Notes about moving from Python to C++

Yung-Yu Chen (@yungyuc)
PyConTW 2020, Tainan, Taiwan

Why move?

- Numerical software: Computer programs to solve scientific or mathematic problems.
  - Other names: Mathematical software, scientific software, technical software.
- Python is a popular language for application experts to describe the problems and solutions, because it is easy to use.
- Most of the computing systems (the numerical software) are designed in a *hybrid architecture*.
  - The computing kernel uses C++.
  - Python is chosen for the user-level API.

What I do

- By training, I am a computational scientist, focusing on applications of continuum mechanics. (I am *not* a computer scientist.)

- In my day job, I write high-performance code for semiconductor applications of computational geometry and photolithography.

- In my spare time, I am teaching a course ‘numerical software development’ in the department of computer science in NCTU.

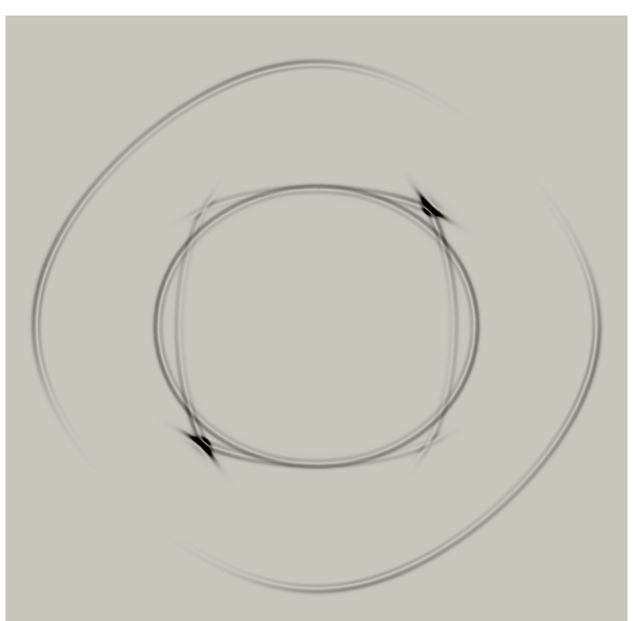
twitter: https://twitter.com/yungyuc
linkedin: https://www.linkedin.com/in/yungyuc/.

Example: PDEs

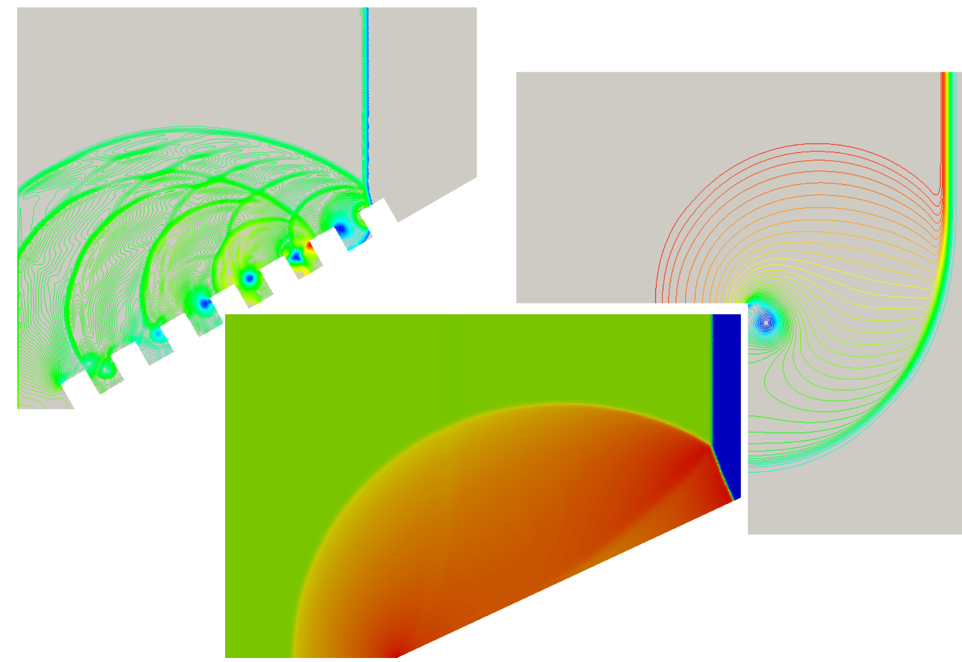
Numerical simulations of conservation laws:

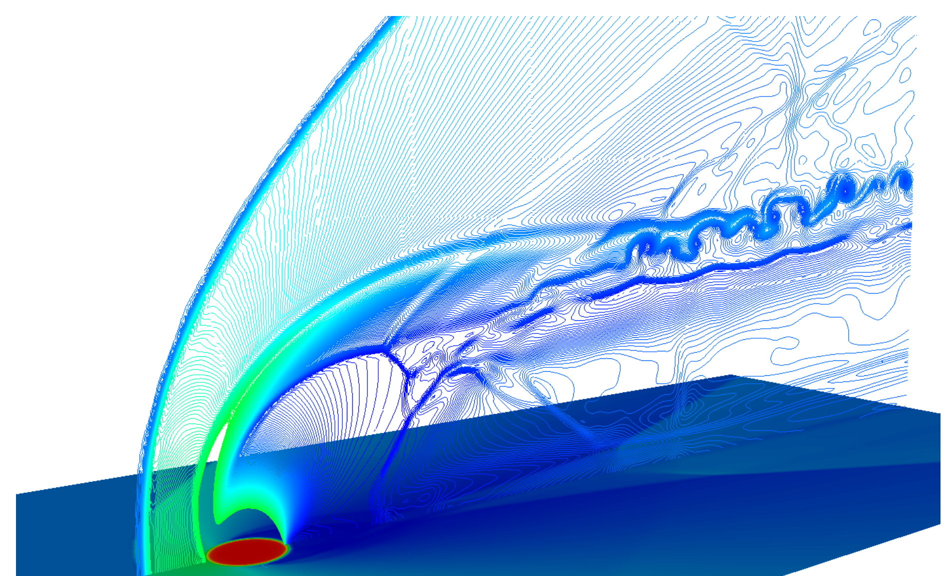
$$\frac{\partial \mathbf{u}}{\partial t} + \sum_{k=1}^{3} \frac{\partial \mathbf{F}^{(k)}(\mathbf{u})}{\partial x_k} = 0$$

Use case: stress waves in anisotropic solids



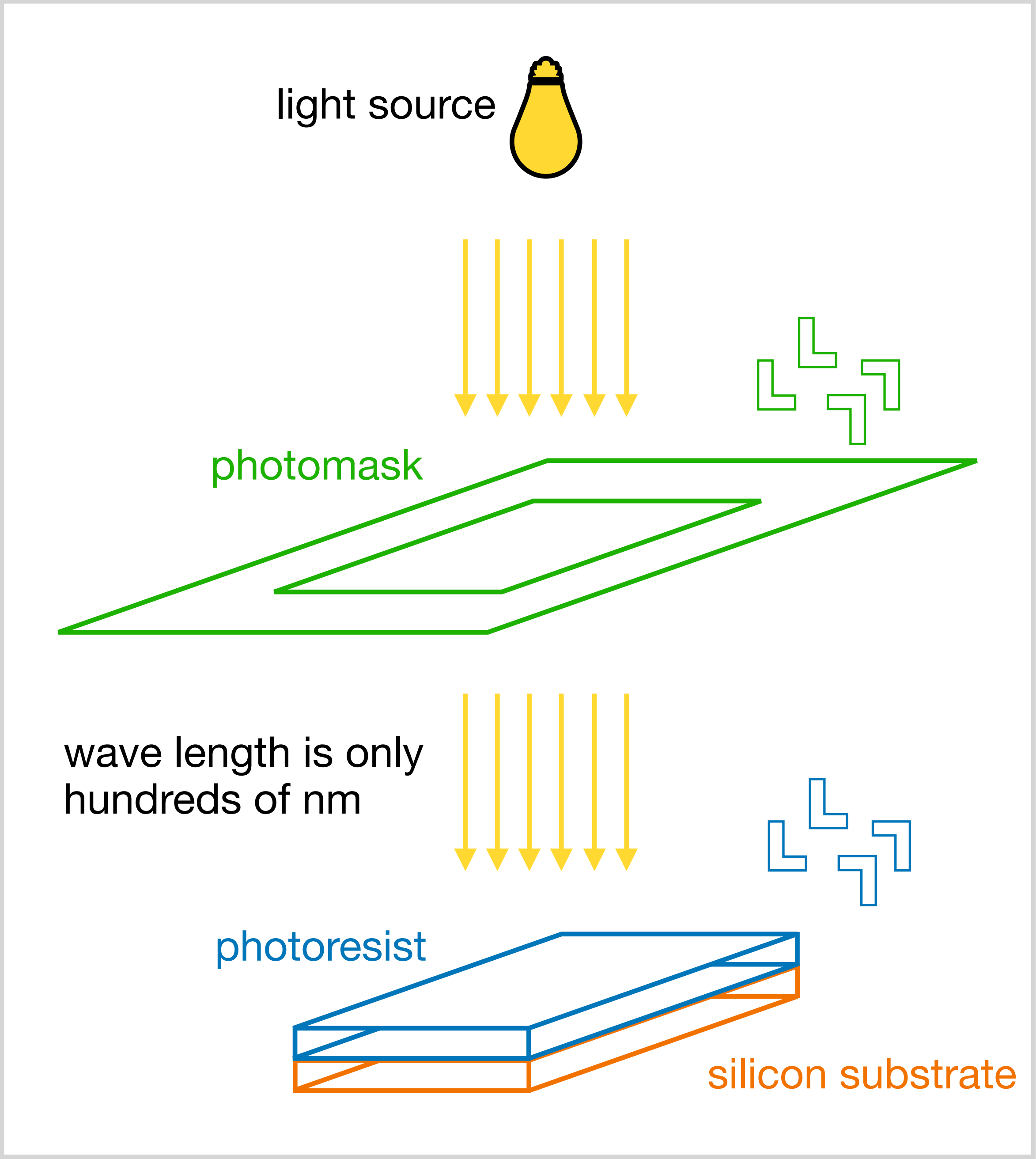
Use case: compressible flows



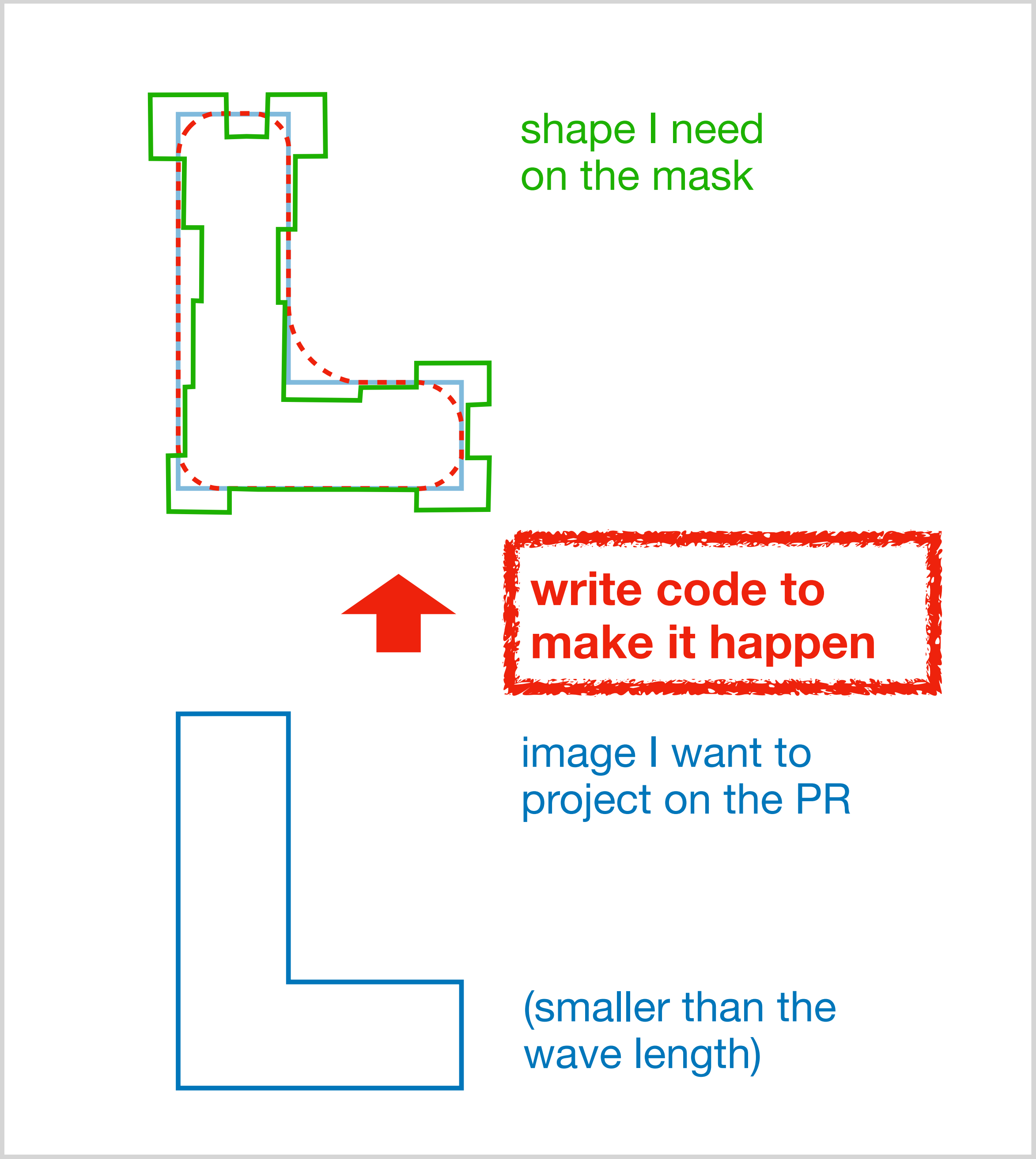


Example: OPC

Photolithography in semiconductor fabrication



Optical proximity correction (OPC)



Python: the good is the bad

```
import math
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    @property
    def length(self):
        return math.hypot(self.x, self.y)
```

```
print(Point(3, 4).length)  →  5.0
```

```
print(Point("3", "4").length)  →  Traceback (most recent call last):
                                    File "ambiguity.py", line 10, in <module>
                                      print(Point("3", "4").length)
                                    File "ambiguity.py", line 8, in length
                                      return math.hypot(self.x, self.y)
                                  TypeError: a float is required
```

- Dynamicity enables convention over compilation for productivity
- It is so when you have good *testing* and *documentation*

What is fast?

High-level language

```
class Data {
    int m_storage[1024];
public:
    int & operator[] (size_t it) { return m_storage[it]; }
};

Data & worker(Data & data, size_t idx, int value) {
    data[idx] = value;
    return data;
}
```

```
worker(Data&, unsigned long, int):
        mov     DWORD PTR [rdi+rsi*4], edx
        mov     rax, rdi
        ret
```

Machine code

Dynamic typing is slow

- Python trades runtime for convenience; everything is indirect and incurs runtime overhead

```python
def work(data, key, name, value):
    # Assign value to the data as a container with key.
    data[key] = value
    # Assign value to the data as an object with name.
    setattr(data, name, value)
    return data
```

- The innocent-looking Python code uses many checking functions under the hood

```python
def work(data, key, name, value):


    data[key] = value
    setattr(data, name, value)
    return data
```

```c
static PyObject *__pyx_pf_6simple_work(
    PyObject *__pyx_v_data, PyObject *__pyx_v_key
  , PyObject *__pyx_v_name, PyObject *__pyx_v_value
) {
  PyObject_SetItem(__pyx_v_data, __pyx_v_key, __pyx_v_value);
  PyObject_SetAttr(__pyx_v_data, __pyx_v_name, __pyx_v_value);
  return __pyx_v_data;
}
```

data[key] = value

```c
int PyObject_SetItem(PyObject *o, PyObject *key, PyObject *value)
{
    PyMappingMethods *m;
    if (o == NULL || key == NULL || value == NULL) {
        null_error();
        return -1;
    }
    m = o->ob_type->tp_as_mapping;                              mapping
    if (m && m->mp_ass_subscript)
        return m->mp_ass_subscript(o, key, value);
    if (o->ob_type->tp_as_sequence) {                           sequence
        if (PyIndex_Check(key)) {
            Py_ssize_t key_value;
            key_value = PyNumber_AsSsize_t(key, PyExc_IndexError);
            if (key_value == -1 && PyErr_Occurred())
                return -1;
            return PySequence_SetItem(o, key_value, value);
        }
        else if (o->ob_type->tp_as_sequence->sq_ass_item) {
            type_error(/* ... omit ... */);
            return -1;
        }
    }
    type_error(/* ... omit ... */);
    return -1;
}
```

(in Objects/abstract.c)

setattr(data, name, value)

```c
int PyObject_SetAttr(PyObject *v, PyObject *name, PyObject *value)
{
    PyTypeObject *tp = Py_TYPE(v);
    int err;                                            attribute name check
    if (!PyUnicode_Check(name)) {
        PyErr_Format(PyExc_TypeError, /* ... omit ... */);
        return -1;
    }
    Py_INCREF(name);
    PyUnicode_InternInPlace(&name);                             check to set
    if (tp->tp_setattro != NULL) {
        err = (*tp->tp_setattro)(v, name, value);
        Py_DECREF(name);
        return err;
    }
    if (tp->tp_setattr != NULL) {
        const char *name_str = PyUnicode_AsUTF8(name);
        if (name_str == NULL) {
            Py_DECREF(name);
            return -1;
        }
        err = (*tp->tp_setattr)(v, (char *)name_str, value);
        Py_DECREF(name);
        return err;
    }
    Py_DECREF(name);
    _PyObject_ASSERT(name, name->ob_refcnt >= 1);
    if (tp->tp_getattr == NULL && tp->tp_getattro == NULL)
        PyErr_Format(PyExc_TypeError, /* ... omit ... */);
    else
        PyErr_Format(PyExc_TypeError, /* ... omit ... */);
    return -1;
}
```

(in Objects/object.c)

Speed is the king

To pursue high-performance and flexibility simultaneously, the systems need to use the best tools at the proper layers. This is a path to develop such systems.

- **Python controls execution flow**
- **C++ manages resources**
- **Use C++ to generate fast assembly code**
- **Glue Python and C++**

PyObject uses malloc

```
PyObject *
_PyObject_New(PyTypeObject *tp)
{
    PyObject *op;
    op = (PyObject *) PyObject_MALLOC(_PyObject_SIZE(tp));
    if (op == NULL)
        return PyErr_NoMemory();
    return PyObject_INIT(op, tp);
}
```

PyObject_MALLOC() is a wrapper to the standard C memory allocator malloc().

Python call stack

```python
import inspect

def show_stack():
    # Print calling stacks.
    f = inspect.currentframe()
    i = 0
    while f:
        name = f.f_code.co_name
        if '<module>' == name:
            name += ': top-level module'
        print("#%d line %d: %s" % (i, f.f_lineno, name))
        f = f.f_back
        i += 1

    # Obtain a local variable defined in an outside stack.
    f = inspect.currentframe().f_back
    print('local_val is:', f.f_locals['local_var'])

    del f

def caller():
    local_var = "defined outside"
    show_stack()

caller()
```

```
#0 line 11: show_stack
#1 line 23: caller
#2 line 25: <module>: top-level module

local_val is: defined outside
```

- Python maintains its own stack
- It’s different from that of the ‘real machine’
- The two (C/C++ and Python) call stacks work very differently

Best practice: glue

- For performance and compile-time checking, we do not want to use Python
- For flow control and configuration, we do not want to use C++
- Glue tools:
    - Python C API : https://docs.python.org/3/c-api/index.html
    - Cython : https://cython.org/
    - Boost.Python : https://www.boost.org/doc/libs/1_74_0/libs/python/doc/html/index.html
    - Pybind11 : https://github.com/pybind/pybind11

I only listed tools that require manual generation of wrapping code and compilation.

Pick pybind11

Tools	Pros	Cons
Python C API	always available	manual reference count is hard to get right
Cython	easy to use	no comprehensive C++ support
Boost.Python	comprehensive C++ support	requires boost
Pybind11	easy-to-use comprehensive modern C++ support	require C++11

Build system: cmake

- Again, there are many tools for build systems
  - Plain make
  - Python distutils, setuptools, pip, etc.
  - cmake
- cmake:
  - Cross platform
  - Consistent and up-to-date support

How cmake works

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.9)
project(pybmod)

find_package(pybind11 REQUIRED)
include_directories(${pybind11_INCLUDE_DIRS})

pybind11_add_module(
    _pybmod
    pybmod.cpp
)
target_link_libraries(_pybmod
    PRIVATE ${MKL_LINKLINE})
set_target_properties(_pybmod
    PROPERTIES CXX_VISIBILITY_PRESET "default")

add_custom_target(_pybmod_py
    COMMAND ${CMAKE_COMMAND} -E
        copy $<TARGET_FILE:_pybmod>
        ${PROJECT_SOURCE_DIR}
    DEPENDS _pybmod)
```

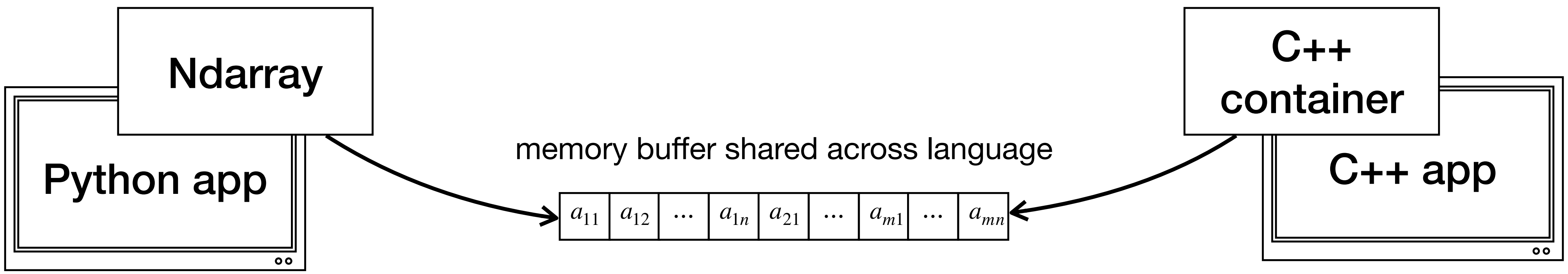
pybmod.cpp

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

std::vector<double> distance(
    std::vector<double> const & x
  , std::vector<double> const & y) {
    std::vector<double> r(x.size());
    for (size_t i = 0 ; i < x.size() ; ++i)
    {
        r[i] = std::hypot(x.at(i), y.at(i));
    }
    return r;
}

PYBIND11_MODULE(_pybmod, mod) {
    namespace py = pybind11;
    mod.doc() = "simple pybind11 module";
    mod.def("distance", &distance);
}
```

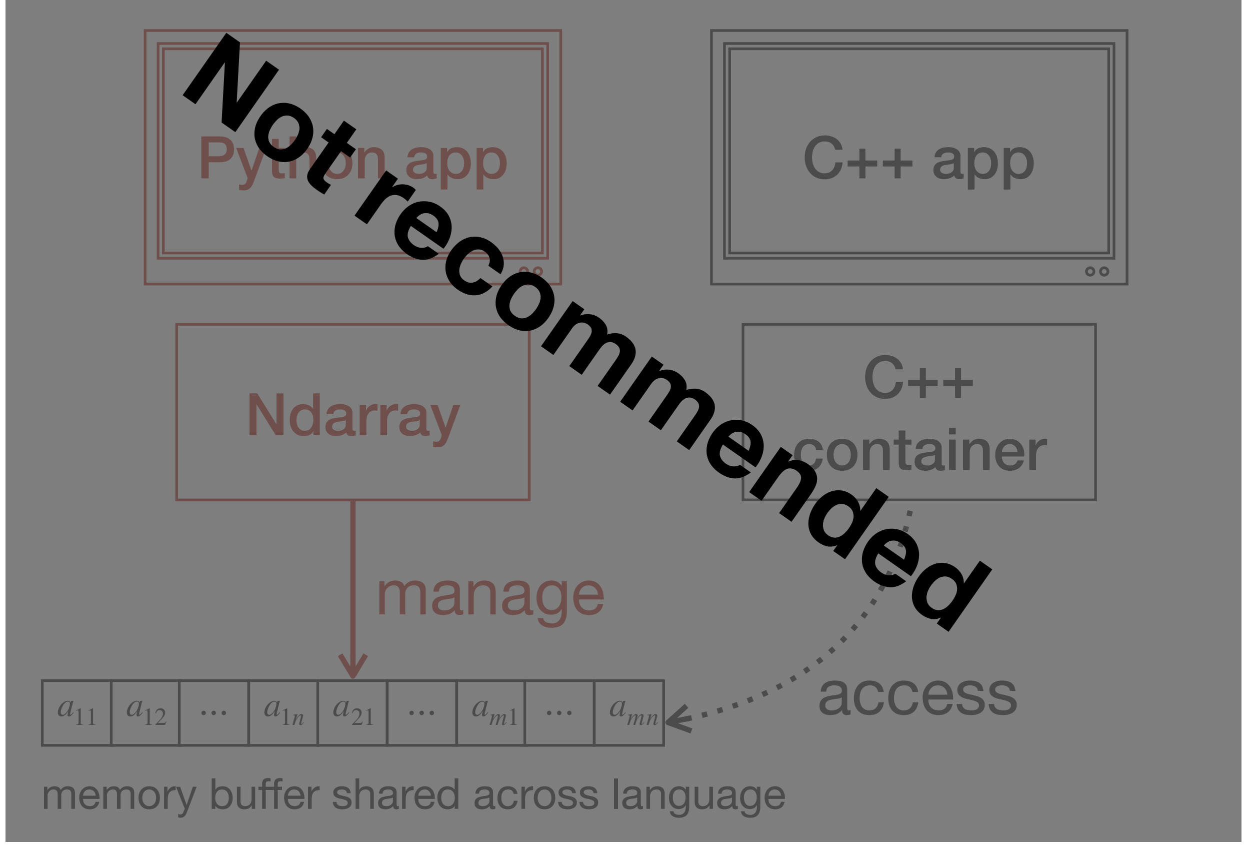
```
$ mkdir -p build ; cd build
$ cmake .. -DCMAKE_BUILD_TYPE=Release
$ make -C build _pybmod_py
$ python3 -c 'import _pybmod ; print(_pybmod.distance([3, 8], [4, 6]))'
[5.0, 10.0]
```

Zero-copy: do it where it fits

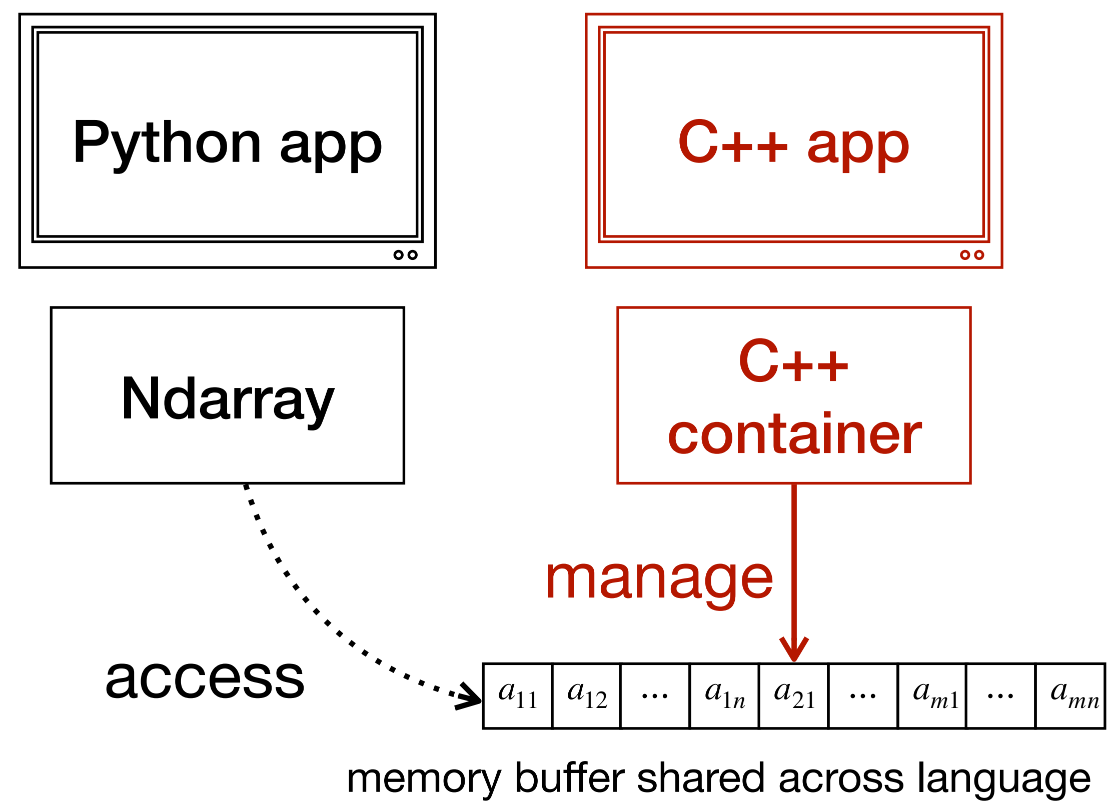
for performance



Top (Python) - down (C++)



Bottom (C++) - up (Python)



Share buffer from C++

Two handy approaches by pybind11

Python buffer protocol:

- The C++ class itself is turned into an array description

```
cls
.def_buffer
(
    [](wrapped_type & self)
    {
        namespace py = pybind11;
        return py::buffer_info
        (
            self.data() /* Pointer to buffer */
          , sizeof(int8_t) /* Size of one scalar */
          , py::format_descriptor<char>::format()
            /* Python struct-style format descriptor */
          , 1 /* Number of dimensions */
          , { self.size() } /* Buffer dimensions */
          , { 1 } /* Strides (in bytes) for each index */
        );
    }
)
```

Return a distinct numpy ndarray:

- The C++ class can return multiple buffers

```
cls
.def_property_readonly
(
    "ndarray"
  , [](wrapped_type & self)
    {
        namespace py = pybind11;
        return py::array
        (
            py::detail::npy_format_descriptor<int8_t>::dtype()
            /* Numpy dtype */
          , { self.size() } /* Buffer dimensions */
          , { 1 } /* Strides (in bytes) for each index */
          , self.data() /* Pointer to buffer */
          , py::cast(self.shared_from_this())
            /* Owning Python object */
        );
    }
)
.def_property_readonly(“ndarray2”, /* ... */)
```

Check assembly

- Make sure compilers generate good code for performance hotspot
    - Checking instead of writing: we don't have time to use assembly for everything
- If compilers don't do a good job, before jumping into hand-written assembly, consider to use intrinsics: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

```
void calc_distance(
    size_t const n
  , double const * x
  , double const * y
  , double * r) {
    for (size_t i = 0 ; i < n ; ++i)
    {
        r[i] = std::sqrt(x[i]*x[i] + y[i]*y[i]);
    }
}
```

-O3 -mavx2

-O3

```
vmovupd ymm0, ymmword [rsi + r9*8]
vmulpd ymm0, ymm0, ymm0
vmovupd ymm1, ymmword [rdx + r9*8]
vmulpd ymm1, ymm1, ymm1
vaddpd ymm0, ymm0, ymm1
vsqrtpd ymm0, ymm0
```

AVX: 256-bit-wide vectorization

```
movupd xmm0, xmmword [rsi + r8*8]
mulpd xmm0, xmm0
movupd xmm1, xmmword [rdx + r8*8]
mulpd xmm1, xmm1
addpd xmm1, xmm0
sqrtpd xmm0, xmm1
```

SSE: 128-bit-wide vectorization

```
# Use redare2 to print the assembly.
r2 -Aqc "e scr.color=0 ; s sym.calc_distance_unsignedlong_doubleconst__doubleconst__double ; pdf" \
_pybmod.cpython-37m-darwin.so
```

Manage resources

- Python uses reference count to manage object lifecycle. You can do the in C++ by using shared pointer.

```
class ConcreteBuffer
  : public std::enable_shared_from_this<ConcreteBuffer>
{
private:
    struct ctor_passkey {};
public:
    static std::shared_ptr<ConcreteBuffer> construct(size_t nbytes)
    {
        return std::make_shared<ConcreteBuffer>(nbytes, ctor_passkey());
    }
    ConcreteBuffer(size_t nbytes, const ctor_passkey &)
      : m_nbytes(nbytes)
      , m_data(allocate(nbytes))
    {}
};
```

- Reference counting involves a lock and is slow. You didn't experience it in Python because (i) other parts in Python are even slower and (ii) you didn't have a choice.

Translate reference count

- If reference count is used in C++, make sure it is correctly translated between Python.

```cpp
pybind11::class_
<
    ConcreteBuffer
  , std::shared_ptr<ConcreteBuffer>
>(
    mod, "ConcreteBuffer"
  , "Contiguous memory buffer"
);
```

```cpp
class ConcreteBuffer
  : public std::enable_shared_from_this<ConcreteBuffer>
{
private:
    struct ctor_passkey {};
public:
    static std::shared_ptr<ConcreteBuffer> construct(size_t nbytes)
    {
        return std::make_shared<ConcreteBuffer>(nbytes, ctor_passkey());
    }
    ConcreteBuffer(size_t nbytes, const ctor_passkey &)
      : m_nbytes(nbytes)
      , m_data(allocate(nbytes))
    {}
};
```

- If reference count is used in C++, make sure it is correctly translated between Python.

Run Python in C++

```cpp
py::str cppfunction() {
    py::list lst = py::list();
    lst.append("one");
    lst.append("two");
    lst.append("three");
    return py::str(", ").attr("join")(lst);
}
```

```python
>>> joined = cppfunction()
>>> print(joined)
one, two, three
```

- When C++ code puts together the architecture, oftentimes we want to keep Python code as little as possible.
- But Python is used for scripting and it's necessary to return Python objects for the scripts.
- Pybind11 allows to write concise C++. It's much more maintainable than in-line Python.

```cpp
// No, don't do this!
PyRun_SimpleString("', '.join('one', 'two', 'three')");
```

Import and run

Pybind11 API is very Pythonic

```cpp
void show_modules() {
    py::module sys = py::module::import("sys");
    py::print(py::len(sys.attr("modules")));
    py::print(sys.attr("getrecursionlimit")());
}
```

```python
def show_modules():
    import sys
    print(len(sys.modules))
    print(sys.getrecursionlimit())
```

```
>>> show_modules()
1101
3000
```

Spend time in compilation to save time in runtime

Compiler does a lot of good. Make use of it as much as possible.

```
template < typename T >
class SimpleArray
{
public:
    using value_type = T;
    using shape_type = small_vector<size_t>;
    // Determine content value element size during compile time.
    static constexpr size_t ITEMSIZE = sizeof(value_type);
    static constexpr size_t itemsize() { return ITEMSIZE; }
    // Straight constructor.
    explicit SimpleArray(size_t length)
      : m_buffer(ConcreteBuffer::construct(length * ITEMSIZE))
      , m_shape{length}
      , m_stride{1}
    {}
    template< class InputIt > SimpleArray(InputIt first, InputIt last)
      : SimpleArray(last-first)
    {
        // Let STL decide how to optimize memory copy.
        std::copy(first, last, data());
    }
};
```

Compiler reduces runtime errors

- Wrap the wrappers: reduce duplicated code
- Do it in compile time to reduce runtime errors

```cpp
template
<
    class Wrapper
  , class Wrapped
  , class Holder = std::unique_ptr<Wrapped>
  , class WrappedBase = Wrapped
>
class WrapBase
{
public:
    using wrapper_type = Wrapper;
    using wrapped_type = Wrapped;
    using wrapped_base_type = WrappedBase;
    using holder_type = Holder;

    // Treat inheritance hierarchy.
    using class_ = typename std::conditional_t
    <
        std::is_same< Wrapped, WrappedBase >::value
      , pybind11::class_
        < wrapped_type, holder_type >
      , pybind11::class_
        <wrapped_type, wrapped_base_type, holder_type>
    >;

    // Singleton.
    static wrapper_type & commit(pybind11::module & mod)
    {
        static wrapper_type derived(mod);
        return derived;
    }

    class_ & cls() { return m_cls; }

protected:
    // Register through construction.
    template <typename... Extra>
    WrapBase(
        pybind11::module & mod
      , char const * pyname, char const * pydoc
      , const Extra & ... extra
    )
      : m_cls(mod, pyname, pydoc, extra ...)
    {}

private:
    class_ m_cls;
};
```

Decouple resource management from algorithms

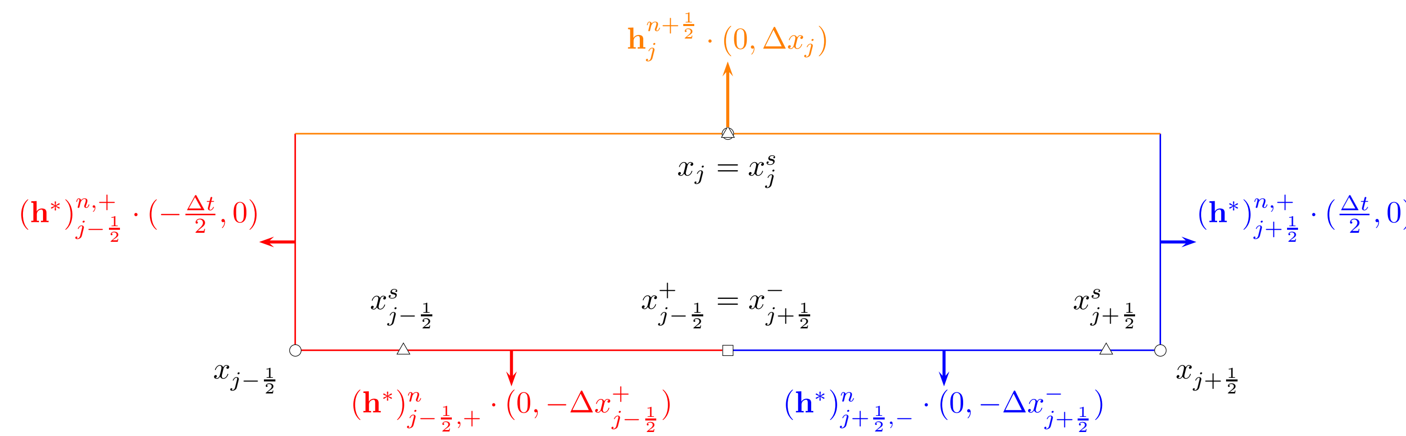
- Fixed-size contiguous data buffer

```
class ConcreteBuffer
  : public std::enable_shared_from_this<ConcreteBuffer>
{
private:
    struct ctor_passkey {};
public:
    static std::shared_ptr<ConcreteBuffer> construct(size_t nbytes)
    { return std::make_shared<ConcreteBuffer>(nbytes, ctor_passkey()); }
    ConcreteBuffer(size_t nbytes, const ctor_passkey &)
      : m_nbytes(nbytes), m_data(allocate(nbytes))
    {}
private:
    using unique_ptr_type =
        std::unique_ptr<int8_t, std::default_delete<int8_t[]>>;
    size_t m_nbytes;
    unique_ptr_type m_data;
};
```

- Descriptive data object owning the data buffer

```
template < typename T >
class SimpleArray
{
public:
    using value_type = T;
    using shape_type = small_vector<size_t>;
    static constexpr size_t ITEMSIZE = sizeof(value_type);
    explicit SimpleArray(size_t length)
      : m_buffer(ConcreteBuffer::construct(length * ITEMSIZE))
      , m_shape{length}
      , m_stride{1}
    {}
private:
    std::shared_ptr<ConcreteBuffer> m_buffer;
    shape_type m_shape;
    shape_type m_stride;
};
```

Encapsulate complex calculation without losing performance



3 solution elements around a compound conservation element

```
// A solution element.
class Selm : public ElementBase<Selm>
{
public:
    value_type dxneg() const { return x()-xneg(); }
    value_type dxpos() const { return xpos()-x(); }
    value_type xctr() const { return (xneg()+xpos())/2; }

    value_type const & so0(size_t iv) const { return field().so0(xindex(), iv); }
    value_type       & so0(size_t iv)       { return field().so0(xindex(), iv); }

    value_type const & so1(size_t iv) const { return field().so1(xindex(), iv); }
    value_type       & so1(size_t iv)       { return field().so1(xindex(), iv); }

    value_type const & cfl() const { return field().cfl(xindex()); }
    value_type       & cfl()       { return field().cfl(xindex()); }

    value_type xn(size_t iv) const { return field().kernel().calc_xn(*this, iv); }
    value_type xp(size_t iv) const { return field().kernel().calc_xp(*this, iv); }
    value_type tn(size_t iv) const { return field().kernel().calc_tn(*this, iv); }
    value_type tp(size_t iv) const { return field().kernel().calc_tp(*this, iv); }
    value_type so0p(size_t iv) const { return field().kernel().calc_so0p(*this, iv); }
    void update_cfl() { return field().kernel().update_cfl(*this); }
};
```

(Part of a code for the space-time conservation element and solution element (CESE) method)

Conclusion

- Speed is the king for numerical calculation
- Use Python for configuration and C++ for number crunching: glue them
- Keep in mind: Software engineering, zero-copy data buffer, read assembly, manage resources, use compiler for speed and safety
- More information: https://github.com/yungyuc/nsd

Thank you