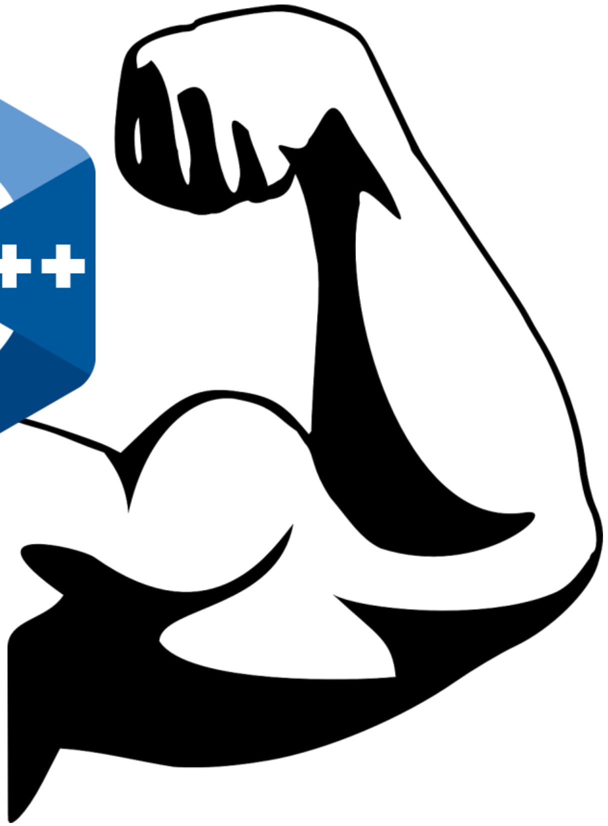
Python as glue

Codes available on GitHub: https://github.com/Paul-St-Young/thw-python-as-glue

Yubo “Paul” Yang, THW-IL, 2018/04/17

Images from https://pixabay.com/vectors



Step 1: Basic Interface

Python can import *shared library*

Choices:

f2py (numpy)

swig (KMClib)

ctype (pyscf)

cython ([yt](yt))

numba

boost.python

pybind11 (Eigen)

…

Criteria:

easy to write and debug

compatible with np.array

separable into native languages

f2py

```
NAME=example

all:
	f2py -c -m $(NAME) $(NAME).f90

clean:
	rm $(NAME).so
```

```fortran
integer function add(i, j)
  integer, intent(in) :: i, j
  add=i+j
end function
```

pybind11

```
NAME=example
CXX=g++
OFLAGS=-O3
CFLAGS=$(OFLAGS) -shared -fPIC
OBJS=example.o

%.o: %.cpp
	$(CXX) $(CFLAGS) -c $<

all: $(OBJS)
	$(CXX) $(CFLAGS) $(OBJS) -o $(NAME).so

clean:
	rm *.o $(NAME).so
```

```cpp
#include <pybind11/pybind11.h>

int add(int i, int j)
{
  return i+j;
}

PYBIND11_MODULE(example, m)
{
  m.def("add", &add, "add two integers");
}

```

https://carbon.now.sh
https://team411.github.io/src2img

Step 2: Same Instruction Multiple Data (SIMD)

Python driver

```python
import numpy as np
from forlib import example as fex
from cpplib import example as cex

def add(i, j):
  return i+j

def check_correct(vec1, vec2):
  pyvec = add(vec1, vec2)
  cvec = cex.add(vec1, vec2)
  fvec = cex.add(vec1, vec2)
  print('cpp=python', np.allclose(cvec, pyvec))
  print('fortran=python', np.allclose(fvec, pyvec))
```

('cpp=python', True)
('fortran=python', True)
('python/fortran', 1.689771011187475)
('python/cpp', 0.26767694027701155)
('python/python', 1.0473552503390706)

Timing: C++ is slowest

Fortran backend

```fortran
subroutine add(vec1, vec2, vecout, n)
  integer, intent(in) :: n
  real*8, intent(in) :: vec1(n), vec2(n)
  real*8, intent(out) :: vecout(n)
  vecout = vec1+vec2
end subroutine
```

C++ backend

```cpp
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>

namespace py=pybind11;

double add(double i, double j)
{
  return i+j;
}

PYBIND11_MODULE(example, m)
{
  m.def("add", py::vectorize(add), "add two integers");
}
```

Step 3: Linear Algebra

Timing: numpy is fastest!

('python/fortran', 0.05105879678631556)
('python/cpp', 0.049230687039213586)
(numpy/numpy', 0.9886795048143053)

```python
def matmul(mat, vec):
  return np.dot(mat, vec)
```

Fortran backend

```fortran
subroutine mmul(mat, vec, vecout, m, n)
  integer, intent(in) :: m, n
  real*8, intent(in) :: mat(m, n), vec(n)
  real*8, intent(out) :: vecout(n)
  vecout = matmul(mat, vec)
end subroutine
```

C++ backend

```cpp
#include <pybind11/pybind11.h>
#include <pybind11/eigen.h>
#include <Eigen/Dense>

namespace py=pybind11;
typedef Eigen::MatrixXd Matrix;
typedef Eigen::VectorXd Vector;

Vector mmul(
  Eigen::Ref<const Matrix>& mat,
  Eigen::Ref<const Vector>& vec)
{
  return mat*vec;
}

PYBIND11_MODULE(example, m)
{
  m.def("mmul", &mmul, "multiply matrix vector");
}
```

Step 4: General Computing (e.g. distance table)

$$r_{ij} = |\boldsymbol{r}_i - \boldsymbol{r}_j|$$

Timing: C++ and Fortran are both O(1000) times faster than Python for loops

('python/cpp', 734.6365928831605)
('python/fortran', 747.8374215630347)
('python/python', 1.0)

```python
def disp_in_box(pos, lbox):
  nint = np.round(pos/lbox)
  return pos-nint*lbox

def get_dtable(pos, lbox):
  natom = len(pos)
  dtable = np.zeros([natom, natom])
  for i in range(natom):
    for j in range(natom):
      dtable[i, j] = np.linalg.norm(
        disp_in_box(pos[i]-pos[j], lbox)
      )
  return dtable
```

```fortran
subroutine distance_table(pos, lbox, natom, ndim, dtable)
  integer, intent(in) :: natom, ndim
  real*8, intent(in) :: pos(natom, ndim), lbox
  real*8, intent(out) :: dtable(natom, natom)
  integer i, j
  real*8 drij(ndim)
  do i=1,natom
    do j=i+1,natom
      drij = pos(i, :) - pos(j, :)
      drij = drij - lbox*nint(drij/lbox)
      dtable(i, j) = norm2(drij)
    end do
  end do
end subroutine
```

```cpp
Eigen::Ref<Matrix> distance_table(
  Eigen::Ref<const Matrix>& pos, double lbox)
{
  const int natom = pos.rows();
  const int ndim = pos.cols();
  Matrix dtable = Matrix::Zero(natom, natom);
  Vector drij(ndim);
  for (int i=0;i<natom;i++)
  {
    for (int j=i; j<natom; j++)
    {
      drij = pos.row(i) - pos.row(j);
      for (int idim=0;idim<ndim;idim++)
      {
        drij(idim) -= lbox*std::round(drij(idim)/lbox);
      }
      dtable(i, j) = drij.norm();
    }
  }
  return dtable;
}
```

Step 4: General Computing (e.g. distance table)

$$r_{ij} = |\boldsymbol{r}_i - \boldsymbol{r}_j|$$

Timing: vectorization saves some face
 but imposes other limitations

('pyvec/cpp', 6.621653701352949)
('pyvec/fortran', 6.754378146746862)
('pyvec/pyvec', 1.0)

```python
def disp_in_box(pos, lbox):
  nint = np.round(pos/lbox)
  return pos-nint*lbox

def get_dtable_vec(pos, lbox):
  dptable = disp_in_box(
    pos[:, np.newaxis] - pos[np.newaxis, :]
  , lbox)
  dtable = np.linalg.norm(dptable, axis=-1)
  return dtable
```

```fortran
subroutine distance_table(pos, lbox, natom, ndim, dtable)
  integer, intent(in) :: natom, ndim
  real*8, intent(in) :: pos(natom, ndim), lbox
  real*8, intent(out) :: dtable(natom, natom)
  integer i, j
  real*8 drij(ndim)
  do i=1,natom
    do j=i+1,natom
      drij = pos(i, :) - pos(j, :)
      drij = drij - lbox*nint(drij/lbox)
      dtable(i, j) = norm2(drij)
    end do
  end do
end subroutine
```

```cpp
Eigen::Ref<Matrix> distance_table(
  Eigen::Ref<const Matrix>& pos, double lbox)
{
  const int natom = pos.rows();
  const int ndim = pos.cols();
  Matrix dtable = Matrix::Zero(natom, natom);
  Vector drij(ndim);
  for (int i=0;i<natom;i++)
  {
    for (int j=i; j<natom; j++)
    {
      drij = pos.row(i) - pos.row(j);
      for (int idim=0;idim<ndim;idim++)
      {
        drij(idim) -= lbox*std::round(drij(idim)/lbox);
      }
      dtable(i, j) = drij.norm();
    }
  }
  return dtable;
}
```

Step 5: Classes

Binding to C++ class is straight-forward. Interface *borrowed* from boost.python.

Clean support for operator overloading (including pickling!)

constructor

getters/setters

methods

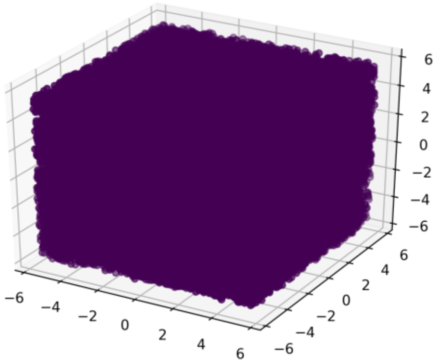
operators

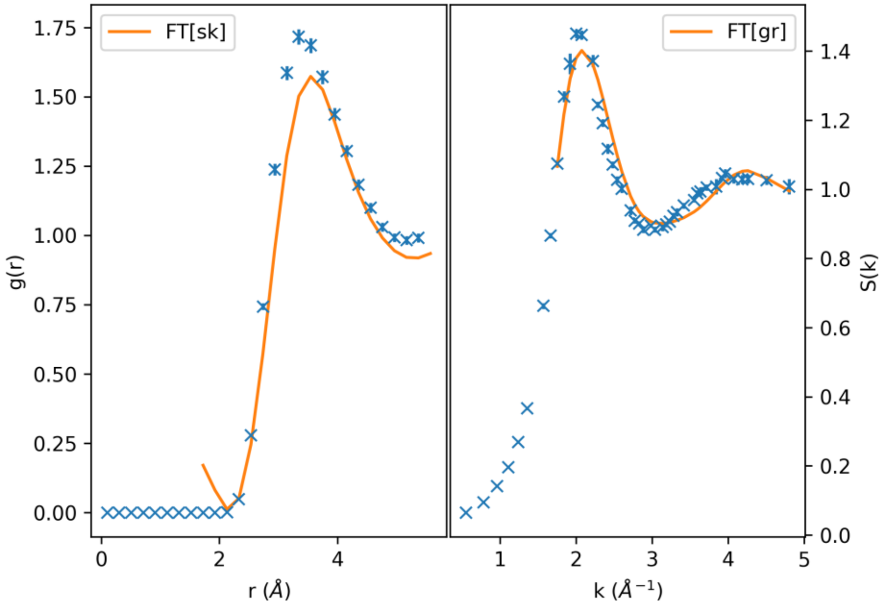
```cpp
PYBIND11_MODULE(example, m)
{
  py::class_<MatrixS>(m, "MatrixS")
    .def(py::init<int, int>())
    .def_readwrite("m", &MatrixS::m)
    .def_readonly("n", &MatrixS::n)
    .def_readonly("mat", &MatrixS::mat)
    .def("randomize", &MatrixS::randomize)
    .def("getitem", &MatrixS::getitem)
    //.def("__getitem__", &MatrixS::getitem)//, py::is_operator())
    .def("__repr__", &MatrixS::to_string)
    .def(py::self + py::self)
    .def(double() * py::self)
    .def(py::self * double())
  ;
}
```

Practical Applications: quick prototype, pytest, input automation, visualization …

McMillan He simulation GitHub repository “McMillanHe”

1. C++ core class McMillanHe performs variational Monte Carlo random walk
2. Fortran analysis module grsk.f90 performs analysis on random walk samples for g(r) and S(k)
3. Python workflow script mmh.py coordinate simulation, analysis, and visualization.






```
[yyang173@localhost McMillanHe]$ pytest -v .
========================= test session starts ==========================
platform linux2 -- Python 2.7.15, pytest-3.7.4, py-1.6.0, pluggy-0.7.1 -
- /usr/bin/python2
cachedir: .pytest_cache
rootdir: /home/yyang173/Desktop/pyglue/mcmillan/nonboost/McMillanHe, ini
file:
collected 6 items

tests/test_mmh.py::test_init PASSED                                [ 16%]
tests/test_mmh.py::test_get_lbox PASSED                            [ 33%]
tests/test_mmh.py::test_set_lbox PASSED                            [ 50%]
tests/test_mmh.py::test_get_a1 PASSED                              [ 66%]
tests/test_mmh.py::test_set_a1 PASSED                              [ 83%]
tests/test_mmh.py::test_wfval PASSED                               [100%]

======================= 6 passed in 0.11 seconds =======================
```

Now we can use Sphinx instead of doxygen!

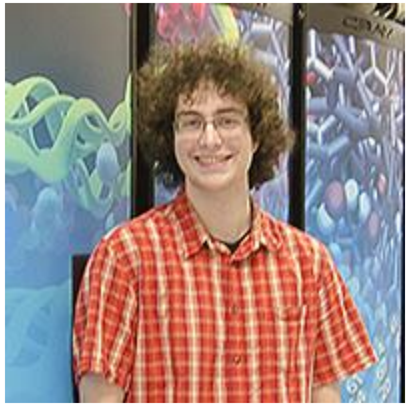
References and Acknowledgement

f2py
[1] f2py user guide https://docs.scipy.org/doc/numpy/f2py
[2] scipy endorsement
[3] illustrative example
[4] our very own Katy

pybind
[1] pybind11 readthedocs
[2] boost.python projects
[3] pybind vs. swig

tutorial repositories
[1] basic steps
[2] McMillan He simulation

Many thanks to Ryan Levy for guiding me away from my boost.python debacle in favor of pybind11, which plays nicely with Eigen.



Conclusions:

- Thanks to packages like **f2py** and **pybind**, it is exceedingly easy to call C and Fortran from Python.
- However, SIMD and linear algebra can (should?) be left to numpy.
- Nested for loops can speed up O(10-100) times in C or Fortran.

TL;DR Flex your F/C muscles when you have to, but leave most complexity to your *Smart Pythonic "Brain".*