Software News and Updates

cclib: A Library for Package-Independent Computational Chemistry Algorithms

NOEL M. O'BOYLE,[1] ADAM L. TENDERHOLT,[2] KAROL M. LANGNER[3]
[1]*Cambridge Crystallographic Data Centre, 12 Union Road, Cambridge CB2 1EZ, United Kingdom*
[2]*Department of Chemistry, Stanford University, Stanford, California 94305*
[3]*Institute of Physical and Theoretical Chemistry, Wrocław University of Technology, Wyb. Wyspianskiego 27, 50-370 Wrocław, Poland*

Received 30 May 2007; Revised 20 July 2007; Accepted 26 July 2007
DOI 10.1002/jcc.20823
Published online 11 September 2007 in Wiley InterScience (www.interscience.wiley.com).

Abstract: There are now a wide variety of packages for electronic structure calculations, each of which differs in the algorithms implemented and the output format. Many computational chemistry algorithms are only available to users of a particular package despite being generally applicable to the results of calculations by any package. Here we present cclib, a platform for the development of package-independent computational chemistry algorithms. Files from several versions of multiple electronic structure packages are automatically detected, parsed, and the extracted information converted to a standard internal representation. A number of population analysis algorithms have been implemented as a proof of principle. In addition, cclib is currently used as an input filter for two GUI applications that analyze output files: PyMOlyze and GaussSum.

 J Comput Chem 29: 839–845, 2008

Key words: computational chemistry; algorithms; Python

Introduction

Computational chemists carrying out semi-empirical, *ab initio*, or density functional theory calculations may choose from a wide variety of modern computational chemistry packages. Each package is characterized by the presence of different computational methods and levels of theory, as well as by how these methods are implemented. From a technical point of view, the programming languages used to write these packages include Fortran, C, C++, and Python. Because of differences in design, the lack of an API (an application programming interface, which would allow access to internal functions of the package from a user's own program), and the proprietary nature of several of these packages, modern computational chemistry packages are not interoperable. This leads to a situation where, for example, a particular charge analysis method is only available to the users of a particular program even though such methods are generally applicable to calculations by any package.

How then can computational chemists ensure that their algorithms are available to users of any package? Typically, they choose several packages that they are interested in, and write routines to parse the necessary information from the text file containing the results of the calculation (the log file), or to extract this information from a binary file produced during the calculation (the checkpoint file). The former often appears to be the best choice as the log file may be easily viewed, it is the output file with which users of the software are most familiar, and users do not always retain the checkpoint file as they tend to be quite large. However, log files from different programs have completely different structures, are typically quite free-format, the units used for data may be different (and indeed, are sometimes not specified in the file), and the same data may be present under a different heading. On top of this, the specifics of a log file from a particular package may depend on the nature of the calculation, on the version of the software, or on the operating system on which the calculation is run. As a result, depending on the set of log files available to the developer, it is unlikely that the resulting parser will be sufficiently robust to deal with the log files of all users. Even then, as new versions of the package become available, the author must constantly update the parser or it quickly becomes obsolete.

Correspondence to: N. M. O'Boyle; e-mail: noel.oboyle2@mail.dcu.ie

Here we describe cclib, a programming library whose goal is to overcome these difficulties by providing developers with a single interface to the results of all computational chemistry packages. A library shared by many programs has the advantage that a bug found by the users of any of the client programs will result in the library being improved for all users. Since the cclib library focuses on parsing output files, considerable effort goes into ensuring that the parsers are robust, up to date, and that any bug reports are quickly dealt with. This means that the individual developers do not have to worry about writing their own parsers, and so can concentrate on the algorithms they are implementing. In addition, their algorithms will work equally well for users of any of the computational chemistry packages supported by cclib, and will continue to work with new versions of the packages with different output file formats simply by keeping their version of cclib up to date.

cclib is aimed at two distinct sets of users. The first set consists of computational chemists who need to repeatedly extract information from log files, and who typically used a combination of command-line tools such as "grep" and "cut", or just copied text directly to spreadsheet software and edited it there. Using cclib these users can, in just a couple of lines of code, extract many different pieces of information from the log file. The other set of users are the developers of software that, as a necessary first step, needs to parse computational chemistry output files. This could be molecular visualization software, software that processes the results of a calculation, or software that implements a computational algorithm. Software developed using cclib will be independent of the particular package and version, as well as being platform independent.

cclib is designed with the following goals:

1. To facilitate the implementation of algorithms that are not specific to a particular computational chemistry package;
2. To provide a simple and standard interface to the results of computational chemistry calculations;
3. To standardize the information extracted from log files;
4. To maximize interoperability with other open source computational chemistry and cheminformatic software libraries.

cclib is released under an open source license, the GNU Lesser General Public License (LGPL), to encourage contributions from outside developers to develop a shared community resource. The LGPL maximizes the impact of cclib by allowing its incorporation into both open source and closed source programs, subject to certain conditions.[1]

Related Software

Current developments in grid computing infrastructure have resulted in several parallel efforts to facilitate communication between monolithic computational chemistry codes as part of a workflow. These methods rely on converting the output, either the binary file or the associated log file, to a standardized format, which can then be converted to an input file for the next program in the workflow. Borini et al.[2] have developed a standardized binary file format based on HDF5 files, a file format designed for storing scientific data. Wrapper scripts extract data from the binary files associated with calculations and write them to a HDF5 file. Their Q5Cost library, a FORTRAN library, then provides an API to the contents of the HDF5 file. The scope of the API is currently limited to atomic orbital, molecular orbital, and wavefunction data. There are also parallel efforts to convert the log file into a more easily parsed format, principally XML, to enable workflows in a grid operating environment. The eCCP and eMinerals UK eScience projects have developed an XML format based on CML and the XML used by the Visualization Tool Kit (VTK), and have modified several codes of interest to the environmental science community to read and write XML.[3] Baldridge et al.[4] have modified GAMESS to produce XML output via a Java library.

Another related project is GaussDal, an Open Source project for the management of data from calculations using a database.[5] This has been developed with workflows in mind and in particular it has been incorporated into the SciCraft visual programming environment.[6]

cclib differs from related work in that it does not rely on an additional file format or database, and is aimed at users who are developing algorithms or simply need to extract data from output files, although enabling workflows is a possible application. In addition, unlike some of the other efforts, cclib does not require any changes to be made to the underlying code which, in a field where most computational chemistry codes are proprietary, would be of limited use.

Description

cclib is primarily a programming library, and is composed of four modules: parser, bridge, method, and progress. At the core of cclib are the parser classes, of which there are five in cclib 0.7: one for ADF,[7] GAMESS (this works for both GAMESS US[8] and PC GAMESS[9]), GAMESS-UK,[10] Gaussian,[11] and Jaguar.[12] These may either be instantiated directly with the name of a log file or via the `ccopen()` function, which automatically detects which package the log file corresponds to and then creates the appropriate instance. Calling the `parse()` method parses the file and extracts the information shown in Table 1, if present. This information is made available through attributes of the parser object.

cclib aims to present a unified interface to the information contained in a variety of computational chemistry log files (Table 1). This unity extends to the data itself. For example, molecular coordinates are presented in Angstrom, and vibrational frequencies in cm^{-1}, no matter what units are used in the underlying log file. As well as unit conversions, cclib standardizes conventions such as those used to denote orbital symmetry. For the symmetry labeled *BU* by GAMESS and Gaussian, ADF uses *B.u*, GAMESS-UK uses *bu*, and Jaguar uses *Bu*; cclib normalizes all of these to *Bu*. In other cases all of the programs disagree: A'' is alternatively represented by *AAA* (ADF), A'' (GAMESS), a'' (GAMESS-UK), A'' (Gaussian), and *App* (Jaguar).

The problem of different symmetry labels discussed above highlights a general difficultly encountered when trying to parse log files, that of a lack of detailed documentation. Of the four programs, only ADF provides a manual (available on-line) that describes the symmetry labels possible for molecules belonging to different point groups. Since many of these programs do not

Table 1. Information Extracted by the Parser Module from Computational Chemistry Output Files.

Attribute name	Description	Units	Data type
aonames	Atomic orbital names		List of strings
aooverlaps	Atomic orbital overlap matrix		Array of rank 2
atomcoords	Atom coordinates	Å	Array of rank 3
atomnos	Atomic numbers		Array of rank 1
coreelectrons	Number of core electrons in an atom's pseudopotential		Array of rank 1
etenergies	Energies of electronic transitions	cm^{-1}	Array of rank 1
etoscs	Oscillator strengths of electronic transitions		Array of rank 1
etrotats	Rotatory strengths of electronic transitions		Array of rank 1
etsecs	Singly excited configurations for electronic transitions		List of lists
etsyms	Symmetries of electronic transitions		List of strings
fonames	Fragment molecular orbital names		List of strings
fooverlaps	Fragment molecular orbital overlap matrix		Array of rank 2
gbasis	Coefficients and exponents of Gaussian basis functions		PyQuante format
geotargets	Criteria target values for geometry convergence		Array of rank 1
geovalues	Criteria values for geometry convergence		Array of rank 2
homos	Molecular orbital index of the HOMO(s)		Array of rank 1
mocoeffs	Molecular orbital coefficients		List of arrays of rank 2
moenergies	Molecular orbital energies	eV	List of arrays of rank 1
mosyms	Molecular orbital symmetries		List of lists
mpenergies	Möller-Plesset corrected electronic energies	eV	Array of rank 2
natom	Number of atoms		Integer
nbasis	Number of basis functions		Integer
nmo	Number of molecular orbitals		Integer
scfenergies	Electronic energy of the molecule	eV	Array of rank 1
scftargets	Criteria target values for SCF convergence		Array of rank 2
scfvalues	Criteria values for SCF convergence		List of arrays of rank 2
vibdisps	Cartesian displacement vectors	ΔÅ	Array of rank 3
vibfreqs	Vibrational frequencies	cm^{-1}	Array of rank 1
vibirs	IR intensities	km/mol	Array of rank 1
vibramans	Raman intensities	A^4/amu	Array of rank 1
vibsyms	Symmetries of vibrations		List of strings

provide the source code, the only way to obtain the full list of labels is to run calculations with molecules of various symmetries.

The following example shows how cclib is used in practice to calculate the HOMO-LUMO gap (in eV), given the output of a single point energy calculation using Gaussian03:

```
from cclib.parser import ccopen
mylogfile = ccopen('methane.log')
mylogfile.parse()
homo = mylogfile.homos[0]
energies = mylogfile.moenergies[0]
homolumogap = energies[homo+1] -
 energies[homo]
```

During the parsing process status updates, warnings and error messages are written to a logger associated with the parser. Typically, these messages are written to standard output, but they can be redirected to a file, handled by a dialog box, or the log level can be adjusted to show, for example, only warning messages. In addition, the parser provides hooks to a Progress class, which can be used to monitor graphically (for example, using a progress bar in a GUI application) or at the command line, progress in parsing the file. Examples of these are included in the cclib distribution in the *progress* module.

cclib provides implementations of computational chemistry algorithms in the *methods* module. Using the molecular orbital coefficients, **C**, and the overlap matrix, **S**, extracted from the log file, a number of orbital-based population methods have been implemented. The c^2 or CSPA method[13] disregards the fact that basis functions may overlap, and simply defines the contribution, c_{ai}, of a particular atomic orbital, a, to a particular molecular orbital, i, as the square of the molecular orbital coefficient:

$$c_{ai} = C_{ai}^2 \tag{1}$$

Mulliken population analysis[14] (MPA) deals with overlapping basis functions by splitting the overlap population equally between the two basis functions:

$$c_{ai} = C_{ai}\left(\hat{\mathbf{S}}\hat{\mathbf{C}}\right)_{ai} = \sum_b C_{ai}C_{bi}S_{ab} \tag{2}$$

where b runs over all of the atomic orbitals.

The overlap population itself can be used to analyze the bonding interaction between basis functions, atoms, or groups of atoms in a molecule. If A and B are the sets of basis functions

on the atoms of two groups in a molecule, then the overlap population between A and B, OP_{AB}, for a particular molecule orbital, i, is defined as:

$$OP_{AB,i} = \sum_{a \in A} \sum_{b \in B} 2C_{ai}C_{bi}S_{ab} \quad (3)$$

cclib allows the user to calculate the density matrix, $\mathbf{P}$, which is defined as the sum over the N occupied orbitals of the outer product of the vector of coefficients for a particular molecular orbital, $\mathbf{C}_k$, with itself:

$$\mathbf{P} = \sum_{k}^{N} \mathbf{C}_k \mathbf{C}_k{}^t \quad (4)$$

The density matrix, $\mathbf{P}$, is used in the cclib implementation of Mayer's bond order analysis,[15] where the bond order, B_{AB}, between two atoms or fragments A and B is defined as:

$$B_{AB} = 2\sum_{a \in A} \sum_{b \in B} [(\mathbf{P}^{\alpha}\mathbf{S})_{ba}(\mathbf{P}^{\alpha}\mathbf{S})_{ba} + (\mathbf{P}^{\beta}\mathbf{S})_{ba}(\mathbf{P}^{\beta}\mathbf{S})_{ba}] \quad (5)$$

where $\mathbf{P}^{\alpha}$ and $\mathbf{P}^{\beta}$ indicate the density matrices for alpha electrons and beta electrons respectively.

Charge Decomposition Analysis (CDA) is a method developed by Dapprich and Frenking[16] that describes the interaction between two molecular fragments, A and B, in terms of (i) the mixing between the occupied orbitals of A and the empty orbitals of B, (ii) the mixing between the occupied orbitals of B and the empty orbitals of A, and (iii) the mixing between the occupied orbitals of A and the occupied orbitals of B. Based on the CDA 2.1 code,[17] we have implemented CDA as part of cclib.

Also included in the *methods* module are functions to calculate the magnitude of the wavefunction and the electron density at grid points in a volume. These functions use the open source quantum chemistry package, PyQuante,[18] to handle the calculation (see below under Interoperability). The resulting Volume object can be written to disk in Gaussian Cube format or as a Visualization Tool Kit (VTK) file.

Development

cclib is implemented using the Python programming language, which is an object-oriented platform-independent scripting language. This is an ideal language for parsing text files and is widely used in this context, for example, in the bioinformatics community (Biopython[19]). Its platform-independence means that the same Python code will work equally well on a computer running MacOSX, Linux, or Windows. Equally important though is the fact that Python can be regarded as a "glue language." That is, it can interface with code written in C,[20] C++,[21] Fortran,[22] and Java.[23] This is of great importance as many scientific algorithms are written in C and Fortran in particular. Finally, through the Numeric extension to Python,[24] efficient matrix multiplication and linear algebra routines are available, which are essential for implementing computational chemistry algorithms.*

cclib is developed using an open source development model,[25] and takes full advantage of the development resources provided by its hosting site, SourceForge (http://www.sf.net). A strong emphasis on design is a key aspect of cclib's development, and communication between developers is crucial to this goal. Initial discussion takes place on the developers' mailing list. As a consensus emerges, the specifications (for example, the type of data structure, and data type required for a particular attribute parsed from a log file) are recorded on the cclib wiki (a web site edited using a web browser). These specifications are then implemented by the developers. All changes to the code are versioned using the Subversion source code versioning system and require a log message, which is then relayed to the other developers. Any difficulties or comments that arise in the implementation process are entered on the wiki, so that a full record exists of design choices and decisions made.

Another notable feature of the cclib development model is the extensive use of unit tests. Unit tests are short pieces of code designed to test a single unit of functionality of a program. There are two unit test frameworks that are part of the standard Python library, *unittest* and *doctest*, and both are used by cclib. *doctest* runs tests embedded in the documentation strings of modules and functions, and is best suited to testing functions whose correct behavior can be verified by examining a small number of outputs. For example, the symmetry labels produced by ADF are standardized by a function with doctests for every possible symmetry that can be found in an ADF log file, thus giving a strong guarantee that it is bug-free.

Unittest is a more general purpose unit test framework, which cclib uses to ensure consistency between parsers, and internal consistency between the data extracted from the same file. For every computational chemistry package handled by cclib, five standard calculations are performed on the molecule 1,4-divinylbenzene at the B3LYP/STO-3G level of theory (BLYP and a minimal basis set in the case of ADF): a geometry optimization, a single point energy calculation (with overlap matrix and molecular orbital coefficients printed to the log file), an unrestricted single point energy calculation for the singly charged cation, a vibrational frequency calculation (with IR and Raman intensities), and a TD-DFT calculation. Unit tests ensure that the extracted data for each new package agrees with those extracted from the others. For example, to ensure that each of the parsers is extracting atom coordinates in Angstrom (not Bohr), the minimum C—C distance in the molecule is compared to a known value in Angstrom. Other unit tests verify that the extracted data is internally consistent; for example, for the geometry optimization of divinylbenzene, cclib has tests to ensure that the number of geometries extracted is equal to (or one more than, in the case of ADF) the number of tests for convergence extracted.

All log files used to develop the parsers are stored along with the source code on Subversion. If, after release, a log file is found that the current parser cannot parse, then a test for the

*The next release of cclib will be based on NumPy, the successor to Numeric.

bug is added to a regression test suite, and the bug is fixed. The test suite ensures that a bug, once fixed, will stay fixed. Periodically, a release is made of all of the log files that have historically broken our parsers. In effect, these log files define the behavior and ability of our current parsers, and may be useful to others as a test set for developing similar software.

Interoperability

Since one of the goals of cclib is to provide a bridge from computational chemistry output files to analysis algorithms, it is appropriate for cclib to also promote interoperability with other open source Python software for the analysis of chemical information. This is also in line with the goals of the Blue Obelisk movement, a group of chemists and programmers who promote interoperability in chemical informatics.[26] To this end, cclib contains a *bridge* module which can use the molecular information parsed by cclib to create a Biopython PDB object,[19] an OpenBabel OBMol object,[27] or a PyQuante Molecule object.[18]

Biopython contains many algorithms of interest in structural biology such as *Superimposer* which aligns two molecular conformations by minimizing the root mean squared deviation between the atoms. Of more general interest is the OpenBabel library, a C++ cheminformatics library which provides Python bindings. OpenBabel allows conversion of molecular data to any of more than 60 different molecular file formats, including input file formats for several computational chemistry packages. In addition, it contains many algorithms to deal with molecular structures such as ring perception, detection of aromaticity, and detection of chirality. The following example shows how to create a PDB file containing the final step of a geometry optimization:

```
from cclib.parser import ccopen
from cclib.bridge import
 makeopenbabel
import pybel # Provided by OpenBabel
logfile = ccopen('mylogfile.out')
logfile.parse()
OBmol = makeopenbabel(logfile.
 atomcoords[-1], logfile.atomnos)
pybel.Molecule(OBmol).write
 ('pdb', 'finalstep.pdb')
```

PyQuante is an open source suite of programs for developing quantum chemistry methods. It is written in Python with speed-critical code in C. Since cclib extracts the Gaussian basis set in PyQuante format (see Table 1), it is possible to use the *bridge* module to create a PyQuante Molecule and carry out an electronic structure calculation using the extracted basis set. The cclib functions for calculating the electron density and the magnitude of the wavefunction at grid points in a volume interface with PyQuante in this way.

Applications

cclib allows software developers to concentrate on the implementation of algorithms or processing of results without having to worry about the details of particular log file formats. Here we describe some applications of the cclib library in computational chemistry software: ccget, cda, GaussSum, and PyMOlyze.

ccget

An example command-line application called ccget is included in the cclib distribution. Given the name of a computational chemistry log file, ccget lists the attributes that cclib can parse from that log file, or alternatively, the user can specify a particular attribute and its value will be displayed. Although some users may find ccget useful, it is intended more as an example of how to incorporate cclib into a Python script.

cda

A Python script, *cda*, which performs a Charge Decomposition Analysis, is included with cclib. Given three log files as input, a "supermolecule" and two constituent molecules, it analyses the donor–acceptor interactions and then presents the results in table form.

GaussSum

GaussSum[28] is a Python application with a graphical user interface based on Tkinter that allows the user to monitor calculations, convolute spectra, and perform Mulliken population analysis (Fig. 1). The convergence of the self-consistent field (SCF) procedure, as well as the progress of a geometry optimization calculation, is followed by comparing the values of particular criteria to their target values. GaussSum calculates an overall measure of progress by summing progress values for each criterion. These progress values are either 0, if the target value has been reached for that criterion, or the logarithm of the difference between the target value and the current value. Oscillatory behavior in SCF convergence may indicate to the user that the calculation should be restarted with a different convergence algorithm, whereas oscillatory behavior close to convergence of a geometry optimization may indicate that a finer integration grid is needed to increase numerical accuracy in the calculation of the Hessian due to a shallow well.

GaussSum can also be used to relate calculations of physical properties to experimental measurements. The results of a TD-DFT calculation may be convoluted using Gaussian curves to create UV–visible spectra and circular dichroism spectra. In addition, based on the results of a vibrational frequency calculation it can use Lorentzian curves to convolute the IR spectrum and Raman spectrum. As well as allowing the user to specify the full width at half maximum of the curves used to convolute the data, it is possible to scale the vibrational frequencies either individually or with an overall scaling factor. The information extracted by cclib is written to a file in a tab-separated format suitable for reading with spreadsheet software.

Other features include the calculation of the percent contribution of a particular group (for example, a ligand) to a particular molecular orbital based on a Mulliken population analysis. This information can be convoluted using Gaussian curves to create a density of states (DOS) curve or an overlap population density of states (also referred to as crystal orbital overlap population (COOP)) spectrum.

PyMOlyze

PyMOlyze[29] is a Python application with a graphical user interface based on the Qt library, which uses cclib both to perform parsing of computational chemistry log files and to perform population analysis (Fig. 2). Molecular fragments can be defined very intuitively using a drag-and-drop interface that allows individual atoms or basis functions to be added to molecular fragments. Once the fragments are defined, a number of methods are available for analyzing the electronic structure:

- Mulliken Population Analysis (MPA)
- C-squared Population Analysis (SCPA)
- Charge Decomposition Analysis (CDA)
- Mayer's Bond Orders
- Overlap Population Analysis (OPA)
- Fragment Analysis

PyMOlyze also provides several features for the manipulation of molecular coordinates. The first is a Cartesian coordinate editor, which is linked to a molecular viewer where the atom being edited is highlighted. If two atoms are selected, it is possible to rotate the molecule so that the selected atoms are aligned parallel to a particular axis. If three atoms are selected, the molecule can be rotated to place all three in the *XY*, *YZ*, or *XZ* plane. Other features include a molecular viewer that can show the change in molecular structure during the course of an optimization, and an interface to OpenBabel that allows the molecular structure to be saved as any one of several file formats.

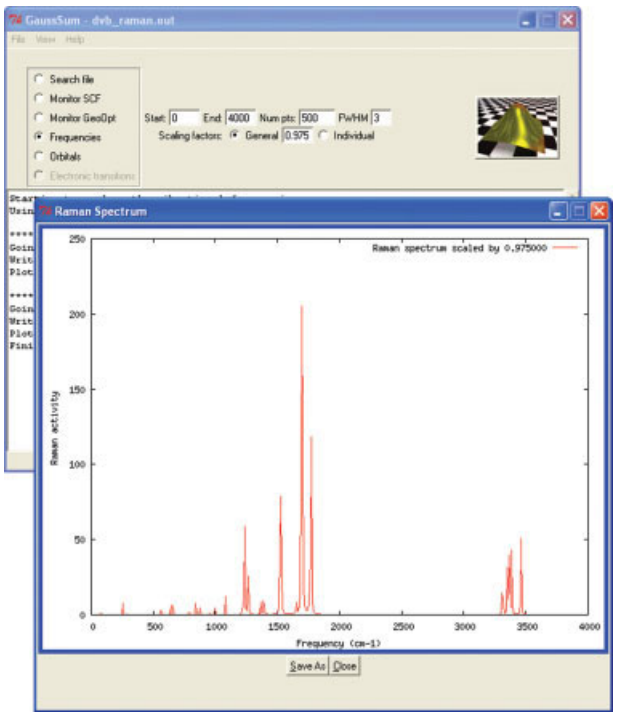

Figure 1. Screenshot of GaussSum showing the predicted Raman spectrum from a vibrational frequency calculation.

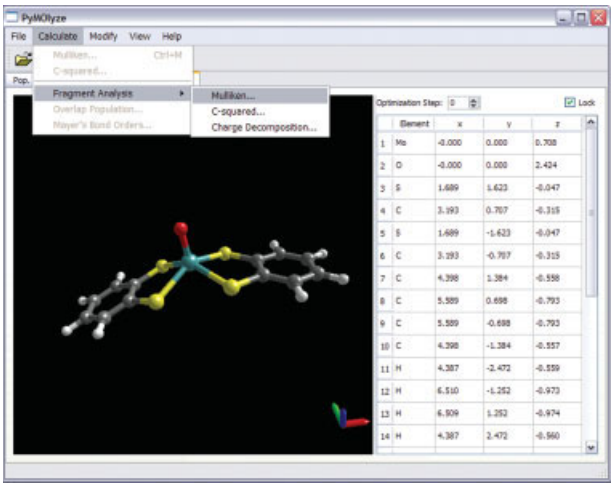

Figure 2. Screenshot of PyMOlyze showing the molecular viewer, Cartesian coordinates pane and available population analysis methods.

Conclusions

cclib provides a solution to the problem of writing algorithms that use information taken from computational chemistry output files. The information extracted from an output file is standardized, so that an application or algorithm that uses cclib will work equally well for any of several computational chemistry packages. In addition, by using cclib, developers and users can avoid the unnecessary duplication of effort involved in writing a parser themselves.

The task of parsing information from output files would be much easier if information were written using a standard machine-readable format, one that adheres to agreed conventions by means of dictionaries and ontologies so that information and units would be trivial to extract. This is an approach that is gaining momentum in the area of computational materials science where XML output is in development for packages such as GULP and CRYSTAL. However, until this approach becomes widespread in the computational chemistry community, libraries such as cclib insulate users from the differences between the output files of various packages, and allow developers and users to concentrate on writing algorithms or analyzing results with the knowledge that their scripts and programs will work with the output of any of a growing number of computational chemistry packages.

Acknowledgments

We thank Prof. Gernot Frenking and group members for their assistance during our implementation of CDA. Web-site hosting, mailing lists, and a code repository were provided by SourceForge (http://www.sf.net). An RSS feed of commit log messages was provided by CIA (http://cia.vc). We are grateful to the authors of GAMESS and PC GAMESS for providing their programs free of charge, and to CFS and Schrödinger, respectively, for providing licenses for GAMESS-UK and Jaguar free of

charge. We thank several users for submitting bug reports and example log files for regression testing (for full details, see the file THANKS in the cclib distribution), and Egon Willighagen for a critical reading of the manuscript.

References

1. GNU Lesser General Public License, http://www.gnu.org/copyleft/lesser.html (Accessed on 18 May 2007).
2. Borini, S.; Monari, A.; Rossi, E.; Tajti, A.; Angeli, C.; Bendazzoli, G. L.; Cimiraglia, R.; Emerson, A.; Evangelisti, S.; Maynau, D.; Sanchez-Marin, J.; Szalay, P. G. J Chem Inf Model 2007, 47, 1271.
3. (a) Allan, R.; Couch, P.; Knowles, P.; Sherwood, P. Proceedings of the UK e-Science All Hands Meeting 2004; (b) Couch, P. A.; Sherwood, P.; Sufi, S.; Todorov, I. T.; Allan, R. J.; Knowles, P. J.; Bruin, R. P.; Dove, M. T.; Murray-Rust, P. Proceedings of the UK e-Science All Hands Meeting 2005, 426; (c) White, T. O. H.; Murray-Rust, P.; Couch, P. A.; Tyer, R. P.; Bruin, R. P.; Todorov, I. T.; Wilson, D. J.; Dove, M. T.; Austen, K. F.; Parker, S. C. Proceedings of the UK e-Science All Hands Meeting 2006.
4. Baldridge, K. K.; Greenberg, J. P.; Sudholt, W.; Mock, S.; Altintas, I.; Amoreira, C.; Potier, Y.; Birnbaum, A.; Bhatia, K.; Taufer, M. Proc IEEE 2005, 93, 510.
5. Alsberg, B. K.; Bjerke, H.; Navestad, G. M.; Åstrand, P.-O. Comp Phys Commun 2005, 171, 133.
6. (a) Alsberg, B. K.; Kirkhus, L.; Tangstad, T.; Anderssen, E. In Knowledge Exploration in Life Science Informatics, Proc.; In Lect. Notes Artif. Intell., 3304; Lopez, J. A.; Benfenati, E.; Dubitzky, W., Eds.; Springer: Berlin 2004; pp. 58–68; (b) Alsberg, B. K.; Kirkhus, L.; Tangstad, T. SciCraft—A General Data Analysis Tool, Version 0.99.0, 2007, Available at http://www.scicraft.org.
7. (a) te Velde, G.; Bickelhaupt, F. M.; van Gisbergen, S. J. A.; Guerra, C. F.; Baerends, E. J.; Snijders, J. G.; Ziegler, T. J Comput Chem 2001, 22, 931; (b) Fonseca Guerra, C.; Snijders, J. G.; te Velde, G.; Baerends, E. J. Theor Chim Acta 1998, 99, 391; (c) ADF2006.01, SCM, Theoretical Chemistry, Vrije Universiteit, Amsterdam, The Netherlands, Available at http://www.scm.com.
8. (a) Schmidt, M. W.; Baldridge, K. K.; Boatz, J. A.; Elbert, S. T.; Gordon, M. S.; Jensen, J. H.; Koseki, S.; Matsunaga, N.; Nguyen, K. A.; Su, S. J.; Windus, T. L.; Dupuis, M.; Montgomery, J. A. J Comput Chem 1993, 14, 1347; (b) Gordon, M. S.; Schmidt, M. W. In Theory and Applications of Computational Chemistry, the First Forty Years; Dykstra, C. E.; Frenking, G.; Kim, K. S.; Scuseria, G. E., Eds.; Elsevier: Amsterdam, 2005; (c) GAMESS, March 24, 2007, Available at http://www.msg.ameslab.gov/GAMESS/GAMESS.html.
9. Granovsky, A. A. PC GAMESS, Version 7.0, 2006, Available at http://classic.chem.msu.su/gran/gamess/index.html.
10. (a) Guest, M. F.; Bush, I. J.; van Dam, H. J. J.; Sherwood, P.; Thomas, J. M. H.; van Lenthe, J. H.; Havenith, R. W. A.; Kendrick, J. Mol Phys 2005, 103, 719; (b) GAMESS-UK, Version 7.0, 2006, Available at http://www.cfs.dl.ac.uk/gamess-uk/index.shtml.
11. Frisch, M. J.; Trucks, G. W.; Schlegel, H. B.; Scuseria, G. E.; Robb, M. A.; Cheeseman, J. R.; Montgomery, J. A. Jr.; Vreven, T.; Kudin, K. N.; Burant, J. C.; Millam, J. M.; Iyengar, S. S.; Tomasi, J.; Barone, V.; Mennucci, B.; Cossi, M.; Scalmani, G.; Rega, N.; Petersson, G. A.; Nakatsuji, H.; Hada, M.; Ehara, M.; Toyota, K.; Fukuda, R.; Hasegawa, J.; Ishida, M.; Nakajima, T.; Honda, Y.; Kitao, O.; Nakai, H.; Klene, M.; Li, X.; Knox, J. E.; Hratchian, H. P.; Cross, J. B.; Bakken, V.; Adamo, C.; Jaramillo, J.; Gomperts, R.; Stratmann, R. E.; Yazyev, O.; Austin, A. J.; Cammi, R.; Pomelli, C.; Ochterski, J. W.; Ayala, P. Y.; Morokuma, K.; Voth, G. A.; Salvador, P.; Dannenberg, J. J.; Zakrzewski, V. G.; Dapprich, S.; Daniels, A. D.; Strain, M. C.; Farkas, O.; Malick, D. K.; Rabuck, A. D.; Raghavachari, K.; Foresman, J. B.; Ortiz, J. V.; Cui, Q.; Baboul, A. G.; Clifford, S.; Cioslowski, J.; Stefanov, B. B.; Liu, G.; Liashenko, A.; Piskorz, P.; Komaromi, I.; Martin, R. L.; Fox, D. J.; Keith, T.; Al-Laham, M. A.; Peng, C. Y.; Nanayakkara, A.; Challacombe, M.; Gill, P. M. W.; Johnson, B.; Chen, W.; Wong, M. W.; Gonzalez, C.; Pople, J. A. Gaussian 03, Revision C.01, Gaussian, Inc., Wallingford, CT, 2004.
12. Jaguar, Version 6.5, 2005, Schrodinger, LLC, New York, NY.
13. Ros, P.; Schuit, G. C. A. Theor Chim Acta 1966, 4, 1.
14. Mulliken, R. S. J Chem Phys 1955, 23, 1833.
15. Mayer, I. Chem Phys Lett 1983, 97, 270.
16. Dapprich, S.; Frenking, G. J Phys Chem 1995, 99, 9352.
17. CDA, Version 2.1, 2004, Available at http://www.uni-marburg.de/fb15/ag-frenking/cda.
18. Muller, R. P. PyQuante, Version 1.5.1, 2007, Available at http://pyquante.sf.net.
19. BioPython, Version 1.42, 2006, Available at http://biopython.org.
20. SWIG, Version 1.3.31, 2006, Available at http://www.swig.org.
21. Boost.Python, Version 2, 2002, Available at http://www.boost.org/libs/python/doc.
22. Peterson, P. F2PY, 2005, Available at http://cens.ioc.ee/projects/f2py2e.
23. Menard, S. JPype, Version 0.5.2.1, 2006, Available at http://jpype.sf.net.
24. Ascher, D.; Dubois, P. F.; Hinsen, K.; Hugunin, J.; Oliphant, T. Numerical Python, Version 24.2, 2005, Available at http://numpy.scipy.org.
25. Fogel, K. Producing Open Source Software; O'Reilly, 2005.
26. Guha, R.; Howard, M. T.; Hutchison, G. R.; Murray-Rust, P.; Rzepa, H.; Steinbeck, C.; Wegner, J. K.; Willighagen, E. L. J Chem Inf Model 2006, 46, 991.
27. Open Babel, Version 2.1.0, 2007, Available at http://openbabel.sf.net.
28. O'Boyle, N. M. GaussSum, Version 2.0.5, 2007, Available at http://gausssum.sf.net.
29. Tenderholt, A. L. PyMOlyze, Version 2.0, 2007, Available at http://pymolyze.sf.net.