Science & Technology Facilities Council

Technical Report
RAL-TR-2015-004

A Python interface to CASTEP

G Corbett, J Kermode, D Jochym, K Refson

May 2015

©2015 Science and Technology Facilities Council

This work is licensed under a Creative Commons Attribution 3.0 Unported License.

Enquiries concerning this report should be addressed to:

RAL Library
STFC Rutherford Appleton Laboratory
Harwell Oxford
Didcot
OX11 0QX

Tel: +44(0)1235 445384
Fax: +44(0)1235 446403
email: libraryral@stfc.ac.uk

Science and Technology Facilities Council reports are available online at: http://epubs.stfc.ac.uk

ISSN 1358-6254

Neither the Council nor the Laboratory accept any responsibility for loss or damage arising from the use of information contained in any of their reports or in any communication about their tests or investigations.

A Python interface to CASTEP

Greg Corbett[1], James Kermode[2], Dominik Jochym[1] and Keith Refson[1, 3]

[1]STFC Rutherford Appleton Laboratory
Harwell Oxford
Didcot
OX11 0QX
United Kingdom

[2]Centre for Predictive Modelling
School of Engineering
University of Warwick
Coventry
CV4 7AL
United Kingdom

[3]Department of Physics
Royal Holloway
University of London
Egham
TW20 0EX
United Kingdom

Abstract

This report documents a successful pilot project and feasibility study for adding a Python interface to the CASTEP first principles materials modelling code. Such an interface will allow the growing Python community within the scientific computing field access to CASTEP functionality, without the requirement of learning Fortran.

To achieve this, changes have been made to the CASTEP source code to allow:

- Serially re-entrant calling of a major task routine, specifically `electronic_minimisation()`.
- Automated generation of a Python interface.

The reasoning behind these changes has been documented and coding practices that may hinder a full move to serial re-entrancy in future have been noted. To demonstrate the success of the project, top-level task control logic has been written in Python -- using the Fortran 2003 computational core to perform multiple calls to `electronic_minimisation()`.

1 Introduction

CASTEP is a UK-developed cutting edge scientific computer program, based on quantum mechanics, for modelling of materials at the atomic level. Quantum mechanics-based simulation using density functional theory (DFT) methods is a major activity in materials science, solid-state physics and chemistry which is used by both academic, researchers and industry. CASTEP is written in portable, modular Fortran 2003 and according to good software design practice and design principles. It has a user base of well over a hundred UK university research groups, plus many more worldwide, and is the third-ranked code in terms of CPU cycles used by the HECToR national HPC service until its close in 2014. As of 2015, over 5000 peer-reviewed scientific articles have been published which cite CASTEP.

At present CASTEP is operated using a simple, ASCII file I/O model. The user prepares input files containing keywords to specify the calculation to be performed, initiates a run of a single executable program, which writes text based output files during and at the end of the run. This has benefits for code portability - eliminating I/O and graphics library dependencies, for use in a batch queuing environment where interactivity with a running application is frequently disallowed, and for massively parallel use on supercomputer-class machines, where the MPI model does not support any asynchronous control. Some limited interactivity and computational steering is allowed by means of updating an input file during the run. Typically, runs may take hours or days even on supercomputer-class machines using thousands of processors. Final analysis and graphics preparation is done by post processing analysis of CASTEP's output files - commonly on a different machine with a richer software environment than supercomputers typically provide.

While this model works well in many cases, it is a generation old, and fails to support a more recent style of use, characterized by the phrase "high-throughput simulation". Rather than individually set up a few, large calculations, the idea is to run thousands of smaller ones, which have been set up, controlled and analysed automatically by another computer program. The Materials Genome project [1] attempts to run large numbers of calculations on variant structures with the aim of discovering new materials. Ab-initio Random Structure Search [2] and evolutionary algorithms perform searches through crystal structure space over thousands of variants. And projects such as the Atomic Simulation Environment [3] are attempting to build a workflow platform for scientists closely integrating calculation, visualisation and analysis. From an STFC point of view, integration with the MANTID workflow framework [4] for neutron scatterers and DAWN [5] for Synchrotron X-Ray users has been identified as an important goal. In each of these cases, the end-user no longer interacts directly with the simulation code but through an additional software layer, delegating run setup, control and steering, as well as automatic parsing, sorting, data-sifting and possibly database entry. For this, an API and interface to the CASTEP routines would have great benefits compared to file-based I/O.

The eventual goal will be to provide an API, application program interface, interface which will allow CASTEP's high-level internal routines to be accessible from Python code – essentially to turn CASTEP into a library. This will allow far more flexibility than current parser-driven Fortran driver code by

- Allowing the major "driver" logic of a calculation to be written and rewritten by a larger set of peripheral scientist/software developers cum end-end users.

- Making “parameter-sweep” type calculations easy to implement using looping, for example convergence testing, including extraction, collation and graphing of calculation results
- Providing a framework for direct implementation of “high-throughput” and AIRSS-style calculations,
- Allowing tight integration with a variety of existing frameworks including AIRSS, ASE, MANTID, DAWN and more.

Python is the strongest choice of language because

- It is available on all major platforms and operating systems including HPC-class machines
- It is widely taught at undergraduate level, and as a foundation for computational physicists [6]
- It provides a low entry-barrier for new developments around CASTEP
- It is a scripting language, so short-circuits the compile-edit-run cycle during the development process
- A number of software tools are available to automate or semi-automate the generation of interfaces with CASTEP Fortran code
- A wide range of scientific libraries such as NUMPY, GUI and graphics libraries are available and under active development.

However some challenges must be overcome before CASTEP is fit for use as a library;

- It was not designed as a library and the current Fortran API will require refactoring for library use.
- It was not designed to be re-entrant and makes extensive use of module-scope (“static”) variables.
- There is extensive caching of intermediate computation results which cannot be avoided for computational efficiency. More flexible catalogued caching will be required.
- The new API should provide direct access to CASTEP's basic data types (“classes”) so they can be manipulated by Python code.
- The resulting code must be maintainable. It is vital to avoid or hide complexity which would inhibit further development.

The twofold goals of this pilot project are more modest; to explore and document the current features of the code which are obstacles to re-entrancy, and to assess the feasibility of automatic interface generation using available software tools.

2 Technology

There is often a need for programs written in different languages to interface with each other. Common reasons include the fact that one language may not be suited to all requirements, for example, a language that is very efficient at matrix operations may not be the language of choice for a GUI. It may also be undesirable or impractical to port code to a new language. For example, the effort to port LAPACK [7], a larger numerical linear algebra package, to a different language has been stated to represent more than fifty person-years of programming time [8]. Ongoing maintenance following a code port to another language may require much additional work, and accumulation of unavoidable, if unintentional, differences as the code evolves [8].

One alternative approach is to provide an interface layer of software implementing an inter-language interface. An interface between Python and CASTEP is desirable as it would allow the growing Python community with in the scientific computing field access to CASTEP functionality. Some of the issues that would arise from building such an interlanguage interface can be anticipated. Common problems include array index origin differences, array storage-order differences, run time differences such as I/O handling and differing representations of data types [8]. Another approach would be a loosely-coupled approach using AWK and shell scripts to make such programs work together. However this approach has many inherent problems and limitations that will surface in the long run [9].

This project looked into the suitability of existing technologies that automated the process of building Python wrappers. Manually wrapping the CASTEP source in Python would be impractical due to the codebase having approximately 250,000 lines of code. Tools researched included:

- f2py [10] [11]: now part of NumPy, the fundamental package for scientific computing with Python [12]
- f90wrap [13]: developed by James Kermode to overcome shortcomings in f2py.
- Forthon [14]: an alternative Python-Fortran connection tool developed by Berkeley Lab.
- PyFort [15]: another Python-Fortran connection tool developed by Lawrence Livermore National Laboratory
- Fwrap [16]: another Python-Fortran connection tool
- Cython [17] [18], in conjunction with a C interface implemented using ISO_C_BINDING

This project was intended as a proof of concept that top-level task control logic for the underlying Fortran code could be implemented in Python, and therefore picked a single, promising route using the f2py and f90wrap tools. These were chosen in part because of their shallower learning curve - trivial examples of a Python-Fortran interface using derived types were constructed quickly - and because of the availability of expert support from f90wrap developer James Kermode. No attempt was made to evaluate other tools but it is recommended that any follow-on project should undertake such a comparison prior to committing significant effort to a port.

2.1 f2py

f2py scans Fortran code to produce signature files (.pyf files). The signature files contain all the information (function names, arguments and their types, etc.) that is needed to construct Python bindings to Fortran (or C) functions. The syntax of signature files is borrowed from the Fortran 90/95 language specification and has some f2py specific extensions. The signature files can be modified to dictate how Fortran (or C) programs are called from Python [19].

All basic Fortran types, multi-dimensional arrays of all basic types as well as attributes and statements are supported by f2py but derived types are not [19]. This is unlikely to change as it “may require the next edition of f2py” [20] and development appears to have stalled since around 2005.

Practice has shown that f2py generated interfaces (to C or Fortran functions) are less error prone and even more efficient than handwritten extension modules. The f2py generated interfaces are easy to maintain and any future optimization of f2py generated interfaces transparently apply to extension modules by just regenerating them with the latest version of f2py [21].

f2py seemed the most viable option initially, with trivial examples of a Python-Fortran interface able to be constructed quickly, although additional work would be needed to overcome the lack of derived type support.

2.2 f90wrap

f90wrap is a tool that overcomes the short comings of f2py with regards to derived types. f90wrap parses Fortran code and first creates a Fortran wrapper where references to derived types have been replaced with opaque pointers, which f2py can handle. These f90wrap files are then passed to a slightly modified version of f2py, f2py-f90wrap, which creates the shared library file [22].

3 Serially Re-entrant CASTEP

A serial re-entrant method is one that has been defined as a method that must not leave anything behind that changes the path of a subsequent process [23]. A serially re-entrant CASTEP would be able calculate to properties of one material, then calculate the properties of another material and have the results be consistent with the same materials simulated on different execution images. To identify and investigate obstacles in the code which prevent re-entrant use, in the style of a library, a top-level control module was developed which ran two separate calculations within a single execution image.

The execution of this single image was compared to the same calculations running as separate images, differences were tracked down using a comparative debugging approach with the singular runs acting as the working baseline. This process was aided with the TotalView debugger [24].

Changes were made to both top-level control module as well as the underlying code in the Functional, Fundamental and Utility directories. A patch file with these changes is available on CCPForge, available to registered CASTEP users.

For a description of the underlying structure of the CASTEP source code, see Appendix E.

3.1 Top-level control module

In order to simulate a second material with CASTEP in a single execution image, the following steps must be undertaken before beginning the second simulation.

- De-allocate `main_model`; this acts as a container for the properties of the system that have been calculated.
- Finalise and then initialise the `trace` module which provides timing data via the `trace_entry()` and `trace_exit()` subroutines. Internally, `trace` times how long CASTEP spent between the entry and exit point and associates that time with the subroutine [25].
- Read the new cell file. A cell file defines the data associated with a unit cell and stores the current unit cell which is to be modelled.
- Read or generate new pseudopotentials in `ion_atom`. The `ion` module provides the data and operations which describe the properties of the ionic cores of the atoms in the system. Note, `ion` was not de-allocated, as such a method does not exist, instead the top-level control module used `ion_read()` to overwrite data.
- Read the new param file, which stores all the parameters that control the calculation and provides default values
- Call `cell_distribute_kpoints()`, this subroutine distributes arrays containing k-point data among processors in a parallel run. In a serial run, it copies the initial data structure into the `current_cell` data structure.
- Initialise `ion`, again, without de-allocating it previously as no routine exists.
- Initialise `basis` again, `basis` contains all of the code for generating and storing the distribution of the FFT grids and plane wave basis set
- Call `ion_real_initalised()` which initialises memory within the ion module
- initialise the new `main_model`

- Call `ewald_reset()` to explicitly set the `ewald` module to behave as if this is a first pass. This module calculates the ion-ion contribution to the total energy (and its derivatives) using the Ewald sum technique.

After the second `electronic_minimisation()`, clean up, such as de-allocating the model, exiting `trace` and finalising CASTEP should be called.

3.2 Changes to CASTEP Functional/, Fundamental/, Utility/ source

For full technical details on the changes to `trace`, `ewald` and `ion` please refer to Appendices B, C and D. The following is a brief summary of the changes.

3.2.1 trace.f90

CASTEP contains a `trace` module which provides, amongst other functionality, timing data via the `trace_entry` and `trace_exit` subroutines. There is a call to `trace_entry()` at the start of most CASTEP subroutines, and a corresponding call to `trace_exit()` at each exit point. Internally the `trace` module times how long CASTEP spent between the entry and exit point, and associates that time with the subroutine. The `trace` module also supports the association of a subroutine with a class of operation, allowing the large amount of timing data to be reported per class of operation rather than per individual subroutine [25].

Because the `trace` module was not being reset between the runs, it was maintaining some state after the first run. This saved state was removed by finalising and then re-initialising the module between the two runs, ensuring all allocated arrays are de-allocated by the finalising method, removing the `save` attribute from variables and reworking the code so no variable inherits the `save` variable implicitly.

The above changes did not have a noticeable effect on the reliability of timings produced by the `trace` module; however this was not tested rigorously.

3.2.2 ewald.f90

The `ewald` module was not designed with serial re-entrancy in mind. It maintains a lot of persistent data, to avoid re-computing an unchanged Ewald contribution. This contribution remains the same provided the unit cell has not changed.

Each subroutine in `ewald` had a `first_pass` variable which is one of the ways CASTEP uses to determine whether it needs to recalculate the Ewald contribution. Because these variables are subroutine level variables, they cannot be changed outside the subroutine. Changing the `first_pass` variables was necessary to clear the persistent data after reading a new .cell file.

The first solution was to create a module level `first_pass` variable that all subroutines would share and a method to reset that variable when required. However, all the `ewald` subroutines sharing the same `first_pass` variable led to a failure of the CASTEP test suite. As such, a separate `first_pass` module variable was created for the offending subroutine whilst the other subroutines continued to use their subroutine level variable.

In future, the way the `ewald` module maintains persistent data will need to be changed to allow for serial re-entrancy.

3.2.3 ion.F90

The changes made to the `ion` module highlight a situation where the cause of the problem and the effects of the problem are fairly separate.

In the `ewald` module, an array could not be de-allocated during the second run. This was tracked down to be a memory leak in ion, in which an array was not being de-allocated and re-allocated between the two runs. The cause of this error was an if-statement of the form, “if not allocated, then allocate”. The intention was very likely to cache a computational result between calls under the assumption that the pseudopotentials are never changed during a run. Re-entrant use breaks this assumption. During the first run the behaviour is as expected, but for the second run the array remained allocated and as such was not reallocated, causing the memory leak. The if-statement was changed to read, “if the array is allocated, then de-allocate and later re-allocate.

However, that this is not the optimal, general, solution as there is frequently a need to preserve data from one call to the next. The current workaround could result in certain cases failing, where data is meant to be preserved, similar to what was observed with the naïve attempt to reset `first_pass` in `ewald`. Perhaps a method to de-allocate `ion` is needed, this would allow a programmer to explicitly tell CASTEP to de-allocate arrays.

4 Python interface to CASTEP

A hand written Python interface to CASTEP is impractical to achieve due to its large codebase. The tools f90wrap and f2py were used to automate this process. f90wrap generates Fortran wrappings of the selected source code, so that the derived types are concealed through the use of opaque pointers. These wrappers are then passed to a patched version of f2py to generate the Python library file to allow Python and CASTEP to interface.

4.1 Wrapping

Because f90wrap is still under development, it was decided to focus attention on porting the cut down, top-level control file from the first half of the project to Python, rather than the complete functionality of CASTEP. As such the following modules were identified as being necessary after a visual inspection of control.

Utility/	Fundamental/	Functional/
• constants	• parameters	• model
• algor	• cell	• electronic
• comms.serial	• basis	• ewald
• io	• ion	
• trace	• density	
• license	• wave	

`constants` was identified as being the “easiest” module to wrap first, as it did not rely on any other modules. The process identified features of Fortran that the f90wrap tool was not initially programmed to handle, although these were incorporated into the tool relatively quickly. From there, the Utility modules listed were wrapped, followed by Fundamental and Functional, with the effort necessary decreasing with each module. `ewald` was not needed to be wrapped originally for a simple proof of concept Python control logic, but was needed for a serial re-entrant example and as such f90wrap did not need to be modified specifically to handle it. Because the code style is very consistent in CASTEP, `ewald` was wrapped without any errors or changes.

Some changes to the Fortran code base were necessary to allow the current f90wrap to create a CASTEP library file that could be used in a Python script. A patch file is also available on CCPForge, available to registered CASTEP users.

1) Array declarations of the form[1]:

```
real(kind=dp), dimension(:,:,:), intent(out) :: eigenvalues
```

needed to become

```
real(kind=dp), dimension(size(occ,1),size(occ,2),size(occ,3)),
intent(out) :: eigenvalues
```

This is considered a general limitation of Fortran/Python interfacing, rather than a fault in f2py or f90wrap. The explicit dimensions are required because these arrays are declared as `intent(out)` in Fortran. The Python equivalent of an `intent(out)` argument is an extra returned item, and since this extra argument doesn't exist on entry to the subroutine we need to know how big it will be in order to allocate it.

For a subroutine with a single `intent(out)` argument, the Fortran and corresponding Python calls look similar like this:

A Fortran soubroutine "sub" with intent(out) argument "out_array":

```
! size is the known dimension of the array
allocate(out_array(size))
! do something with out_array
```

Python method calling the Fortran subroutine "sub":

```
out_array = sub() # out_array is implicitly allocated by the Fortran
                  #subroutine
```

There are two alternatives to providing explicit dimensions. Firstly, f2py could automatically add an extra integer argument to specify the size of each unknown dimension, so the Python call about would become:

```
size = 100  #user has to know in advance how big return array will
            #be
out_array = sub(size)
```

Alternatively, `intent(out)` arguments could be converted to `intent(in, out)`, and require the Python user to take care of allocating memory, but this is error-prone as they need to ensure the memory is Fortran-contiguous:

```
size = 100 # still need to know size in advance
out_array = np.zeros(size, order='F')
sub(out_array)
```

2) '`target`' attribute to module elements of derived type are needed i.e.:

[1] In this example, occ is the "KS Band Occupancies" and is an array passed to the `electronic_minimisation` routine.

```
type(unit_cell), public, save ::current_cell ! Declare current_cell
```

becomes

```
type(unit_cell), public, save, target :: current_cell ! Declare
current_cell
```

The target attribute is required for module-level derived type instances because Python accesses such variables by passing around an opaque pointer to them. The Fortran standard insists that anything that can be the target of a pointer must have the target attribute. This is not a problem for derived type instances created from Python on-the-fly with `allocate()`, or for derived types within other derived types, so it only affect module-level variables like `current_cell` and `current_params`.

3) Patch `io_initialise()` so that it works without command line arguments

The motivation for this project was to be able to swap in and out different materials and perform calculations on them in the fashion of a library; hence requiring command line input of material properties is no longer required or useful.

4.2 Usage

After the successful creation of the CASTEP library file, it can be used from Python by adding the following to a Python file.

```
import castep
```

Methods and constants are then accessed by:

```
castep.modulename.methodname
castep.modulename.constant
```

where "`castep`" refers to the shared object library imported, so if the module has been aliased, the alias name must be used in the place of "`castep`".

For example:

```
print castep.constants.pi
print castep.constants.cmplx_i
```

would print 3.141592665359 and 1j respectively

```
castep.cell.cell_read("test")
```

would read the cell file called test.cell

```
current_cell = castep.cell.get_current_cell()
```

would then set Python variable `current_cell` to be the cell read by the preceding `cell_read()` call.

Data from within the current cell could then be accessed by, e.g.:

```
        current_cell.num_ions_in_species[ispec]
```

Python classes corresponding to Fortran types, such as `cell`, are automatically created by f90wrap and can be accessed by

```
        typename.membername
```

as in the `current_cell` example above.

Two example Python CASTEP top level control modules haves been implemented. One, called `singlepoint.py` performs one electronic minimisation. The other, called `castep_reentrant.py` performs two successive, serially re-entrant, electronic minimisations. For this example to work correctly, both the `serial_re_entrancy` patch and the `PyCASTEP` patch must be applied. Both files can be found on CCPForge, available to registered CASTEP users.

5 Summary

The two aims of the project were to:

1. Explore and document the current features of the code which are obstacles to re-entrancy
2. Assess the feasibility of automatic interface generation using available software tools

These aims have been met.

A serially re-entrant Fortran example, involving two calls to the `electronic minimisation()` routine within the same execution image, has been created. See section 3 for details on how serial re-entrancy was achieved. This example highlighted three coding practices that may inhibit full serial re-entrancy. Full serial re-entrancy is be required before a future move of CASTEP away from an ASCII file I/O model to an API interface would be possible.

A Python interface to CASTEP has also been generated using the f90wrap tool. Using this interface, simple examples of a Python top level driver have been created. A serially re-entrant Python example has also been created. See section 4.2 for details on using the interface. The interface is maintainable as any future changes to CASTEP can be propagated by regenerating the interface with the latest version of f90wrap.

Having achieved both these aims, "high-throughput simulation" and “parameter-sweep” type calculations will be easier to implement and be implementable by a larger set of peripheral scientist/software developers cum end-end users. The interface also allows for tighter integration with a variety of existing frameworks including AIRSS, ASE, MANTID and DAWN.

6 Future

The work undertaken to create a serial re-entrant example of CASTEP execution and to create a partial Python wrapper of CASTEP serve as a proof of concept only. At this stage, it cannot be said that the entirety of CASTEP is serially re-entrant. However the work has shown that a subsequent full scale project is feasible. Such a project has been adopted by the CASTEP developers.

This pilot project has provided insight into what a full migration to serial re-entrancy might involve.

6.1 Recommendations for serial re-entrancy

Very few changes to the code base have had to be made. This speaks to the robustness and good coding practices of the CASTEP code. Below is a summary of coding practises that may impact serial-re-entrancy in the future.

1) Variables with the `save` attribute have the potential to be a problem for serial re-entrancy. Also any variable that is declared and initialised at the same time, i.e.:

```
integer :: current_index=-1
```

automatically inherits the `save` attribute and hence is likely to be a problem for serial re-entrancy, as after one run there is no guarantee of the state of `current_index`. Such instances should be avoided where possible and replaced with an explicit initialisation in the initialise method or a reset method. The modified `trace_init()` method now explicitly sets `current_index=-1` in `trace_init()`, rather than at declare time.

2) If-statements of the form:

```
if(.not.allocated(QLnm_to_index)) then
      allocate(QLnm_to_index(0:2*lmax,max_ps_projectors, &
      max_ps_projectors,current_cell%num_species),stat=ierr)
end if
```

may cause problems with serial re-entrancy as well. This is because on subsequent runs the arrays could be allocated, hence they would not be reallocated, and remain filled with persistent data. Worse still, this can lead to memory leaks which do not manifest as problems until much later in the code, as was the case with `ion`. In this specific case of `ion`, the above has been replaced by:

```
if(allocated(QLnm_to_index)) then
             deallocate(QLnm_to_index, stat=ierr)
end if
allocate(QLnm_to_index(0:2*lmax,max_ps_projectors, &
max_ps_projectors,current_cell%num_species),stat=ierr)
```

although a more general approach may be to ensure modules such as ion have explicit de-allocate methods, which would allow data to persist if not explicitly de-allocated. The fact that `ion` is the only module that needed to be (re)initialised and didn't have a de-allocate method indicates that this may be the better, more general, approach for future modules and modifications.

3) Initialise and finalise

Certain modules, such as `ion`, have a initialise method but not a finalise method. This has resulted in the initialise method also serving as a re-initialise method for this project, requiring additional checks to be included to check for arrays being allocated arrays. To remain consistent with the style of other modules, `ion` could have a finalise method that explicitly dealt with the de-allocation of persistent data, rather than relying on the implicit clean-up that the initialise method conducts.

4) Re-initalise

If a module relies heavily on persistent data, the module could be given a re-initialise method. This method would fall short of a full de-allocate and only de-allocate and re-allocate data that is no longer needed. The reset method added in `ewald` is an example of such a method.

6.2 A Python wrapping of CASTEP

The Python wrapper created for the serial re-entrant example is very close to a complete wrapping of CASTEP. Even if additional changes to CATSEP or f90wrap are needed these are unlikely to be major changes and can be implemented quickly. A complete Python wrapper would open up a new potential user base for CASTEP and provide an opportunity to write unit tests for low level CASTEP routines in Python, using the wrapper generated by f90wrap. It would also enable new science by rapid Python development of high level functionality.

7 Bibliography

[1] The Materials Project, “Materials Project,” 20 Feb 2015. [Online]. Available: https://www.materialsproject.org/. [Accessed 20 02 2015].

[2] UCL, “Ab Initio Random Structure Searching,” UCL, 19 11 2012. [Online]. Available: http://www.cmmp.ucl.ac.uk/~ajm/airss/airss.html. [Accessed 20 02 2015].

[3] Danmarks Tekniske Universitet, “Atmoic Simulation Environment,” Danmarks Tekniske Universitet, 20 02 2015. [Online]. Available: https://wiki.fysik.dtu.dk/ase/. [Accessed 20 02 2015].

[4] Mantid, “Mantid,” Mantid, 09 02 2015. [Online]. Available: http://www.mantidproject.org/Main_Page. [Accessed 20 02 2015].

[5] DAWN Science, “DAWN,” DAWN Science, 13 02 2015. [Online]. Available: http://www.dawnsci.org/. [Accessed 20 02 2015].

[6] Software Carpentry, “Software Carpentry,” Software Carpentry, 19 02 2015. [Online]. Available: http://software-carpentry.org/. [Accessed 20 02 2015].

[7] LAPACK, “LAPACK — Linear Algebra PACKage,” netlib, 19 11 2013. [Online]. Available: http://www.netlib.org/lapack/. [Accessed 04 02 2015].

[8] N. H. F. Beebe, “Using C and C++ with Fortran,” Department of Mathematics - University of Utah, 16 11 2001. [Online]. Available: http://www.math.utah.edu/software/c-with-fortran.html. [Accessed 04 02 2015].

[9] M. F. Sanner, “PYTHON: A PROGRAMMING LANGUAGE FOR SOFTWARE INTEGRATION AND DEVELOPMENT,” *J Mol Graph Model ,* vol. 17, no. 1, pp. 57-61, 1999.

[10] P. Peterson, “F2PY: a tool for connecting Fortran and Python programs.,” *International Journal of Computational Science and Engineering,* vol. 4, no. 4, pp. 296-305, 2009.

[11] P. Peterson, “F2PY Users Guide and Reference Manual,” SciPy.org, 02 04 2005. [Online]. Available: http://docs.scipy.org/doc/numpy-dev/f2py/. [Accessed 08 10 2014].

[12] NumPy, “NumPy,” NumPy, 23 10 October. [Online]. Available: http://www.numpy.org/. [Accessed 04 04 2015].

[13] J. Kermode, “f90wrap,” GitHub, 28 08 2014. [Online]. Available: https://github.com/jameskermode/f90wrap. [Accessed 09 10 2014].

[14] D. Grote, “Forthon,” Berkeley Lab, 31 05 2014. [Online]. Available: http://hifweb.lbl.gov/Forthon/. [Accessed 04 04 2015].

[15] P. F. Dubois, “Pyfort -- The Python-Fortran connection tool,” sourceforge, 12 10 2012. [Online]. Available: http://pyfortran.sourceforge.net/. [Accessed 08 10 2014].

[16] K. Smith, “Fortran for Speed, Python for Comfort — fwrap v0.1.0,” sourceforge, 2010. [Online]. Available: http://fwrap.sourceforge.net/. [Accessed 13 10 2014].

[17] D. S. Seljebotn, “Fast numerical computations with Cython,” in *Proceedings of the 8th Python in Science Conference (Vol. 37)*, 2009.

[18] Cython, “Cython: C-Extensions for Python,” Cython, 12 02 2015. [Online]. Available: http://cython.org/. [Accessed 20 02 2015].

[19] M. Bertini, “F2py,” SciPy.org, 27 01 2011. [Online]. Available: http://wiki.scipy.org/F2py. [Accessed 08 10 2014].

[20] P. Virtanen, “numpy/FAQ.txt,” GitHub, 02 01 2014. [Online]. Available: https://github.com/numpy/numpy/blob/master/doc/f2py/FAQ.txt. [Accessed 08 10 2014].

[21] P. Peterson, “README.txt,” GitHub, 02 01 2014. [Online]. Available: https://github.com/numpy/numpy/blob/master/doc/f2py/README.txt. [Accessed 09 10 2014].

[22] J. Kermode, “useage.rst,” GitHub, 17 08 2014. [Online]. Available: https://github.com/jameskermode/f90wrap/blob/master/docs/usage.rst. [Accessed 10 10 2014].

[23] IBM, “IBM Knowledge Center,” IBM, 01 01 2013. [Online]. Available: http://www-01.ibm.com/support/knowledgecenter/SSB23S_1.1.0.9/com.ibm.ztpf-ztpfdf.doc_put.09/gtpa2/pgrentt.html. [Accessed 04 04 2015].

[24] RogueWave, “TotalView,” RogueWave, [Online]. Available: http://www.roguewave.com/products-services/totalview. [Accessed 04 02 2015].

[25] N. Drakos, “Implementation,” hector.ac.uk, 27 10 2014. [Online]. Available: http://www.hector.ac.uk/cse/distributedcse/reports/castep04/castep04/node8.html. [Accessed 03 02 2015].

[26] M. Segall, P. Lindan, M. Probert, C. Pickard, P. Hasnip, S. Clark, K. Refson and M. Payne, “Module Specification for New Plane Wave Code,” 2003.

[27] S. Clark, “Intro to Castep.ppt,” 20 08 2009. [Online]. Available: http://www.tcm.phy.cam.ac.uk/castep/oxford/castep.pdf. [Accessed 03 02 2015].

[28] B. Jesson, “CASTEP: Quantum Mechanical Atomistic Simulation Code,” 20 07 2004. [Online]. Available: http://www.csar.cfs.ac.uk/about/csarfocus/focus5/castep.pdf. [Accessed 11 02 2015].

8 Appendices

A. Applying Patches

- To run the serial re-entrant CASTEP example, apply the `serial_re_entrancy` patch to Mercurial changeset 478e6b98142b.
- To generate the partial Python wrapper using f90wrap and run the Python example, apply the `PyCASTEP` patch to Mercurial changeset 8c3fd681241a.
- To run the serially re-entrant Python example, both patches must be applied.

B. trace.f90

CASTEP contains a `trace` module which provides, amongst other functionality, timing data via the `trace_entry()` and `trace_exit()` subroutines. There is a call to `trace_entry()` at the start of most CASTEP subroutines, and a corresponding call to `trace_exit()` at each exit point. Internally the `trace` module times how long CASTEP spent between the entry and exit point, and associates that time with the subroutine. The `trace` module also supports the association of a subroutine with a class of operation, allowing the large amount of timing data to be reported per class of operation rather than per individual subroutine [25].

Because this `trace` module was not being reset between the runs, it was in an unknown sate for the start of the second run. Using comparative debugging, two variables were identified to be maintaining some state after the first run.

The first of the two variables, `child_index`, was corrected by adding a `trace_finalise`() and a second `trace_init()` was added.

This alone was insufficient to fix the maintained state problem as a second variable, `current_index`, was set at definition time and was not being re-initialised in `trace_init()`. This was due to the variable being initialised and instantiated in one line, meaning it inherited the `save` attribute. A line of code was added to `trace_init()` to re-initialised `current_index` and the one line initialise/instantiate was removed.

Also, `trace_finalise()` was not de-allocating an array which `trace_init()` was later allocating, which cause problems when `trace_init()` was called a second time. A de-allocation clause was added to `trace_finalise()` to de-allocate the array if allocated.

The above changes did not have a noticeable effect on the reliability of timings produced by the `trace` module; however this was not tested rigorously.

C. ewald.f90

Subroutine `ewald_reset()` was added to `ewald` to allow for explicit setting of `first_pass` to true.

`first_pass` was initialized to true when it was declared, within a function, and so subsequent calls to `ewald` functions had a persistent value for `first_pass`, namely false.

However in some cases we wish the data to persist, to prevent re-calculating a constant term. If the cell changes, then we need to recalculate and `first_pass` should be set to true.

The first attempted solution was to have all the subroutines in the ewald module sharing a first_pass variable. But this caused a total failure of the test suite. Presumably because functions like calculate_energy() would be followed by calculate_forces(), but would need to behave as if it was a first pass.

As such, a separate first_pass_calculate_energy variable was added. In future, all the routines that use first_pass variables may need their own private version, however hopefully only one reset method will be needed.

D. ion.F90

In ewald, old_ion_position could not be deallocated during a second run. This was tracked down to be a memory leak in ion, in which an array was not being deallocated and reallocated between the two runs. The cause of this error was

```
if(.not.allocated(QLnm_to_index)) then
      allocate(QLnm_to_index(0:2*lmax,max_ps_projectors, &
      max_ps_projectors,current_cell%num_species),stat=ierr)
end if
```

During the second run, QLnm_to_index was not reallocated because it was already allocated. This code was changed to:

```
if(allocated(QLnm_to_index)) then
             deallocate(QLnm_to_index, stat=ierr)
end if
allocate(QLnm_to_index(0:2*lmax,max_ps_projectors, &
max_ps_projectors,current_cell%num_species),stat=ierr)
```

It has been noted, however, that this is not the optimal, general, solution as there is frequently a need to preserve data from one call to the next. The current workaround could result in certain cases failing, where data is meant to be preserved, similar to what was observed with the naïve attempt to reset first_pass in ewald. Perhaps a method to de-allocate ion is needed, this would allow a programmer to explicitly tell CASTEP to de-allocate arrays.

There were several more if(.not.allocated…) present in ion, that may cause problems for future serial re-entrancy.

E. CASTEP code structure

The underlying CASTEP code is split into three trees called “Functional”, “Fundamental” and “Utility”. The `Functional' modules provide a physical function for the calculation. The `Fundamental' modules are objects providing fundamental data, data types and operations that may be used in any of the functional modules. The Utility modules contain low level code for I/O, FFTs, MPI communication, physical and mathematical constants and numerical algorithms.

The structure is visualised in Appendix F. Module dependencies within each category of module are shown by solid line (lines are omitted from the highest level modules for clarity). Modules may only use those below them in this diagram [26].

F. CASTEP code structure diagram

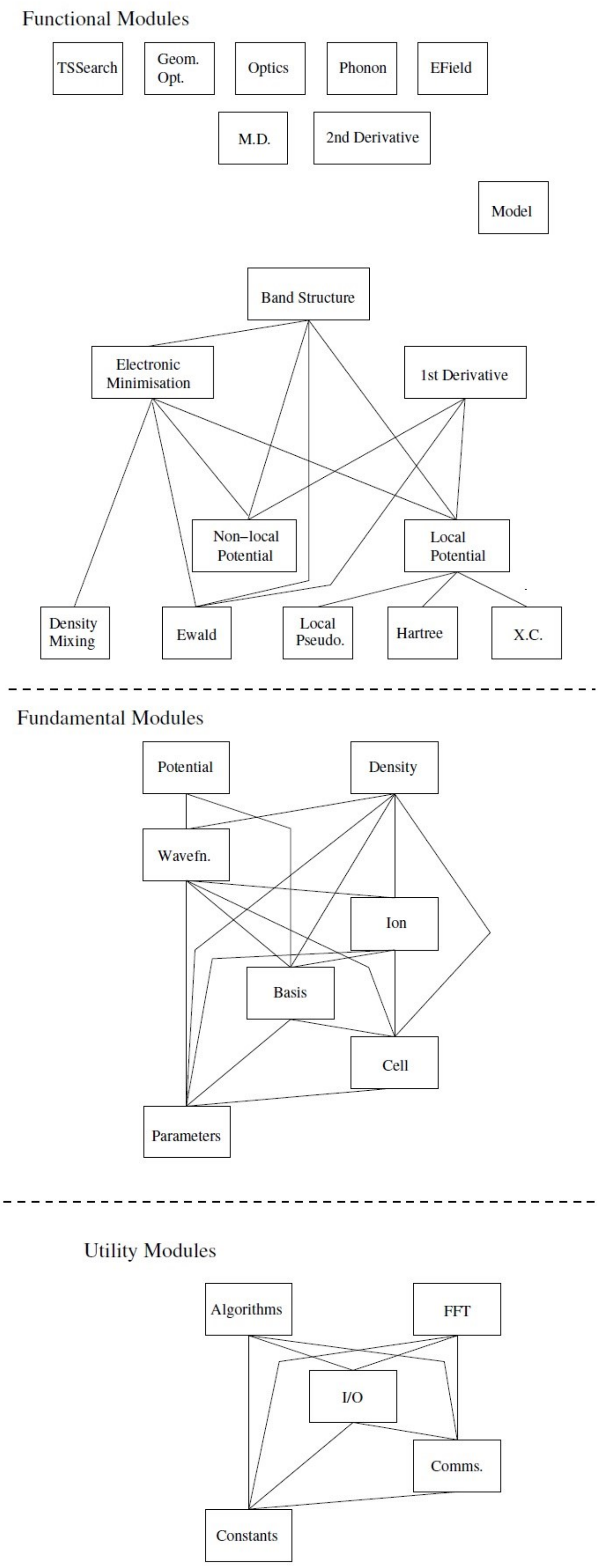

Figure 1: The CASTEP code structure diagram [26]