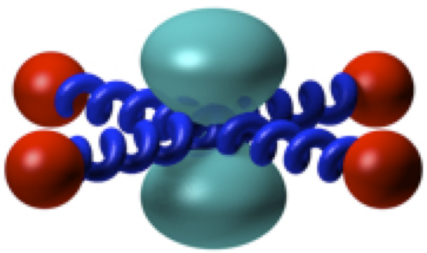
Interoperability between electronic structure codes with the quippy toolkit

James Kermode

Department of Physics
King's College London

6 September 2012



Outline

Outline

The libAtoms, QUIP and quippy packages

- The libAtoms[1] package is a software library written in Fortran 95 for the purposes of carrying out molecular dynamics simulations.

[1] http://www.libatoms.org
[2] Csányi et al., PRL (2004)
[3] http://www.jrkermode.co.uk/quippy

The libAtoms, QUIP and quippy packages

- The libAtoms[1] package is a software library written in Fortran 95 for the purposes of carrying out molecular dynamics simulations.
- The QUIP (QUantum mechanics and Interatomic Potentials) package, built on top of libAtoms, implements a wide variety of interatomic potentials and tight binding quantum mechanics, and is also able to call external packages.

[1] http://www.libatoms.org
[2] Csányi et al., PRL (2004)
[3] http://www.jrkermode.co.uk/quippy

The libAtoms, QUIP and quippy packages

- The libAtoms[1] package is a software library written in Fortran 95 for the purposes of carrying out molecular dynamics simulations.
- The QUIP (QUantum mechanics and Interatomic Potentials) package, built on top of libAtoms, implements a wide variety of interatomic potentials and tight binding quantum mechanics, and is also able to call external packages.
- Various hybrid combinations are also supported in the style of QM/MM, including 'Learn on the Fly' scheme (LOTF).[2]

[1] http://www.libatoms.org
[2] Csányi et al., PRL (2004)
[3] http://www.jrkermode.co.uk/quippy

The libAtoms, QUIP and quippy packages

- The libAtoms[1] package is a software library written in Fortran 95 for the purposes of carrying out molecular dynamics simulations.
- The QUIP (QUantum mechanics and Interatomic Potentials) package, built on top of libAtoms, implements a wide variety of interatomic potentials and tight binding quantum mechanics, and is also able to call external packages.
- Various hybrid combinations are also supported in the style of QM/MM, including 'Learn on the Fly' scheme (LOTF).[2]
- quippy[3] is a Python interface to libAtoms and QUIP.

[1] http://www.libatoms.org
[2] Csányi et al., PRL (2004)
[3] http://www.jrkermode.co.uk/quippy

Main QUIP authors and contributors

- University of Cambridge
  - Albert Bartók-Partay
  - Gábor Csányi
  - Wojciech Szlachta
  - Csilla Várnai
- King's College London
  - James Kermode
- Naval Research Laboratory, Washington DC
  - Noam Bernstein
- Fraunhofer IWM, Freiburg
  - Lars Pastewka
  - Michael Moseler

Potentials implemented in QUIP

Classical interatomic potentials

- BKS (silica)
- Brenner (carbon)
- EAM (fcc)
- Fanourgakis-Xantheas
- Finnis-Sinclair (bcc)
- Flikkema-Bromley
- GAP (general many-body)
- Guggenheim-McGlashan
- Lennard-Jones
- Morse
- Partridge-Schwenke (water monomer)
- Si-MEAM (silicon)
- Stillinger-Weber (carbon, silicon, germanium)
- Sutton-Chen
- Tangney-Scandolo (silica, titania etc)
- Tersoff (silicon, carbon)

Plus several tight binding parameterisations (Bowler, DFTB, GSP, NRL-TB, . . .)

Potentials implemented in QUIP

External packages

- CASTEP — DFT, planewaves, ultrasoft pseudopotentials
- CP2K — DFT, mixed Gaussian/planewave basis set, various pseudopotentials. Driver supports QM, MM and QM/MM.
- MOLPRO — All electron quantum chemistry code. DFT, CCSD(T), MP2
- VASP — DFT, planewaves, PAW or ultrasoft pseudopotentials
- Interface to OpenKIM project
- Relatively easy to add new codes

Potentials implemented in QUIP

Atomic Simulation Environment (ASE)

- QUIP also has a full interface to the Atomic Simulation Environment, ASE.[4]
- ASE adds support for several more codes e.g. ABINIT, Elk, Exciting, GPAW, SIESTA, ...
- ASE also works with the CMR[5] database system

[4] https://wiki.fysik.dtu.dk/ase

[5] https://wiki.fysik.dtu.dk/cmr

Outline

Benefits of scripting interfaces

1. Primary Goals

A scripting interface provides access to all the functionality of a code written in a low level language (e.g. Fortran, C) via a high level language (e.g. Java, Python, Perl, TCL, ...).

- Can be generated either automatically or manually from the source code of the low level code

Primary benefits of scripting interfaces are:

- Preparation of input files
- Analysis and post processing of results

Benefits of scripting interfaces

2. Increased flexibility

- Can be used interactively, or via shell commands/scripts
- Interactive visualisation and structure modification
- Interoperability with other codes
- Batch processing
- Assemble existing components in new ways
- Good for less experienced programmers

Benefits of scripting interfaces

3. Improvements to the main code

- Simplify top level programs
- Encourages good software development practices
  - *e.g.* modularity, well defined APIs
- Speed up development of new algorithms and routines
- Eases creation and maintenance of unit/regression tests

Why Python?

Python has become increasingly popular for scientific computing in recent years, due to

- Clean, easy-to-learn syntax
- Very high level object-oriented language
- Availability of packages: many fast, robust mathematical and scientific tools, principally `numpy` and `scipy`,[6] but now also many more.

[6] Both available from http://www.scipy.org

Why Python?

Python has become increasingly popular for scientific computing in recent years, due to

- Clean, easy-to-learn syntax
- Very high level object-oriented language
- Availability of packages: many fast, robust mathematical and scientific tools, principally `numpy` and `scipy`,[6] but now also many more.

But aren't interpreted languages like Python really slow?

[6] Both available from http://www.scipy.org

Why Python?

Python has become increasingly popular for scientific computing in recent years, due to

- Clean, easy-to-learn syntax
- Very high level object-oriented language
- Availability of packages: many fast, robust mathematical and scientific tools, principally `numpy` and `scipy`,[6] but now also many more.

But aren't interpreted languages like Python really slow?

- Providing numerically intensive parts are vectorised, Python code can run surprisingly fast
- Mixture of high and low level languages can be ideal to maximise overall efficiency of developing *and* running codes

[6] Both available from http://www.scipy.org

Wrapping Fortran codes in Python

1. Automatic wrapper generation with `f2py`

`f2py`,[7] part of `numpy`, allows Fortran routines to be called from Python

- `f2py` scans Fortran 77/90/95 codes and automatically generates Python interfaces
- Portable, compiler independent
- Produces easy-to-use Python extension modules
- Supports all basic Fortran types, multi-dimensional arrays

[7] P. Peterson, *Int. J. Comp. Sci. Eng.* **4** 296-305 (2009)

Wrapping Fortran codes in Python

1. Automatic wrapper generation with `f2py`

`f2py`,[7] part of `numpy`, allows Fortran routines to be called from Python

- `f2py` scans Fortran 77/90/95 codes and automatically generates Python interfaces
- Portable, compiler independent
- Produces easy-to-use Python extension modules
- Supports all basic Fortran types, multi-dimensional arrays
- But no support for derived types or overloaded interfaces

[7] P. Peterson, *Int. J. Comp. Sci. Eng.* **4** 296-305 (2009)

Wrapping Fortran codes in Python

2. Derived type support via `f90wrap`

`f90wrap` adds support for Fortran 90 derived types and generic interfaces to `f2py`

- Based on Fortran 90 documentation generator `f90doc`.[8]
- Opaque interface layer wraps derived types using `transfer()`.[9]
- Thin object-oriented layer on top gives wrapped code natural (Pythonic) look and feel
- Currently part of `quippy`, but to be released in future as a standalone utility

---

[8] Ian Rutt, f90doc: automatic documentation generator for Fortran 90 (2004)

[9] Pletzer, A et al., Exposing Fortran Derived Types to C and Other Languages, *Comp. Sc. Eng.*, **10**, 86 (2008).

Outline

quippy features

- General purpose tool for:
  - Manipulating atomic configurations
  - Visualising and analysing results
  - Performing classical and *ab initio* calculations

quippy features

- General purpose tool for:
  - Manipulating atomic configurations
  - Visualising and analysing results
  - Performing classical and *ab initio* calculations
- Other similar tools exist, with different focuses, *e.g.*:
  - `ASE` – atomic simulation environment
  - `MMTK` – molecular modelling toolkit
  - `OpenBabel` – toolbox for chemical file format conversion
  - `pizza.py` – LAMMPS toolkit
  - `PyMOL` – visualisation suite, optimized for biomolecules
  - `pymatgen` – Materials Project collaboration

Creating structures

Python code

```
from quippy import *
dia = diamond(5.44, 14)
print "n:", dia.n
print "positions:"
print dia.pos.T
```

Creating structures

Python code

```
from quippy import *
dia = diamond(5.44, 14)
print "n:", dia.n
print "positions:"
print dia.pos.T
```

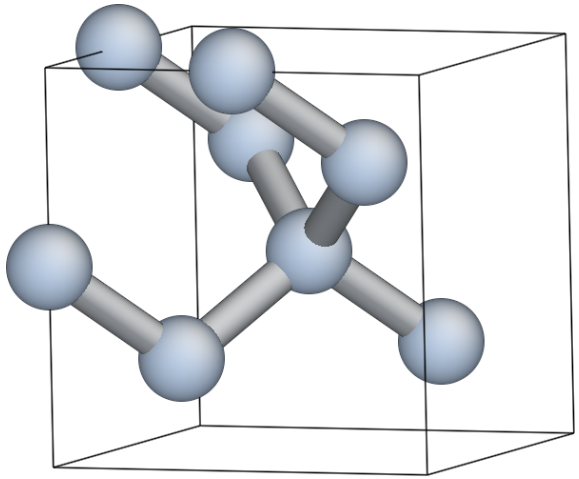
Output

```
n: 8
positions:
[[ 0.    0.    0.  ]
 [ 1.36  1.36  1.36]
 [ 2.72  2.72  0.  ]
 [ 4.08  4.08  1.36]
 [ 2.72  0.    2.72]
 [ 4.08  1.36  4.08]
 [ 0.    2.72  2.72]
 [ 1.36  4.08  4.08]]
```

Interactive visualisation

Display values

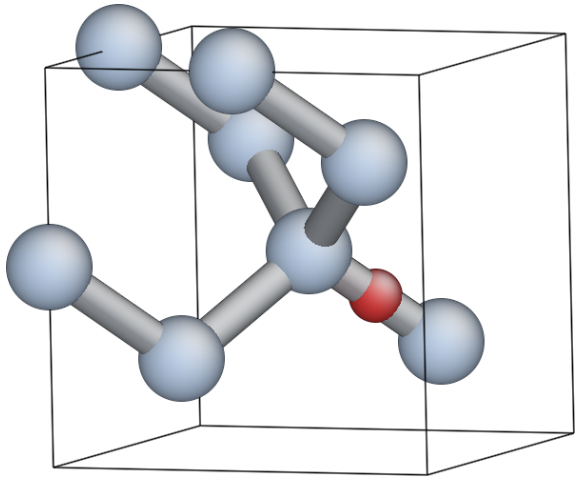
```
>>> dia.pos[1]
[ 0.  0.  0.]
>>> view(dia)
```



Interactive visualisation

Display values

```
>>> dia.pos[1]
[ 0.  0.  0.]
>>> view(dia)
```



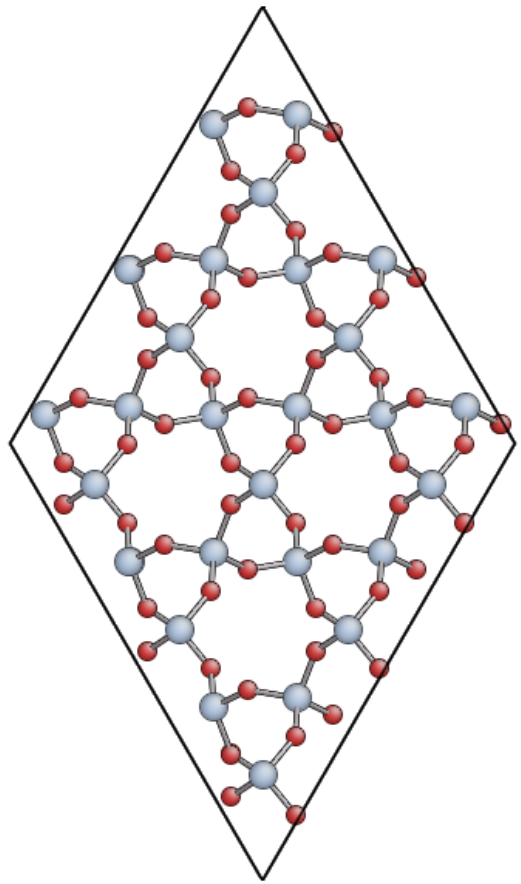
Modify data

```
O_pos = (dia.pos[1]+dia.pos[7])/2.
dia.add_atom(pos=O_pos, z=8)
redraw()
```

Manipulating atoms

α-quartz cell

```
unit = alpha_quartz(a=4.92,
                    c=5.40)
aq = supercell(unit, 3, 3, 3)
view(aq)
```



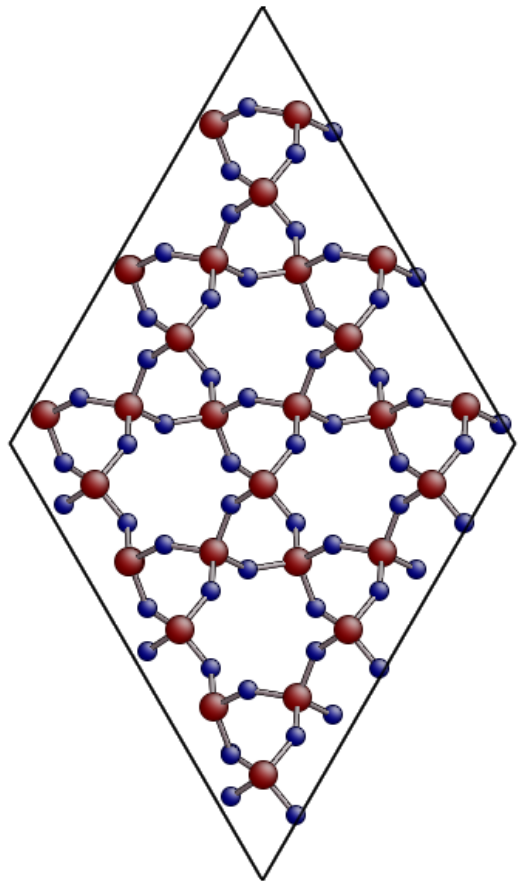
Manipulating atoms

α-quartz cell

```
unit = alpha_quartz(a=4.92,
                    c=5.40)
aq = supercell(unit, 3, 3, 3)
view(aq)
```

Custom colouring

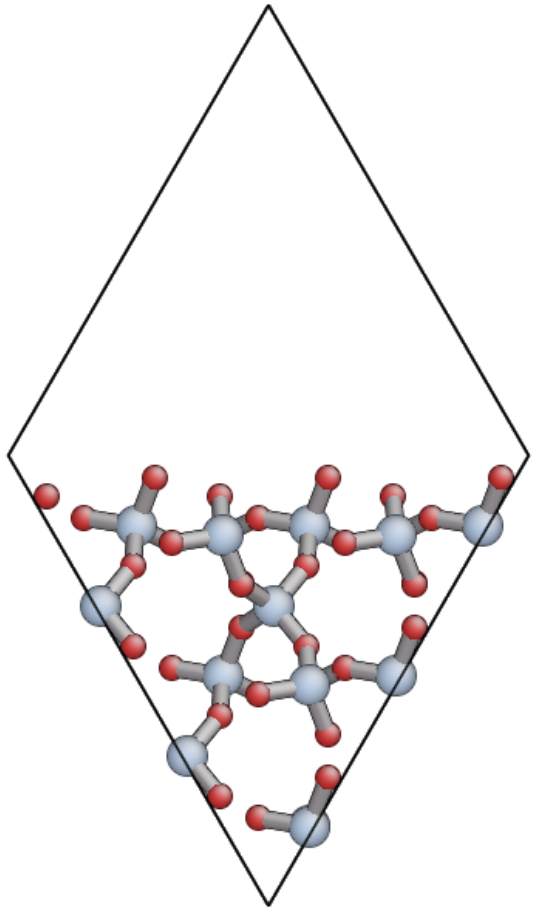
```
aq.add_property("charge", 0.0)
aq.charge[aq.z==8]  = -1.4
aq.charge[aq.z==14] =  2.8
aux_property_coloring("charge")
```



Manipulating atoms

Filtering atoms

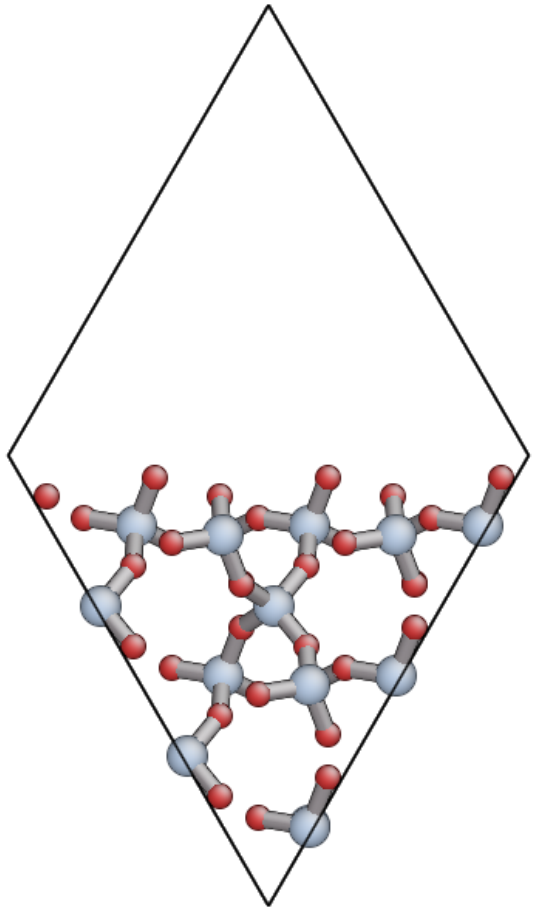
```
aq.map_into_cell()
aq2 = aq.select(aq.pos[2,:] > 0)
view(aq2)
```



Manipulating atoms

Filtering atoms

```
aq.map_into_cell()
aq2 = aq.select(aq.pos[2,:] > 0)
view(aq2)
```



Saving results

Configurations can be written out in number of formats, e.g.

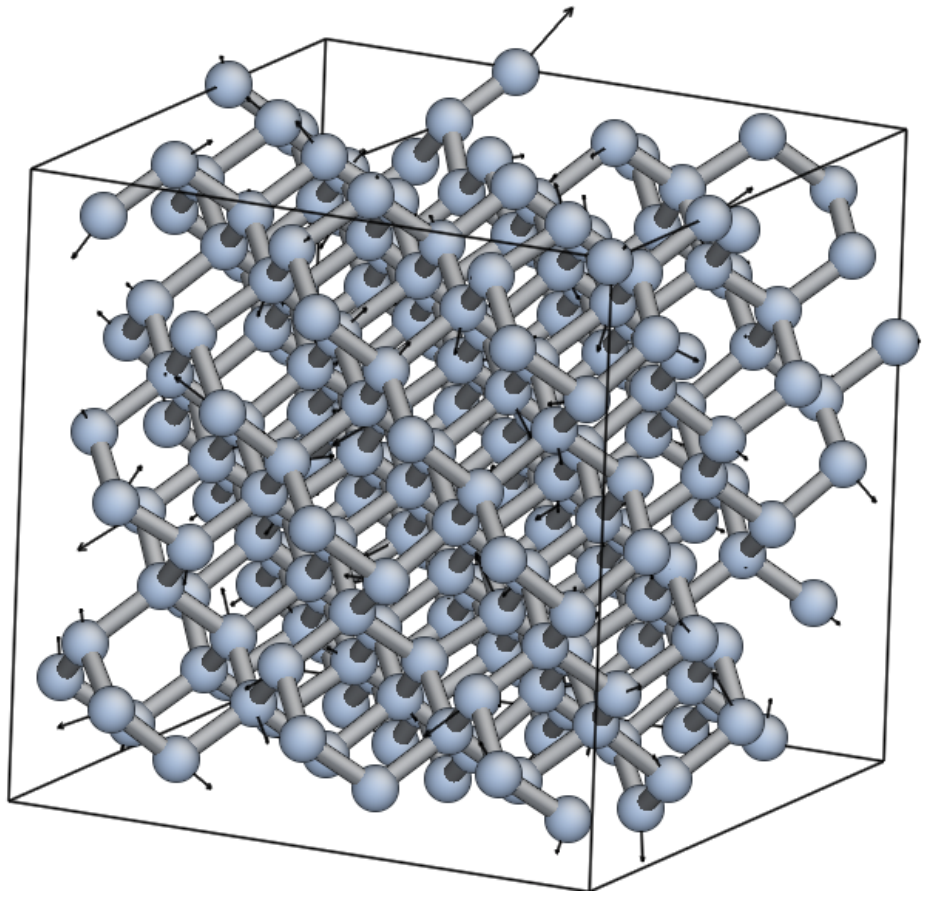
```
aq2.write('aq.xyz')   # XYZ
aq2.write('aq.cell')  # CASTEP
aq2.write('aq.cube')  # Gaussian
aq2.write('INCAR')    # VASP
```

Post-processing of results

Reading configurations

Individual snapshots or entire trajectories can be read in, also in a variety of formats

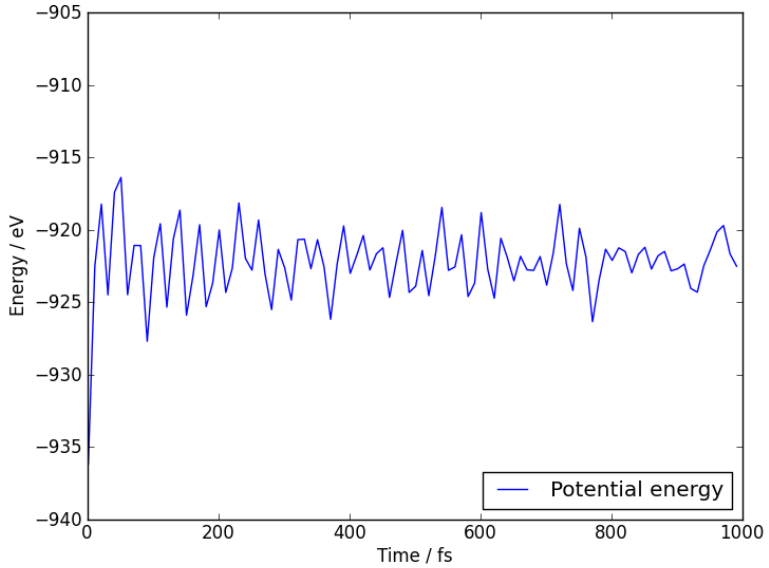
```
first = Atoms('md.xyz')
final = Atoms('md.xyz@-1')
traj = AtomsList('md.xyz')
view(traj)
draw_arrows('force')
```



Post-processing of results

Plotting with the `matplotlib` library

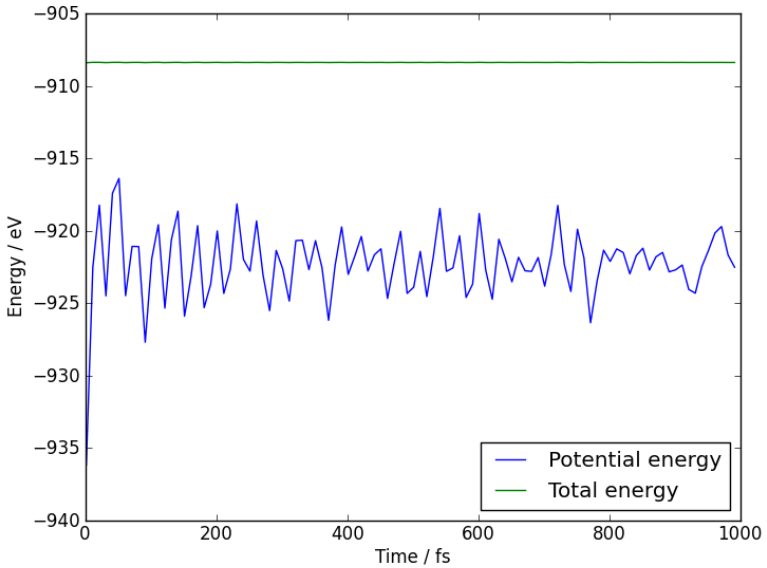
```
from pylab import *

plot(traj.time, traj.energy,
     label='Potential energy')
xlabel('Time / fs')
ylabel('Energy / eV')
legend(loc='lower right')
```



Post-processing of results

Calculate kinetic energy, and add total energy to the plot:

```
ke = array([0.5*sum(at.mass*
                    at.velo.norm2())
                    for at in traj])
plot(traj.time,
     ke + traj.energy,
     label='Total energy')
```



Post-processing of results

Maxwell-Boltzmann distribution

$$f(v)\,\mathrm{d}v = 4\pi \left(\frac{m}{2\pi k_B T}\right)^{3/2} v^2 \exp\left[-\frac{mv^2}{2k_B T}\right] \mathrm{d}v$$

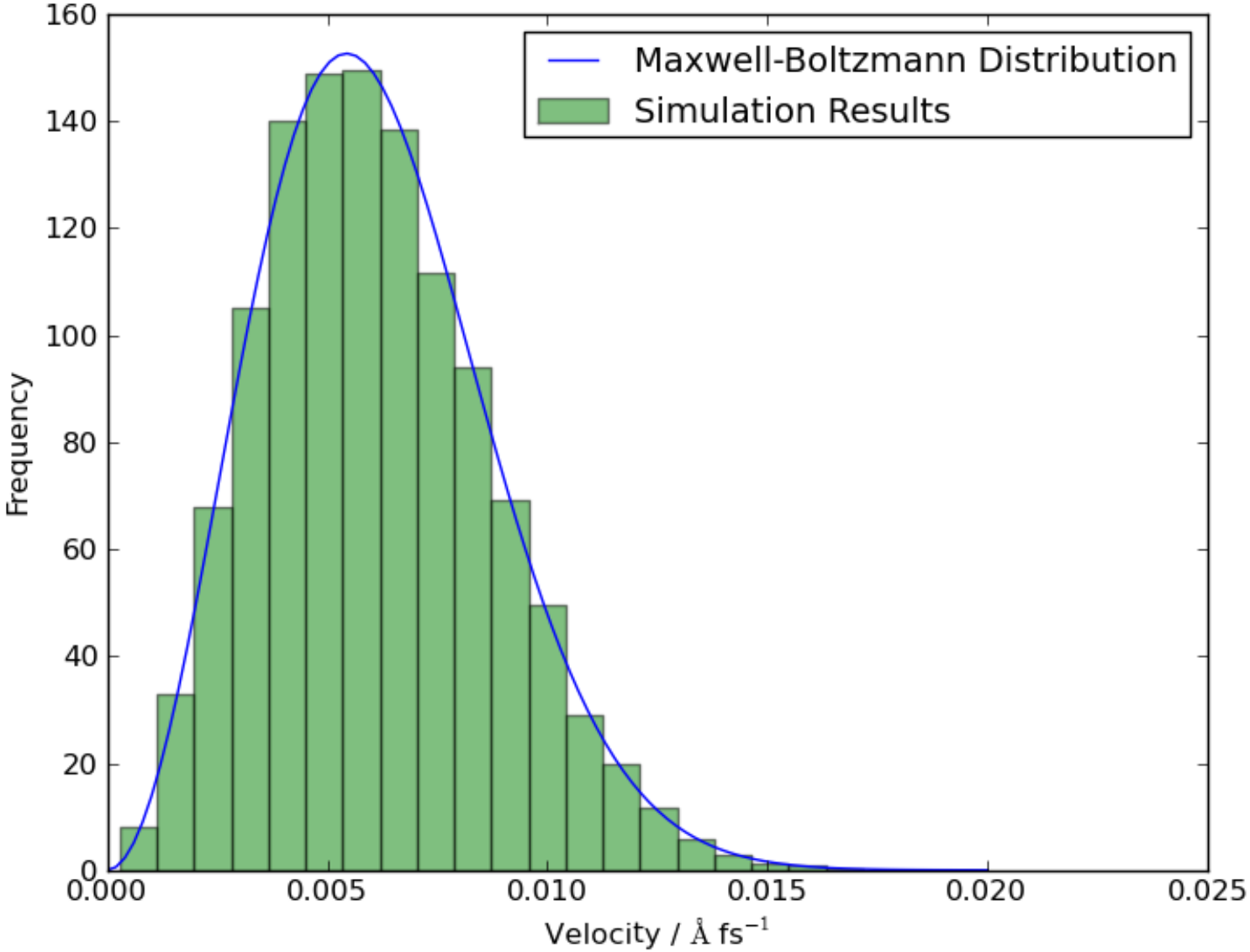
Post-processing of results

Maxwell-Boltzmann distribution

$$f(v)\,\mathrm{d}v = 4\pi \left(\frac{m}{2\pi k_B T}\right)^{3/2} v^2 \exp\left[-\frac{mv^2}{2k_B T}\right] \mathrm{d}v$$

```
speeds = [at.velo.norm() for at in traj[-50:]]
all_speeds = hstack(speeds)
hist(all_speeds, normed=True, bins=20, alpha=0.5)

v = linspace(0.0, 0.02, 100)
plot(v, max_bolt(traj[0].mass[1], 500.0, v))
```

Post-processing of results



Performing calculations

- As well as preparing structures and post-processing results, `quippy` allows calculations to be run
- In `QUIP` and `quippy`, all calculations are performed with a Potential object (very similar to "calculator" concept in ASE)

Performing calculations

- As well as preparing structures and post-processing results, `quippy` allows calculations to be run
- In `QUIP` and `quippy`, all calculations are performed with a Potential object (very similar to "calculator" concept in ASE)
- Types of potential
  - Internal: interatomic potential or tight binding
  - External: file-based communication with external code or callback-based communication with a Python function
  - Plus flexible combinations of other potentials

Performing calculations

- As well as preparing structures and post-processing results, `quippy` allows calculations to be run
- In `QUIP` and `quippy`, all calculations are performed with a Potential object (very similar to "calculator" concept in ASE)
- Types of potential
  - Internal: interatomic potential or tight binding
  - External: file-based communication with external code or callback-based communication with a Python function
  - Plus flexible combinations of other potentials
- Internal potentials use XML parameter strings
- External potentials use template parameter files

Performing calculations

Creating a Potential

Internal potential

```
sw_pot = Potential('IP SW')
```

Internal potential

```
sw_pot = Potential('IP SW')
```

External potential

```
castep = Potential('FilePot',
                   command='./castep-driver.sh')
```

Driver script can be a shell script, an executable program using QUIP or a quippy script. It can even invoke code on a remote machine.

Internal potential

```
sw_pot = Potential('IP SW')
```

External potential

```
castep = Potential('FilePot',
                   command='./castep-driver.sh')
```

Driver script can be a shell script, an executable program using QUIP or a quippy script. It can even invoke code on a remote machine.

Higher level functionality

- Any of these codes or potentials can be used for higher level calculations
- Within QUIP
  - Molecular dynamics and QM/MM (any combination of codes)
  - Geometry optimisation with CG, damped MD and FIRE
  - Transition state searches with NEB and string method

[10] http://spglib.sourceforge.net

[11] http://phonopy.sourceforge.net

Higher level functionality

- Any of these codes or potentials can be used for higher level calculations
- Within QUIP
  - Molecular dynamics and QM/MM (any combination of codes)
  - Geometry optimisation with CG, damped MD and FIRE
  - Transition state searches with NEB and string method
- By interoperating with other packages
  - Global minimisation with basin or minima hopping via `ASE`
  - Symmetry analysis via `spglib` [10]
  - Phonon band structure via `phonopy` [11]
  - ...

[10] http://spglib.sourceforge.net

[11] http://phonopy.sourceforge.net

Outline

Comparing codes

For verification and validation, we would like to compare structural properties predicted by a number of DFT codes

- Let's try this with the H_2 molecule for a few codes
- PBE XC-functional
- Basis set parameters have been converged for each code

Comparing codes

For verification and validation, we would like to compare structural properties predicted by a number of DFT codes

- Let's try this with the H_2 molecule for a few codes
- PBE XC-functional
- Basis set parameters have been converged for each code

CASTEP calculation

```
h2 = h2_molecule(0.7)
castep.calc(h2, energy=True, force=True)
print h2.energy
print h2.force
```

Alternative invocation methods

ASE-compatible calculator interface

```
h2.set_calculator(castep)
e = h2.get_potential_energy()
f = h2.get_forces()
```

Command line interface

Most of these tools can also be used without the `quippy` Python interface, using the QUIP `eval` tool

```
eval init_args="FilePot command=./castep-driver.sh"\
  at_file=h2.xyz F E
```

There is also a command line tool `convert.py` which can convert between file formats, e.g. .xyz to/from `CASTEP` .cell, `VASP INCAR`, etc. (plus more formats via `ASE` and/or `OpenBabel`)

Performing calculations

Changing parameters

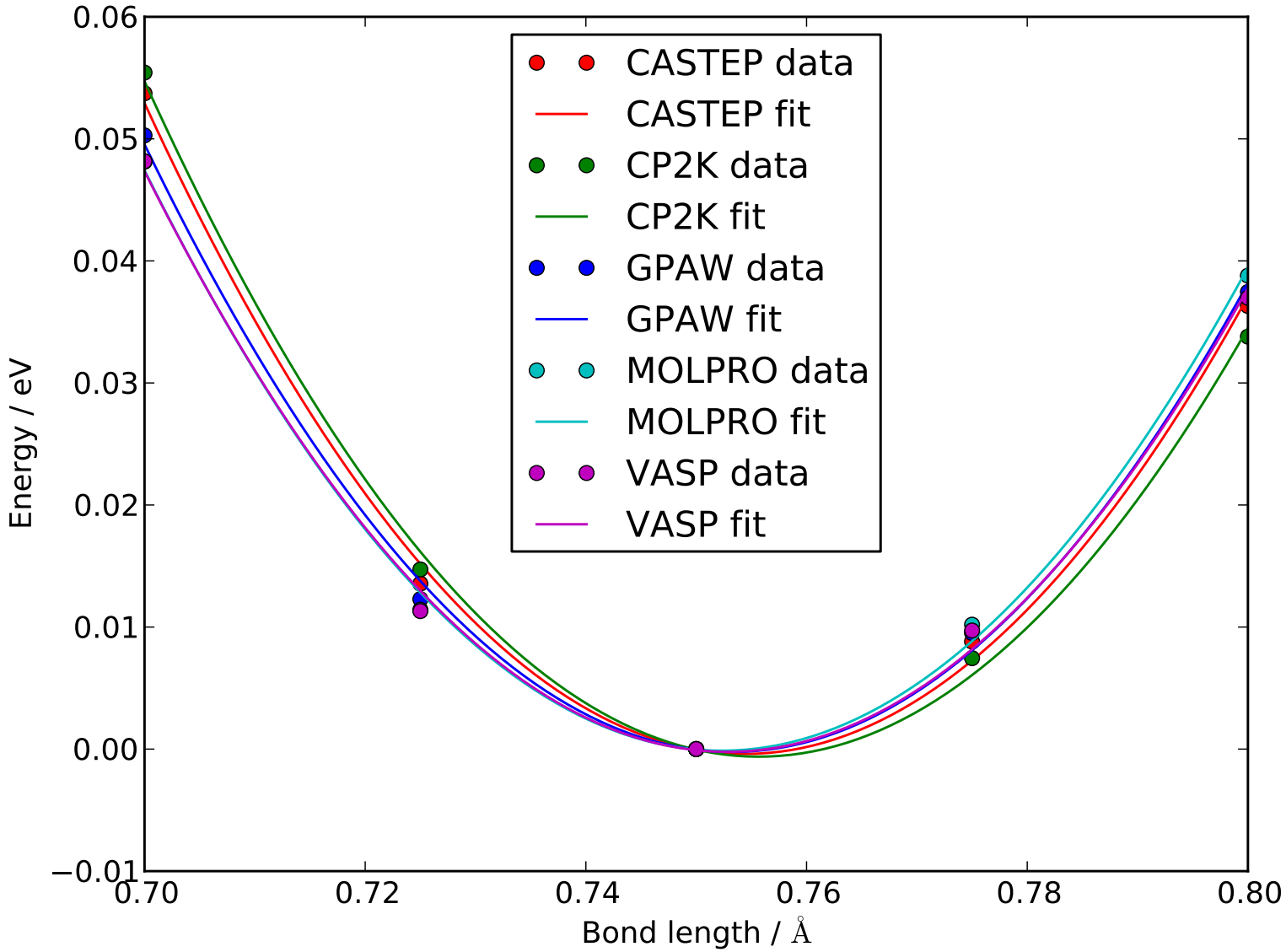
The template input files and other options can be changed by passing extra arguments to the `calc()` routine, e.g. to do a geometry optimisation instead of a single point calculation:

```
castep.calc(h2, energy=True,
            template='h2',
            task='geometryoptimisation')
```

- Parameters can be set interactively while testing, but runs can then of course be automated with scripts
- As well as energies, forces and stress tensors, our output parsers can extract other information such as bond populations

Comparing codes

Plotting the results



Comparing codes

Harmonic fit to data

Code	Bond length / Å	Force constant / eV/Å^2
CASTEP	0.754	36.0
CP2K	0.756	35.7
GPAW	0.753	35.1
MOLPRO	0.752	34.8
VASP	0.753	34.1

Comparing codes

Going beyond GGA

- The framework is rather general, so we can easily connect to codes which go beyond GGA
- e.g. MP2 and CCSD(T) with the `molpro` quantum chemistry code

Code	Bond length / Å	Force constant / eV/Å^2
MOLPRO, MP2	0.739	35.2
MOLPRO, CCSD(T)	0.745	34.6

Robustness

- These tools were initially developed for multiscale QM/MM simulations, where typical production runs require $\sim 10^4$ DFT calculations
- Also used for fitting interatomic potentials to large QM databases (up to $\sim 10^5$ atomic environments)
- Robustness is important!
- `CASTEP`, `VASP` and `CP2K` interfaces now particularly robust
  - Convergence checks
  - Fall back on more reliable density mixers
  - Automatic wavefunction reuse when possible
  - `CP2K` and `VASP` interfaces allow persistent connections (fast!)

Outline

Advantages and disadvantages of `quippy`

- General purpose — arbitrary, extensible data model
- All speed critical code is in Fortran, so it's fast and scales well to large systems ($\sim 10^6$ atoms)
- Interactive visualisation with `AtomEye`[12] plugin (which also scales well to large systems)
- Robust interfaces to several DFT codes
- Fully interoperable with ASE for many more
- Fortran wrapping makes it more complex to use
- Harder to install than a pure Python package

[12] J. Li *Modell. Simul. Mater. Sci. Eng.* (2003)
Modified version: http://jrkermode.co.uk/AtomEye

Summary and Conclusions

- Adding a scripting interfaces to codes gives lots of benefits relevant to validation and verification
- Python and `f2py` do a good job of wrapping Fortran codes
- Wrapping Fortran 90 codes which make heavy use of derived types is also possible with `f90wrap`
- `libAtoms`, `QUIP` and `quippy` provide a uniform interface to a number of electronic structure codes
- Freely available from http://www.libatoms.org (GPLv2)