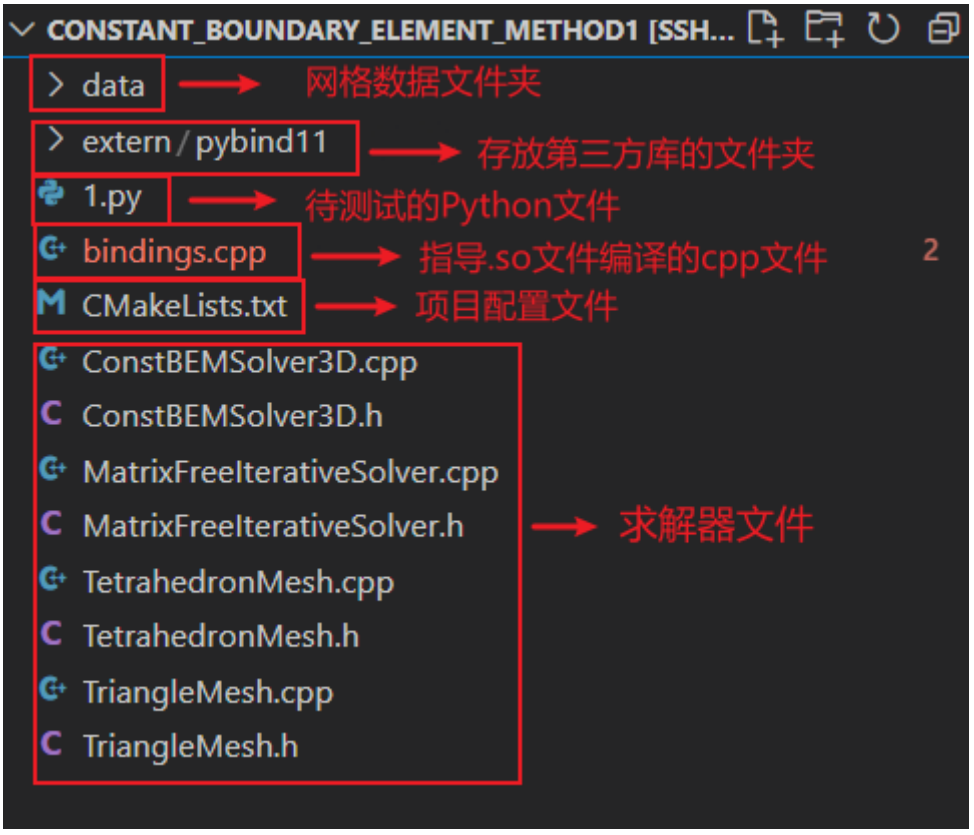
准备依赖

pybind11是个C++的**header-only**的库。因此无需安装，只需要有头文件即可。
这是pybind11的github：https://github.com/pybind/pybind11， 可以直到github下安装，或者用git克隆：

```
git clone https://github.com/pybind/pybind11 --depth=1
```

项目结构

文件结构



CMakeLists.txt 文件

```
cmake_minimum_required(VERSION 3.23)
project(ConstBEMSolver_3D)

set(CMAKE_CXX_FLAGS "${CMAKE_C_FLAGS} -Wall -O3")

# Add pybind11
add_subdirectory(extern/pybind11)

# Add the source files
set(SOURCES
    ConstBEMSolver3D.cpp
    MatrixFreeIterativeSolver.cpp
    TriangleMesh.cpp
    bindings.cpp
)
```

```
# Create the Python module
pybind11_add_module(ConstBEMSolver_3D ${SOURCES})
```

上面最后一行代码是调用了pybind11的CMake函数，作用是创建一个python模块，并导入源文件。

bindings.cpp 文件

```
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>
#include "ConstBEMSolver3D.h"
#include "MatrixFreeIterativeSolver.h"
#include "TriangleMesh.h"

namespace py = pybind11;

// 当返回值为数组时
py::array_t<double> getVar(ConstBEMSolver_3D& self) {
    // 获取 density_ 数组的指针
    double* arrayPtr = self.getVar();

    // 获取数组的大小
    size_t size = self.getNElement();

    // 创建 NumPy 数组，使用 buffer_info 来管理内存
    auto array = py::array(py::buffer_info(
    arrayPtr,             // 指向数据的指针
    sizeof(double),       // Size of one scalar
    py::format_descriptor<double>::format(), // Python struct-style format
descriptor
    1,                    // Number of dimensions
    { size },             // Shape of the array
    { sizeof(double) }    // Strides (in bytes) for each axis
    ));

    // 返回 NumPy 数组
    return array;
}

// 参数是一维数组
void setVar(ConstBEMSolver_3D& self, py::array_t<double> array) {
    // 请求 NumPy 数组的 buffer_info
    py::buffer_info buf = array.request();

    // 获取数组的指针
    double* ptr = static_cast<double*>(buf.ptr);

    // 调用原始的 setVar 函数
    self.setVar(ptr);
}

// 解决Python GIL线程锁释放的问题
void preconditionsolve_with_GIL(ConstBEMSolver_3D& self) {
    // Release GIL to allow other Python threads to run
    py::gil_scoped_release release;
```

```
    self.preconditionsolve();

    // Re-acquire GIL before returning to Python
    py::gil_scoped_acquire acquire;
}

PYBIND11_MODULE(ConstBEMSolver_3D, m)
{
    py::class_<TriangleMesh>(m, "TriangleMesh")
        .def(py::init<>())
        .def("read_off", &TriangleMesh::read_off);


    py::class_<ConstBEMSolver_3D>(m, "ConstBEMSolver_3D")
        .def(py::init<TriangleMesh*>())

        .def("LinfError", &ConstBEMSolver_3D::LinfError)
        .def("outputTecPlotDataFile", &ConstBEMSolver_3D::outputTecPlotDataFile)
        .def("computeRHS", &ConstBEMSolver_3D::computeRHS)

        .def("getVar", &getVar)  // 绑定一个double* 型的成员变量
        .def("setVar", &setVar) // 绑定一个参数是double* 型的函数
        .def("preconditionsolve", &preconditionsolve_with_GIL); // 绑定一个需要释放
GIL线程锁实现多线程运行的函数

}
```

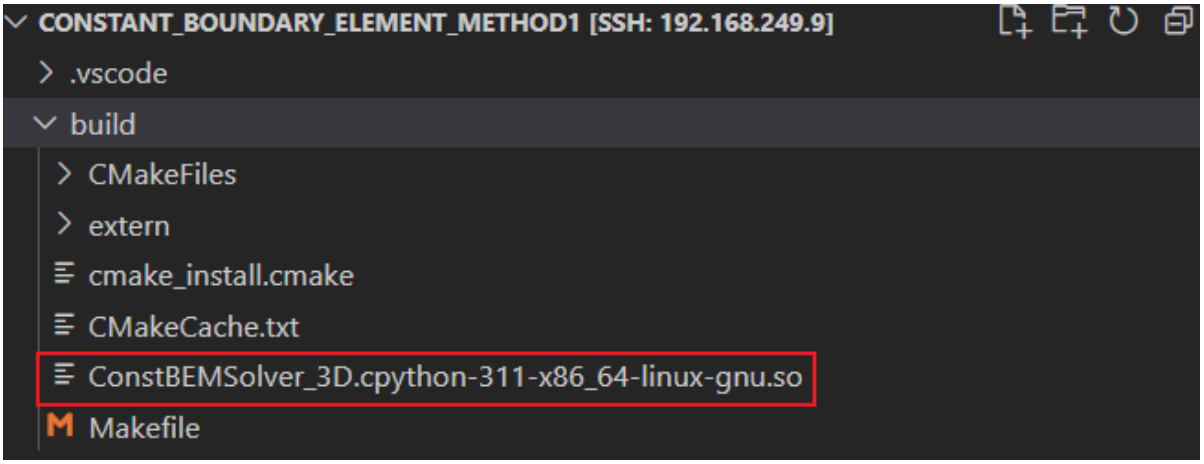
- 导入必要的头文件 包括代编译的源代码，Pybind11的部分头文件。
- 使用PYBIND11_MODULE函数创建一个Python模块，第一个参数为模块的名字，第二个参数类型为 _py::model__ _的变量（m），它代表了正在创建的Python模块，并且是创建绑定的主要接口。
- 然后通过 py::class_ 添加两个类，TriangleMesh和ConstBEMSolver_3D，并添加需要在Python脚本中使用的成员函数
    - 构造函数，C++中的构造函数直接通过默认的 py::init<>() 即可添加，如果有参数像 ConstBEMSolver_3D(TriangleMesh *mesh)，则在<>中添加即可，py::init<TriangleMesh>()。
    - 绑定其他成员变量时，.def( ) 第一个参数是对应Python模块中的函数名，第二个参数是待绑定的C++函数
        - 无参数无返回值函数，则直接跟上绑定函数即可
        - 有参数有返回值，但参数和返回值都是基本类型，int，double，直接跟上绑定函数即可
        - 无参数有返回值，返回值类型是数组，即double*，需通过一个中间函数将绑定函数返回的C++数组转化为Python的ndarray即可
        - 有参数无返回值，参数类型是数组，即double*，需通过一个中间函数，该函数的参数类型为py::array_t&（Pybind11提供的C++环境下的ndarray类型），将接收到的Python代码中的ndarray转化为C++数组后，再将数组传至绑定函数即可
        - 有参数有返回值，参数和返回值类型都为数组时，结合上述两步即可
    - 若遇到需要再Python脚本中进行多线程处理的函数，则需在绑定过程中通过一中间函数手动释放GIL线程锁，在待绑定的函数前后输入py::gil_scoped_release release，py::gil_scoped_acquire acquire 即可

编译过程

在Linux操作系统下，直接使用CMake+make即可。
在项目目录下，输入指令

```
mkdir build
cd build
cmake ..
make
```

编译结束后，后在build目录下产生一个.so文件 即Linux系统下的C++动态库文件。



将该文件移动至测试Python脚本下即可导入并测试。

测试

测试的Python脚本

```
import ConstBEMSolver_3D
import numpy as np

# 创建 TriangleMesh 对象并读取文件
mesh = ConstBEMSolver_3D.TriangleMesh()
mesh.read_off("./data/surfacemesh_of_sphere_one.off")

# 创建 ConstBEMSolver_3D 对象并调用成员函数
solver = ConstBEMSolver_3D.ConstBEMSolver_3D(mesh)
solver.computeRHS()
solver.preconditionsolve()
solver.LinfError()
solver.outputTecPlotDataFile('./data/numerical_solution.dat')

print("iterated over.")
print(solver.getVar())  # 测试输出
print(type(solver.getVar()))  # 测试输出类型

array = np.array([1.0, 2.0, 3.0])
solver.setVar(array)  # 测试传参
print(solver.getVar())

print('over')
```

输出结果如下

```
(base) test@test-virtual-machine:~/code/python/pybind11/constant_boundary_element_method1$ python 1.py
iteration steps =1
iteration steps =2
iteration steps =3
iteration steps =4
iteration steps =5
iteration steps =6
iteration steps =7
iteration steps =8
iteration steps =9
iteration steps =10
iteration steps =11
iteration steps =12
iteration steps =13
iteration steps =14
iteration steps =15
iteration steps =16
iteration steps =17
iteration steps =18
iteration steps =19
iteration steps =20
iteration steps =21
dof= 5120  Linf err:0.00118932
dof= 5120  L2 err:1.86209e-05
iterated over.
[1.00504108 1.00504108 1.00504112 ... 1.00424059 1.00431373 1.0041555 ]
<class 'numpy.ndarray'>
[1. 2. 3. ... 0. 0. 0.]
over
```

测试成功!