# Hydra

## A library for data analysis in massively parallel platform

Deepanshu Thakur

Presented at GSoC/Hydra meeting, CERN, June , 2017


Google
Summer of Code

# Outline

- Callback functions bindings
- General workaround that can be used for simple variables
- What I have achieved till now
- A thought regarding CRTP
- Summary

# Callback functions bindings

It is actually very easy to bind a callback in pybind11.

```
int func_callback_stateless(std::function<double(unsigned int, double, double)> &f) {
  int a = 10;
  double b = 5.95, c = 5.99;
  return f(a, b, c);
}
PYBIND11_MODULE(foo, m) {
  m.def("func_arg", &func_callback_stateless);
}
```

# Interesting thing regarding callback function taking pointers and references

Take a look at this function.

```
double wierd(unsigned int n, double* a, double *b) {
  double d = (*a) * (*b);
  *a = 100.0; // Try to change the value of a
  return d;
}
```

I use this function and it did the exact same thing as expected. The value of variable a was not changed. Which is expected because int, bool, string, double are immutable in python.

# Workaround

A typical workaround could be to write a small wrapper lambda function that returns a tuple with all output arguments.

```
int change(int* a) {
  *a *= 10;
  return 42;
}
PYBIND11_MODULE(foo, m) {
  m.def("c", [](int *i) { int rv = change(i); return std::make_tuple(rv, *i); });
}
```

# Workaround (continue … )

And in python we can do

```
In [9]: a = 21

In [10]: ret, a = foo.c(a)

In [11]: ret

Out[11]: 42

In [12]: a

Out[12]: 210
```

But again the issue is when we are trying to pass the array as pointers from python, which is common in C++

# Solving fundamental problem regarding callback

Though I am still trying to figure out, how can we bind the CRTP idiom (thoughts on this on next slide), I was able to use the callbacks like

```
std::function<double(unsigned int, std::vector<double>&, double*)> functor_t2;
```

From pybind11. I am trying to figure out a way in which change in vector<double> from python function will create effects on passed C++ vector. We can create a wrapper and than use that in C++ and changing values from C++ will change values in python list too. For example

```
PYBIND11_MAKE_OPAQUE(std::vector<double>);

void function(std::size_t size, double* a)

 for (std::size_t i = 0; i < size; ++i) a[i] += 42;



void wrapper(std::vector<double>& vec)

  for (auto i = vec.begin(); i != vec.end(); ++i) *i += 100;

PYBIND11_MODULE(foo, m) {

  py::bind_vector<std::vector<double>>(m, "DVector");

  m.def("function", &wrapper);

}
```

# And in python

```
a = [1, 2, 3]

a = foo.DVector(a)

foo.function(a)

a

Out: DVector[101, 102, 103]
```

# CRTP

We can’t bind the C++ CRTP directly in Python, in a “statically way”. Because the CRTP depends on a template type that we might not know before compile time (that is, before binding it to python, the template type “could” be defined by the user). So one solution we can adopt is, all of our pybind bindings can be parameterized by a C++ template, say T. Then we can have  a function to ask at runtime, the type user want to instantiate (say Foo, of course it had to be defined in a C++ code somewhere accessible by our bindings).  Once the user submi the type he wanted (Foo), the Python function can compile our pybind bindings (by calling g++/clang in a new process), replacing the template parameter T by the actual type Foo the user wanted... and so can have a shared library .so with our python bindings adapted to the Foo type for the CRTP.

# Summary

- We can actually bind the CRTP with the specified technique. This can be useful with fwrapper<N> for “N” which has not been already defined by our Macro trick. So say we compiled the library upto N <= 100 than we can use the above specified trick at runtime to compile the bindings for N > 100.
- I am still looking for a way to pass an array a pointer.
- One thing we can do is to create a wrapper that accepts python list and call another C++ function that convert that list into C++ array.