Furion 帮助手册

作者：周豪

日期：2023 年 9 月 20 日

目录

1 编译、运行以及管理

1.1 CMake 配置文件 CMakeLists.txt

CMake 编写一种平台无关的 CMakeList.txt 文件来定制整个编译流程，然后再根据目标用户的平台进一步生成所需的本地化 Makefile 和工程文件，如 Unix 的 Makefile 或 Windows 的 Visual Studio 工程。相比于使用 Visual Studio 直接编译，使用 CMake 速度要快很多。

首先在 Visual Studio 中创建 CMake 项目，名称为 Fusion。在 Fuion 文件夹里修改 CMakeLists.txt 文件：

```
# 会自动创建两个变量，PROJECT_SOURCE_DIR 和 PROJECT_NAME
# ${PROJECT_SOURCE_DIR}：本 CMakeLists.txt 所在的文件夹路径
# ${PROJECT_NAME}：本 CMakeLists.txt 的 project 名称
cmake_minimum_required(VERSION 3.26.4)          # 指定 CMake 的最低版本：
project(Furion)
set(CMAKE_CXX_STANDARD 17)          # 使用 set 命令来配置编译器选项和其他项目变量。
#include_directories(C:/software/C_library/eigen-3.4.0)      # 修改后的程序不需要 Eigen 库
include_directories(C:/software/C_library/MPI/Include)          # 多核并行 MS MPI 库
include_directories(${CMAKE_SOURCE_DIR})
find_package ( MPI )
message ( STATUS "MPI_FOUND=$MPI_FOUND}" )
message ( STATUS "MPI_CXX_INCLUDE_DIRS=${MPI_CXX_INCLUDE_DIRS}" )
message ( STATUS "MPI_LIBRARIES=${MPI_LIBRARIES}" )
if ( MPI_FOUND )
        list ( APPEND PRJ_INCLUDE_DIRS ${MPI_CXX_INCLUDE_DIRS} )
        list ( APPEND PRJ_LIBRARIES ${MPI_LIBRARIES} )
endif ()
file(GLOB SOURCES "*.cpp")          # 编译当前目录下所有 cpp 源文件
add_executable(Furion ${SOURCES})
```

Visual Studio 会自动读取 cmake，生成构建系统，并创建 bulid 文件夹，进入文件夹，输入 make 进行编译（也可以输入 make -j10 多核并行编译，10 代表核心数）

注意事项：

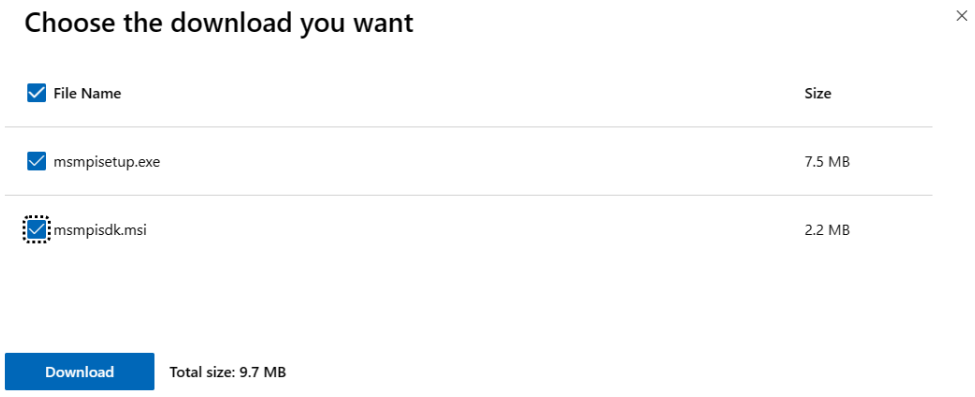
1. 如果项目包含多个子目录，可以使用 add_subdirectory 命令将它们包含到构建过程中。
2. 如果项目依赖于其他 CMake 项目，可以使用 ExternalProject 模块来管理它们的依赖关系。
3. 使用 install 命令来定义如何安装生成的可执行文件、库文件和其他资源。

1.2 Microsoft MPI

由于计算的光路没有相互作用，可以很方便的实现 CPU 并行计算。C++ 在 Windows 系统下面的并行计算 MPI 使用的是 Microsoft MPI 库。

下载地址：`https://www.microsoft.com/en-us/download/details.aspx?id=100593`

点击后进入页面显示为，全选两项下载：



所有都要选上，其中 msmpisetup.exe 是 mpi 运行软件，msmpisdk 是安装需要的库，缺一不可。

参考教程：Win10 下 Microsoft MPI（MSMPI）的下载安装 from 知乎

安装完成后，CMakeLists.txt，：include_directories（Path/MPI/Include）,Path 表示安装 MPI 的文件夹地址。

在 Visual Studio 生成可执行文件 Furion.exe 后，直接在 Visual Studio 只能是单核运行。且可执行文件需要在根目录才能正确运行（涉及到文件输出以及作图），因此写了个 Bash 脚本完成文件的移动、多核并行指令等功能（相关指令及其意义已集成在 run.bat 文件中）。

1.3 Git 管理项目

GitHub 是一个基于 Web 的代码托管平台，它提供了版本控制、协作和项目管理工具，用于开发和共享软件项目。

版本控制：GitHub 使用 Git 作为版本控制系统，允许跟踪和管理代码库的不同版本。可以轻松回滚到早期的代码版本，查看更改历史记录以及协作开发。允许用户

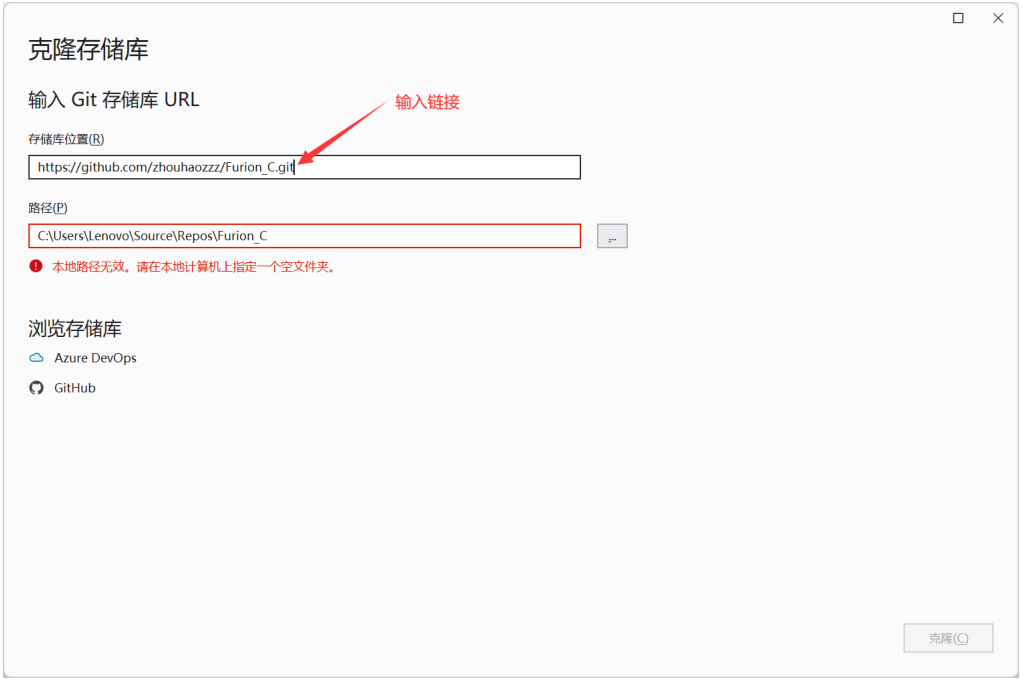
在其平台上托管代码库（称为仓库），并提供了用于代码上传、下载和管理的工具。

分支和合并：分支功能，在不影响主代码库的情况下创建新的分支，进行实验性开发或修复 bug。分支完成后，可以合并回主分支。

Pull 请求：通过 Pull 请求（PR）向项目的主仓库提交代码更改。这使得代码审查和团队协作更加灵活，贡献者可以将自己的更改提交给项目维护者进行审查和合并。

在 Visual Studio 菜单栏选择 Git-> 克隆仓库：输入链接

`https://github.com/zhouhaozzz/Furion_C.git`



即可克隆至本地。相关提取、拉取、推送和同步进行版本控制的操作步骤参考网页：**`https://learn.microsoft.com/zh-cn/visualstudio/version-control/git-fetch-pull-sync?view=vs-2022`**

2 Furion C++ 指南

2.1 头文件.h

在 C++ 中，头文件（header file）通常使用.h 扩展名，并包含了函数声明、类声明、宏定义以及其他需要在多个源文件之间共享的代码。头文件的主要目的是将接口部分与实现部分分离，以便在不暴露具体实现的情况下，在多个源文件中共享代码。

例如 **Furion.h：**

```
#pragma once
#ifndef FUR_FURION_H_
#define FUR_FURION_H_                    //用于防止头文件重复包含的预处理器指令。
#include <iostream>                      //引用相应库
using namespace std;                     //引用命名空间
#define Pi 3.1415926536                  //自定义宏
#define E 2.71828
namespace Furion_NS                      //自定义命名空间
{
    class Furion                 //创建了一个名为 Furion 的 C++ 类，
    {
    public:                      //在 public 访问修饰符下开始定义类的成员。
            class Grating* grating;          //声明其他头文件中定义的类
            Furion(int rank1, int size1);    //构造函数 Furion(), 用于创建对象时初始化对象的成员变量和执行其他必要的初始化操作。构造函数的名称必须与类的名称相同，并且没有返回类型（甚至不是 void）。
            ~Furion();           //析构函数的主要作用是在对象生命周期结束时执行清理工作，例如释放分配的资源、关闭文件、删除对象等。析构函数的名称与类名相同，但前面加上一个波浪号（ ）。
            void init();                     //成员函数
            size_t i = 0;                    //自定义成员变量
            const static int n = 100000;
            double Lambda[5] =  1, 1.55, 2, 2.5, 3 ;          }; }
#endif
```

以及 **G_Beam.h**：

```
#pragma once
#ifndef FUR_G_BEAM_H_
#define FUR_G_BEAM_H_
#include "Furion.h"
#include "Furion_Plot_Sigma.h"          //使用其类他成员需要先引用相应头文件名称
namespace Furion_NS
{
    class G_Beam
    {
    public:
        double* XX = new double[Furion::n];          //创建了一个名为 XX 的指向 double 类型的动态数组，并使用 new 运算符为其分配了堆内存，在整个程序的生命周期内存在，大小通常较大，并且受系统物理内存限制。它适合存储较大的数据结构和对象。
        class Furion_Plot_Sigma f_p_s;          //声明其他头文件中定义的类，以便使用其中的成员函数
    };
}
#endif
```

注意：堆内存可以在整个程序的生命周期内存在，因此不需要使用时必须进行销毁操作：delete，但访问速度较慢，因此需要酌情使用。如果需要声明栈内存可以写为：double* XX[Furion::n]。

注意：成员函数引用了指针数组，在成员函数内部对指针数组的改变会直接修改数组的值。如果想要引用一个指针并保护它的值不被修改，可以使用 const 关键字来声明指向常量的指针。

如果数组的大小未知，可以使用 **vector** 提供一个动态数组：

```
std::vector<int> myVector;          // 创建一个整数类型的空向量
myVector.push_back(42);          // 在向量末尾添加元素
int value = myVector[0];          // 访问第一个元素
int secondValue = myVector.at(1);          // 访问第二个元素
myVector.pop_back();          // 删除向量末尾的元素
```

子类头文件的编方法：例如 **G_Cylinder_Ellipse.h**:

```
#include "g_oe.h"          //引用父类头文件
namespace Furion_NS {
    class G_Cylinder_Ellipse : public G_Oe          表示继承与父类 G_Oe
    {
    public:
        void intersection(double* T) override;          override 用于显式指定派生类中的成员函数是重写（覆盖）基类中的虚函数。而在相应父类函数中开头要添加 virtual 关键字
```

```
    };
}
#endif
```

注意事项：

1. 头文件保护：使用预处理器指令来确保头文件只被包含一次。这可以防止重复定义错误。
2. 只包含必要的内容：头文件应该只包含其他源文件需要的声明和接口信息。避免在头文件中包含大量实现细节，以减小头文件的大小并提高编译效率。
3. 使用前向声明：尽量使用前向声明而不是包含其他头文件，以减少编译时间和减少依赖关系。前向声明指的是在头文件中声明一个类或函数，而不包含其定义。
4. 避免使用 using 指令：因为它可能引入命名冲突。最好在源文件中使用 using 来限定作用域。
5. 避免全局变量：尽量避免在头文件中定义全局变量，因为全局变量可能引起命名冲突和不可预测的行为。
6. 不要在头文件中实现函数：头文件通常只包含函数的声明，而不包含函数的实现。函数的实现应该放在源文件中。

2.2 源文件.cpp

在 C++ 中，源文件通常是包含程序代码的文本文件，这些文件以.cpp 扩展名结尾。源文件包含了程序的实际代码，用于定义类、函数、变量等。

例如 **G_Beam.cpp**：

```
#include "G_Beam.h"
using namespace Furion_NS;                    //使用 Furion_NS 命名空间，而不必使用 Furion_NS:: 限定符来访问这些成员。
G_Beam::G_Beam(double* XX, double* YY, double* phi, double* psi, double lambda) : XX(XX), YY(YY), phi(phi), psi(psi), n(Furion::n), lambda(lambda)    //构造函数接受一些参数，包括四个 double* 类型的指针（XX、YY、phi 和 psi）和一个 double 类型的变量 lambda。：冒号表示构造函数的初始化列表的开始。分别初始化了类 G_Beam 的成员变量 XX、YY、phi、psi、n 和 lambda。这些成员变量的名称与参数名称相同，因此通过这种方式将参数值赋给成员变量。
{
}
G_Beam:: G_Beam() { }
G_Beam G_Beam::translate(double distance)
{           for (int i = 0; i < n; i++)
    {
            XX[i] = XX[i] + distance * tan(phi[i]) * cos(psi[i]);
```

```
            YY[i] = YY[i] + distance * tan(psi[i]);          //默认使用类中初始化的变量，由于定义为指针数组，因此函数执行后直接修改对应地址的值，无需返回相应数组
    ındent      }
        return G_Beam(XX, YY, phi, psi, lambda);          这里返回的是整个 G_Beam 类 }
    void G_Beam::plot_sigma(double distance, int rank1)
    {
        G_Beam beam = translate(distance);          因为 translate 返回 G_Beam 类，因此需要初始化一个 G_Beam 类接收
        f_p_s.Furion_plot_sigma(beam.XX, beam.YY, beam.phi, beam.psi, rank1);          使用其他类的成员函数
    }
```

注意事项：

1. 错误处理：可以使用 try-catch 块来捕获和处理异常，并且返回异常。

3 C++ 矩阵运算

3.1 Eigen 矩阵运算库可行性分析

要实现类似于 Matlab 里的矩阵运算如：点乘、叉乘、求逆、行列式等等可以直接使用 for 循环，也可以使用 Eigen 库。

Eigen 支持各种矩阵和向量操作，包括基本的线性代数运算、特征值分解、奇异值分解等。它还支持稠密和稀疏矩阵。被设计成高性能的线性代数库。它的性能接近于手工优化的 C/C++ 代码，因此适用于对性能有要求的应用程序。

但经过实践发现，在本项目中使用 Eigen 库，或者说大部分矩阵运算都使用 Eigen 库的话，效率反而大大降低。分析原因：

- 例 1：对矩阵求其正弦值：matrix.array().sin() ：只能先将矩阵转换为数组表达式，然后对每个元素应用逐元素的操作。
- 例 2:如果要实现 MATLAB 里例如:(Furion_rotz(obj.chi)*[X;Y;repmat(-ds,1,n)]) 的功能。需要预先创建 [X;Y;repmat(-ds,1,n)] 矩阵，如果矩阵较大，使用 Eigen 创建将花费较长时间。
- 例 3：Eigen 库里并不包含 MATLAB 矩阵运算的所有运算规则。但对于某些简单的任务，可能需要编写较多的代码，这可能会增加复杂性。

经检验，完全使用 Eigen 库的 C++ 代码效率只有 MATLAB 的 20%。因此推荐尽量使用 for 循环进行手工优化。 在没有熟练运用 Eigen 库时盲目使用性价比不高。

3.2 矩阵叉乘试例

例子：3×3 矩阵与 $3 \times n$ 矩阵的叉乘：

```
void G_Oe::matrixMulti(double *L2, double *M2, double *N2, double *matrix, double *L, double *M, double *N, int n) //XYZ:1*3; LMN:1*n
{
    for (int i = 0; i < n; i++)
    {
            L2[i] = matrix[0]*L[i] + matrix[1]*M[i] + matrix[2]*N[i];
            M2[i] = matrix[3]*L[i] + matrix[4]*M[i] + matrix[5]*N[i];
            N2[i] = matrix[6]*L[i] + matrix[7]*M[i] + matrix[8]*N[i];
    }
```

}

实现矩阵的列相乘并形成新矩阵。

然鹅涉及到相关线性代数运算比如求行列式、特征值等复杂的运算过程仍需要使用 Eigen，同时也提供了使用 Eigen 库的示范版本。