JCIM JOURNAL OF CHEMICAL INFORMATION AND MODELING

Article

Cite This: J. Chem. Inf. Model. 2019, 59, 3191−3197

pubs.acs.org/jcim

QMflows: A Tool Kit for Interoperable Parallel Workflows in Quantum Chemistry

Felipe Zapata,[#,†] Lars Ridder,[†] Johan Hidding,[†] Christoph R. Jacob,[‡] Ivan Infante,[*,#,§] and Lucas Visscher[*,#]

[#]Division of Theoretical Chemistry, Faculty of Science, Vrije Universiteit Amsterdam, de Boelelaan 1083, 1081 HV Amsterdam, The Netherlands

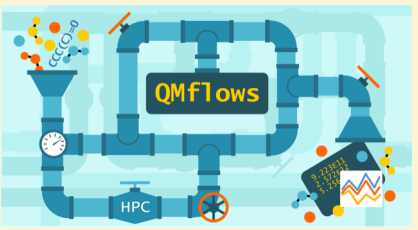
[†]Netherlands eScience Center, Science Park 140 (Matrix I), 1098 XG Amsterdam, The Netherlands

[‡]Institute of Physical and Theoretical Chemistry, TU Braunschweig, Gaußstraße 17, 38106 Braunschweig, Germany

[§]Department of Nanochemistry, Istituto Italiano di Tecnologia, Via Morego 30, 16163 Genova, Italy

Supporting Information

ABSTRACT: We present the QMflows Python package for quantum chemistry workflow automatization. QMflows allows users to write complex workflows in terms of simple Python scripts. It supports the development of interoperable workflows involving multiple quantum chemistry codes and executes them efficiently on large scale parallel computers. This open source library provides standardized interfaces to a number of quantum chemistry packages and can be easily extended to accommodate additional codes. QMflows features are described and illustrated with a number of representative applications.



INTRODUCTION

In the last years, new pathways to materials discovery that go beyond trial-and-error processes have been implemented with promising results. These paths are usually based on computer prediction algorithms that either find new materials by comparing structures from available databases storing hundreds of thousands of known structures or by exploratory algorithms that generate new materials without much previous knowledge.[1] Either way, these new materials are usually linked to specific desired properties that often need to be computed at the atomistic level and that represent further constraints in the search of novel structures. Such computed properties are usually carried out at the quantum chemical level, and knowledge and availability of the various computational tools are fundamental. Rational design based on computational modeling indeed often requires a combination of theoretical approaches and software and commonly involves composite workflows consisting of a large number of individual calculations.[2,3] Such workflows are well suited for parallel computing as it is much easier to exploit job-type parallelism than to parallelize each computational task individually. There are, however, two complications in exploiting job-type parallelism that should be considered. Complex workflows require the execution of several different program packages, which require different, program-specific input data, which in turn possibly depends on the results of preceding calculations. These dependencies between different tasks are often nontrivial and need to be taken into account in the parallel execution of workflows.

A well-known example of a computational workflow that benefits from automation and parallelization is running a benchmark study to assess the performance of a series of density functional theory (DFT) exchange-correlation functionals.[4−7] Another simple example is the geometry relaxation of a set of related molecules. More complicated applications involve multiple methods and are characterized by interdependencies between the calculations. An example is the prediction of the activation energy of several reactions by performing a multilevel calculation of reactants and transition states.[8]

Overall, three common tasks need to be automated: (i) the preparation of specific inputs for the quantum chemical (QM) software package(s) that is (are) used; (ii) submission of jobs to a queuing system on a supercomputer facility; and (iii) the collection and analysis of the output files, often involving postprocessing steps with yet other software tools. Commonly used *ad hoc* Unix shell or Python scripts for such purposes are often poorly transferable and difficult to maintain and extend. They furthermore require sufficient programming experience to attain good performance. In most research groups, the time available to tune and document such scripts is limited, resulting in a suboptimal use of computational and human resources.

To address these limitations of project-specific scripting, a number of generic solutions for the preparation and execution of QM workflows have been developed: PyADF,[9] Chemshell,[10] FireWorks,[11] Aiida,[12] ASE.[13] While being very helpful in the automation of input/output handling, many of these solutions have still limited abilities for parallel processing or require expert knowledge. Parallelization of the workflow often

Received: May 8, 2019
Published: June 19, 2019

DOI: 10.1021/acs.jcim.9b00384
J. Chem. Inf. Model. 2019, 59, 3191−3197

requires input from the user who should indicate the stages in the simulation that can be run in parallel. Another common limitation in currently available tools arises from the definition of the interfaces to the different QM packages. Usually, the interfaces are specific to the different supported program packages, and different interfaces are required for performing the same task (e.g., a DFT geometry optimization) with different program packages. This hinders the interoperable use of different codes in one workflow.

To facilitate automation and parallelization, while keeping maximum flexibility and limiting maintenance, we have developed the Python package QMflows for the automation of computational chemistry workflows. QMflows is built on top of the PLAMS library[14] that is designed to provide a general interface to QM simulation packages. QMflows furthermore incorporates the Noodles[15] library that is developed for dependency analysis and automatic parallel execution of workflows. We chose to implement the QMflows platform with Python, the most-used programming language for the automation of computations in chemistry. In this way, we aim at supporting both beginners and experienced Python programmers. Beginners will benefit from the simple and unobtrusive syntax to specify job dependencies, while more advanced users will be able to construct complex workflows.

QMflows' main objective is to allow users with only basic programming skills to build complex workflows that can be reused, adapted, and executed in high-performance computing environments. The most relevant features of our library are (i) manipulation of molecular objects to quickly generate and modify molecular structures, combined with interoperability between QM packages to facilitate use of multiprogram workflows; (ii) parallel execution of tasks defined in the workflow by automated analysis of task dependencies; (iii) recovery and restart in case of job failure; and (iv) a file manager for a posteriori retrieval and analysis of data.

A major objective of QMflows is to enable beginners to focus on the scientific aspects of the workflows (the definition of the molecules to be studied, the level of theory to be used for each task, and the molecular properties that are to be retrieved and analyzed) instead of solving technical issues related to specific program packages or to the parallel execution of the workflow. For this kind of use only a basic understanding of the Python scripting language is required, a skill that most computational chemists nowadays possess. Advanced users will be able to extend the existing functionality by programming new functions in Python with only minor adaptations to allow Noodles to perform an automatic dependency analysis. QMflows currently supports the ADF,[16] CP2K,[17] DIRAC,[18] GAMESS-US,[19] and ORCA[20] packages and is distributed as open source software under the LGPL-3.0 license.[21]

METHODS

QMflows has four major modules, as schematically depicted in Figure 1 and described in more detail below.

Module 1: Input Manipulation. The first step takes care of the manipulation of the molecular objects that are studied in the QM calculations. In QMflows, molecules are by default defined with the *Molecule* class of the PLAMS library. An instance of a *Molecule* object holds atoms and coordinates and can optionally store bond information as well. It can be created from the *xyz*, *mol*, and *pdb* data formats and has methods for adding and deleting atoms and for identifying, adding, and deleting bonds. For more complex structure manipulations, a dedicated *Molkit* library is available, which builds on 3D chemical structure functionality offered by the RDKit package.

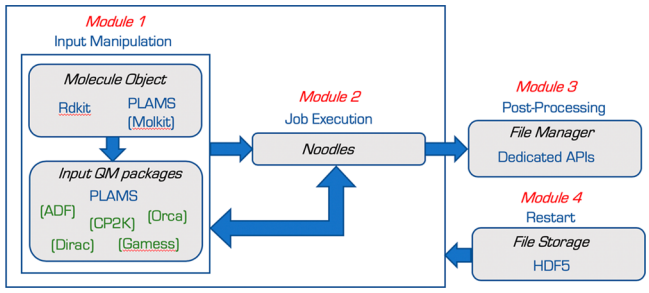


Figure 1. Schematic representation of the QMflows architecture.

The second step is dedicated to the preparation of the input for the individual QM calculations. Here we build on the input model adopted by the PLAMS library, which avoids the use of "hard-coded" interfaces to the QM packages. PLAMS makes use of the fact that most QM packages adopt a hierarchical input structure, consisting of input sections, subsections, and key-value pairs. PLAMS enables the representation of such input as a nested Python dictionary, which can be automatically translated into a correct input file for a specific package. This strategy allows PLAMS users to access the complete set of features offered by a given QM package and not only a subset covered by a predefined wrapper or interface. This design also minimizes maintenance costs of the interfaces as the addition of new features in a package does not require an update of the interface.

On top of the PLAMS interface, QMflows offers a set of *generic* key words and *templates* for easy access to functionalities that are provided by multiple QM packages. These *generic* key words make it possible to use the same input definitions for common features (e.g., basis set, DFT functional) when calling different QM codes and makes it easy to switch between two implementations of a particular method. The translation from generic to *specific* key word is stored as a dictionary that is easy to maintain and extend when the input of one or more of the packages changes. The translation from *generic* to *specific* key words is done automatically at run time, so for standard applications users can use identical run scripts for different quantum chemistry packages.

The *templates* consist of PLAMS settings for common computational chemistry tasks, such as structure optimization, transition state searches, and vibrational frequency calculations. These settings objects can be modified and merged by the user, who can also add *generic* and code-*specific* additional input for a particular task.

An example of a *template* for a geometry optimization is provided below:

```
1. from scm.plams import Molecule
2. from qmflows import (adf,run,templates)

3. molecule = Molecule('files/uranyl.xyz')
4. inp = templates.geometry

5. job = adf(inp, molecule)
6. result = run(job)

7. print(result.molecule)
```

In *line 3* we define a molecule object by reading coordinates from an external *xyz* file. Then we use a generic template to define the settings for the type of calculation (a geometry

DOI: 10.1021/acs.jcim.9b00384
J. Chem. Inf. Model. 2019, 59, 3191−3197

optimization). Note that we do not yet define the QM package. The next line calls the ADF program with the settings and the molecule object as arguments. In this case, the *template* for an ADF geometry optimization makes use of a nonscalar calculation with a DZP basis set, a BLYP exchange-correlation functional, and a frozen core MEDIUM. The advantage of this approach is that, without rewriting any input, we may carry out the geometry optimization with another code, e.g. CP2k, by simply replacing the word *adf* in *line 5* with *cp2k*. In the latter case, the DZP is switched in a basis set of similar quality, a DZVP, and a GTH pseudpotential for the core electrons. All the *templates* set for a given task (scf, geometry optimization, etc.) have been wrapped for each QM code by applying similar, albeit not equal, input parameters. More details on these template input parameters are provided in the online documentation of QMflows.

Below, we also illustrate how a user may modify the basis set and the DFT exchange-correlation functional of the *template* using *generic* key words:

```
1. from scm.plams import Molecule
2. from qmflows import (adf,run,templates)

3. molecule = Molecule('files/uranyl.xyz')
4. inp = templates.geometry

# Generic Settings
# Overriding basisset default
5. inp.basis ='TZ2P'

# Overriding xc default
6. inp.functional = 'bp86'

7. result = run(adf(inp, molecule))

8. print(result.molecule)
```

Before calling the *adf* function in *line 7*, in *lines 5 and 6*, the variables *basis* and *functional* of the default settings object are redefined by the user. These *generic* key words can be used with most QM codes and can be set to program-independent values. QMflows will translate these to the corresponding package-specific key words and values once the actual QM code to be used for the task is known. The current list of generic key words is presented in Table S1, and it will be expanded in the future for enhanced flexibility.

The definition of *templates* and *generic* key words that can be used by multiple packages does not prohibit the use of options that are only available in a specific QM code. In this case, the user can extend the default settings for a given calculation by adding *generic* key words (as demonstrated above) and/or also package-*specific* key words in a dedicated subsection of the input settings indicated by the key word "specific" and the name of the package. This subsection should follow the input tree structure of the specific QM package. An example of this is illustrated in the code snippet below:

```
1. from scm.plams import Molecule
2. from qmflows import (adf,run,templates)

3. molecule = Molecule('files/uranyl.xyz')
4. inp = templates.geometry

# Generic Settings that override the template settings
5. inp.basis = 'TZ2P'
6. inp.functional = 'bp86'

# Specific Settings
7. inp.specific.adf.excitations.lowest = 10
8. inp.specific.adf.relativistic = "scalar Zora"
9. inp.specific.adf.scf.iterations = 100

10. result = run(adf(inp, molecule))

11. print(result.molecule)
```

Until *line 6* the workflow resembles exactly the previous example. In *lines 7−9*, however, we assign three extra key words specific to the ADF code. The first calls the excitations module to perform a time-dependent DFT (TDDFT) calculation computing the lowest 10 roots; the second calls the ZORA module to include scalar relativistic effects; the third overwrites the default setting for the number of iterations employed in the SCF procedure. Only for these specific key words the user should look up the input tree structure and key word definition used by the specific QM package.

Module 2: Workflow Execution and Communication between Jobs. The task of the second module of the package is to efficiently execute the user-defined Python workflows. With the Noodles framework, dependencies between the tasks defined in the QMflows Python script are automatically detected. The idea behind this library is to postpone the execution of tasks to allow for dependency analysis. Rather than immediately executing the Python code found in a user script, Noodles will first construct a dependency graph in which tasks (for example the optimization of a molecular structure or a calculation of NMR shieldings for a given molecule) are represented by nodes connected via edges to represent the dependencies between these tasks (for example the NMR calculation needs the structure from the geometry optimization task). The tasks are always implemented as scheduled functions which return "promised objects" rather than results. The actual results of the workflow are obtained using an explicit run statement at the end of the workflow. This run statement triggers the evaluation of the dependencies and the generation and execution of the actual jobs. The advantage of this approach is that the user script looks almost like a normal Python script, while in the background Noodles takes care of those calls to QMflows tasks that do not depend on each other and are executed in parallel. This is highlighted in the following script where the call to Noodles is delayed until line 12.

```
1.  from scm.plams import Molecule
2.  from noodles import gather
3.  from qmflows import (orca, dftb, run, templates)

# ethanol, N-Ethylmethylamine
4.  files = ['ethylene.xyz', 'aniline.xyz', 'caffeine.xyz']
5.  molecules = [Molecule(f, 'xyz') for f in files]

# Default geometry optimization parameters
6.  inp = templates.geometry

# Initial Geometry optimization guess with DFTB (ADF)
7.  dftb_jobs = [dftb(inp, mol) for mol in molecules]

# Optimization refinement with Orca
8.  inp.functional = "pbe"
9.  inp.basis = '"def2-svp"'
10. orca_jobs = [orca(inp, job.molecule) for job in dftb_jobs]

# Extract the resulting results
11. opt_molecules = [job.molecule for job in orca_jobs]
12. results = run(gather(*opt_molecules))
13. for mol in results:
14.     print(mol)
```

This makes it possible to optimally use the options available for parallel execution that compute servers may offer. For this purpose, Noodles comes with several back ends for different common architectures. For small workflows, the user can simply reserve a number of threads on his or her local computer, while larger workflows can be run via job schedulers on a compute cluster or supercomputer. This flexibility allows users to easily scale workflows from the testing phase to the production phase. A more detailed description of the Noodles

DOI: 10.1021/acs.jcim.9b00384
J. Chem. Inf. Model. 2019, 59, 3191−3197

library, which is used for non-QM workflows as well, is available online.[15]

Module 3: Job Recovery. To enable extension of the set of molecules to be studied or the types of analysis to be carried out, QMflows is from the outset designed to allow for restarts. The workflow and dependencies are stored in a database that is updated with information on successfully completed calculations during the execution of the workflow. When the run command is invoked, noodles traverses the graph of job dependencies and checks against the database for a reference to the job results; if such reference does not exist, then the job is executed, and the resulting output metainformation is stored in the database.

If the execution of the workflow is stopped by the user or fails for technical reasons, the generated database with metadata can be used to restart the workflows. Noodles will walk through the dependencies tree in the same way as when started from scratch but will query the database for already existing results and execute only the tasks that were not yet successfully completed. This makes the restart procedure very straightforward for the user who will need only to resubmit the original job scheduler script (Slurm or PBS) without modifications.

Module 4: Output Postprocessing. When tasks involve different software packages and are run on different resources, data transfer between tasks involves data conversion as well as communication between the different computational resources. To allow for efficient and general data handling, QMflows distinguishes between primary data that results from the execution of a particular task, output data that is to be used in a next step as input, and metadata that describes the type of tasks that was executed. The latter is stored in a database that is also used to monitor the progress of the workflow execution. The primary data is kept in the native format of the quantum chemistry program that was used and at the location where this task was executed. The subset of output data that is needed as input for a next task is converted to the portable HDF5 format for ease of communication between resources. In case primary data of a particular step is not needed for analysis of the results, users may choose to have these data automatically deleted after the task is completed. Otherwise the data of all steps is stored in its original form at the local computing node to allow for a posteriori analysis.

In the simple code snippets presented earlier, the results of the calculations were simply printed to screen using Python commands. In real applications, in which the amount of data produced is much larger, it is desirable to store data to a file and retrieve the subset of data of interest when needed. In QMflows, we have implemented parsers to retrieve the most essential information about a system from output or checkpoint file(s). This currently includes the coordinates of the optimized geometry, the total energy of the system, dipole moments, gradient, excitation energies, and Hessians matrices. For other data, the user can use the parsers offered by the QM code or write its own parser to retrieve the desired output data.

EXAMPLES

To demonstrate QMflows capabilities, we provide some use cases with workflows of increasing complexity.

Example 1: Multilevel Geometry Optimization and Excitation Spectrum. A common starting point for a high-level quantum chemistry calculation of molecular properties is a structure optimized at a lower level of theory. For example, it is quite common to perform a single point post-Hartree−Fock calculation on a DFT-optimized structure. As also DFT optimization can be time-consuming for larger molecules it is preferable to perform a preoptimization step to define a starting geometry that is already close to the minimum. Figure 2 demonstrates the implementation of such a multilevel approach, with three different QM packages interoperating in a single QMflows workflow. In this example, we consider calculating the lowest excitation energies of the coumarin dye for which we perform a preoptimization with DFTB, send the resulting structure to the ADF package for further optimization with the PBE functional, and finally employ the Orca program to compute the 10 lowest electronically excited states at the equation of motion CCSD level of theory. This example demonstrates that QMflows provides the flexibility to assign each optimization task to the QM implementation that is most suitable for this task. The full code is provided in Example S1 in the Supporting Information.

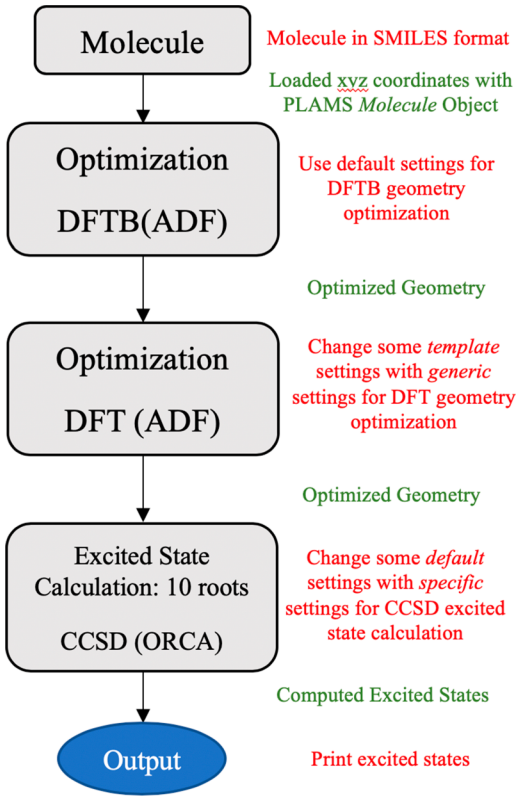


Figure 2. QMflows workflow for a multilevel geometry optimization followed by an excited states calculation on the relaxed structure. The QMflows actions are highlighted in red, while the results of the action are highlighted in green.

Example 2: Screening Absorption Characteristics of Organic Molecules. In the following example a series of organic chromophores is screened to select molecules with absorption wavelength in a desired range. As a simple criterion for a first selection we calculate the DFT HOMO−LUMO gap. The set of molecules is taken from the chemical database GDB-17,[22] which provides structures in *SMILES* format. The flowchart and the workflow to carry out this screening is illustrated in Figure 3.

First, the *SMILES* strings representing the target molecules are converted to 3D chemical structures using Molkit. For each

DOI: 10.1021/acs.jcim.9b00384
J. Chem. Inf. Model. 2019, 59, 3191−3197

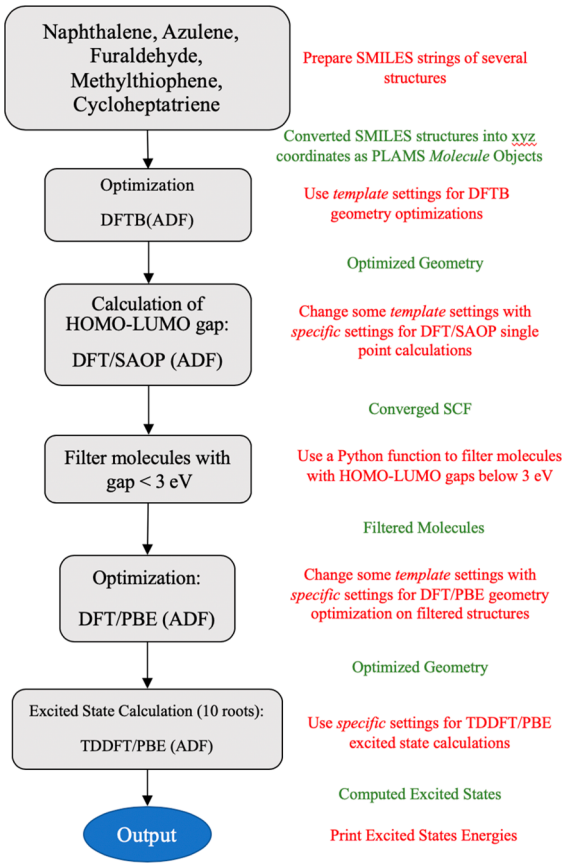


Figure 3. QMflows workflow for a multilevel geometry optimization followed by an excited states calculation on the relaxed structure. The QMflows actions are highlighted in red, while the results of the action are highlighted in green.

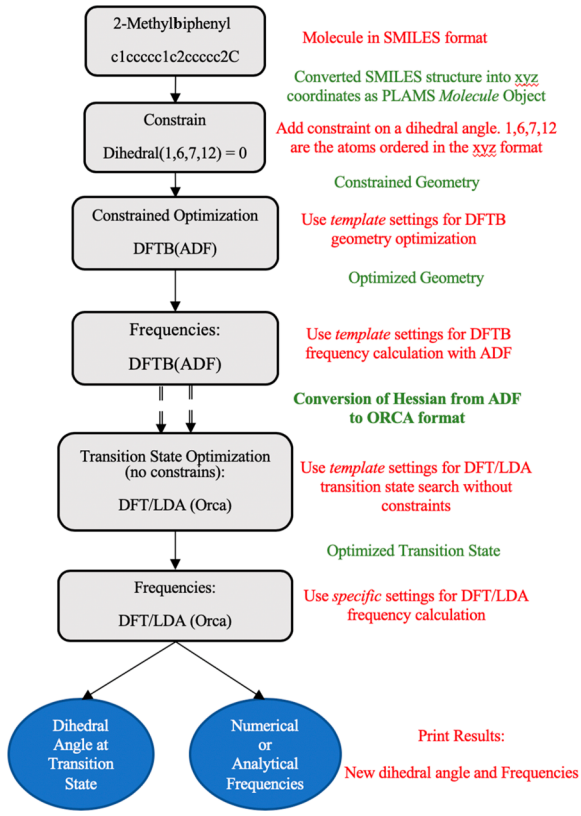


Figure 4. QMflows workflow for a multilevel transition state search. The QMflows actions are highlighted in red, while the results of the action are highlighted in green.

molecule, an initial geometry optimization is performed using the computationally inexpensive DFTB method followed by a single point DFT calculation with the SAOP functional. The HOMO and LUMO energies are extracted, and the best performing candidates, those with HOMO−LUMO gaps within the desired energy range, are selected for further study by DFT structure optimization and TD-DFT calculation of the lowest excited states. This workflow illustrates manipulation of structures, extraction of selected data, and workflows in which the number of jobs is determined at runtime. The full code is provided in Example S2 in the Supporting Information. Notice that there are no dependencies between the calculations for different molecules, and this fact is automatically picked by Noodles that generates a dependency graph where the jobs of different molecules are independent from each other and therefore are marked to run in parallel.

Example 3: Multilevel Transition State Search. Figure 4 shows another workflow to illustrate the integration of calculations with different packages by QMflows. Starting from an automatically generated 2-methylbiphenyl molecule, a constrained geometry optimization is performed with the DFTB package, to obtain an approximate TS state for the rotation between two conformers. Subsequently, a frequency calculation is performed to obtain a DFTB Hessian. This Hessian is then used to initialize the transition state calculation at the DFT level in a different package (ORCA). The full code is provided in Example S3 in the Supporting Information. This example shows how QMFlows allows (partial) interoperability between codes by providing generic functionality that is common to different packages, translating between different input specifications without user intervention.

Example 4: Multilevel Transition State Search with Conditional Workflows. The flowchart in Figure 5 illustrates another possibility for conditional workflows. In this case we check whether the preoptimization was successful and skip this step in case it was not. This is helpful when looping in over a wide class of molecules for which it is difficult to predict whether a certain preoptimizer will provide a converged result. After the preoptimizing step with DFTB, the molecules are further optimized with DFT (Orca) with a final step of TD-DFT calculation including solvent effects with the ADF program. Figure S1 shows the output of the script, indicating that the preoptimizations for molecules containing a boron atom failed (because this atom is not contained in the default parameter set). QMflows allows users to decide whether an error should be propagated to the rest of the dependencies (as a Python *None* value), or the user can provide an alternative function to execute in case of failure, for example an alternative calculation or a default value.

CONCLUSIONS

In summary, we have presented the open source software QMflows, a flexible and powerful Python code capable of easily automatizing and parallelizing quantum chemistry workflows. The software is constructed on four main pillars:

a. Flexible input manipulation. Here QMflows provides generic key words and templates for different tasks (single points, geometry optimizations, etc.) that are

DOI: 10.1021/acs.jcim.9b00384
J. Chem. Inf. Model. 2019, 59, 3191−3197

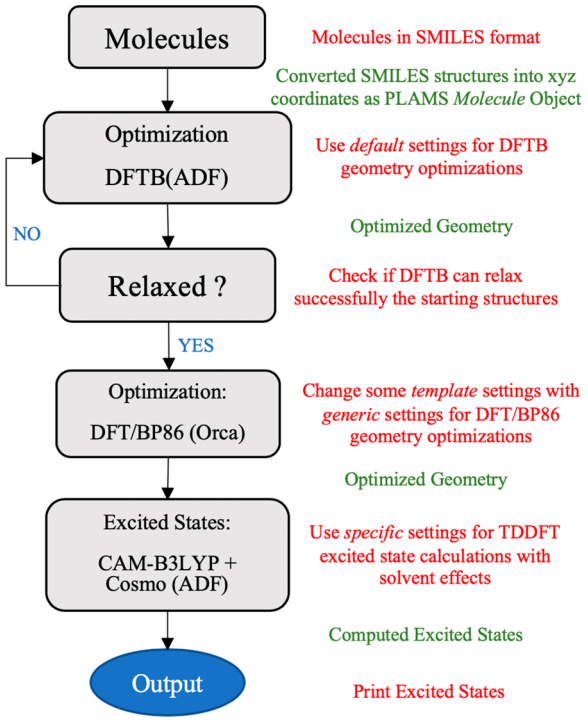


Figure 5. QMflows workflow for a multilevel transition state search using a conditional function. The QMflows actions are highlighted in red, while the results of the action are highlighted in green.

valid for several quantum chemical packages. The users will only need to call the desired QM package and a given task to carry out the calculation. If specific key words are required beyond the provided templates, QMflows allows to add them following the tree structure of the given QM code.

b. Workflow execution. In a complex workflow where several QM packages and tasks are called, QMflows rely on the Noodles framework to construct the dependencies between each task and schedule them as "promised objects". The actual workflow is executed at the end of the script allowing for optimal usage of parallel resources.

c. Efficient restart procedure. Thanks to the Noodles framework, if a job fails for whatever reason, Noodles will check the dependency tree querying a database with the metadata of the successfully executed parts of the workflow. In this way, the user will need only to resubmit the original Slurm or PBS script without any modification, starting from the last successful job.

d. Data postprocessing. QMflows is furnished with output parsers of several QM packages to retrieve essential data. Additionally, QMflows allows for storing metadata in a HDF5 portable format, which can be used for further postprocessing.

■ ASSOCIATED CONTENT

Supporting Information

The Supporting Information is available free of charge on the ACS Publications website at DOI: 10.1021/acs.jcim.9b00384.

Full code snippets for examples presented in manuscript (PDF)

■ AUTHOR INFORMATION

Corresponding Authors

*E-mail: i.a.c.infante@vu.nl.
*E-mail: l.visscher@vu.nl.

ORCID

Christoph R. Jacob: 0000-0002-6227-8476
Ivan Infante: 0000-0003-3467-9376
Lucas Visscher: 0000-0002-7748-6243

Notes

The authors declare no competing financial interest.
This QMflows software is distributed as a Python library that can be downloaded from GitHub at www.github.com/SCM-NV/QMflows.

■ ACKNOWLEDGMENTS

This work was supported by The Netherlands eScience Center (Grant No. 027.014.202) and by The Netherlands Organization of Scientific Research (NWO) through the Innovational Research Incentive (Vidi) Scheme (Grant No. 723.013.002).

■ ABBREVIATIONS

DFT, density functional theory; DFTB, tight-binding DFT; QM, quantum mechanics

■ REFERENCES

(1) Liu, Y.; Zhao, T.; Ju, W.; Shi, S. Materials discovery and design using machine learning. *J. Mater.* **2017**, *3*, 159−177.

(2) Gómez-Bombarelli, R.; Aguilera-Iparraguirre, J.; Hirzel, T. D.; Duvenaud, D.; Maclaurin, D.; Blood-Forsythe, M. A.; Chae, H. S.; Einzinger, M.; Ha, D.-G.; Wu, T.; Markopoulos, G.; Jeon, S.; Kang, H.; Miyazaki, H.; Numata, M.; Kim, S.; Huang, W.; Hong, S. I.; Baldo, M.; Adams, R. P.; Aspuru-Guzik, A. Design of efficient molecular organic light-emitting diodes by a high-throughput virtual screening and experimental approach. *Nat. Mater.* **2016**, *15*, 1120−1127.

(3) Hachmann, J.; Olivares-Amaya, R.; Atahan-Evrenk, S.; Amador-Bedolla, C.; Sánchez-Carrera, R. S.; Gold-Parker, A.; Vogt, L.; Brockway, A. M.; Aspuru-Guzik, A. The Harvard Clean Energy Project: Large-Scale Computational Screening and Design of Organic Photovoltaics on the World Community Grid. *J. Phys. Chem. Lett.* **2011**, *2*, 2241−2251.

(4) Hamprecht, F. A.; Cohen, A. J.; Tozer, D. J.; Handy, N. C. Development and assessment of new exchange-correlation functionals. *J. Chem. Phys.* **1998**, *109*, 6264−6271.

(5) Goerigk, L.; Grimme, S. A thorough benchmark of density functional methods for general main group thermochemistry, kinetics, and noncovalent interactions. *Phys. Chem. Chem. Phys.* **2011**, *13*, 6670.

(6) Santra, B.; Michaelides, A.; Scheffler, M. On the accuracy of density-functional theory exchange-correlation functionals for H bonds in small water clusters: Benchmarks approaching the complete basis set limit. *J. Chem. Phys.* **2007**, *127*, 184104.

(7) Azpiroz, J. M.; Ugalde, J. M.; Infante, I. Benchmark Assessment of Density Functional Methods on Group II−VI MX (M = Zn, Cd; X = S, Se, Te) Quantum Dots. *J. Chem. Theory Comput.* **2014**, *10*, 76−89.

(8) Jacobson, L. D.; Bochevarov, A. D.; Watson, M. A.; Hughes, T. F.; Rinaldo, D.; Ehrlich, S.; Steinbrecher, T. B.; Vaitheeswaran, S.; Philipp, D. M.; Halls, M. D.; Friesner, R. A. Automated Transition State Search and Its Application to Diverse Types of Organic Reactions. *J. Chem. Theory Comput.* **2017**, *13*, 5780−5797.

(9) Jacob, C. R.; Beyhan, S. M.; Bulo, R. E.; Gomes, A. S. P.; Götz, A. W.; Kiewisch, K.; Sikkema, J.; Visscher, L. PyADF - A scripting framework for multiscale quantum chemistry. *J. Comput. Chem.* **2011**, *32*, 2328−2338.

DOI: 10.1021/acs.jcim.9b00384
J. Chem. Inf. Model. 2019, 59, 3191−3197

(10) Metz, S.; Kästner, J.; Sokol, A. A.; Keal, T. W.; Sherwood, P. ChemShell-a modular software package for QM/MM simulations. *Wiley Interdiscip. Rev.: Comput. Mol. Sci.* **2014**, *4*, 101−110.
(11) Jain, A.; Ong, S. P.; Chen, W.; Medasani, B.; Qu, X.; Kocher, M.; Brafman, M.; Petretto, G.; Rignanese, G.-M.; Hautier, G.; Gunter, D.; Persson, K. A. FireWorks: a dynamic workflow system designed for high-throughput applications. *Concurr. Comput. Pract. Exp.* **2015**, *27*, 5037−5059.
(12) Pizzi, G.; Cepellotti, A.; Sabatini, R.; Marzari, N.; Kozinsky, B. AiiDA: automated interactive infrastructure and database for computational science. *Comput. Mater. Sci.* **2016**, *111*, 218−230.
(13) Larsen, A. H.; Mortensen, J. J.; Blomqvist, J.; Castelli, I. E.; Christensen, R.; Dułak, M.; Friis, J.; Groves, M. N.; Hammer, B.; Hargus, C.; Hermes, E. D.; Jennings, P. C.; Jensen, P. B.; Kermode, J.; Kitchin, J. R.; Kolsbjerg, E. L.; Kubal, J.; Kaasbjerg, K.; Lysgaard, S.; Maronsson, J. B.; Maxson, T.; Olsen, T.; Pastewka, L.; Peterson, A.; Rostgaard, C.; Schiøtz, J.; Schütt, O.; Strange, M.; Thygesen, K. S.; Vegge, T.; Vilhelmsen, L.; Walter, M.; Zeng, Z.; Jacobsen, K. W. The atomic simulation environment—a Python library for working with atoms. *J. Phys.: Condens. Matter* **2017**, *29*, 273002.
(14) Software for Chemistry & Materials B.V. (SCM): 2018.
(15) Hidding, J.; Weel, B.; Zapata, F.; Borgdorff, J. *NLeSC/noodles 0.2.3*; DOI: 10.5281/ZENODO.205986.
(16) te Velde, G.; Bickelhaupt, F. M.; Baerends, E. J.; Fonseca Guerra, C.; van Gisbergen, S. J. A.; Snijders, J. G.; Ziegler, T. Chemistry with ADF. *J. Comput. Chem.* **2001**, *22*, 931−967.
(17) Hutter, J.; Iannuzzi, M.; Schiffmann, F.; Vandevondele, J. Cp2k: Atomistic simulations of condensed matter systems. *Wiley Interdiscip. Rev. Comput. Mol. Sci.* **2014**, *4*, 15−25.
(18) Visscher, L.; Jensen, H. J. Aa.; Bast, R.; Saue, T. with contributions from Bakken, V.; Dyall, K. G.; Dubillard, S.; Ekström, U.; Eliav, E.; Enevoldsen, T.; Faßhauer, E.; Fleig, T.; Fossgaard, O.; Gomes, A. S. P.; Hedegård, E. D.; Helgaker, T.; Henriksson, J.; Iliaš, M. *Dirac, A relativistic ab initio electronic structure program, Release DIRAC17*.
(19) Gordon, M. S.; Schmidt, M. W. In *Theory and Applications of Computational Chemistry*; Elsevier: 2005; pp 1167−1189, DOI: 10.1016/B978-044451719-7/50084-6.
(20) Neese, F. The ORCA program system. *Wiley Interdiscip. Rev. Comput. Mol. Sci.* **2012**, *2*, 73−78.
(21) Zapata, F.; Ridder, L.; Hidding, J.; Infante, I.; Visscher, L. *QMflows*; DOI: 10.5281/ZENODO.1045523.
(22) Ruddigkeit, L.; van Deursen, R.; Blum, L. C.; Reymond, J.-L. Enumeration of 166 Billion Organic Small Molecules in the Chemical Universe Database GDB-17. *J. Chem. Inf. Model.* **2012**, *52*, 2864−2875.

DOI: 10.1021/acs.jcim.9b00384
J. Chem. Inf. Model. 2019, 59, 3191−3197