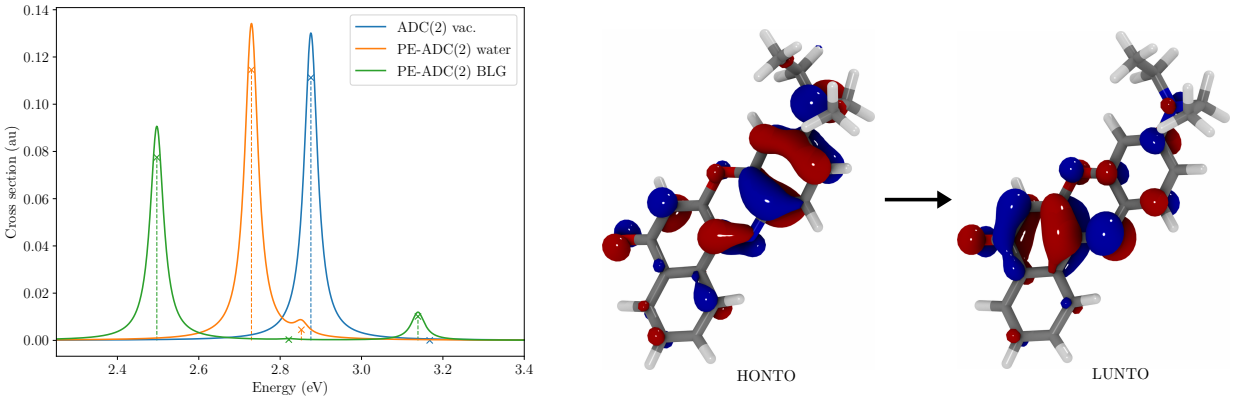


Figure 7: Absorption spectrum (left) with three singlet excited states of nile red in vacuum, water, and BLG. Highest occupied NTO (HONTO) and lowest unoccupied NTO (LUNTO) for the first singlet excited state in water (right). Spectra are broadened empirically with a Lorentzian ($\gamma = 0.02$ eV).

To illustrate the capabilities of `adcc` in a biomolecular application, the solvent shift of nile red in water and protein environment is modelled using the polarisable embedding (PE) model in combination with ADC (PE-ADC) [41]. Geometries of the chromophore and parameters for the water and the β-lactoglobulin (BLG) protein environments were used from previous work [42]. Using `pyscf` as host program, three singlet excited states were computed at the ADC(2)/cc-pVDZ [32] level of theory using `adcc`. Calculations included 420 basis

functions and ran for 8 hours on 32 cores on a single node (see supporting information for hardware details). The PE-HF ground state calculations in `pyscf` employed `cppe` [42], which is also a modular `python` library. Perturbative corrections of the excitation energies [41] were computed directly in the `python` job scripts. The resulting spectra for nile red in vacuum, water, and BLG were obtained by convolution with a Lorentzian function (γ = 0.02 eV). Natural transition orbitals (NTOs) were also generated in the `python` job script by decomposing the AO transition density matrix with `numpy` (`numpy.linalg.svd`) and subsequently writing the orbitals to disk with the `cubegen` utility in `pyscf`. Finally, cube files of NTOs were rendered with VMD [43].

The resulting excitation spectra and NTOs of the first excited states of nile red embedded in water are depicted in Figure 7. The NTOs clearly show the $\pi\pi^*$ nature of the lowest transition. Consequently, the absorption cross section is high, and the transition is strongly red-shifted in the embedded systems compared to vacuum. Rather than discussing the properties of nile red, which has been done in previous work [42], the given example shows that `adcc` is capable of tackling systems of medium size with good performance. As explained before, full flexibility is still granted, here by a) computing perturbative corrections to the ADC excitation energies and b) generating NTOs in user code with negligible programming effort. In the same manner, other solvent models available through the supported host programs could be combined with `adcc` as well, e.g., continuum solvation models. Furthermore, users could expand their scripts with more advanced wave function analysis [44], making the analysis more interactive and tailored to the problem at hand.

CURRENT STATE AND SUPPORTED FEATURES

Presently the `adcc` code base allows to model electronically excited states using various levels of the algebraic-diagrammatic construction scheme for the polarisation propagator (ADC). This includes ADC(2), ADC(2)-x, and ADC(3) [12, 45] for the treatment of valence-excited states as well as the respective core-valence separation (CVS) variants [34, 46] for tackling core-excited states. Both restricted as well as unrestricted Hartree-Fock references are supported and few-reference ground states as well as their excitations can be approached using

the spin-flip modification [47] of ADC. For reducing the number of active occupied or virtual orbitals and thus lowering the computational cost, the frozen-core and frozen-virtual [48] approximations can be additionally applied to all methods implemented in `adcc`. The selection of occupied or virtual orbitals to be frozen as well as the selection of the core space is completely arbitrary, i.e., not limited to contiguous blocks of occupied or virtual orbitals. The tensor operations required in the ADC working equations are evaluated in the `C++` core of `adcc` utilising the block-sparse tensor library `libtensor` [18] for exploiting symmetry and parallelising operations. As a result `adcc` is able to easily address medium-sized problems, such as the cc-pVDZ ADC(2) calculations of nile red shown previously.

In `adcc` ADC calculations are started and controlled from a `python` module, which exposes predefined ADC workflows for all aforementioned methods. In line with the building-block approach taken by `adcc`, the Hartree-Fock (HF) reference needs to be prepared in an external host program. Currently four host programs are supported out-of-the box, such that in practice the respective SCF datastructures are directly understood by `adcc`, namely `psi4` [1, 2], `pyscf` [3], `molsturm` [4] and `veloxchem` [14]. Further programs can be added by implementing about 20 functions from `python`, by employing a dictionary, or by using an HDF5 file to pass precomputed data from the host program to `adcc`.

The default workflow of `adcc` computes a set of requested excited states and offers `python` functions for simple post-processing, such as plotting of the excitation spectrum. Key individual ADC quantities, such as transition or state properties or respective transition or state density matrices can be directly accessed as `numpy` arrays [19]. Beyond this, any individual step of `adcc` and its intermediate results may be requested from `python`. This allows unusual or novel ADC computations to be easily realised and simplifies the extension of `adcc` beyond its present capabilities in user code. For a laptop-scale problem, this makes `adcc` highly suitable for interactive use, allowing to grasp the quantum-chemical properties of a system through one's own code rather than through a "black box" provided by traditional program packages. This greatly facilitates a hands-on approach to computational spectroscopy for teaching or research.

SUMMARY AND OUTLOOK

The hybrid `C++`/ `python` module `adcc` for the simulation of excited states based on the algebraic-diagrammatic construction for the polarisation propagator (ADC) has been presented. Instead of aiming for a complete framework for spectroscopy simulations, our philosophy is to integrate with existing software as much as possible and only provide a single building block, i.e., a module for ADC calculations. For this reason our `C++` core layer mainly contains code to interface with two libraries, `libtensor` [18] for performing tensor operations and `libadc` [16] for the implementation of the ADC working equations. Similarly, on the `python` layer we aim to integrate both with the conventional `python` ecosystem, i.e., with libraries such as `numpy` [19] or `matplotlib` [20], as well as `python`-based quantum chemistry software: Running calculations with `adcc` by supplying a Hartree-Fock reference from `pyscf` [3], `psi4` [1, 2], `molsturm` [4], or `veloxchem` [14] is supported out of the box.

The required interfaces, both on the `C++` and the `python` layer, are kept simple and are well-documented with ideally all functionality of the `C++` core being available from `python` as well. This has the advantage that large parts of `adcc`, including our iterative solver algorithms, could be implemented in `python`. The result is a flexible module with extensible workflows, which has been demonstrated in the given examples. Despite this flexibility, `adcc` easily performs calculations with about 400 basis functions making `adcc` not only a useful tool for method development, but also for practical research calculations or teaching.

In the future we plan to extend `adcc` to other ADC methods of similar mathematical structure, such as IP-ADC [49, 50, 51, 52]. For this we expect our focus on `python` and open interfaces to accelerate developments. A clear aim is also to enhance deeper integration of `adcc` into other quantum-chemical software projects. As a result not only the feature set of each involved project would grow, but the implied possibility to mix and match software building blocks for a scientific simulation at wish, generates an environment for sustainable scientific innovation.

FUNDING INFORMATION

MS was supported by the Deutsche Forschungsgemeinschaft (DFG) by means of the research training group "CLiC" (GRK 1986, Complex Light Control). TF was supported by a grant from the Swedish Research Council (Grant No. 2017-00356). This work was supported by the Heidelberg Graduate School of Mathematical and Computational Methods for the Sciences (GSC220).

RESEARCH RESOURCES

Calculations in this work were supported by the state of Baden-Württemberg through bwHPC (bwForCluster MLS&WISO) and the German Research Foundation (DFG) through grant INST 35/1134-1 FUGG.

ACKNOWLEDGMENTS

The authors thank Patrick Norman for stimulating the redesign of `adcc` from an earlier version of the code and Adrian L. Dempwolff for discussions and helpful comments during the preparation of the manuscript.

FURTHER READING

The details of the `python` interface of `adcc` as well as guides for installation and for getting started can be found in the `adcc` documentation. The `adcc` documentation is available online at https://adc-connect.org.

References

[1] Parrish RM, Burns LA, Smith DGA, Simmonett AC, DePrince AE, Hohenstein EG, et al. Psi4 1.1: An Open-Source Electronic Structure Program Emphasizing

Automation, Advanced Libraries, and Interoperability. J Chem Theory Comput. 2017;13(7):3185–3197. doi:10.1021/acs.jctc.7b00174.

[2] Smith DGA, Burns LA, Sirianni DA, Nascimento DR, Kumar A, James AM, et al. Psi4NumPy: An Interactive Quantum Chemistry Programming Environment for Reference Implementations and Rapid Development. J Chem Theory Comput. 2018;14(7):3504–3511. doi:10.1021/acs.jctc.8b00286.

[3] Sun Q, Berkelbach TC, Blunt NS, Booth GH, Guo S, Li Z, et al. The Python-based Simulations of Chemistry Framework (PySCF). WIREs Comput Mol Sci. 2017;8(1). doi:10.1002/wcms.1340.

[4] Herbst MF, Dreuw A, Avery JE. Towards quantum-chemical method development for arbitrary basis functions. J Chem Phys. 2018 Aug;149(8):84106. doi:10.1063/1.5044765.

[5] Verstraelen T, Tecmer P, Heidar-Zadeh F, González-Espinoza CE, Chan M, Kim TD, et al.. HORTON 2.1.1; 2017. Available from: http://theochem.github.com/horton/.

[6] Muller R. PyQuante: Python Quantum Chemistry;. Accessed on 15th October 2019. http://pyquante.sourceforge.net.

[7] Field MJ. The pDynamo program for molecular simulations using hybrid quantum chemical and molecular mechanical potentials. J Chem Theory Comput. 2008;4(7):1151–1161. doi:10.1021/ct800092p.

[8] Unsleber JP, Dresselhaus T, Klahr K, Schnieders D, Böckers M, Barton D, et al. Serenity: A subsystem quantum chemistry program. J Comp Chem. 2018;39(13):788–798. doi:10.1002/jcc.25162.

[9] Enkovaara J, Romero NA, Shende S, Mortensen JJ. GPAW - massively parallel electronic structure calculations with Python-based software. Procedia Comput Sci. 2011;4:17–25. doi:10.1016/j.procs.2011.04.003.

[10] Larsen AH, Mortensen JJ, Blomqvist J, Castelli IE, Christensen R, Dułak M, et al. The atomic simulation environment—a Python library for working with atoms. J Phys: Condens Matter. 2017;29(27):273002. doi:10.1088/1361-648X/aa680e.

[11] Di Remigio R, Steindal AH, Mozgawa K, Weijo V, Cao H, Frediani L. PCMSolver: An open-source library for solvation modeling. Int J Quantum Chem. 2019;119(1):e25685. doi:10.1002/qua.25685.

[12] Schirmer J. Beyond the random-phase approximation: A new approximation scheme for the polarization propagator. Phys Rev A. 1982 Nov;26:2395–2416. doi:10.1103/PhysRevA.26.2395.

[13] Dreuw A, Wormit M. The algebraic diagrammatic construction scheme for the polarization propagator for the calculation of excited states. WIREs Comput Mol Sci. 2014;5(1):82–95. doi:10.1002/wcms.1206.

[14] Rinkevicius Z, Li X, Vahtras O, Ahmadzadeh K, Brand M, Ringholm M, et al. VeloxChem: a Python-driven density-functional theory program for spectroscopy simulations in high-performance computing environments. WIREs Computational Molecular Science. 2019;p. 0. Submitted.

[15] Hehre WJ, Stewart RF, Pople JA. Self-Consistent Molecular-Orbital Methods. I. Use of Gaussian Expansions of Slater-Type Atomic Orbitals. J Chem Phys. 1969;51(6):2657–2664. doi:10.1063/1.1672392.

[16] Wormit M, Rehn DR, Harbach PHP, Wenzel J, Krauter CM, Epifanovsky E, et al. Investigating excited electronic states using the algebraic diagrammatic construction (ADC) approach of the polarisation propagator. Mol Phys. 2014;112(5-6):774–784. doi:10.1080/00268976.2013.859313.

[17] Davidson ER. The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices. J Comp Phys. 1975;17(1):87 – 94. doi:10.1016/0021-9991(75)90065-0.

[18] Epifanovsky E, Wormit M, Kuś T, Landau A, Zuev D, Khistyaev K, et al. New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations. J Comput Chem. 2013;34(26):2293–2309. doi:10.1002/jcc.23377.

[19] van der Walt S, Colbert SC, Varoquaux G. The NumPy Array: A Structure for Efficient Numerical Computation. Comp Sci Eng. 2011;13(2):22–30. doi:10.1109/MCSE.2011.37.

[20] Hunter JD. Matplotlib: A 2D graphics environment. Comp Sci Eng. 2007;9(3):90–95. doi:10.1109/MCSE.2007.55.

[21] Pérez F, Granger BE. IPython: a System for Interactive Scientific Computing. Comput Sci Eng. 2007 May;9(3):21–29. doi:10.1109/MCSE.2007.53.

[22] Kluyver T, Ragan-Kelley B, Pérez F, Granger B, Bussonnier M, Frederic J, et al.. Jupyter Notebooks - a publishing format for reproducible computational workflows; 2016. https://jupyter.org/. doi:10.3233/978-1-61499-649-1-87.

[23] Wormit M. Development and application of reliable methods for the calculation of excited states : from light-harvesting complexes to medium-sized molecules [Ph.D. thesis]. Goethe-Universität Frankfurt (Main); 2009.

[24] Herbst MF. Development of a modular quantum-chemistry framework for the investigation of novel basis functions [Ph.D. thesis]. Ruprecht-Karls-Universität Heidelberg; 2018. doi:10.11588/heidok.00024519.

[25] Hughes J. Why functional programming matters. In: Turner D, editor. Research Topics in Functional Programming. Addison-Wesley; 1990. p. 17–42.

[26] Krishnan R, Binkley JS, Seeger R, Pople JA. Self-Consistent Molecular Orbital Methods. XX. A Basis Set for Correlated Wave Functions. J Chem Phys. 1980;72:650. doi:10.1063/1.438955.

[27] Shao Y, Gan Z, Epifanovsky E, Gilbert ATB, Wormit M, Kussmann J, et al. Advances in molecular quantum chemistry contained in the Q-Chem 4 program package. Mol Phys. 2015;113(2):184–215. doi:10.1080/00268976.2014.952696.

[28] Herbst MF. ctx: Key-value C++ datastructures for organised hierarchical storage; 2019. doi:10.5281/zenodo.2590706.

[29] Jakob W, Rhinelander J, Moldovan D. pybind11 – Seamless operability between C++11 and Python; 2017. Https://github.com/pybind/pybind11.

[30] Jones E, Oliphant T, Peterson P, et al.. SciPy: Open source scientific tools for Python; 2001–. Accessed on 09th Sepetember 2019. Available from: http://www.scipy.org/.

[31] HDF5 Reference Manual; 2011. Release 1.8.8.

[32] Dunning TH. Gaussian basis sets for use in correlated molecular calculations. I. The atoms boron through neon and hydrogen. J Chem Phys. 1989;90(2):1007–1023. doi:10.1063/1.456153.

[33] Barth A, Schirmer J. Theoretical core-level excitation spectra of N_2 and CO by a new polarisation propagator method. J Phys B: At Mol Phys. 1985;18:867–885. doi:10.1088/0022-3700/18/5/008.

[34] Trofimov AB, Moskovskaya TÉ, Gromov EV, Vitkovskaya NM, Schirmer J. Core-Level Electronic Spectra in ADC(2) Approximation for Polarization Propagator: Carbon Monoxide and Nitrogen Molecules. J Struct Chem. 2000;41:483–494. doi:10.1007/BF02742009.

[35] Wenzel J, Wormit M, Dreuw A. Calculating Core-Level Excitations and X-Ray Absorption Spectra of Medium-Sized Closed-Shell Molecules with the Algebraic-Diagrammatic Construction Scheme for the Polarization Propagator. J Comput Chem. 2014;35:1900–1915. doi:10.1002/jcc.23703.

[36] Vidal ML, Feng X, Epifanovsky AI E anf Krylov, Coriani S. New and efficient equation-of-motion coupled-cluster framework for core-excited and core-ionized states. J Chem Theory Comput. 2019;15:3117–3133. doi:10.1021/acs.jctc.9b00039.

[37] Peng R, Copan AV, Sokolov AY. Simulating X-ray absorption spectra with linear-response density cumulant theory. J Phys Chem A. 2019;123:1840–1850. doi:10.1021/acs.jpca.8b12259.

[38] McLaren R, Clark SAC, Ishii I, Hitchcock AP. Absolute oscillator strengths from K-shell electron-energy-loss spectra of the fluoroethenes and 1,3-perfluorobutadiene. Phys Rev A. 1987;36:1683–1701. doi:10.1088/0022-3700/10/12/028.

[39] Fransson T, Coriani S, Christiansen O, Norman P. Carbon X-ray absorption spectra of fluoroethenes and acetone: A study at the coupled cluster, density functional, and static-exchange levels of theory. J Chem Phys. 2013;138:124311. doi:10.1063/1.4795835.

[40] Wenzel J, Holzer A, Wormit M, Dreuw A. Analysis and Comparison of CVS-ADC Approaches up to Third Order for the Calculation of Core-Excited States. J Chem Phys. 2015;142:214104. doi:10.1063/1.4921841.

[41] Scheurer M, Herbst MF, Reinholdt P, Olsen JMH, Dreuw A, Kongsted J. Polarizable Embedding Combined with the Algebraic Diagrammatic Construction: Tackling Excited States in Biomolecular Systems. J Chem Theory Comput. 2018;14(9):4870–4883. doi:10.1021/acs.jctc.8b00576.

[42] Scheurer M, Reinholdt P, Kjellgren ER, Olsen JMH, Dreuw A, Kongsted J. CPPE: An Open-Source C++ and Python Library for Polarizable Embedding. J Chem Theory Comput. 2019;Just Accepted Manuscript. doi:10.1021/acs.jctc.9b00758.

[43] Humphrey W, Dalke A, Schulten K. VMD: Visual molecular dynamics. J Mol Graph. 1996 feb;14(1):33–38. doi:10.1016/0263-7855(96)00018-5.

[44] Plasser F, Wormit M, Dreuw A. New tools for the systematic analysis and visualization of electronic excitations. I. Formalism. J Chem Phys. 2014 jul;141(2):024106. doi:10.1063/1.4885819.

[45] Trofimov AB, Stelter G, Schirmer J. A consistent third-order propagator method for electronic excitation. J Chem Phys. 1999;111(22):9982–9999. doi:10.1063/1.480352.

[46] Wenzel J, Wormit M, Dreuw A. Calculating X-Ray Absorption Spectra of Open-Shell Molecules with the Unrestricted Algebraic-Diagrammatic Construction Scheme for the Polarization Propagator. J Chem Theory Comput. 2014;10:4583–4598. doi:10.1021/ct5006888.

[47] Lefrancois D, Wormit M, Dreuw A. Adapting algebraic diagrammatic construction schemes for the polarization propagator to problems with multi-reference electronic ground states exploiting the spin-flip ansatz. J Chem Phys. 2015;143(12):124107. doi:10.1063/1.4931653.

[48] Yang C, Dreuw A. Evaluation of the restricted virtual space approximation in the algebraic-diagrammatic construction scheme for the polarization propagator to speed-up excited-state calculations [article]. J Comput Chem. 2017 6;38:1528–1537. doi:10.1002/jcc.24794.

[49] Schirmer J, Cederbaum LS, Walter O. New approach to the one-particle Green's function for finite Fermi systems [article]. Phys Rev A. 1983;28:1237–1259. doi:10.1103/physreva.28.1237.

[50] von Niessen W, Schirmer J, Cederbaum LS. Computational methods for the one-particle green's function [article]. Comput Phys Rep. 1984 4;1:57–125. doi:10.1016/0167-7977(84)90002-9.

[51] Schirmer J, Angonoa G. On Green's function calculations of the static self-energy part, the ground state energy and expectation values [article]. J Chem Phys. 1989 8;91:1754–1761. doi:10.1063/1.457081.

[52] Dempwolff AL, Schneider M, Hodecker M, Dreuw A. Efficient implementation of the non-Dyson third-order algebraic diagrammatic construction approximation for the electron propagator for closed- and open-shell molecules [article]. J Chem Phys. 2019 2;150:064108. doi:10.1063/1.5081674.