Python interface to FHI-aims

Jan Hermann
Fritz-Haber-Institut der Max-Planck-Gesellschaft, Berlin, Germany

Background

- FHI-aims is written almost exclusively in modern Fortran, while many of the provided post-processing tools are written in Python

Fortran[1]

- Unique memory management design and extremely optimized compilers enable writing fast numerical implementations without unnecessary boilerplate and almost no overhead
- Limited handling of I/O, necessity of explicit static typing (variable declarations) and absence of higher-level language constructs hinders prototyping and data processing
- Ideal for implementing fast numerical libraries

Python

- The built-in data structures and language expressions together with the extensive standard library and thousands of third-party packages make Python one of the most expressive popular programming languages
- Its complexity and flexibility severely complicate fast implementations of a Python interpreter, making execution of pure Python code several orders of magnitude slower than of an equivalent code in compiled languages
- Perfect as a “glue”

Python Extensions

- The official Python interpreter is implemented in C (CPython), making it easy to extend Python with C code or any other code that can be called from C, such as Fortran
- Without further tools, writing Python extensions requires a deep understanding of how the abstract Python objects are implemented on the C level

C Foreign Function Interface (cFFI)

- Python package (available also for PyPy) that can automatically generate the Python/C glue code given C header files
- Furthermore, tools are provided on the Python side to comfortably work with the raw C objects

cFFI vs. f2py

- cFFI can work with any C-compatible language, f2py is limited to Fortran
- f2py generates the C/Fortran glue code automatically, while it needs to be written by hand for cFFI
- cFFI can handle derived types
- f2py severely limits public interfaces of the Fortran functions

Extending vs. Embedding

- Extending a higher-level language (Python) with a lower-level language (Fortran) is almost always more flexible than embedding it
- Python is designed to be extended, not embedded
- Extending requires initial design, whereas embedding can be done in any project with almost no initial investment

Acknowledgements

Advisors: Alexandre Tkatchenko
Matthias Scheffler

Info

E-mail: jhermann@fhi-berlin.mpg.de

Summary & outlook

- Python scripts as well as the interactive Python command line can be launched from within FHI-aims
- Both Fortran variables and subroutines can be wrapped to their Python counterparts and accessed as well as modified from Python
- A fairly comprehensive interface would be achieved by interfacing all variables in the `physics` module, possibly with automated code generation
- In principle, extending Python rather than embedding it would be a more flexible solution

Compiling FHI-aims with Python interface

- The Python interface is compiled optionally since it requires that the Fortran compiler supports the 2003 standard
- Requirements: Python 2 or 3, cFFI Python package (available via pip)
- Using GNU Make:

```
USE_CFFI = yes
USE_C_FILES = yes
PYTHON = <path to Python>
```

- Using Waf:

```
./waf configure --with-cffi
   --python=<path to Python>
```

Launching Python command line in FHI-aims

```
python_hook post_scf REPL
```

- Single line in `control.in` specifies at which point of the the calculation should the Fortran execution drop to Python command line

```
  ...
  X Energy LDA        :        -55.704281060 Ha
  C Energy LDA        :         -2.847874033 Ha
  ------------------------------------
  End decomposition of the XC Energy
  ------------------------------------
  Executing Python hook post_solver (REPL)...
There is a local variable `ctx`. See
`help(ctx)` for details. Press CTRL+D to con-
tinue the aims run. Type `exit(1)` to abort
aims.
+>>> ctx
<AimsContext ‘KS_eigenvector, occ_numbers, rho,
partition_tab, KS_eigenvalue, rho_gradient,
hirshfeld_volume, batches, elements, kinetic_
density, coords’>
+>>>
```

- The context object `ctx` holds Numpy arrays that are wrappers around the same memory as the corresponding Fortran arrays
- Any modifications of the Numpy arrays are propagated back to FHI-aims
- The embedded command line is identical to launching Python directly, with access to the same packages

Running Python scripts within FHI-aims

```
python_hook post_scf load-hirshfed.py parallel
```

- The specified script has to define function `build` which is called with the context object as a single argument
- Option `parallel` specifies that the script is run on all nodes (the default is to run only on the root node)
- Example: Override Hirshfeld volumes before any van der Waals method

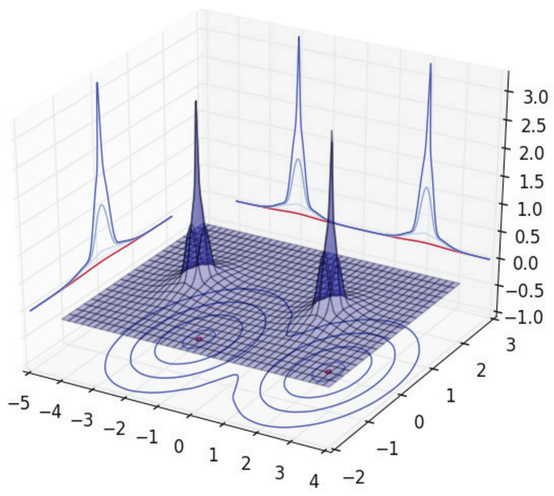
```
def build(ctx):
   ctx.hirshfeld_volume[:] = 0.8
```

Working with quantities defined on the real-space grid

- The context object provides a convenience function to collect grid data from all MPI nodes

```
points, rho, partition_tab = \
   ctx.gather_all_grids(
      [‘rho’, ‘partition_tab’]
   )
```

- Plots can be generated directly by FHI-aims with the use of `matplotlib` ot other plotting packages (shown is electron density of argon dimer)



Calling FHI-aims Fortran functions from the embedded Python

- Fortran subroutines can be wrapped to their Python equivalents and called from the embedded Python command line or scripts

$$\delta n(\mathbf{r}, \omega) = \int \mathrm{d}\mathbf{r}' \chi_0(\mathbf{r}, \mathbf{r}', \omega) \delta v(\mathbf{r}', \omega)$$

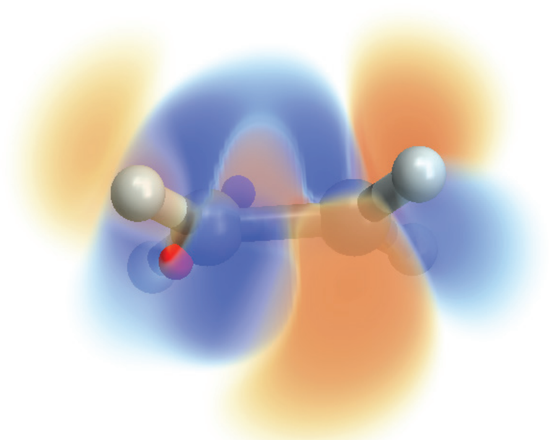
- Example: Evaluation of the density response function is performed by the following Fortran function that implements the Adler–Wiser formula

```
subroutine evaluate_chi_0( &
   c, c_cmplx, omega, f, &
   r_grid, r_prime, u, &
   chi_0 &
)
```

- When wrapping this routine for Python, KS orbitals, energies and occupation numbers are taken from the `physics` module, resulting in the following Python function

```
chi_0 = ctx.evaluate_chi_0(
   r_grid, r_prime, 0j)
```

- Shown is a response to two delta function perturbation located on a carbon atom in an ethylene molecule, plotted with Mathematica after exporting the data from Python to HDF5



Implementation

- Embedding Python via cFFI leads to dynamic linking of the FHI-aims binary to the Python library
- Any call to a Python function from the embedding program initializes the CPython interpreter which then handles any subsequent Python calls

Calling the Python interface

- This is the only change that needs to be done in the FHI-aims codebase
- Leads to an empty stub call when FHI-aims is not compiled with the Python interface

```
if (python_hooks%post_scf%registered) then
   call run_python_hook(python_hooks%post_scf)
end if
```

Python entry point

- C function signature

```
extern int call_python_cffi(
   struct AimsContext_t *, char *,
   char *, int);
```

- Python implementation

```
@ffi.def_extern()
def call_python_cffi(
      c_ctx, filename, event, rank):
   ...
```

- Fortran call

```
retcode = call_python_cffi( &
   ctx, c_filename, &
   ‘run’ // c_null_char, myid &
)
```

Calling from Python back to Fortran

- C function signature

```
void c_evaluate_chi_0(
   double *r_grid, int n_grid,
   double *r_prime, double *u,
   double *chi_0);
```

- Fortran implementation

```
subroutine c_evaluate_chi_0( &
   r_grid, n_grid, r_prime, u, chi_0 &
) bind(c)

integer(c_int), value :: n_grid
real(c_double) :: r_grid(n_grid, 3)
real(c_double) :: r_prime(3)
complex(c_double_complex) :: u
real(c_double) :: chi_0(n_grid)
...
```

- Python call

```
aims.c_evaluate_chi_0(
   ffi.cast(‘double *’, r_grid.ctypes.data),
   r_grid.shape[0],
   ffi.cast(‘double *’, r_prime.ctypes.data),
   ffi.cast(‘double *’, u.ctypes.data),
   ffi.cast(‘double *’, chi_0.ctypes.data),
)
```

Converting Fortran arrays to Numpy

- Fortran to C

```
c_ctx%coords = c_loc(coords)
```

- C to Python

```
ctx.coords = np.ndarray(
   shape=(3, n_atoms),
   buffer=ffi.buffer(
      c_ctx.coords, 3*n_atoms*8
   ),
   order=’F’,
   dtype=float
)
```