Friedrich-Alexander-Universität
Naturwissenschaftliche Fakultät

Using embedded Python within openCFS

Fabian Wein

Friedrich-Alexander-Universität Erlangen-Nürnberg

15. December 2022

Combining Python and C/C++

FAU

Extending

- automatically generate wrappers around C++ classes
- use these wrapped classes from standard Python
- example: VTK, Cubit
- example wrapper: Boost.Python

Embedding

- launch Python interpreter from within C/C++ application
  $\rightarrow$ execute Python code from within C/C++ application
- expose C functions to Python
- this is the presented approach for openCFS

Python Interpreters

Python implementations

- CPython: reference implementation in C
- IronPython: implementation in C#, compatible to CPython
- PyPy: implementation in Python

CPython

- practically all `python` we use is CPython
- a Python installation has several optional packages
- e.g. NumPy is implemented in Python/C/Fortran
- CPython is not designed for multiprocessing

Python C-API

- low level API with manual shared reference counting
- virtually no overhead
- same API as for CPython internal usage (e.g. to integrate NumPy)
- many C-Macros
- outstanding good documentation
- Python/C API Reference Manual $\rightarrow$ `docs.python.org/3/c-api/index.html`

Python installations

- often several Python version installed
- each has own set of installed libs (e.g. via `pip`)
- it is sufficient to call specific `bin/python`
- correct installed libraries are automatically found
- no environment variables required

Embedding Python

- enable `USE_EMBEDDED_PYTHON`
- CMake applies black magic for finding instance
- optionally enforce by setting manually
  - `-DPYTHON_INCLUDE_DIR=/usr/include/python3.8`
  - `-DPYTHON_LIBRARY=/usr/lib/x86_64-linux-gnu/libpython3.8.so`
- `PYTHON_EXECUTABLE (Found PythonInterp:  ..)` is something different

Portability

Local

- works locally nicely for Linux, macOS and Windows
- `cfs` is dynamically linked to python instance
- confirm with `ldd bin/cfs` $\rightarrow$ libpython3.8.so.1.0

On other system

- Linux/macOS: probably help with setting `LD_LIBRARY_PATH`
- practically setting a symbolic link in lib(64) to any version works
- for Windows the exact same Python version needs to installed

Bundling with openCFS?

- portability could be solved by bundling
- see ParaView or Cubit

Disadvantages

- quite fat dependency
- necessary Python packages shall be included (numpy, scipy, . . .)
- some Python packages have system dependencies (e.g. h5py)
- some packages are exotic (e.g. autograd)

Compromise

- Windows: agree on specific Python installation (`repo.anaconda.com/archive` ?)
- Linux and macOS: not really a problem

Performance

- initializing CPython interpreter is fast (< 1 sec)
- calling Python functions is very fast
- calling openCFS functions from Python has no overhead
- performance bottleneck is Python performance
- no penalty from bridging

Program counter

- aka instruction pointer
- a logical processor has a logical program flow
  - sequentially processing machine code
  - machine code can emulate higher language program counter
  - mentally skip interrupts and multiprocessing
- openCFS $\rightarrow$ CPython $\rightarrow$ Python code (return) $\rightarrow$ openCFS ...
- additionally: ... Python code $\rightarrow$ call openCFS function $\rightarrow$ Python code (return) ...
- embedded Python code cannot run in debugger

Python code

- CPython interpreter can evaluate strings or files
- console Python interpreter and openCFS interpreter is the same (binary libpython3.8.so)
- Python code can import arbitrary packages at run time
- openCFS interpreter provides module `cfs` which can be imported by `import cfs`
- usually openCFS expects certain Python functions

Example Python code

```python
def set_cfs_mesh(opt): # openCFS expects this function
  print('\nset_cfs_mesh() from mesh_tool.py with', str(opt))
  ..
  type = opt['type'] # we got a dictionary from openCFS
  ..
  mesh = create_2d_mesh(x_res, y_res, width=width, height=height)
  call_cfs_input_reader(mesh)

def call_cfs_input_reader(mesh):
  import cfs # import here to run file also from console
  cfs.set_nodes(mesh.nodes); # calls PythonKernel::mesher_set_nodes()
```

Dual use Python code

```
if __name__ == '__main__': # executed from command line
  parser = argparse.ArgumentParser()
  ..
  class FakeCFS:
    def evalobj(self)
  ..
  cfs = FakeCFS()
else:
  import cfs  # not from command line but imported (from openCFS)
cfs.evalobj()
```

openCFS PythonKernel

- service class `Utils/PythonKernel`
- always exists as object `python` like `domain` or `param`
- gate keeper of the unique CPython instance
- service functions for modules (Python files)
- optionally handles root Python file (parses from xml)
- static service functions for dictionary, array and matrix conversion

```
PyObject* CreatePythonDict(const StdVector<pair<string, string> > obj);
PyObject* CreatePythonList(const StdVector<TYPE>& list) ;
void ConvertPythonList(Container<TYPE>& vec, PyObject* list);
string ToString(PyObject* obj);
void CheckPythonFunction(PyObject* pyobject, const char* name);
..
```

Data handling

- each Python object is a shared pointer object
  $\rightarrow$ manually decrement/increment for each list item
- numpy arrays are plain data array
- somehow initializing numpy arrays by openCFS fails?!

```
Vector::Vector(PyObject* obj, bool decref);
void Vector::Fill(PyObject* obj, bool decref = true);
void Vector::Export(PyObject* obj);
Matrix<T> Numpy2DArrayToMatrix(PyObject* numpy, Matrix<T>& out, bool dr);
```

openCFS Testsuite

- `NodalPythonCoef`: generate coef values by Python
- `PythonCallback`: example for registering Python hook
- `PythonMesher`: generate regular mesh by python with xml parametrization
- `PythonMesherVar`: custom mesh generation with xml parametrization
- `python/ocm`: simple optimizer calling cfs functions
- `python/mma-multiload`: complex optimizer using `numpy.linalg.solve`
- `python/ocm-matlab`: slow bridge to Matlab via `import matlab.engine`
- `spaghetti_*`: highly complex optimization parametrization, once even with `autograd`
- `[local_]python_function`: example function and gradient for optimization

NodalPythonCoef Example

- test case is for proof of concept
- use Python functions as alternative to muParser
- Python functions do not call openCFS functions
- Python file contained as global PythonKernel kernel

NodalPythonCoef xml

```
<python file="coef.py" path="."/>
..
<bcsAndLoads>
  <force name="inclusion_nodes">
    <!-- is of type DT_InputPython and parsed fully by PythonKernel -->
    <python function="coef_eval" init="coef_init">
      <option key="center" value="[.5, .5]"/>
    </python>
 </force>
</bcsAndLoads>
```

NodalPythonCoef coef.py

```python
print('hallo from coef.py'
# 'import cfs' would work but is not needed
def coef_init(opt): # optional function for openCFS
  cs = opt['center']
  global c # create variable with global scope
  c = eval(cs) # "[.5, .5]" -> list

def coef_eval(coord): # function expected from openCFS
  return [coord[0] - c[0], coord[1] - c[1]]
```

CoefFunctionPython I

```
CoefFunctionPython::CoefFunctionPython(PtrParamNode pn, unsigned int dim)
  : CoefFunction()
{
  init_     = pn->Get("init")->As<string>();
  function_ = pn->Get("function")->As<string>();

  auto opt = PythonKernel::ParseOptions(pn->GetList("option"));

  if(init_ != "") {  /* next slide */  }

  eval_ = PyObject_GetAttrString(python->GetKernel(), function_.c_str());
  PythonKernel::CheckPythonFunction(eval_, function_.c_str()); // throws
}
```

Call Python with dict

```
PyObject* func = PyObject_GetAttrString(python->GetKernel(), init_.c_str())
PythonKernel::CheckPythonFunction(func, init_.c_str()); // throws

PyObject* arg = PyTuple_New(1);
// PyTuple_SetItem() steals the reference from dict
PyTuple_SetItem(arg, 0, PythonKernel::CreatePythonDict(opt));

PyObject* call = PyObject_CallObject(func, arg);
PythonKernel::CheckPythonReturn(call); // throws

Py_XDECREF(call);
Py_XDECREF(arg);
Py_XDECREF(func);
```

Service Function

```
PyObject* CoefFunctionPython::CallFunction(const LocPointMapped& lpm) {
  Vector<double> pointCoord;
  lpm.GetGlobal(pointCoord); // extract coord from lpm (not standard!)

  PyObject* arg = PyTuple_New(1);
  PyTuple_SetItem(arg, 0, PythonKernel::CreatePythonList(pointCoord));

  PyObject* ret = PyObject_CallObject(eval_, arg);
  PythonKernel::CheckPythonReturn(ret);

  Py_XDECREF(arg);
  return ret; // dont't forget to decref ret
}
```

Calling Python

```
void CoefFunctionPython::GetVector(Vector<double>& vec,
                                   const LocPointMapped& lpm) {
  PyObject* ret = CallFunction(lpm);
  PythonKernel::ConvertPythonList<double>(vec, ret);
  Py_XDECREF(ret);
}

void CoefFunctionPython::GetScalar(double& val,
                                   const LocPointMapped& lpm) {
  PyObject* ret = CallFunction(lpm);
  val = PyFloat_AsDouble(ret);
  Py_XDECREF(ret);
}
```

PythonMesher Example

- we use a Python script which calls openCFS functions
- dual use script `share/python/mesh_tools.py`
- separate Python file for mesher (not global kernel file)

PythonMesher xml

```
<fileFormats>
  <input>
    <!-- mesh_tool.py has a reference implementation of set_cfs_mesh() --
    <python fileName="mesh_tool.py" path="cfs:share:python">
      <option key="type"  value="bulk2d"/>
      <option key="x_res" value="2"/>
      <!-- set_cfs_mesh() can interpret many more options -->
   </python>
 </input>
</fileFormats>
```

PythonMesher mesh_tool.py

```
def set_cfs_mesh(opt): # the only expected function from SimInputPython
  if not ('type' in opt and 'x_res' in opt):
    # exceptions are propagated to openCFS
    raise "set at least the options type and x_res"

  type = opt['type']
  x_res  = int(opt['x_res'])
  y_res  = int(opt['y_res'])  if 'y_res' in opt else None
  ..
  mesh = create_2d_mesh(..)
  call_cfs_input_reader(mesh)
```

PythonMesher mesh_tool.py

```
def call_cfs_input_reader(mesh):
  import cfs
  cfs.set_nodes(mesh.nodes); # mesh.nodes is numpy.ndarray
  regions = list(set([e.region for e in mesh.elements]))
  cfs.set_regions(regions)
  ..
  for at in list(set([e.type for e in mesh.elements])):
    ..
    cfs.add_elements(total, cfs_type, npd)
  for name, nodes in mesh.bc:
    ..
    cfs.add_named_nodes(name, npn)
  ..
```

SimInputMesh.hh

```
class SimInputPython: virtual public SimInput {
    friend class PythonKernel;
    void InitModule() override;
    virtual void ReadMesh(Grid* grid) override; // calls set_cfs_mesh()
    void SetNodes(PyObject* args); // called from Python
    void SetRegions(PyObject* args);
    void AddElements(PyObject* args);
    ..
    PyObject* module = NULL;
};
```

SimInputMesh::SetNodes()

```
void SimInputPython::SetNodes(PyObject* args) { // read numpy array
  PyArrayObject *array = NULL;
  PyArg_ParseTuple(args, "O!", &PyArray_Type, &array);
  PythonKernel::CheckPythonReturn((PyObject*) array);
  unsigned int rows = PyArray_DIM(array,0);
  unsigned int cols = PyArray_DIM(array,1);
  grid->AddNodes(rows); // reserve space with GridCFS::AddNodes()
  Vector<double> loc(3); loc.Init(0.0); // third dimension keeps 0
  for(unsigned int i = 0; i < rows; i++)  {
    loc[0] = *((double*) PyArray_GETPTR2(array,i,0));
    loc[1] = *((double*) PyArray_GETPTR2(array,i,1));
    grid->SetNodeCoordinate(i+1, loc); }
  Py_XDECREF(args); }
```

PythonKernelFunctions.cc

```
PyObject* PythonKernel::mesher_set_nodes(PyObject *self, PyObject *args)
{
  if(!CheckMesher()) return NULL;
  mesher_->SetNodes(args);
  Py_RETURN_NONE;
}
PyMethodDef PythonKernel::cfs_methods[] =
{
  {"set_nodes", PythonKernel::mesher_set_nodes, METH_VARARGS,
   "Set node coordinates via numpy array with 2 or 3 columns"},
  ..
}
```

Purpose/Potential

- add callable cfs function to PythonKernelFunctions.cc
- this flat collection of functions shall grow and grow
  - read/set solution, rhs, system matrix, ...
- potential in rapid development and testing
  - result functions
  - coef functions
  - reader and writer
  - system solver (w/o file interface)

Getting started

- study/debug corresponding Testsuite example
- study/enhance `unittests/embeddedpython.cc`
- read Python/C API Reference Manual
  → `docs.python.org/3/c-api/index.html`
- browse code for `PyObject*` for inspiration

Final words

- before copy & pasting Python C-API code extend PythonKernel
- development in Python $\approx 10\times$ faster than in C++
- reduce “dead” C++ code in openCFS from failed attempts by using Python
  $\rightarrow$ instead of unused special openCFS code leave reusable extension
- there is nothing like a free lunch:

$$\left.\frac{\partial \text{ learning curve}}{\partial t}\right|_{t=0} \gg 0$$