开发约定与工作流规范 (Development Conventions & Workflow)

1. Git 提交规范 (Git Conventional Commits)

本项目强制采用 **Conventional Commits** 规范。提交信息必须清晰描述变更性质。

1.1 消息格式

<type>(<scope>): <subject>

[可选: body]

1.2 Type 定义

- **feat**: 新功能 (Feature)。
- **fix**: 修复 Bug (包括编译错误、逻辑错误、测试失败)。
- **test**: 添加或修改测试代码。
- **build**: 构建系统变更 (CMakeLists.txt, setup.py, 依赖库配置)。
- **refactor**: 代码重构 (不改变外部行为)。
- **docs**: 文档变更。
- **style**: 代码格式调整 (Clang-format, Black 等)。
- **chore**: 杂项 (配置更新、无关代码的变动)。

1.3 Scope 定义 (建议)

- cpp: C++ 核心算法层。
- py: Python 顶层逻辑/胶水层。
- cmake: 编译配置。
- geo: 几何计算模块。
- phys: 力学计算模块。
- input: 数据输入/预处理。

1.4 示例

- feat(cpp): implement ScoreCalculator class skeleton
- fix(build): resolve linker error for Eigen3
- test(py): add unit test for point cloud downsampling
- fix(geo): correct normal vector direction

2. 严格开发循环协议 (The Strict Development Loop)

为了保证代码质量和可追溯性，开发必须严格遵循以下 6步循环。严禁在该循环结束前跳过任何

提交环节。

步骤 1: Feature 开发 (Coding)

- 编写一个小粒度的子功能代码（例如：只定义头文件，或只实现一个函数）。
- **Action**: 立即提交。
- **Commit**: feat(<scope>): implement <sub-feature name>

步骤 2: 编译/运行检查 (Build Check)

- 执行编译 (C++) 或 运行 (Python)。
- **Case A**: 编译通过 -> 进入步骤 4。
- **Case B**: 编译失败/语法错误 -> 进入步骤 3。

步骤 3: 修复编译错误 (Fix Compilation)

- 修改代码直到编译通过。
- **Action**: 提交修复。
- **Commit**: fix(build): resolve compilation errors in <module>
- 注意：不要合并到上一个 *feat* 提交，保留修复痕迹。

步骤 4: 单元测试 (Unit Test)

- 编写或运行针对该功能的单元测试（Python pytest 或 C++ GTest）。
- **Action**: 提交测试代码（如果是新写的）。
- **Commit**: test(<scope>): add tests for <feature>

步骤 5: 修复逻辑错误 (Fix Logic)

- 运行测试。
- **Case A**: 测试通过 -> 循环结束，回到步骤 1 开发下一个 Feature。
- **Case B**: 测试失败 -> 修复代码逻辑。
- **Action**: 提交逻辑修复。
- **Commit**: fix(<scope>): fix logic error in <function> (test failed)

步骤 6: 循环结束 (Cycle Complete)

- 确认当前子功能 编译通过 且 测试通过。
- 开始下一个小任务。

3. 颗粒度控制 (Granularity Control)

3.1 核心原则

- **One Logic, One Commit**: 一个提交只做一件事。
- **Compile First**: 哪怕功能没写完，只要声明了类结构且能编译，就可以提交。
- **No Giant Commits**: 严禁一次性提交几百行未经验证的代码。

3.2 C++ 开发场景示例

1. 定义 ScoreCalculator.h -> feat(cpp): define ScoreCalculator header
2. 在 main.cpp 中 include 并实例化 -> 编译报错（找不到路径）-> 修复 CMake -> fix(build): add include directories
3. 实现 filterByGeoScore 的空函数 -> feat(cpp): add empty filterByGeoScore implementation
4. 编写具体逻辑 -> 编译报错（类型不匹配）-> 修复 -> fix(cpp): fix type mismatch in Eigen matrix
5. 编写 Python 测试调用该函数 -> test(py): add binding test
6. 测试发现返回值不对 -> 修复逻辑 -> fix(cpp): fix return value logic

4. 异常处理流程

4.1 遇到难以解决的编译错误

如果在 步骤 **3** 卡住超过 30 分钟：

1. 将当前“坏的代码”进行 WIP (Work In Progress) 提交，以免丢失进度。
    - chore(wip): save progress on <feature> (compilation failed)
2. 寻求解决方案。
3. 解决后，提交修复。
    - fix(build): resolve complex compiler error

4.2 依赖变更

如果在开发过程中需要引入新库（如修改 CMakeLists.txt 或 requirements.txt）：

- 必须单独提交一个 build 类型的 Commit。
- build(deps): add OpenMP dependency

5. 分支策略 (Branching)

- **main**: 仅包含通过所有测试的稳定代码。
- **dev**: 主开发分支。
- **feature/<name>**: 具体功能开发分支（推荐）。
    - 所有上述循环在 feature 分支进行。
    - 完成一个完整的 Issue（如 P2-1）后，通过 PR 合并回 dev。