Software News and Updates
PyADF — A Scripting Framework for Multiscale Quantum Chemistry

CHRISTOPH R. JACOB,[1] S. MAYA BEYHAN,[2] ROSA E. BULO,[2] ANDRÉ SEVERO PEREIRA GOMES,[3] ANDREAS W. GÖTZ,[4] KARIN KIEWISCH,[2] JETZE SIKKEMA,[2] LUCAS VISSCHER[2]

[1]*Center for Functional Nanostructures, Karlsruhe Institute of Technology (KIT), Wolfgang-Gaede-Str. 1a, 76131 Karlsruhe, Germany*

[2]*Amsterdam Center for Multiscale Modeling (ACMM), VU University Amsterdam, De Boelelaan 1083, 1081 HV Amsterdam, The Netherlands*

[3]*Laboratoire PhLAM, Université de Lille 1, CNRS UMR 8523, Bat P5, F-59655 Villeneuve d'Ascq Cedex, France*

[4]*San Diego Supercomputer Center, University of California San Diego, 9500 Gilman Drive MC0505, La Jolla, California 92093-0505*

Received 18 January 2011; Revised 7 March 2011; Accepted 20 March 2011
DOI 10.1002/jcc.21810
Published online 3 May 2011 in Wiley Online Library (wileyonlinelibrary.com).

Abstract: Applications of quantum chemistry have evolved from single or a few calculations to more complicated workflows, in which a series of interrelated computational tasks is performed. In particular multiscale simulations, which combine different levels of accuracy, typically require a large number of individual calculations that depend on each other. Consequently, there is a need to automate such workflows. For this purpose we have developed PyADF, a scripting framework for quantum chemistry. PyADF handles all steps necessary in a typical workflow in quantum chemistry and is easily extensible due to its object-oriented implementation in the Python programming language. We give an overview of the capabilities of PyADF and illustrate its usefulness in quantum-chemical multiscale simulations with a number of examples taken from recent applications.

© 2011 Wiley Periodicals, Inc. J Comput Chem 32: 2328–2338, 2011

Key words: multiscale; scripting; workflow; embedding

Introduction

In modern applications of quantum-chemical program packages, one usually needs to perform series of calculations. For instance, for the same molecule a single calculation is in general not sufficient, but calculations using different theoretical methods, different basis sets, different technical settings, and in many cases also using different program packages are necessary. Furthermore, such series of calculations are mostly needed not only for a single molecular structure, but for many different ones.[1,2]

In addition, these calculations are commonly part of a collection of tasks that are interrelated, i.e., the results of one calculation serve as input for a subsequent one.[3,4] A basic example of such a workflow would be performing geometry optimizations for a series of molecules using density-functional theory (DFT) with a smaller basis set, followed by single-point energy calculations using a more accurate wave-function theory (WFT) method and/or a calculation of molecular properties. Finally, the appropriate results have to be extracted from the output files and have to be processed to obtain the sought-after quantities (e.g., total energy differences).

Clearly, there is a need to automate these workflows. Many computational chemists have developed their own solutions for this purpose, mostly shell scripts or small programs (see, e.g., ref. 5). However, these usually address only certain steps, such as generating input files or extracting results from output files. Some quantum-chemical program packages take a further step by providing a scripting interface. For instance, NWChem[6] offers a Python interface[7] for executing simple workflows, Molpro[8] provides scripting facilities in its input files and produces XML output for an easier post-processing of results, ADF[9,10] comes with tools for the automatic generation of input files and for easily extracting the

Correspondence to: Ch. R. Jacob; e-mail: christoph.jacob@kit.edu

Contract/grant sponsor: Deutsche Forschungsgemeinschaft (DFG) (Center for Functional Nanostructures), The Netherlands Organization for Scientific Research (NWO) (Veni and Vici grant), Ministère chargé de la Recherche, Région Nord-Pas de Calais, Fond Européen de Développement Economique des Régions (FEDER), German Academic Exchange Service (DAAD) (Postdoc-Programme)

results of calculations,[11] and the MOLECONTROL[12] add-on to TURBOMOLE[13, 14] allows for the execution of series of calculations and simple workflows.

However, as workflows become more and more involved, ad hoc shell-scripting solutions reach their limitations. Furthermore, the existing scripting interfaces are usually specific to one quantum-chemical program package and can often only handle parts of the required workflows. Therefore, more flexible and general solutions will be useful.

In the area of chemo- and bio-informatics, where very large datasets are generated and processed, general-purpose workflow engines that allow the user to organize and schedule different tasks (usually using a graphical user interface) are very common.[15, 16] Such workflow engines have also been adapted to handle computational chemistry problems, in particular in the context of grid computing.[17–20] However, these solutions are mostly either tailored to executing a single program package (see, e.g., ref. 21), or very general and thereby present significant obstacles for their initial use and for the extension to tasks from quantum chemistry.

In the past years, work in our groups has focussed on "quantum-chemical multiscale simulations," which combine different levels of quantum-chemical descriptions. These are based on the frozen-density embedding (FDE) scheme initially proposed by Wesolowski and Warshel[22] (following earlier work by Senatore and Subbaswamy[23] and by Cortona[24]) and its extension to WFT-in-DFT embedding, first proposed by Carter and coworkers.[25–28]

Particularly for such quantum-chemical multiscale simulations one encounters very complex workflows that are beyond the capabilities of standard tools. Typically, they involve hundreds of individual calculations, and the results of a subset of these calculations are needed as input for following steps. For instance, applications of the FDE scheme to calculate solvent effects on molecular properties[29–31] require the construction of an approximate solvent electron density, followed by an FDE calculation of the molecular property of interest for the embedded solute molecule. This needs to be done for a large number of snapshots taken from a molecular dynamics simulation. Similarly, a subsystem-DFT treatment of proteins[32] requires calculations for all subsystems (e.g., the individual amino acids), each embedded in an environment constructed from all other subsystems. For WFT-in-DFT embedding calculations (see, e.g., ref. 33), it is necessary to combine different quantum-chemical program packages and to pass the embedding potential and/or the frozen environment electron density between these programs.

To automate these rather complicated workflows, we have developed PYADF, a scripting framework for quantum chemistry. It handles all the steps required in typical workflows of quantum chemistry: generation of input files, execution of the different programs, error handling, as well as extraction and post-processing of the results and offers a very flexible and extensible framework for combining these different steps. PYADF is written in the Python programming language[34] (for an introduction to Python from the perspective of computational sciences, see, e.g., refs. 35 and 36).

Even though the functionality available in PYADF currently focusses on quantum-chemical multiscale simulations, it is in no way limited to this type of calculations. Instead, it provides a general framework for processing workflows in quantum chemistry, which can easily be extended to additional computational tasks. Despite its name — which indicates its historical origin as an extension to the ADF package — PYADF is not specific to a single program, but works with a number of different quantum chemistry codes. In this article, we give an overview of the PYADF scripting framework, and illustrate its usefulness by discussing a number of examples.

This work is organized as follows. First, we outline the design of PYADF and give an overview of its most important features. This is followed by a demonstration of the capabilities of PYADF for automating workflows commonly encountered in quantum chemistry and an overview of various applications of PYADF in quantum-chemical multiscale simulations. Finally, we summarize and give an outlook on planned and ongoing developments.

PYADF — Design and Overview

The driving idea behind PYADF is the definition of a framework that will provide mechanisms for both controlling the execution of different computational tasks and for managing the communication between these tasks. This should be achieved in such a way that users are provided simple, yet powerful ways to define their computational workflows.

To this end, PYADF makes use of object-oriented programming techniques in the high-level programming language Python. The central paradigm of object-oriented programming is the definition of classes, i.e., objects that contain both data (also know as "attributes") and actions (the so-called "methods"). The definition of different classes allows us to group together the different aspects involved in one step of the workflow into a single entity. This way, as many details as possible are hidden from the user, who only needs to know how to use these classes on a higher level. Such an object-oriented design has further advantages. It is possible to establish a hierarchical relation between classes (known as "inheritance"), so that one can utilize existing classes to construct new ones. The classes provided by PYADF can then easily be extended by the user. This makes it, for instance, rather straightforward to incorporate scientific codes from third parties.

The input files to PYADF are, in effect, simply Python scripts, so the full power of the language and its numerous extensions is immediately available to the user. To illustrate how we utilize the classes provided by PYADF to arrive at a very simple, high-level definition of a basic workflow, a sample input file is given in Figure 1.

This input defines an elementary workflow: The molecular coordinates are read from a file, a single point calculation is performed with ADF, and finally the magnitude of the dipole moment is extracted from the resulting output and printed.

In the first line, a molecule object (i.e., an instance of the `molecule` class) is created. In the simplest form, which is used here, it is initialized by reading the molecular coordinates from

```
mol = molecule('h2o.xyz')

job = adfsinglepointjob(mol, basis='TZVP')

results = job.run()

print results.get_dipole_magnitude()
```

Figure 1. A minimal PYADF input file.

an xyz-file. Internally, PyAdf uses the Openbabel library[37–39] for storing the molecular coordinates so that any file format supported by Openbabel can be understood. The `molecule` class of PyAdf has a number of methods for manipulating molecular coordinates (e.g., joining different molecules, splitting a large system into its molecular fragments, adding hydrogen atoms to protein structures, and aligning molecules). Most of these methods also rely on functionality provided by Openbabel.

In the second line, a job object is initialized. PyAdf provides a number of different job classes for different types of calculations. Here, the class `adfsinglepointjob`, which represents a single point DFT calculation with the Adf program, is used. There are other job classes for other types of calculations (e.g., `adfgeometryjob` for geometry optimizations, `adfnmrjob` for the calculation of NMR chemical shifts, and `adfcpljob` for calculating spin–spin coupling constants with Adf, or `daltonsinglepointjob` and `diracsinglepointjob` for single point calculations with the Dalton[40] and Dirac[41] programs, respectively). These job classes are part of a class hierarchy, i.e., they are related by inheritance. A complete list of the types of calculations currently available in PyAdf can be found in the PyAdf documentation (available at `http://www.pyadf.org`).

Creating a job object only defines the type of calculation and allows one to specify the technical settings (such as basis set and exchange–correlation functional) that should be used in this calculation. It does not execute the calculation itself. This is achieved by calling the job's `run` method, as shown in the third line of the script considered here. This method will generate the necessary input files, call the appropriate executable(s), and save the output file(s) produced by the program (both standard output and binary restart files, depending on the program).

The job's `run` method returns a results object. This results object is an instance of a results class corresponding to the type of the job. Using it, the results of the calculation can be accessed. For instance, on the fourth line of the considered input file, the magnitude of the dipole moment is extracted with the `get_dipole_magnitude` method. There is a hierarchy of such results classes that is analogous to the job class hierarchy. Each of them provides methods for accessing the different quantities that have been calculated. Which quantities are available obviously depends on the type of the calculation. A full list of predefined computational tasks and of the associated classes can be found in the PyAdf documentation.

Internally, PyAdf implements a file manager, which stores the files produced by each of the calculations. The results objects then use this file manager to access these files. If a certain quantity is requested, the corresponding method of the results object knows how to extract the quantity from these files — either by using a regular expression to extract it from the output file or by reading it from binary restart files. For the convenience of the user, the output of PyADF clearly identifies the related files for each job, so that these files can easily be inspected in case of a problem.

PyAdf also provides extensive restart facilities. Because Python is an interpreted language, errors in the PyAdf input file will be detected only at runtime. Furthermore, it is of course possible that one of the programs called by PyAdf encounters an error condition during its execution. As a consequence, it can happen that a PyAdf run is aborted after a large number of calculations have been performed. In this case, PyAdf generates an archive of its results that can be imported when re-running with a corrected input file. In such a restarted run, the calculations that have already been completed earlier will not be executed again. This is achieved by generating a checksum of the input file(s) for each calculation. Before a calculation is actually executed, PyAdf checks whether a calculation with the same checksum has already been performed. If so, a results object for this previous calculation is returned, using the files from the previous run.

Automating Common Workflows in Quantum Chemistry with PyAdf

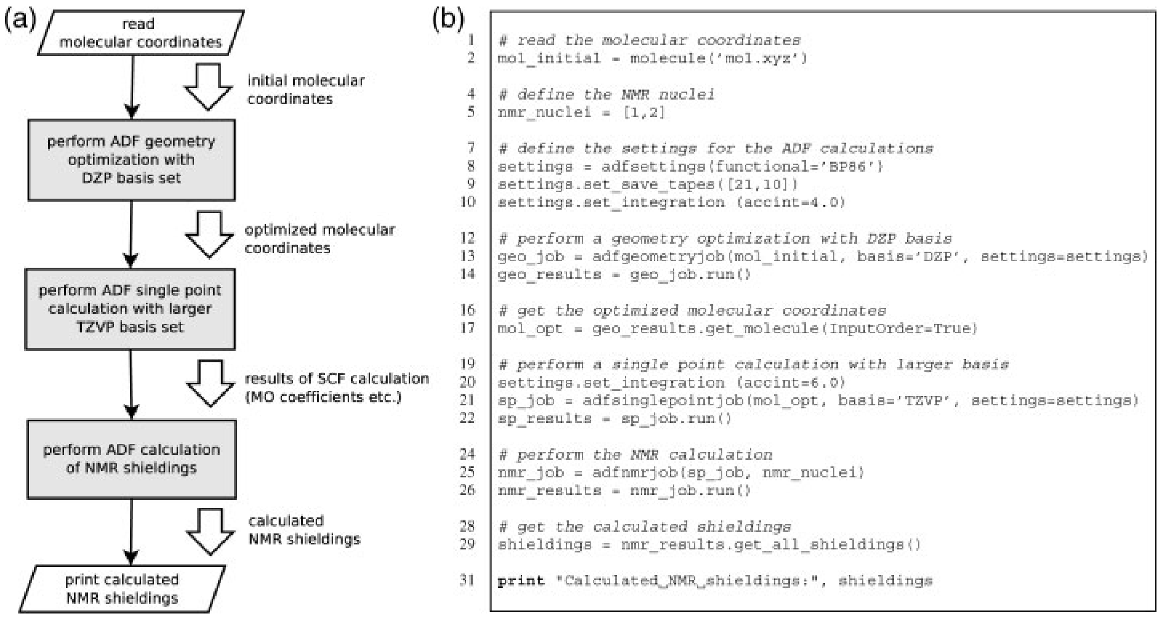
A Simple Example: Calculation of NMR Shieldings

To demonstrate how the building blocks of PyAdf described in the previous section (i.e., the `molecule` class as well as job and result classes) can be used to construct quantum-chemical workflows, we first consider a simple example: a geometry optimization followed by a calculation of NMR shieldings. A flowchart of this workflow is shown in Figure 2a, which also indicates which information has to be exchanged between the different tasks.

First, the molecular coordinates are read from a file. Starting from this initial structure, a geometry optimization is then performed using a small basis set. This is followed by a single point calculation for the optimized geometry, employing a larger basis set. Subsequently, a calculation of the NMR shieldings is performed. This NMR calculation requires the results (most importantly, the MO coefficients) of the previous single point calculation. Finally, the calculated NMR shieldings are printed.

The input file in Figure 2b shows how such a workflow can be realized with PyAdf. First, a molecule object is initialized by reading the molecular coordinates from an xyz-file and a list with the numbers of the nuclei for which the NMR shieldings should be calculated is created (lines 1–5). After that, the settings for the subsequent Adf calculations are initialized, specifying the exchange–correlation functional and the numerical integration accuracy (lines 7–10). Then, an `adfgeometryjob` instance is created and the job is run (lines 12–14). The `run` method returns a results object, from which the optimized molecular coordinates (in form of another molecule object) are obtained with the `get_molecule` method (line 17). This new molecule object is then used to initialize and run an `adfsinglepointjob`, for which a higher integration accuracy and a larger basis set are used (lines 19–22). The new results object is then used to initialize and run an `adfnmrjob` (lines 24–26). Finally, the `get_all_shieldings` method of the NMR job's results object is used to extract the calculated shieldings, which can then be printed (lines 28–31).

Since all the features of the Python language are available in PyAdf input files, it is easy to extend the simple workflow outlined above. Among the many possibilities is the application of this workflow to a large number of molecules. This can be achieved by a loop over all xyz-files in a given directory, allowing the calculations to be performed for each of these molecules. If needed, the numbers of the nuclei for which the NMR shieldings have to be calculated could be determined for each of these molecules individually. To this end, one can, for instance, use the functionality provided by Openbabel to identify the nuclei of interest using a SMARTS pattern.[42]


(a)
read molecular coordinates
initial molecular coordinates
perform ADF geometry optimization with DZP basis set
optimized molecular coordinates
perform ADF single point calculation with larger TZVP basis set
results of SCF calculation (MO coefficients etc.)
perform ADF calculation of NMR shieldings
calculated NMR shieldings
print calculated NMR shieldings
(b)

```
# read the molecular coordinates
mol_initial = molecule('mol.xyz')

# define the NMR nuclei
nmr_nuclei = [1,2]

# define the settings for the ADF calculations
settings = adfsettings(functional='BP86')
settings.set_save_tapes([21,10])
settings.set_integration (accint=4.0)

# perform a geometry optimization with DZP basis
geo_job = adfgeometryjob(mol_initial, basis='DZP', settings=settings)
geo_results = geo_job.run()

# get the optimized molecular coordinates
mol_opt = geo_results.get_molecule(InputOrder=True)

# perform a single point calculation with larger basis
settings.set_integration (accint=6.0)
sp_job = adfsinglepointjob(mol_opt, basis='TZVP', settings=settings)
sp_results = sp_job.run()

# perform the NMR calculation
nmr_job = adfnmrjob(sp_job, nmr_nuclei)
nmr_results = nmr_job.run()

# get the calculated shieldings
shieldings = nmr_results.get_all_shieldings()

print "Calculated NMR shieldings:", shieldings
```

Figure 2. (a) Flowchart of a simple quantum-chemical workflow. The steps in gray boxes stand for the calculations performed with an external program. The large arrows indicate results that are passed between the different tasks. (b) A PyADF input file for realizing this workflow.

Furthermore, a calculation of the NMR shieldings for a reference compound could be added, so that the calculated shieldings can be converted to chemical shifts inside PyADF before they are printed.

Running Calculations for Large Test Sets of Molecules

In recent years, work in our groups has focussed on quantum-chemical multiscale simulations, and in particular on the frozen-density embedding (FDE) scheme.[22] In this DFT-based scheme, the total electron density $\rho_{\text{tot}}(\boldsymbol{r})$ is divided into the densities of N subsystems $\rho^{(n)}(\boldsymbol{r})$ ($n = 1, \ldots, N$), with

$$\rho_{\text{tot}}(\boldsymbol{r}) = \sum_{n=1}^{N} \rho^{(n)}(\boldsymbol{r}). \tag{1}$$

Given an (approximate) density for all other subsystems, the density in one active subsystem can be determined from a set of KS-like equations, in which the effect of the frozen environment density enters through an effective embedding potential (see, e.g., refs. 22,43 and 44 for details). By iteratively updating each of the subsystem densities in so-called freeze-and-thaw cycles,[45] one obtains a subsystem-DFT scheme[24] that can be used as an efficient alternative to conventional Kohn–Sham (KS) DFT calculations.

However, even though the frozen-density embedding potential is in principle exact (in the sense that it should lead to the same total electron density as a KS-DFT calculation on the full system), additional approximations are required for the kinetic-energy component of the embedding potentials. Likewise, when calculating energies, approximations have to be introduced for the nonadditive kinetic energy, i.e., an approximate kinetic-energy density functional has to be used. This naturally raises the question how accurate these approximations are. To assess the quality of the available approximations (for an overview, see, e.g., ref. 46) one possibility is to compare the interaction energies between two fragments obtained from a subsystem-DFT calculation to those obtained from a supermolecular KS-DFT calculation.

Using this strategy, three of us have recently presented a comparison of various kinetic-energy density functionals within such subsystem-DFT calculations for a large test set of intermolecular complexes and transition metal coordination compounds.[47] The workflow applied in this study is illustrated in Figure 3. It consists of a loop over all systems in the test set. For each of these (usually bimolecular) systems, the coordinates of the two subsystems are read and calculations for the supermolecule as well as for the isolated fragments are performed. This is followed by a subsystem-DFT calculation (i.e., the densities of both subsystems are updated iteratively), which uses the densities of the isolated fragments as initial guess. Finally, the reference value for the interaction energy is calculated as $E_{\text{int}}^{(\text{ref})} = E_{\text{supermol}} - E_1 - E_2$ and the subsystem-DFT interaction energy is calculated as $E_{\text{int}}^{(\text{FDE})} = E_{\text{FDE}} - E_1 - E_2$ (with corrections for the basis set superposition error where appropriate). After the loop over the test set is complete, a statistical analysis of the errors in the interaction energies can be performed (e.g., calculation of the root-mean square deviation).

Such a workflow can be easily realized in PyADF: Using the functionality of the Python language, a loop over the test set can be performed. For each complex, single point calculations for both the supermolecule and the individual fragments can be executed with the help of the `adfsinglepointjob` class. Frozen-density embedding and subsystem-DFT jobs are handled by the class `adffragmentsjob`. Such jobs require a list of fragments (i.e., molecule objects and possibly the results objects of previous calculations) and for each of these fragments, it can then be chosen

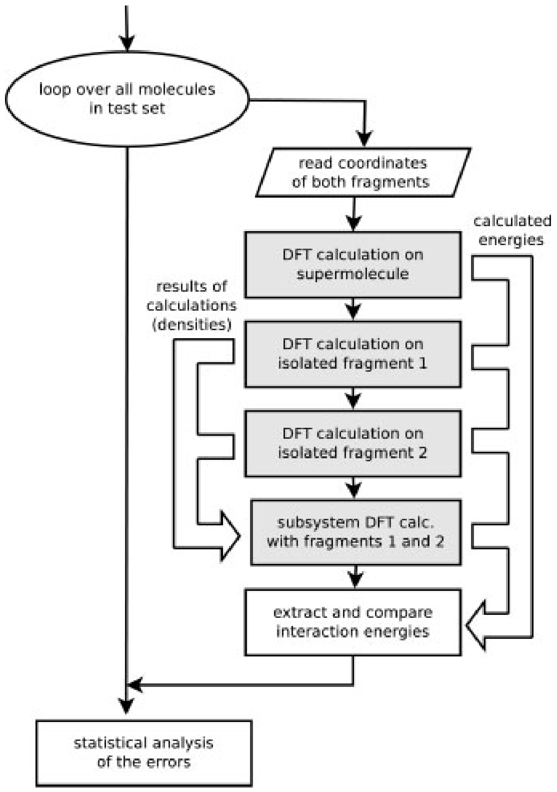

loop over all molecules in test set
read coordinates of both fragments
calculated energies
DFT calculation on supermolecule
results of calculations (densities)
DFT calculation on isolated fragment 1
DFT calculation on isolated fragment 2
subsystem DFT calc. with fragments 1 and 2
extract and compare interaction energies
statistical analysis of the errors

Figure 3. Workflow encountered for testing the accuracy of kinetic-energy functionals within subsystem DFT. The steps in gray boxes stand for the calculations performed with an external program. The large arrows indicate results that are passed between the different tasks.

whether it is active, frozen, or a frozen fragment that is iteratively updated in freeze-and-thaw cycles (see also the description of the flexible FDE implementation in the ADF package[44]). In the case considered here, two fragments (one active and one that is updated in freeze-and-thaw cycles) are used. Ultimately, the required energies can be extracted from the corresponding results objects, and their further processing can be done using the functionality of Python.

Since workflows similar to the one described here are quite common (another example would be the assessment of different exchange–correlation functionals for test sets of molecules), PYADF provides a convenience class `datasetjob` for such applications.

Postprocessing of Results and Plotting

Besides comparing energies, the quality of any quantum-chemical calculation can also be assessed by comparing the computed electron density with that of a reference calculation. PYADF provides the general functionalities for such a comparison. As an example, we will again consider subsystem-DFT calculations. With the exact kinetic-energy functional, a subsystem-DFT calculation would yield the same electron density as a supermolecular KS-DFT calculation on the full system.[43,45] Therefore, the difference between the density from the supermolecular KS-DFT calculation and the subsystem-DFT density obtained with an approximate kinetic-energy functional can be used to judge the quality of these approximations.

Performing the subsystem-DFT calculation for a test set of intermolecular complexes can be performed using PYADF as described in the previous section. However, in contrast to an assessment of the energy (a single real number), comparisons of the electron density (a function of three variables) require additional functionality for the postprocessing of the results. It is necessary to obtain the values of the different densities on a grid and compare them to each other, either by plotting the difference density or by quantifying the deviation by integrating the absolute value of the difference density.[48]

An excerpt from an input file showing how these postprocessing steps can be performed using PYADF is depicted in Figure 4. The results objects of both the supermolecular KS-DFT calculation and the subsystem-DFT calculation have a `get_density` method, which returns a "density object" (lines 6 and 9, respectively). This density can, for instance, be written in a cube file with the `get_cubfile` method (lines 7 and 10), which can in turn be used to prepare an isosurface plot. Example isosurface plots of the densities from a supermolecular KS-DFT calculation and from a subsystem-DFT calculation are shown in Figure 4 for the example of the coordination complex formed from BH_3 and NH_3. Despite the fact that this is a case for which the available GGA-type kinetic-energy functionals[50] have been shown to fail,[51,52] the two isosurface plots can hardly be distinguished. Therefore, it is more instructive to look at the difference density. In PYADF, the difference density can be obtained by simply subtracting the two density objects (line 12), resulting in a new density object that again can be written in a cube file (line 13).

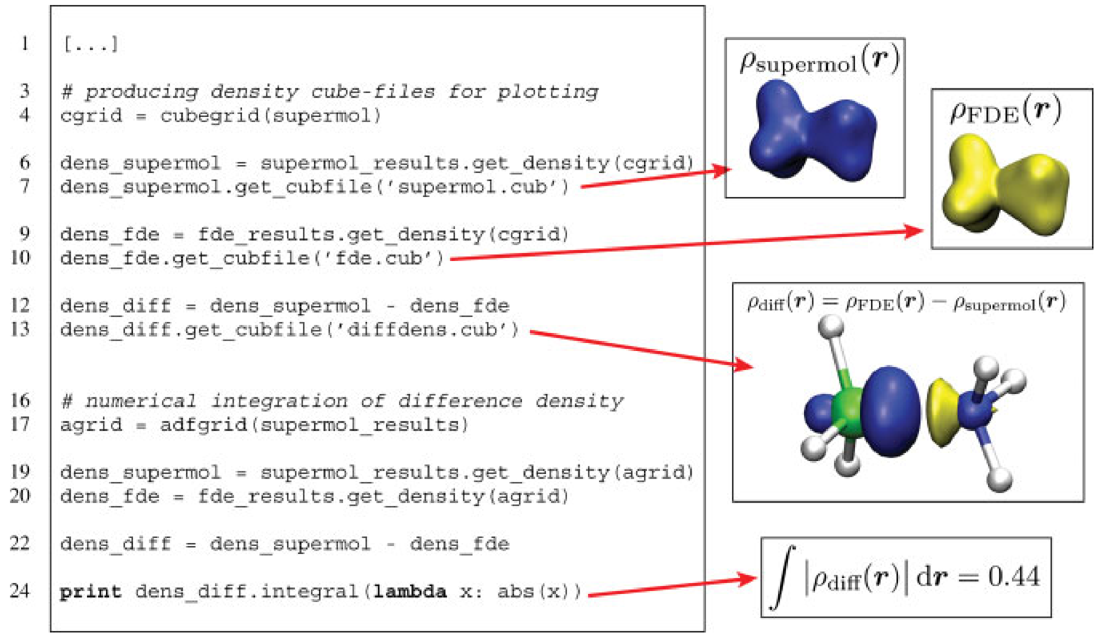
The `get_density` method allows one to choose on which grid the density is needed. For plotting, one typically needs an evenly spaced grid, as it is selected on line 4. Such a grid is, however, not suitable for performing an accurate numerical integration. For this purpose, the more precise numerical integration grid employed by ADF can also be chosen (line 17), to calculate the difference density on this grid in a completely analogous way (lines 19–22). In this representation, it is then possible to calculate the numerical integral over the absolute value of the difference density accurately (line 24).

In a similar way, molecular orbitals, localized orbitals, as well as the Kohn–Sham potential and its individual components can be handled. For instance, the recent work on accurate frozen-density embedding potentials presented in ref. 52 used PYADF extensively for manipulating and plotting the different components of the potentials.

Multiscale Simulations with PYADF

Solvent Effects on Molecular Properties

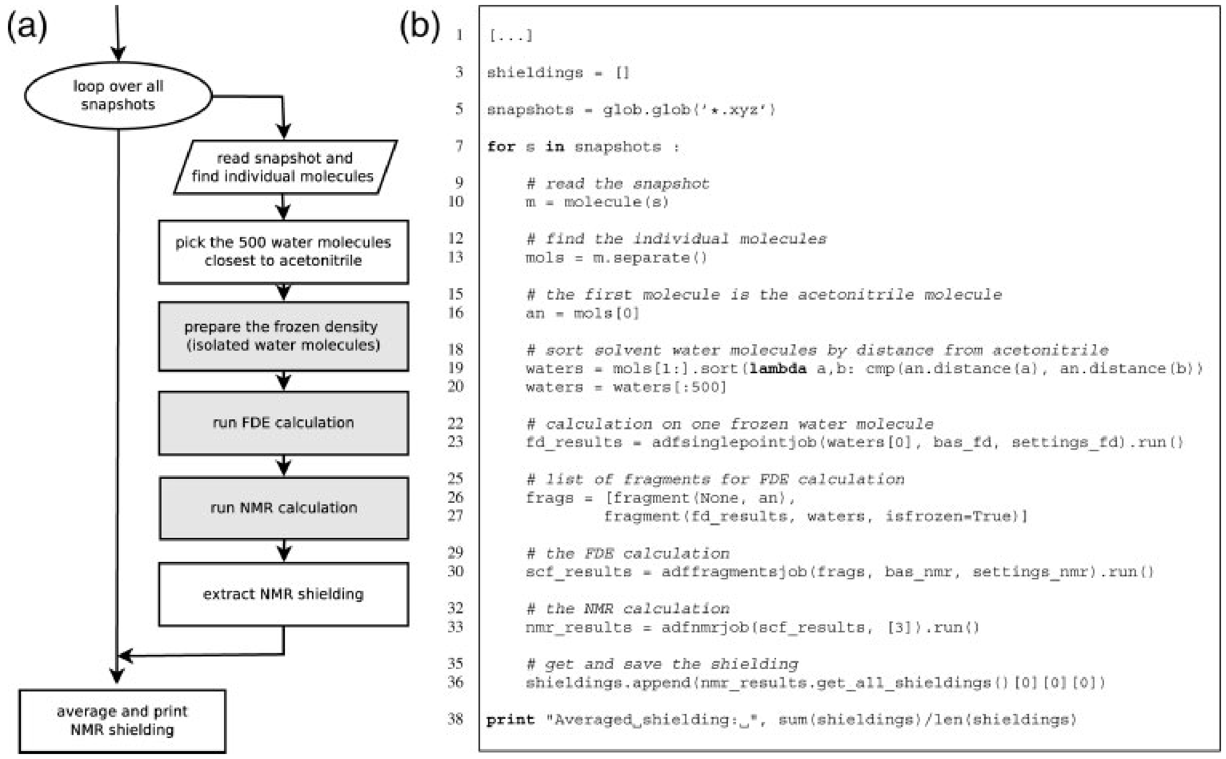
An important multiscale application of the FDE scheme is the calculation of solvent effects on molecular properties (e.g., electronic excitation energies,[29,30] ESR hyperfine coupling constants,[53] or NMR shieldings[31]). Such calculations employ the sequential molecular dynamics followed by quantum-mechanics calculations (S-MD/QM) strategy.[54] This strategy is illustrated for the calculation of the NMR shielding of acetonitrile in water in Figure 5, together with a simplifed version of the PYADF input file used for such calculations in ref. 31.



```
[...]

# producing density cube-files for plotting
cgrid = cubegrid(supermol)

dens_supermol = supermol_results.get_density(cgrid)
dens_supermol.get_cubfile('supermol.cub')

dens_fde = fde_results.get_density(cgrid)
dens_fde.get_cubfile('fde.cub')

dens_diff = dens_supermol - dens_fde
dens_diff.get_cubfile('diffdens.cub')


# numerical integration of difference density
agrid = adfgrid(supermol_results)

dens_supermol = supermol_results.get_density(agrid)
dens_fde = fde_results.get_density(agrid)

dens_diff = dens_supermol - dens_fde

print dens_diff.integral(lambda x: abs(x))
```

$\rho_{\mathrm{supermol}}(\boldsymbol{r})$
$\rho_{\mathrm{FDE}}(\boldsymbol{r})$
$\rho_{\mathrm{diff}}(\boldsymbol{r}) = \rho_{\mathrm{FDE}}(\boldsymbol{r}) - \rho_{\mathrm{supermol}}(\boldsymbol{r})$
$\int |\rho_{\mathrm{diff}}(\boldsymbol{r})|\, \mathrm{d}\boldsymbol{r} = 0.44$

Figure 4. A PyADF input file for analyzing the densities from FDE calculations along with isosurface plots of the (difference) densities for BH_3NH_3. Isosurface plots have been prepared with VMD.[49]



(a)
loop over all snapshots
read snapshot and find individual molecules
pick the 500 water molecules closest to acetonitrile
prepare the frozen density (isolated water molecules)
run FDE calculation
run NMR calculation
extract NMR shielding
average and print NMR shielding
(b)

```
[...]

shieldings = []

snapshots = glob.glob('*.xyz')

for s in snapshots :

    # read the snapshot
    m = molecule(s)

    # find the individual molecules
    mols = m.separate()

    # the first molecule is the acetonitrile molecule
    an = mols[0]

    # sort solvent water molecules by distance from acetonitrile
    waters = mols[1:].sort(lambda a,b: cmp(an.distance(a), an.distance(b))
    waters = waters[:500]

    # calculation on one frozen water molecule
    fd_results = adfsinglepointjob(waters[0], bas_fd, settings_fd).run()

    # list of fragments for FDE calculation
    frags = [fragment(None, an),
             fragment(fd_results, waters, isfrozen=True)]

    # the FDE calculation
    scf_results = adffragmentsjob(frags, bas_nmr, settings_nmr).run()

    # the NMR calculation
    nmr_results = adfnmrjob(scf_results, [3]).run()

    # get and save the shielding
    shieldings.append(nmr_results.get_all_shieldings()[0][0][0])

print "Averaged␣shielding:␣", sum(shieldings)/len(shieldings)
```

Figure 5. (a) Flowchart of the workflow used for calculating the solvent effect on the NMR shieldings using FDE. The steps in gray boxes stand for the calculations performed with an external program. (b) A PyADF input file for realizing this workflow.

First, using periodic boundary conditions, a classical molecular dynamics simulation of an acetonitrile molecule in water is performed. From this simulation, a sufficiently large number of snapshots are extracted. For each of these snapshots, a calculation of the NMR shielding is then performed for the solute molecule surrounded by the 500 nearest solvent molecules. Since a supermolecular calculation of molecular properties would not be feasible for such large systems, the FDE scheme is used in this step. The calculation is performed for the solute molecule as the active subsystem embedded in a frozen environment. For this solvent environment, an approximate electron density obtained by adding the densities calculated for isolated water molecules is used. Finally, the NMR shieldings calculated for each snapshot are averaged.

In the loop over all snapshots, all xyz-files from the current directory are used (line 5–7, see Fig. 5b). These xyz-files of the snapshots have to be generated using the molecular dynamics program. For each snapshot, the `separate` method of the `molecule` class is used to divide the snapshot into individual molecules, which are returned as a list `mols` (line 13). In this example, the first molecule in the list is the acetonitrile molecule (line 16), but if this is not the case it would be easy to add code for identifying it in this list. The remaining water molecules are then sorted by their distance to the acetonitrile molecule (line 19) and the 500 closest ones are chosen (line 20). In lines 25–27, the list of fragments for the FDE calculation is set up: The acetonitrile is the active fragment, and the list of the 500 water solvent molecules is used to create frozen fragments, each with the density of an isolated water molecule. In the present case, the classical molecular dynamics simulation used a water model with a fixed geometry. Therefore, a single frozen density calculated for one isolated water molecule (line 23) can be used for all 500 solvent molecules. If the geometries of the water molecules differ, it is possible to introduce a loop over all frozen solvent molecules, in which a specific frozen density is calculated for each of them. Similarly, it would also be possible to update the densities of some of the solvent molecules in freeze-and-thaw iterations by choosing the appropriate options for these fragments. In the remainder of the input, the FDE calculation and the NMR shielding calculation are performed, and after the loop over all snapshots is finished the calculated shieldings are averaged.

Subsystem-DFT for Proteins

The currently available kinetic-energy functionals are not suitable to apply the FDE scheme to subsystems connected by covalent bonds.[51,52] However, this problem can be circumvented by using a more general partitioning, as it was initially proposed in the molecular fractionation with conjugate caps (MFCC) method.[55–57] This partitioning is shown in Figure 6a. Instead of partitioning a covalent bond directly, caps are introduced to terminate the dangling bonds between the fragments. The auxiliary molecule obtained by joining the caps is then subtracted again. Recently, two of us showed how such a partitioning can be applied within the FDE scheme and demonstrated that this 3-partition FDE (3-FDE) scheme presents an efficient method for a subsystem treatment of proteins.[32]

In ref. 32 the 3-FDE scheme was applied to the protein ubiquitin, containing 76 amino acids, which was chosen because it is small enough for a supermolecular calculation on the full protein to be possible for comparison. However, already for such a small protein a scripting framework such as PyADF is invaluable. First, PyADF's molecule class provides methods for partitioning the protein, introducing the caps, and generating the corresponding cap fragments. For ubiquitin, the protein is partitioned into its 76 amino acids, each terminated by caps, resulting in the creation of 75 cap fragments (see Fig. 6). This partitioning can be performed automatically by PyADF. Besides cutting the protein at the peptide bonds, as was required for ubiquitin, an extension that allows for partitioning by cutting through disulfide bridges[58] is also available. In addition, more general partitionings using larger fragments (e.g., around an active center) are being developed.[59]

Once a partitioning is established, calculations on each of the isolated fragments have to be performed to obtain an initial density. This corresponds to the MFCC scheme and is handled in PyADF by the class `adfmfccjob`. The initial guess is then improved by 3-FDE calculations on each of the subsystems. These calculations are repeated iteratively until the freeze-and-thaw cycles are converged. All these individual calculations are automatically executed by PyADF's `adf3fdejob`. Finally, PyADF is indispensable for post-processing of the final results, e.g., for adding (and subtracting) the densities of all the individual subsystems.

WFT-in-DFT Embedding

The multiscale simulations discussed so far are solely based on DFT (DFT-in-DFT embedding). However, in many cases DFT with the currently available exchange–correlation functionals is not accurate enough. One prominent example is the failure of (adiabatic) time-dependent (TD) DFT for charge-transfer excitations.[60–64] Therefore, there is considerable interest in schemes for embedding a wavefunction based *ab initio* description in an environment treated with DFT (WFT-in-DFT embedding). This allows one to systematically improve the accuracy by employing a hierarchy of accurate WFT treatments for the subsystem of interest.

The FDE scheme can be extended to WFT-in-DFT embedding by using the FDE embedding potential (which has been derived in a DFT context) as an additional local one-electron potential in the WFT calculation.[25–28] It can be shown that using such an embedding potentials is formally exact (i.e., it is exact if the exact exchange–correlation and kinetic-energy functionals are used and the limit of an exact WFT description is reached).[65,66]

In ref. 33, three of us proposed a simplified scheme for the calculation of local excitation energies with WFT-in-DFT embedding. In many cases in which TD-DFT fails (such as in the case of charge-transfer excitations), the density obtained for the ground-state is still rather accurate. Therefore, it is reasonable to determine the embedding potential in a DFT-in-DFT embedding calculation — possibly using freeze-and-thaw cycles for (parts of) the environment — and to import the resulting DFT-in-DFT embedding potential in the WFT calculations. Such a scheme was applied in ref. 33 to investigate local excitations of a neptunyl ion (NpO_2^{2+}) embedded as a defect in a $Cs_2UO_2Cl_4$ crystal.

The workflow corresponding to this simplified scheme is shown in Figure 7a. After an initial frozen density is determined (typically requiring several calculations on the fragments constituting the environment), a DFT-in-DFT embedding calculation is performed. The environment density (or part of it) is then updated iteratively in

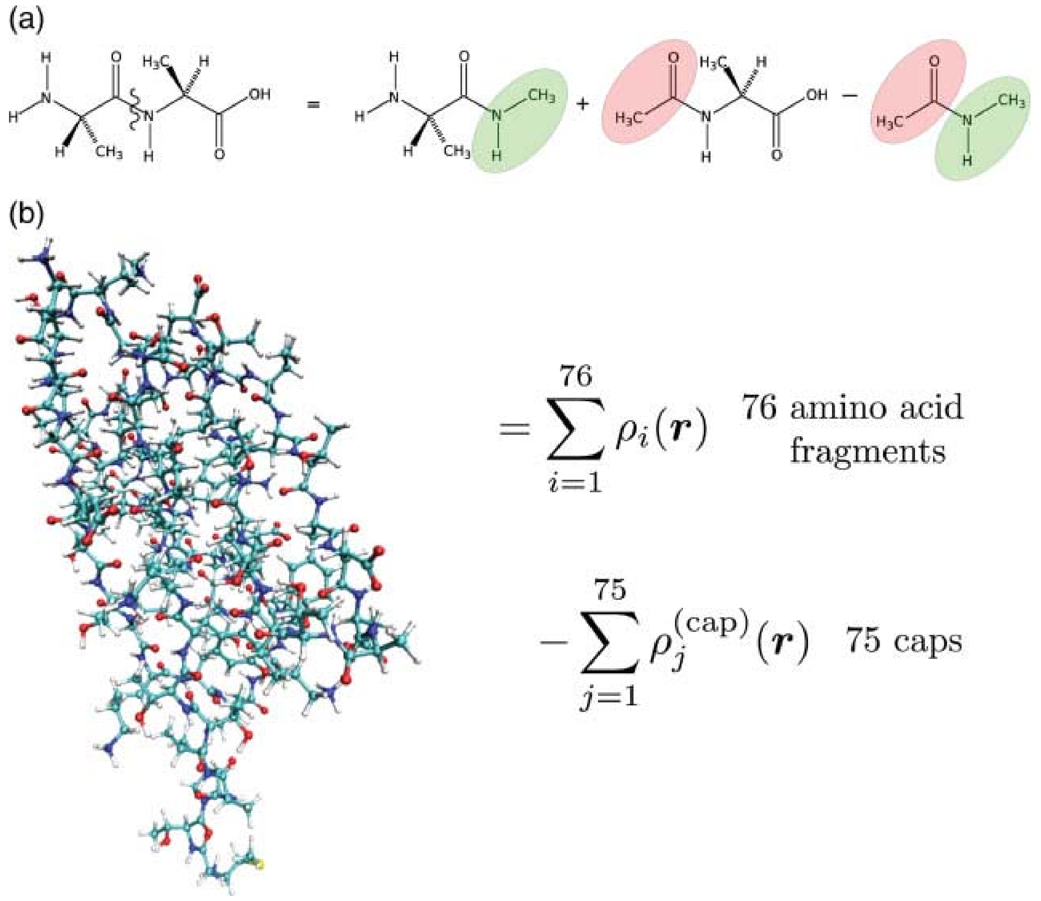

(a)
(b)
$= \sum_{i=1}^{76} \rho_i(\boldsymbol{r})$ 76 amino acid fragments
$- \sum_{j=1}^{75} \rho_j^{(\mathrm{cap})}(\boldsymbol{r})$ 75 caps

Figure 6. (a) Partitioning employed in subsystem-DFT calculations for proteins illustrated for dialanine. (b) For the calculations on ubiquitin in ref. 32, the protein is partitioned into its 76 amino acids and 75 cap fragments.

freeze-and-thaw cycles. After these freeze-and-thaw cycles are converged, the values of the embedding potential at the points of the numerical integration grid are read from ADF's binary results files and exported to a text file. This embedding potential as well as the coordinates and weights of the numerical integration grid itself are then imported in the WFT code. As soon as a WFT job is requested and is passed the results object of a DFT-in-DFT embedding calculation, PYADF internally takes care of creating the necessary files (input files, imported potential, etc.) and of passing them to the WFT code. Currently, DIRAC10 as well as locally modified versions of DALTON and NWCHEM support such an import of an embedding potential.

The drawback of the simplified scheme of ref. 33 is that the DFT ground-state density is used to determine the embedding potential, not the possibly more accurate ground-state density from the WFT calculation. In those cases where DFT fails to provide an acceptable ground-state density, a more complete WFT-in-DFT embedding scheme is needed. The workflow of such a scheme is presented in Figure 7b.

After a first DFT-in-DFT embedding calculation on the subsystem of interest is performed, the embedding potential is again exported and used in the WFT calculation. The values of the density, its gradient and Laplacian, and of the Coulomb potential generated by the density are then exported on the numerical integration grid. Such an export is currently supported by a development version of DIRAC for HF, DFT, and MP2 calculations. PYADF then takes care of converting the XML files written by DIRAC to the binary format that can be imported by ADF. Then, the environment density is updated in ADF, using the WFT density for the active subsystem and with this new environment density, a new embedding potential is exported. This is repeated iteratively, until these freeze-and-thaw iterations are converged. For such WFT-in-DFT calculations, PYADF provides the class `wftindftjob`, that implements all the steps in the workflow of Figure 7b.

Conclusions and Outlook

PYADF is a scripting framework for quantum chemistry that can be used for automating quantum-chemical workflows. This is achieved by providing job classes, which can be used to set up and execute different types of quantum-chemical calculations, and results classes that are returned when running a job and which can be used to extract the results of the calculations. In addition, PYADF offers powerful

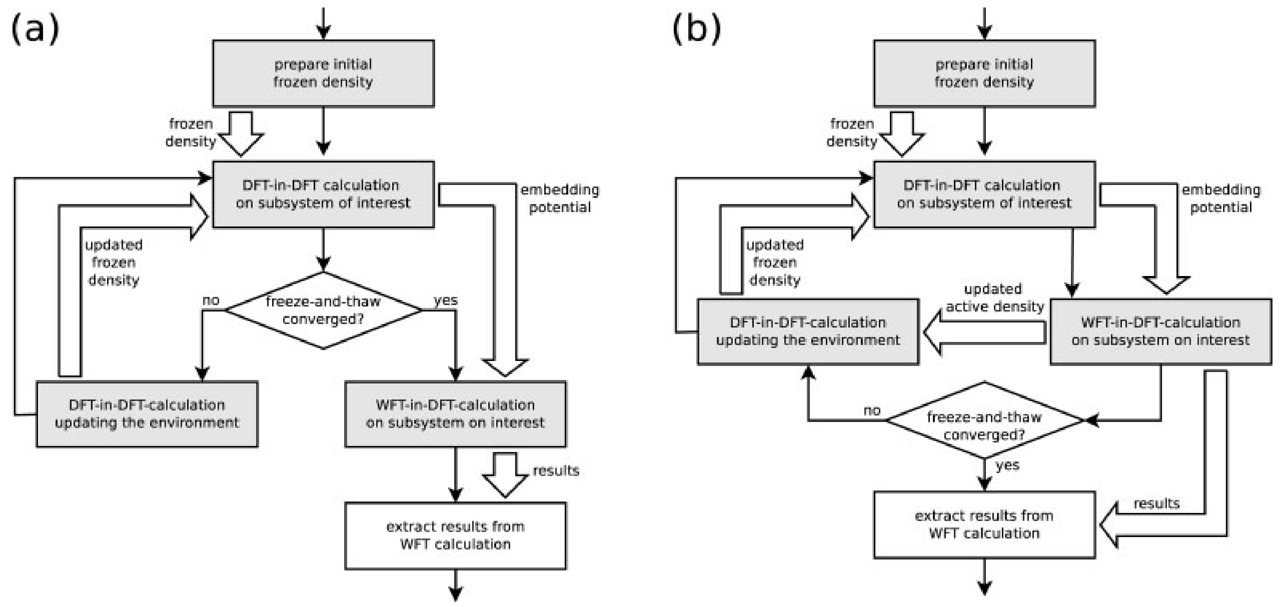

(a)
prepare initial frozen density
frozen density
DFT-in-DFT calculation on subsystem of interest
embedding potential
updated frozen density
no
freeze-and-thaw converged?
yes
DFT-in-DFT-calculation updating the environment
WFT-in-DFT-calculation on subsystem on interest
results
extract results from WFT calculation
(b)
prepare initial frozen density
frozen density
DFT-in-DFT calculation on subsystem of interest
embedding potential
updated frozen density
updated active density
DFT-in-DFT-calculation updating the environment
WFT-in-DFT-calculation on subsystem on interest
no
freeze-and-thaw converged?
yes
results
extract results from WFT calculation

Figure 7. The workflow in WFT-in-DFT embedding calculations. The steps in gray boxes stand for the calculations performed with an external program. The large arrows indicate results that are passed between the different tasks. (a) The simplified scheme of ref. 33, in which the embedding potential from a DFT-in-DFT embedding calculation is imported in the WFT calculation. (b) A freeze-and-thaw WFT-in-DFT embedding setup, in which the embedding potential is updated iteratively, using the density from the WFT calculation.

features for reading and manipulating molecular coordinates and for the post-processing of the results (e.g., for handling orbitals, densities, and potentials available on a numerical grid). Since the full power of the Python language is available in PyADF input files, these building blocks can be combined into complicated workflows.

These features are particularly useful in multiscale simulations. We have illustrated this using examples of applications of PyADF from the recent research in our groups, such as the assessment of kinetic-energy functionals in the FDE scheme, the explicit treatment of solvent effects on molecular properties, subsystem-DFT calculations for proteins, and WFT-in-DFT embedding calculations. Other recent and ongoing work in our groups also relies on PyADF. For instance, the code for performing adaptive QM/MM molecular dynamics simulations developed by Bulo et al.[67,68] uses PyADF for executing the QM calculations.

PyADF is under active development. Currently, it supports ADF, DALTON, DIRAC, and NWCHEM calculations, even though the types of calculations available for each program package differ. We are working on extending these existing interfaces to support additional types of calculations. In this context, we also plan to add a unified interface to similar types of calculations performed with different program packages by providing an additional layer of job classes (e.g., `singlepointjob`, which will then execute either `adfsinglepointjob`, `daltonsinglepointjob`, or `diracsinglepointjob`). Furthermore, interfaces to additional programs, most importantly TURBOMOLE (using the MOLECONTROL environment), are being developed.

Another important topic we are considering is parallelization. Currently, PyADF executes all external tasks sequentially, even though each task can use multiple processors if the program packages support parallelization. A parallel version of PyADF to allow for coarse-grain parallelization of potentially concurrent tasks in a workflow is currently being developed. To achieve this, external tasks will not be executed immediately when a job's `run` method is called, but instead be gathered into a "job queue." Only when the results of one job are requested, the accumulated tasks will be executed in parallel. This way, an "automatic parallelization" can be achieved, in which PyADF input files are still written in a sequential way, and the problem of determining the dependencies between the different jobs and of executing the external tasks in parallel is handled behind the scenes by the scripting framework.

PyADF version 1.0 is available free of charge at `http://www.pyadf.org` under the GNU General Public License (GPL). On the website, one can also find an extensive documentation and examples of input files, including those for the examples discussed here.

Acknowledgments

The authors thank Drew A. McCormack (Amsterdam) for many inspiring discussions.

References

1. Cramer, Ch. J. Essentials of Computational Chemistry, Wiley: New York, 2002.

2. Heine, T.; Joswig, J.-O.; Gelessus, A. Computational Chemistry Workbook: Learning Through Examples; Wiley-VCH: Weinheim, 2009.
3. Boese, A. D.; Oren, M.; Atasoylu, O.; Martin, J. M. L.; Kállay, M.; Gauss, J. J Chem Phys 2004, 120, 4129.
4. Curtiss, L. A.; Redfern, P. C.; Raghavachari, K. J Chem Phys 2007, 126, 084108.
5. van Zeist, W.-J.; Fonseca Guerra, C.; Bickelhaupt, F. M. J Comput Chem 2008, 29, 312.
6. Valiev, M.; Bylaska, E. J.; Govind, N.; Kowalski, K.; Straatsma, T. P.; Van Dam, H. J. J.; Wang, D.; Nieplocha, J.; Apra, E.; Windus, T. L.; de Jong, W. A. Comput Phys Commun 2010, 181, 1477.
7. Environmental Molecular Sciences Laboratory (EMSL) at Pacific Northwest National Laboratory (PNNL), NWCHEM 6.0 Python interface, 2010. Available at http://www.nwchem-sw.org/index.php/Python, Accessed on 6 March 2010.
8. Werner, H.-J.; Knowles, P. J.; Manby, F. R.; Schütz, M.; Celani, P.; Knizia, G.; Korona, T.; Lindh, R.; Mitrushenkov, A.; Rauhut, G.; Adler, T. B.; Amos, R. D.; Bernhardsson, A.; Berning, A.; Cooper, D. L.; Deegan, M. J. O.; Dobbyn, A. J.; Eckert, F.; Goll, E.; Hampel, C.; Hesselmann, A.; Hetzer, G.; Hrenar, T.; Jansen, G.; Köppl, C.; Liu, Y.; Lloyd, A. W.; Mata, R. A.; May, A. J.; McNicholas, S. J.; Meyer, W.; Mura, M. E.; Nicklaß, A.; Palmieri, P.; Pflüger, K.; Pitzer, R.; Reiher, M.; Shiozaki, T.; Stoll, H.; Stone, A. J.; Tarroni, R.; Thorsteinsson, T.; Wang, M.; Wolf, A. MOLPRO, version 2010.1, a package of ab initio programs. Available at: http://www.molpro.net, 2010.
9. Baerends, E. J.; Ziegler, T.; Autschbach, J.; Bashford, D.; Bérces, A.; Bickelhaupt, F. M.; Bo, C.; Boerrigter, P. M.; Cavallo, L.; Chong, D. P.; Deng, L.; Dickson, R. M.; Ellis, D. E.; van Faassen, M.; Fan, L.; Fischer, T. H.; Fonseca Guerra, C.; Ghysels, A.; Giammona, A.; van Gisbergen, S. J. A.; Götz, A. W.; Groeneveld, J. A.; Gritsenko, O. V.; Grüning, M.; Gusarov, S.; Harris, F. E.; van den Hoek, P.; Jacob, C. R.; Jacobsen, H.; Jensen, L.; Kaminski, J. W.; van Kessel, G.; Kootstra, F.; Kovalenko, A.; Krykunov, M. V.; van Lenthe, E.; McCormack, D. A.; Michalak, A.; Mitoraj, M.; Neugebauer, J.; Nicu, V. P.; Noodleman, L.; Osinga, V. P.; Patchkovskii, S.; Philipsen, P. H. T.; Post, D.; Pye, C. C.; Ravenek, W.; Rodríguez, J. I.; Ros, P.; Schipper, P. R. T.; Schreckenbach, G.; Seldenthuis, J. S.; Seth, M.; Snijders, J. G.; Solà, M.; Swart, M.; Swerhone, D.; te Velde, G.; Vernooijs, P.; Versluis, L.; Visscher, L.; Visser, O.; Wang, F.; Wesolowski, T. A.; van Wezenbeek, E. M.; Wiesenekker, G.; Wolff, S. K.; Woo, T. K.; Yakovlev, A. L. ADF, Amsterdam density functional program, 2010. Available at: http://www.scm.com, Accessed on 6 March 2010.
10. te Velde, G.; Bickelhaupt, F. M.; Baerends, E. J.; Fonseca Guerra, C.; van Gisbergen, S. J. A.; Snijders, J. G.; Ziegler, T. J Comput Chem 2001, 22, 931.
11. Scientific Computing and Modelling, ADFPREP and ADFREPORT, 2008. Available at: http://www.scm.com/Doc/ Doc2010.01/ ADF/Utilities, Accessed on 6 March 2010.
12. Doll, C.; Schäfer, A.; Sittel, F., MOLECONTROL, Available at: www.turbomole.com, Accessed on 6 March 2010, 2010.
13. Ahlrichs, R.; et al. TURBOMOLE. Available at: http://www.turbomole.com, Accessed on 6 March 2010.
14. Ahlrichs, R.; Bär, M.; Häser, M.; Horn, H.; Kölmel, Ch. Chem Phys Lett 1989, 162, 165.
15. Kuhn, T.; Willighagen, E.; Zielesny, A.; Steinbeck, Ch. BMC Bioinf 2010, 11, 159.
16. Tiwari, A.; Sekhar, A. K. T. Comput Biol Chem 2007, 31, 305.
17. Gomes, A.; Merzky, A.; Visscher, L. In Proceedings of the Computational Science – ICCS 2006, Pt. 3, Lecture Notes in Computer Science; Springer: Berlin, 2006; pp. 97–104.
18. Sudholt, W.; Altintas, I.; Baldridge, K. In Proceedings of the Computational Science – ICCS 2006, Pt. 3, Alexandrov, V. N.; VanAlbada, G. D.; Sloot, P. M. A.; Dongarra, J., Eds.; Lecture Notes in Computer Science; Springer-Verlag: Berlin, 2006; pp. 69–76.
19. Koehler, M.; Ruckenbauer, M.; Janciak, I.; Benkner, S.; Lischka, H.; Gansterer, W. N. In Proceedings of the Computational Science and its Applicatiosn - ICCSA 2010, pt. 4, Taniar, D.; Gervasi, O.; Murgante, B.; Pardede, E.; Apduhan, B. O., Eds.; Lecture Notes in Computer Science; Springer-Verlag: Berlin, 2010; pp. 13–28.
20. Wang, J.; Korambath, P.; Kim, S.; Johnson, S.; Jin, K.; Crawl, D.; Altintas, I.; Smallen, S.; Labate, B.; Houk, K. N. In Proceedings of the ICCS 2010 - International Confercence on Computational Science, Procedia Computer Science; Elsevier: Amsterdam, 2010; pp. 1169–1178.
21. Sanna, N.; Castrignano, T.; De Meo, P. D.; Carrabino, D.; Grandi, A.; Morelli, G.; Caruso, P.; Barone, V. Theor Chem Acc 2007, 117, 1145.
22. Wesolowski, T. A.; Warshel, A. J Phys Chem 1993, 97, 8050.
23. Senatore, G.; Subbaswamy, K. R. Phys Rev B 1986, 34, 5754.
24. Cortona, P. Phys Rev B 1992, 46, 2008.
25. Govind, N.; Wang, Y. A.; da Silva, A. J. R.; Carter, E. A. Chem Phys Lett 1998, 295, 129.
26. Govind, N.; Wang, Y. A.; Carter, E. A. J Chem Phys 1999, 110, 7677.
27. Klüner, T.; Govind, N.; Wang, Y. A.; Carter, E. A. Phys Rev Lett 2001, 86, 5954.
28. Klüner, T.; Govind, N.; Wang, Y. A.; Carter, E. A. J Chem Phys 2002, 116, 42.
29. Neugebauer, J.; Louwerse, M. J.; Baerends, E. J.; Wesolowski, T. A. J Chem Phys 2005, 122, 094115.
30. Neugebauer, J.; Jacob, Ch. R.; Wesolowski, T. A.; Baerends, E. J. J Phys Chem A 2005, 109, 7805.
31. Bulo, R. E.; Jacob, Ch. R.; Visscher, L. J Phys Chem A 2008, 112, 2640.
32. Jacob, Ch. R.; Visscher, L. J Chem Phys 2008, 128, 155102.
33. Gomes, A. S. P.; Jacob, Ch. R.; Visscher, L. Phys Chem Chem Phys 2008, 10, 5353.
34. van Rossum, G.; et al., PYTHON. Available at: http://www.python.org, Accessed on 6 March 2010.
35. McCormack, D. A. Scientific Scripting with Python, 1st ed., lulu.com, Raleigh, NC, USA, 2009. Accessed on 6 March 2010.
36. Langtangen, H. P. Python Scripting for Computational Science, 3rd ed.; Springer, Berlin, 2008.
37. The Open Babel package. Available at: http://openbabel.sourceforge.net/ Accessed on 6 March 2010.
38. Guha, R.; Howard, M. T.; Hutchison, G. R.; Murray-Rust, P.; Rzepa, H.; Steinbeck, Ch.; Wegner, J.; Willighagen, E. L. J Chem Inf Model 2006, 46, 991.
39. O'Boyle, N.; Morley, C.; Hutchison, G. Chem Cent J 2008, 2, 5.
40. DALTON, a molecular electronic structure program, Release 2.0. Available at: http://www.kjemi.uio.no/software/dalton/dalton.html, Accessed on 6 March 2010, 2005.
41. Saue, T.; Visscher, L.; Jensen, H. J. Aa. DIRAC, a relativistic ab initio electronic structure program, Release DIRAC10. Available at: http://dirac.chem.vu.nl, Accessed on 6 March 2010, 2010.
42. SMARTS — A Language for Describing Molecular Patterns. Available at: http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html, Accessed on 6 March 2010.
43. Wesolowski, T. A. In Computational Chemistry: Reviews of Current Trends, vol. 10; Leszczynski, J., Ed.; World Scientific: Singapore, 2006; pp. 1–82.
44. Jacob, Ch. R.; Neugebauer, J.; Visscher, L. J Comput Chem 2008, 29, 1011.
45. Wesolowski, T. A.; Weber, J. Chem Phys Lett 1996, 248, 71.
46. Wang, Y. A.; Carter, E. A. In Theoretical Methods in Condensed Phase Chemistry, Schwartz, S. D., Ed.; Kluwer: Dordrecht, 2000; pp. 117–184.
47. Götz, A. W.; Beyhan, S. M.; Visscher, L. J Chem Theory Comput 2009, 5, 3161.
48. Beyhan, S. M.; Götz, A. W.; Jacob, Ch. R.; Visscher, L. J Chem Phys 2010, 132, 044114.

49. Humphrey, W.; Dalke, A.; Schulten, K. J Mol Graphics 1996, 14, 33.
50. Lembarki, A.; Chermette, H. Phys Rev A 1994, 50, 5328.
51. Fux, S.; Kiewisch, K.; Jacob, Ch. R.; Neugebauer, J.; Reiher, M. Chem Phys Lett 2008, 461, 353.
52. Fux, S.; Jacob, Ch. R.; Neugebauer, J.; Visscher, L.; Reiher, M. J Chem Phys 2010, 132, 164101.
53. Neugebauer, J.; Louwerse, M. J.; Belanzoni, P.; Wesolowski, T. A.; Baerends, E. J. J Chem Phys 2005, 123, 114101.
54. Malaspina, T.; Coutinho, K.; Canuto, S. J Chem Phys 2002, 117, 1692.
55. Zhang, D. W.; Zhang, J. Z. H. J Chem Phys 2003, 119, 3599.
56. Gao, A. M.; Zhang, D. W.; Zhang, J. Z. H.; Zhang, Y. Chem Phys Lett 2004, 394, 293.
57. Mei, Y.; Zhang, D. W.; Zhang, J. Z. H. J Phys Chem A 2005, 109, 2.
58. Chen, X. H.; Zhang, D. W.; Zhang, J. Z. H. J Chem Phys 2004, 120, 839.
59. Kiewisch, K.; Jacob, Ch. R.; Visscher, L. to be published, 2011.
60. Tozer, D. J. J Chem Phys 2003, 119, 12697.
61. Dreuw, A.; Weisman, J. L.; Head-Gordon, M. J Chem Phys 2003, 119, 2943.
62. Gritsenko, O.; Baerends, E. J. J Chem Phys 2004, 121, 655.
63. Neugebauer, J.; Gritsenko, O.; Baerends, E. J. J Chem Phys 2006, 124, 214102.
64. Hieringer, W.; Görling, A. Chem Phys Lett 2006, 419, 557.
65. Wesolowski, T. A. Phys Rev A 2008, 77, 012504.
66. Khait, Y. G.; Hoffmann, M. R. J Chem Phys 2010, 133, 044107.
67. Bulo, R. E.; Ensing, B.; Sikkema, J.; Visscher, L. J Chem Theory Comput 2009, 5, 2212.
68. Nielsen, S. O.; Bulo, R. E.; Moore, P. B.; Ensing, B. Phys Chem Chem Phys 2010, 12, 12401.