Language binding

N.B.

- All this requires C++11 (or more recent)
- extensive Pybind11 documentation is available

Doing computations efficiently: user-friendliness

Computational efficiency of C++ exceeds that of Python by a large margin, but

- this comes with the burden of first needing to write the (more complex) C++ code to begin with (unavoidable)
- the development cycle (write/modify code, compile, test, …) can be tedious and (in case of large projects) overly long
- if providing code for others: reality is that many users prefer a high-level language like Python

It makes very good sense to try and combine the desirable features from both languages

- notably, provide high-level steering & configuration of C++ tools from Python
  - we will restrict ourselves to this functionality

Generic task

Even though the Python interpreter itself has been written in C, many differences between C++ and Python exist at the user level

- actually, there is an alternative to the standard CPython interpreter: Pypy is written in yet a different language (RPython)

Consequence: a “glue” layer is needed to provide inter-operability

In the following, we will use the Pybind11 code suite to provide this layer

- not the only alternative: there are other alternatives like SWIG, cppyy that offer substantially more automation compared to pybind11. However, they require a software setup that is not readily available on our Linux systems, so it seems less prudent to focus on them now
  - corollary: we will keep this topic brief, as the details are too implementation specific

Basics

Installation (tested on university Linux systems):

```
pip install pybind11
```

- functionality is then provided through a header

General usage:

- for each function / class, provide a C++ code snippet indicating the functionality that should be made available to the Python “user” world (and how)
- this snippet should be compiled into a shared object library, which can subsequently be used from the Python side

```
c++ -O3 -Wall -shared -std=c++11 -undefined dynamic_lookup `python3 -m pybind11 \
 --includes` *.cc -o example`python3-config --extension-suffix`
```

  - in the subsequent slides, we will discuss some very basic usage
  - assume all code (including C++ code) will be compiled in one go, into the same shared library: not very elegant, but improving on this would likely require delving into the CMake build system as well

Basics

Installation (tested on university Linux systems):

```
pip install pybind11
```

- functionality is then provided through a header

General usage:

- for each function / class, provide a C++ code snippet indicating the functionality that should be made available to the Python "user" world (and how)
- this snippet ... red object library, which can subsequ... be used from the Python side

this retrieves the include paths for both the pybind11 and Python headers

```
c++ -O3 -Wall -shared -std=c++11 -undefined dynamic_lookup `python3 -m pybind11 \
 --includes` *.cc -o example`python3-config --extension-suffix`
```

- in the subsequent slides, we will discuss some very basic usage
- assume all code (including C++ code) will be compiled in one go, into the same shared library: not very elegant, but improving on this would likely require delving into the CMake build system as well

Basics

Installation (tested on university Linux systems):

```
pip install pybind11
```

- functionality is then provided through a header

General usage:

- for each function / class, provide a C++ code snippet indicating the functionality that should be made available to the Python "user" world (and how)
- this snippet should be compiled into a share
  subsequently be used from the Python sid

this assumes you are using Python3; omit if this is not the case

```
c++ -O3 -Wall -shared -std=c++11 -undefined dynamic_lookup `python3 -m pybind11 \
 --includes` *.cc -o example`python3-config --extension-suffix`
```

- in the subsequent slides, we will discuss some very basic usage
- assume all code (including C++ code) will be compiled in one go, into the same shared library: not very elegant, but improving on this would likely require delving into the CMake build system as well

Access to a C++ function

Example lifted from manual:

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

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

PYBIND11_MODULE(example, m) {
    m.doc() = "pybind11 example plugin"; // optional docstring
    m.def("add", &add, "A function that adds two numbers");
}
```

module name (must match shared library filename, apart from extension)

variable of type pybind11::module

pass address of add()

- N.B. not a regular function but a preprocessor statement

- in Python:

```
>>> import example
>>> dir(example)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'add']
>>> example.add(3,5)
8
```

Access to a C++ class

Example lifted from manual:

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

class Pet {
 public:
  Pet (const std::string& name) : name(name) { }
  void setName(const std::string& name_);
  const std::string& getName() const;
 private:
  std::string name;
};

namespace py = pybind11·

PYBIND11_MODULE(Pet, m) {
    py::class_<Pet>(m, “Pet”, py::dynamic_attr())
        .def(py::init<const std::string&>())
        .def("setName", &Pet::setName)
        .def("getName", &Pet::getName)
    .def("__repr__", [](const Pet& a) { return "<Pet.Pet named '" + a.getName() + "'>";});
}
```

indicate that this is a class or struct

optional; indicate that this class may be *extended* in Python

expose the C++ constructor also as a constructor (of the specified type) in Python

add code allowing for a meaningful print() statement, using a lambda function

- alternatively:

```
.def_property("name", &Pet::getName, &Pet::setName)
```

  - useful for “simple” properties (getter & setter methods)

Dealing with inheritance

Example lifted from manual:

```
…

namespace py = pybind11;

class Parrot: public Pet {
public:
    Parrot(const std::string& name): Pet(name) {}
    const string talk () {return getName() + "wants a cookie!"};
};

PYBIND11_MODULE(Pet, m) {
    # py::class_ declaration   for class Pet (see preceding page)
    …
    py::class_<Parrot, Pet>(m, "Parrot")
      .def(py::init<const std::string&>())
      .def("talk", &Parrot::talk);
}
```

declare Parrot as deriving from Pet

- in Python:

```
>>> import Pet
>>> dir(Pet.Parrot)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', 'getName', 'setName', 'talk']
```

Pointers and other uncovered items

Pointers do not exist within Python, so they must be dealt with separately: tedious!

- recommendation: work with smart pointers (shared_ptr, unique_ptr) only
  - this may require writing a wrapper class converting those to the raw pointers used in the original class

The preceding slides merely list the (in my opinion) most important features of Pybind11 — sufficient for basic usage

- as stated previously: no attempt to be complete here

Other features (consult the manual):

- function overloading, enums, keyword arguments