NCS 用户手册

发行版本 *v2.5.0*

Huawei OptVerse LAB

内容

CHAPTER 1

天筹数值计算求解器简介

天筹数值计算求解器是一款由华为云自主研发的可移植、可扩展的科学计算工具库，主要用于在单机共享内存和分布式集群环境下高效求解 CAE（Computer Aided Engineering）仿真过程产生的底层数学问题，包括稀疏线性方程组、非线性方程组、与特征值等问题。它们的数学描述如下：

考虑线性方程组问题的一般形式：

$$Ax = b$$

其中 $A \in \mathbb{R}^{n \times n}$（$A \in \mathbb{C}^{n \times n}$）是一个稀疏方阵，表示线性方程组的系数矩阵；$b \in \mathbb{R}^{n}$（$b \in \mathbb{C}^{n}$）表示方程组的右端项；$x \in \mathbb{R}^{n}$（$x \in \mathbb{C}^{n}$）为待求未知向量；$\mathbb{R}$、$\mathbb{C}$ 分别对应实数集与复数集。

考虑非线性方程组问题的一般形式：

$$f\left(x\right) = 0$$

其中，$f : \mathbb{R}^{n} \to \mathbb{R}^{n}$（$f : \mathbb{C}^{n} \to \mathbb{C}^{n}$）表示目标方程组对应的非线性函数。

考虑（广义）特征值问题的一般形式：

$$Ax = \lambda Bx$$

其中，$B \in \mathbb{R}^{n \times n}$（$B \in \mathbb{C}^{n \times n}$）为 Hermitian 正定矩阵。当 B 为单位矩阵时，则称该问题为标准特征值问题。

针对上述问题，天筹数值计算求解器提供丰富的求解算法：

- 对于线性问题，天筹数值计算求解器支持使用直接法（如高斯消元法、LU 分解等），或 Krylov 迭代法 & 预处理子的算法组合进行求解。
- 对于非线性问题，天筹数值计算求解器提供基于 Newton-Raphson、Secant 弦截等思想的线搜索方法。
- 对于特征值问题，天筹数值计算求解器主要通过 Krylov-Schur 算法进行高效求解，支持计算大规模稀疏矩阵的最小、最大或多个内部特征值及其特征向量。

综上，天筹数值计算求解器是一个非常强大的科学计算库，涵盖了全面且高效的算法模块，可覆盖 CAE 领域（包括结构力学、计算流体、电磁仿真和动力学分析等）广泛的科学计算需求。

CHAPTER 2

安装说明

2.1 支持平台

数值计算求解引擎目前支持 Linux 系统平台，在以下系统测试可正常运行：

- **EularOS**: EularOS 2.9, EularOS 2.10
- **Ubuntu**: Ubuntu 18.04, Ubuntu 20.04
- **CentOS**: CentOS 8, CentOS 9
- 对于低版本 Linux 系统，需要满足 glibc >= 2.27，libstdc >= 6.0.26

2.2 安装包

数值计算求解引擎的安装包包含三个文件夹，分别是“bin”，“include”以及”lib”，

- bin 是可执行二进制文件库，可以忽略；
- include 中有四个头文件，分别对应直接法（NCSDirectSolver.h），迭代法（NCSIterativeSolver.h），特征值（NCSEigenSolver.h）以及非线性算法（NCSNonLinearSolver.h）的求解接口；
- lib 库中包含数值计算求解引擎的动态库 libNCSolver.so。

2.3 安装依赖

本工程依赖的三方库有：OpenBLAS-0.3.23、OpenMPI-4.1.4、MindSpore-Lite-1.8.1、Scotch-7.0.4、OpenSSL-1.1.1v、XGBoost-1.7.6

2.3.1 OpenBLAS

下载链接：

https://github.com/OpenMathLib/OpenBLAS/releases/download/v0.3.23/OpenBLAS-0.3.23.tar.gz

源码链接：

https://github.com/xianyi/OpenBLAS/tree/v0.3.23

数值计算求解引擎的使用依赖底层的 BLAS 库，具体地，对于本工程，依赖的 BLAS 库是 OpenBLAS，版本是 v0.3.23。下面对 OpenBLAS 的编译安装作简要说明：

- 将 OpenBLAS 下载到路径/user_dir/中并解压，其中 user_dir 是用户自定义的路径，进入解压后的文件夹中

```
cd /usr_dir/OpenBLAS-0.3.23/
```

- 编译并安装，其中 ${INSTALL_PREFIX_DIR}，是用户选择的安装路径，例如：/opt/oroas/3rds/OpenBLAS-0.3.23/

```
make -j DYNAMIC_ARCH=0 BINARY=64 USE_OPENMP=1
make -j DYNAMIC_ARCH=0 BINARY=64 USE_OPENMP=1 install PREFIX=${INSTALL_PREFIX_
↪DIR}
```

- 将 OpenBLAS 加入到系统库路径的环境变量中

```
export LD_LIBRARY_PATH=${INSTALL_PREFIX_DIR}/lib:$LD_LIBRARY_PATH
```

2.3.2 OpenMPI

下载链接：

https://download.open-mpi.org/release/open-mpi/v4.1/openmpi-4.1.4.tar.gz

源码链接：

https://github.com/open-mpi/ompi/tree/v4.1.4

下面对 OpenMPI 的编译安装作简要说明：

- 将 OpenMPI 下载到路径/user_dir/中并解压，其中 user_dir 是用户自定义的路径，进入解压后的文件夹中

```
cd /usr_dir/openmpi-4.1.4/
```

- 编译并安装，其中 ${INSTALL_PREFIX_DIR}，是用户选择的安装路径，例如：/opt/oroas/3rds/openmpi-4.1.4/

```
./configure --prefix=${INSTALL_PREFIX_DIR}
make all -j
make install
```

- 将 OpenMPI 加入到系统库路径的环境变量中

```
export LD_LIBRARY_PATH=${INSTALL_PREFIX_DIR}/lib:$LD_LIBRARY_PATH
```

2.3.3 MindSpore-Lite

主页：

https://www.mindspore.cn/

下载链接：

https://ms-release.obs.cn-north-4.myhuaweicloud.com/1.8.1/MindSpore/lite/release/linux/x86_64/mindspore-lite-1.8.1-linux-x64.tar.gz

下面对 MindSpore-Lite 的编译安装作简要说明：

- 将 MindSpore-Lite 下载到路径/install_dir/中并解压，其中/install_dir/是用户选择的安装路径，例如：/opt/oroas/3rds/
- 将 MindSpore-Lite 加入到系统库路径的环境变量中

```
export LD_LIBRARY_PATH=/install_dir/mindspore-lite-1.8.1-linux-x64/runtime/lib:
↪$LD_LIBRARY_PATH
```

2.3.4 Scotch

源码链接：

https://gitlab.inria.fr/scotch/scotch/-/tree/v7.0.4?ref_type=tags

下载链接：

https://gitlab.inria.fr/scotch/scotch/-/archive/v7.0.4/scotch-v7.0.4.tar.gz

下面对 Scotch 的编译安装作简要说明：

- 将 Scotch 下载到路径/user_dir/中并解压，其中 user_dir 是用户自定义的路径，进入解压后的文件夹中

```
cd  /usr_dir/scotch-v7.0.4/
```

- 编译并安装，其中 ${INSTALL_PREFIX_DIR}，是用户选择的安装路径，例如：/opt/oroas/3rds/scotch-7.0.4/

```
mkdir cmake-build-release
cd cmake-build-release
cmake -DCMAKE_BUILD_TYPE=RELEASE -DCMAKE_INSTALL_PREFIX=${INSTALL_PREFIX_DIR} -
↪DBUILD_SHARED_LIBS=ON -DCMAKE_POSITION_INDEPENDENT_CODE=ON -DCMAKE_EXPORT_
↪COMPILE_COMMANDS=ON -DINSTALL_METIS_HEADERS:BOOL=ON -DINTSIZE=64 -DBUILD_
↪PTSCOTCH=OFF ..
cmake --build . --  -j
cmake --install .
```

- 将 Scotch 加入到系统库路径的环境变量中

```
export LD_LIBRARY_PATH=${INSTALL_PREFIX_DIR}/lib:$LD_LIBRARY_PATH
```

2.3.5 OpenSSL

下载链接：

https://github.com/openssl/openssl/releases/download/OpenSSL_1_1_1v/openssl-1.1.1v.tar.gz

源码链接：

https://github.com/openssl/openssl/tree/OpenSSL_1_1_1v

下面对 OpenSSL 的编译安装作简要说明：

- 将 OpenSSL 下载到路径/user_dir/中并解压，其中 user_dir 是用户自定义的路径，进入解压后的文件夹中

```
cd  /usr_dir/openssl-1.1.1v
```

- 编译并安装，其中 ${INSTALL_PREFIX_DIR}，是用户选择的安装路径，例如：/opt/oroas/3rds/openssl-1.1.1v/

```
./config -no-asm -no-zlib -fPIC --prefix=${INSTALL_PREFIX_DIR}
make -j
make install
```

- 将 Scotch 加入到系统库路径的环境变量中

```
export LD_LIBRARY_PATH=${INSTALL_PREFIX_DIR}/lib:$LD_LIBRARY_PATH
```

2.3.6 XGBoost

下载链接：

https://github.com/dmlc/xgboost/releases/download/v1.7.6/xgboost.tar.gz

源码链接：

https://github.com/dmlc/xgboost/tree/v1.7.6

XGBoost 依赖 dmlc-core

下载链接：

https://github.com/dmlc/dmlc-core/archive/refs/tags/v0.4.tar.gz

源码链接：

https://github.com/dmlc/dmlc-core/tree/v0.4

下面对 XGBoost 的编译安装作简要说明：

- 将 XGBoost 下载到路径/user_dir/中并解压，其中 user_dir 是用户自定义的路径，进入解压后的文件夹中

```
cd  /usr_dir/xgboost
```

- ** 如果 xgboost 目录下的 dmlc-core 非空，可跳过这一步。** 将 dmlc-core 下载到路径/user_dir/并解压，其中 user_dir 是用户自定义的路径。将 dmlc-core 里的文件复制到 xgboot 目录下的 dmlc-core。

```
mkdir -p ./dmlc-core
cp -r /usr_dir/dmlc-core-0.4/* ./dmlc-core/
```

- 编译并安装，其中 ${INSTALL_PREFIX_DIR}，是用户选择的安装路径，例如：/opt/oroas/3rds/xgboost-1.7.6/

```
mkdir cmake-build-release
cd cmake-build-release
cmake -DCMAKE_BUILD_TYPE=RELEASE -DCMAKE_INSTALL_PREFIX=${INSTALL_PREFIX_DIR} -
↪DBUILD_SHARED_LIBS=ON -DCMAKE_POSITION_INDEPENDENT_CODE=ON ..
cmake --build . --  -j
cmake --install .
```

- 将 XGBoost 加入到系统库路径的环境变量中

```
export LD_LIBRARY_PATH=${INSTALL_PREFIX_DIR}/lib64:$LD_LIBRARY_PATH
```

注：不同的安装环境下，三方库的文件夹名可能不同，可能为 lib 或 lib64，在加入到系统库路径时请注意甄别。

此外，建议用户的编译器版本为 gcc-10.3.0，若 EulerOS V2.0SP5 通过 yum 工具安装 gcc 版本不符合使用要求可以通过源码安装指定版本 gcc。具体操作步骤如下。

1. 配置 EulerOS V2.0SP5 的源。
   - 在/etc/yum.repos.d/目录下，创建文件 EulerOS.repo，配置以下内容。[base] name=EulerOS-2.0SP5 base baseurl=http://repo.huaweicloud.com/euler/2.5/os/x86_64/ enabled=1 gpgcheck=1 gpgkey=http://repo.huaweicloud.com/euler/2.5/os/RPM-GPG-KEY-EulerOS
   - 执行 yum clean all 清除原有 yum 缓存。
   - 执行 yum makecache 生成新的缓存。
2. 执行命令 yum install gcc 和 yum install gcc-c++ 安装编译器。执行命令 gcc -v 查看版本信息是否符合需求。
3. 源码编译安装 GCC
   - 进入官网 https://gcc.gnu.org/pub/gcc/releases/ 下载符合需求的 gcc 版本。
   - 下载依赖包 gmp-6.1.0.tar.bz2、mpfr-3.1.4.tar.bz2、mpc-1.0.3.tar.gz、isl-0.16.1.tar.bz2
   - 下载完毕后；将四个依赖包解压放置在 gcc-x.x.x 路径下, 注意解压后名称不能带版本号。
   - 进入到 gcc-x.x.x 路径下，执行 mkdir gcc-build-x.x.x; cd gcc-build-x.x.x;
   - 执行../configure –enable-checking=release –enable-languages=c,c++,fortran –disable-multilib；等待配置完毕 (configure 的过程是用来生成 Makefile 文件的，因为源码提供的 makefile 文件包含所有特性；使用 configure+ 参数，可以指定安装的目录，主机、支持的语言特性等等)；
   - 执行 make -j8 编译，等待一段时间。
   - 编译完成后卸载低版本 yum remove gcc，yum remove g++。
   - 执行 make install, 进行安装。
   - gcc -v 再次查看版本信息

2.4 简单示例

下面是一个利用 cmake 构建工程的简单示例，文件只包含一个“main.cpp”以及一个对应的“CmakeLists.txt”，这两个文件对应的内容分别为：

```
# CmakeLists.txt
cmake_minimum_required(VERSION 3.16)
project(interface-test)

set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -pthread -fPIC -fopenmp")
SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pthread -fPIC -fopenmp")
SET(CMAKE_CXX_FLAGS_RELEASE "$ENV{CXXFLAGS} -O3 -Wall")
```

(续下页)

(接上页)

```
SET(CMAKE_CXX_FLAGS_DEBUG "$ENV{CXXFLAGS} -O0 -Wall -g2 -ggdb")

set(NCS_DIR /solver_install_dir/NCSsolver)    # /solver_install_dir/NCSsolver is␣
↪the where the solver located
set(NCS_INC_DIR ${NCS_DIR}/include/)
set(NCS_LIB_DIR ${NCS_DIR}/lib/libNCSolver.so)

include_directories(${NCS_INC_DIR})
link_directories(${NCS_LIB_DIR})

add_executable(interface-test main.cpp)
target_link_libraries(interface-test PUBLIC -Wl,--start-group ${NCS_LIB_DIR} -Wl,--
↪end-group -ldl)
```

```
// main.cpp

#include <iostream>
#include <vector>

#include <NCSDirectSolver.h>

using namespace std;
using namespace NCS::DIRECT;

int main(int argc, char* argv[])
{
    // 求解矩阵及右端项示例
    //      | 4 2 8  |        | 2  3 |
    // A = |   2 4  |   b =  | 3  4 |
    //      |     25 |        | 1  5 |
    int n = 3;
    std::vector<int> ptr = {0, 3, 5, 6};
    std::vector<int> ids = {0, 1, 2, 1, 2, 2};
    std::vector<double> matValues = {4, 2, 8, 2, 4, 25};
    int nrhs = 2; // 2个右端项
    std::vector<double> rhsValues = {2, 3, 1, 3, 4, 5};

    DirectOptions options;
    options.solveMethod = 0;
    options.matType = 1;
    auto solver = CreateDirectSolver<double>(options);

    // 使用求解方式1求解，使用默认参数求解
    int retCode = solver->Solve(n, ptr.data(), ids.data(), matValues.data(), nrhs,␣
↪rhsValues.data());

    // 输出求解结果
    std::cout << "求解返回码为: " << retCode << std::endl;
    for (double rhsValue : rhsValues) {
        std::cout << rhsValue << "\t";
    }
    std::cout << std::endl;
    return 0;
}
```

针对使用 visual studio 的用户，“CmakeLists.txt”需要不同的配置，这里给出一个构建 mpi 并行计算的示例，文件包含一个“main.cpp”以及一个对应的“CmakeLists.txt”，这两个文件对应的内容分别为：

```
# CmakeLists.txt
cmake_minimum_required(VERSION 3.16)
project(vs_interface-test)
```

(续下页)

(接上页)

```
# 设置 C++17 标准
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
# 配置生成的 Visual Studio 项目为 Release 和 Debug
set(CMAKE_CONFIGURATION_TYPES "Release;Debug" CACHE STRING "" FORCE)
# 设置 MS MPI 环境（假设已安装MSMPI，并且路径正确）
find_package(MPI REQUIRED)
# 设置 MS MPI 库路径（替换为实际的MSMPI路径）
set(MSMPI_LIB_DIR "C:/Program Files (x86)/Microsoft SDKs/MPI/Lib")  #␣
↪替换为实际路径
set(MSMPI_INCLUDE_DIR "C:/Program Files (x86)/Microsoft SDKs/MPI/Include")  #␣
↪替换为实际路径
# 包含 MS MPI 头文件路径
include_directories(${MSMPI_INCLUDE_DIR})
# 包含 NCSolver 头文件路径
set(NCSOLVER_INCLUDE_DIR "D:/windows_build/vs_call/vs_call_cmake/include")  #␣
↪替换为实际路径
include_directories(${NCSOLVER_INCLUDE_DIR})
# 添加 MS MPI 库路径
link_directories(${MSMPI_LIB_DIR})
# 添加可执行文件
add_executable(vs_call_test main.cpp)
# 链接 MS MPI 库
target_link_libraries(vs_call_test PRIVATE MPI::MPI_CXX)
# 设置 libNCSolver.dll.a 所在的目录
set(NCSolver_LIB_DIR "${CMAKE_CURRENT_SOURCE_DIR}/lib")  # 假设 libNCSolver.dll.a␣
↪在 lib 文件夹中
# 将 libNCSolver.dll.a 作为链接库
target_link_libraries(vs_call_test PRIVATE "${NCSolver_LIB_DIR}/libNCSolver.dll.a")
# 设置运行时 DLL 文件的查找路径
# 使用 GLOB 获取所有 .dll 文件
file(GLOB NCSolver_DLL_FILES "${NCSolver_LIB_DIR}/*.dll")
# 在构建后将所有 DLL 文件复制到可执行文件目录
add_custom_command(TARGET vs_call_test POST_BUILD
        COMMAND ${CMAKE_COMMAND} -E copy_if_different
        ${NCSolver_DLL_FILES}
        $<TARGET_FILE_DIR:vs_call_test>)
# 设置调试和发布配置下的输出目录
set_target_properties(vs_call_test PROPERTIES
        RUNTIME_OUTPUT_DIRECTORY_DEBUG "${CMAKE_BINARY_DIR}/Debug"
        RUNTIME_OUTPUT_DIRECTORY_RELEASE "${CMAKE_BINARY_DIR}/Release"
)
# 强制生成 x64 架构的 Visual Studio 工程
if (MSVC)
    set(CMAKE_GENERATOR_PLATFORM X64)
endif ()
# 包含调试/发布的配置文件
if(CMAKE_BUILD_TYPE STREQUAL "Release")
    target_compile_definitions(vs_call_test PRIVATE NDEBUG)
endif()
```

```
// main.cpp

#include <iostream>
#include <string>
#include <assert.h>
#include <mpi.h>
#include <NCSIterativeSolverWrapper.h>

using std::string;
using namespace NCS::KSP;
```

(续下页)

(接上页)

```
using namespace NCS::AI4SOLVER;

int main(int argc, char *argv[])
{
        /* ------------------------- 初始化MPI ------------------------- */
        MPI_Init(&argc, &argv);
        int rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        int worldSize;
        MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
        assert(worldSize == 2);

        /* -------------------- 设置系数矩阵及右端项---------------------- */
        int dim;
        std::vector<int> rowPtr, colIdx;
        std::vector<double> data, rhsValues, iniSol, sol;
        /*
        | 2 1 0 0 0|   |1|   |4 |
        | 1 2 1 0 0| * |2| = |8 |   rank 0
        | 0 1 2 1 0|   |3|   |12|
        | 0 0 1 2 1|   |4|   |16|
        -----------------------------
        | 0 0 0 1 2|   |5|   |14|   rank 1

         求解报错
        */
        if (rank == 0) {
                dim = 4;
                rowPtr = { 0, 2, 5, 8, 11 };
                colIdx = { 0, 1, 0, 1, 2, 1, 2, 3, 2, 3, 4 };
                data = { 2., 1., 1., 2., 1., 1., 2., 1., 1., 2., 1. };
                rhsValues = { 4., 8., 12., 16 };
                iniSol = { 0., 0., 0., 0. };
        }
        else {
                dim = 1;
                rowPtr = { 0, 2 };
                colIdx = { 3, 4 };
                data = { 1., 2. };
                rhsValues = { 14. };
                iniSol = { 0. };
        }
        sol.resize(dim);

        /* ----------------------- 设置求解参数------------------------- */
        // ksp&pc 设置
        auto dataType = DataType::DOUBLE;
        NCSIterativeSolverWrapper* linearSolverWrapper = LinearSolver_
↪new(dataType);
        LinearSolverSetOption(dataType, linearSolverWrapper, "--ksp_type", "cg");
        LinearSolverSetOption(dataType, linearSolverWrapper, "--pc_type", "none");
        LinearSolverSetOption(dataType, linearSolverWrapper, "--ksp_max_iteration",
↪ "10000");
        LinearSolverSetOption(dataType, linearSolverWrapper, "--pc_jacobi_type",
↪"diagonal");
        // 分布式求解设置参数
        LinearSolverSetOption(dataType, linearSolverWrapper, "--global_dimension",
↪"5");     // global dimension, mandatory para
        LinearSolverSetOption(dataType, linearSolverWrapper, "--is_rank0_input_only
↪", "false");   // // distributed matrix input, mandatory para
```

(续下页)

(接上页)

```
        if (rank == 0) {
                // 进程0负责[0,4)行
                LinearSolverSetOption(dataType, linearSolverWrapper, "--cur_start_
↪index", "0");    // start row index,mandatory para
        }
        else {
                // 进程1负责[4,5)行
                LinearSolverSetOption(dataType, linearSolverWrapper, "--cur_start_
↪index", "4");   // start row index,mandatory para
        }

        auto status = LinearSolverSolve_double(dataType, linearSolverWrapper,
                data.data(), colIdx.data(), rowPtr.data(), dim,
                rhsValues.data(), iniSol.data(), sol.data());

        /* ---------------------------- 输出求解结果----------------------------␣
↪*/
        if (rank == 0) {
                int iterNum = LinearSolverGetIter(dataType, linearSolverWrapper);
                auto res = LinearSolverGet(dataType, linearSolverWrapper,␣
↪Get::GetTrueResNorm);
                auto res2 = LinearSolverGet(dataType, linearSolverWrapper,␣
↪Get::GetResNorm);

                std::cout << "status:" << status << std::endl;
                std::cout << "iterNum:" << iterNum << std::endl;
                std::cout << "TrueResNorm:" << res2 << std::endl;
                std::cout << "ResNorm:" << res << std::endl;
        }

        std::cout << "result from rank " << rank << "-->" <<"\t";
        for (auto s : sol) {
                std::cout << s << "\t";
        }
        std::cout << std::endl;

        LinearSolver_delete(linearSolverWrapper);
        /* ---------------------------- 结束MPI ---------------------------- */
        MPI_Finalize();

        return 0;
}
```

编译完成之后生成的 exe 可以直接用 mpiexec -np 2 命令运行，注意保证生成的 exe 能找到所依赖的动态链接库。

CHAPTER 3

C++ API 手册

本章介绍如何调用天筹数值计算求解器各模块的 C++ 接口。

3.1 线性直接法求解接口说明

3.1.1 直接法接口整体说明

线性直接法对用户提供的接口封装在 NCSDirectSolver.h 中：

```
template <typename Scalar, typename Idx = NCSInt>
class NCSDirectSolver
{
public:
    // 端到端求解接口
    virtual RetCode Solve(Idx n, Idx const* indptr, Idx const* indices, Scalar␣
↪const* values, int nrhs, Scalar* rhs) = 0;
    // 符号分解接口
    virtual RetCode Analyze(Idx n, Idx const* indptr, Idx const* indices, Scalar␣
↪const* values) = 0;
    // 数值分解接口
    virtual RetCode Factorize(Idx n, Idx const* indptr, Idx const* indices, Scalar␣
↪const* values) = 0;
    // 回代求解接口
    virtual RetCode BackSolve(int nrhs, Scalar* rhs) = 0;
};

// 根据矩阵类型及求解方法，创建相应solver的工厂方法
template <typename Scalar, typename Idx = NCSInt>
std::unique_ptr<NCSDirectSolver<Scalar>> CreateDirectSolver(DirectOptions const&␣
↪options);
```

模板参数 Scalar 表示浮点数的类型，目前支持如下类型：

- `float`：实数单精度
- `double`：实数双精度
- `std::complex<float>`：复数单精度

- `std::complex<double>`：复数双精度

模板参数 Idx 表示索引的类型，默认为 NCSInt(int) 目前支持如下类型：

- `int`：32 位整数
- `long long int`：64 位整数

适用于 visual studio 调用的 C 接口封装在 NCSDirectSolverWrapper.h 中, 目前只支持 `int` 类型整数：

```
enum class DataType : int32_t { DOUBLE = 0, FLOAT = 1, COMPLEX_DOUBLE = 2, COMPLEX_
↪FLOAT = 3, };
enum class SolverOP : int32_t { Solve = 0, Analyze = 1, Factorize = 2, BackSolve =␣
↪3,};
// 生成直接求解器实例
DLLEXPORT DirectSolverWrapper* DirectSolver_new(cDirectOptions const& coptions);
// 调用double求解器操作Solve、Analyze、Factorize、BackSolve
DLLEXPORT RetCode DirectSolver_double(SolverOP solverOP, DirectSolverWrapper*␣
↪wrapper, int n, int const* indptr,
                          int const* indices, double const* values, int nrhs,␣
↪double* rhs);
// 调用float求解器操作
DLLEXPORT RetCode DirectSolver_float(SolverOP solverOP,DirectSolverWrapper*␣
↪wrapper, int n, int const* indptr,
                    int const* indices, float const* values, int nrhs, float*␣
↪rhs);
// 调用复数double求解器操作
DLLEXPORT RetCode DirectSolver_complex_double(SolverOP solverOP,
↪DirectSolverWrapper* wrapper, int n, int const* indptr,
                   int const* indices, cComplex_double* values, int nrhs, cComplex_
↪double* rhs);
// 调用复数float求解器操作
DLLEXPORT RetCode DirectSolver_complex_float(SolverOP solverOP,
↪DirectSolverWrapper* wrapper, int n, int const* indptr,
                   int const* indices, cComplex_float* values, int nrhs, cComplex_
↪float* rhs);
// 求解状态接口
DLLEXPORT Stats DirectSolver_GetStats(DirectSolverWrapper* wrapper, DataType␣
↪dataType);
// 删除直接求解器实例
DLLEXPORT void DirectSolver_delete(DirectSolverWrapper* wrapper);
```

3.1.2 参数配置接口 DirectOptions

DirectOptions 用于设置求解参数，定义如下：

```
struct DirectOptions {
    MatType matType;                            // 矩阵类型 【必须】
    Method method{Method::SUPERNODAL};          // 求解方法 【必须】
    MatrixStorageType matStorageType{MatrixStorageType::CENTRALIZED_CSR};  //␣
↪矩阵存储格式
    int numThreads{0};                          // 线程数量
    bool weightedMatching{true};                // 开启匹配算法提升精度
    bool stableOrdering{true};                  // 开启提升整齐求解稳定性
    int parallelMode{0};                        //␣
↪并行模式，仅适用于共享内存版的非对称矩阵求解
    LogLevel logLevel{LogLevel::WARN};          // 日志等级
    std::string logFile;                        // 日志文件

    /* MPI模式下的参数 */
    int localBeginId{};  // 当前进程矩阵的起始行号 【MPI模式必须】
    int localEndId{};    // 当前进程矩阵的结束行号 【MPI模式必须】
```

(续下页)

(接上页)

```
    MPI_Comm comm{MPI_COMM_WORLD};  // MPI通讯子
};
```

visual studio 中 cDirectOptions 用于设置求解参数，定义如下：

```
struct cDirectOptions {
  MatType matType{};
  Method method{Method::SUPERNODAL};
  MatrixStorageType matStorageType{MatrixStorageType::CENTRALIZED_CSR};
  int numThreads{0};
  bool weightedMatching{true};
  bool stableOrdering{true};
  int parallelMode{0};
  LogLevel logLevel{LogLevel::WARN};
  char logFile[200] = "";    // 该处改为char[]类型
  /* params that only used in mpi run */
  int localBeginId{};
  int localEndId{};
  MPI_Comm comm{MPI_COMM_WORLD};
  DataType dataType{DataType::DOUBLE};   // 新添加
}
```

注意事项：

- **标注为“【必须】”的参数无默认值，用户必须指定。**
- **标注为“【MPI 模式必须】“的参数，在 MPI 模式下，用户必须指定。**

各参数的定义和说明详见如下子章节：

矩阵类型的参数 *matType* 【必须】

参数 matType 为**必须设置的参数**，用于指定矩阵的类型。

- 数据类型

```
enum class MatType : int32_t {
    REAL_GENERAL                          = 0,  // 一般实矩阵
    REAL_SYMMETRIC_POSITIVE_DEFINITE      = 1,  // 实对称正定矩阵
    REAL_SYMMETRIC_INDEFINITE             = 2,  // 实对称不定矩阵
    COMPLEX_GENERAL                       = 3,  // 一般复矩阵
    COMPLEX_SYMMETRIC                     = 4,  // 复对称矩阵
    COMPLEX_HERMITIAN_POSITIVE_DEFINITE = 5,  // 复共轭对称正定矩阵
    COMPLEX_HERMITIAN_INDEFINITE          = 6,  // 复共轭对称不定矩阵
};
```

- 取值范围
  - MatType::REAL_GENERAL：一般实矩阵
  - MatType::REAL_SYMMETRIC_POSITIVE_DEFINITE：实对称正定矩阵
  - MatType::REAL_SYMMETRIC_INDEFINITE：实对称不定矩阵
  - MatType::COMPLEX_GENERAL：一般复矩阵
  - MatType::COMPLEX_SYMMETRIC：复对称矩阵
  - MatType::COMPLEX_HERMITIAN_POSITIVE_DEFINITE：复共轭对称正定矩阵
  - MatType::COMPLEX_HERMITIAN_INDEFINITE：复共轭对称不定矩阵

  默认值：无。
- 注意事项

- 当指定矩阵为是实对称、复对称或复共轭对称时，只需要输入矩阵的上三角部分（详见下文’矩阵输入格式说明’内容）。

求解方法的参数 *method* 【必须】

参数 `method` 为**必须设置的参数**，用于指定求解算法。目前提供超节点法和分布式超节点法两种算法。

- 数据类型

```
enum class Method : int32_t {
    SUPERNODAL       = 0,  // 超节点法
    DIST_SUPERNODAL  = 1   // 分布式超节点法
};
```

- 取值范围
  - `Method::SUPERNODAL`：超节点法，单线程和多线程版本
  - `Method::DIST_SUPERNODAL`：分布式超节点法，即 MPI 版本

  默认值：无。
- 注意事项：
  - 在多线程环境下，只能使用 `Method::SUPERNODAL`。在 MPI 环境下，只能使用 `Method::DIST_SUPERNODAL`。
  - 若 `method` 参数指定为 `Method::SUPERNODAL`，则矩阵存储类型只能为 `MatrixStorageType::CENTRALIZED_CSR`。

矩阵存储类型 *matStorageType*

参数 `matStorageType` 用于指定矩阵的存储类型。

- 数据类型

```
enum class MatrixStorageType : int32_t {
    CENTRALIZED_CSR = 0,  // 集中式压缩行主序（CSR）格式
    DISTRIBUTED_CSR = 1   // 分布式压缩行主序（CSR）格式
};
```

- 取值范围
  - `MatrixStorageType::CENTRALIZED_CSR`：集中式压缩行主序格式。可用于多线程和分布式（MPI）版。其中，在 MPI 模式下，主进程存储全部矩阵。详见后文使用示例。
  - `MatrixStorageType::DISTRIBUTED_CSR`：分布式压缩行主序格式。只能用于 MPI 模式，每个进程只存储部分矩阵。需指定当前进程负责的起始行号（`localBeginId`）和结束行号（`localEndId`）。后文会给出具体解释以及分布式输入示例。

  默认值：`MatrixStorageType::CENTRALIZED_CSR`
- 注意事项：
  - 系数矩阵、右端项矩阵、解矩阵的存储行号一致。例如，若存储格式为 `MatrixStorageType::DISTRIBUTED_CSR`，进程 1 负责 $[10, 20)$ 行，则该进程只输入系数矩阵、右端项矩阵的 $[10, 20)$ 行，输出解矩阵的 $[10, 20)$ 行。
  - 若矩阵存储类型为 `MatrixStorageType::DISTRIBUTED_CSR`，则求解算法 `method` 只能指定为 `Method::DIST_SUPERNODAL`。
  - 若矩阵存储类型为 `MatrixStorageType::DISTRIBUTED_CSR`，需指定当前进程负责的起始行号（`localBeginId`）和结束行号（`localEndId`）。

线程数量参数 *numThreads*

参数 `numThreads` 用于指定直接法使用的 OpenMP 的线程数量。

- 数据类型 int
- 取值范围
  - ≤ 0：使用环境变量 `OMP_NUM_THREADS` 的线程数。若不指定环境变量 `OMP_NUM_THREADS`，则使用机器的逻辑核数。
  - > 0：用户指定线程数

  默认值：0
- 注意事项
  - 参数 `numThreads` 的优先级高于环境变量 `OMP_NUM_THREADS`。

匹配算法参数 *weightedMatching*

参数 `weightedMatching` 用于指定是否使用 weighted matching 算法对矩阵进行重排，从而提高矩阵分解的精度。

- 数据类型 bool
- 取值范围
  - true：使用 weighted matching 算法对矩阵进行重排
  - false：不使用

  默认值：true
- 注意事项：
  - 当前该参数只对非对称矩阵生效。
  - 使用该参数可能会导致求解性能降低。

排序算法参数 *stableOrdering*

参数 `stableOrdering` 设为 true 可使 ordering 结果稳定，提升整体求解稳定性。

- 数据类型 bool
- 取值范围
  - true：使用确定性重排
  - false：不使用

  默认值：true

并行模式参数 *parallelMode*

参数 `parallelMode` 用于指定 Factorize 阶段的并行模式，目前仅支持共享内存版的非对称矩阵求解。

- 数据类型 int
- 取值范围
  - 0：使用 left-looking
  - 1：使用 right-looking

  默认值：0
- 注意事项：

- 设置为 1 可能会提升求解性能。

日志级别的参数 *logLevel*

参数 `logLevel` 用于指定日志级别。

- 数据类型

```
enum class LogLevel : int32_t {
    OFF = -1,    // 不输出日志
    ERROR = 0,   // 只输出报错日志
    WARN = 1,    // 输出报错及告警日志
    INFO = 2,    // 输出报错、告警、信息日志
    MIN = OFF,
    MAX = INFO
};
```

- 取值范围
  - `LogLevel::OFF`：关闭日志
  - `LogLevel::ERROR`：只输出报错日志
  - `LogLevel::WARN`：输出报错及告警日志
  - `LogLevel::INFO`：输出报错、告警、信息日志

  默认值：`LogLevel::WARN`

日志输出文件的参数 *logFile*

参数 `logFile` 用于指定日志文件。

- 数据类型 std::string
- 取值范围

  默认值：`""`
- 注意事项
  - 默认不生成日志文件，日志只输出到屏幕。若需将日志输出到文件，则需指定具体的日志文件，此时日志将同时输出到文件及屏幕。示例如下：

```
DirectOptions options;
options.logFile = "/tmp/direct.log";
```

当前进程矩阵的起始行号 *localBeginId* 【MPI 模式必须】

参数 `localBeginId` 为 **MPI 模式必须设置的参数**，用于指定 MPI 模式下当前进程的系数矩阵、右端项矩阵、解矩阵的起始行号。（详见’矩阵输入格式说明’内容，可参考后文的分布式输入示例）

- 数据类型 int
- 取值范围
  - ≥ 0：当前进程的矩阵的起始行号。

  默认值：无
- 注意事项
  - 只有在 MPI 模式且采用分布式矩阵输入的情况下，该参数才生效，即：参数 `method` 需设置为 `Method::DIST_SUPERNODAL`，且参数 `matStorageType` 需设置为 `MatrixStorageType::DISTRIBUTED_CSR`。

当前进程矩阵的结束行号 *localEndId* 【MPI 模式必须】

参数 `localBeginId` 为 **MPI 模式必须设置的参数**，用于指定 MPI 模式下当前进程的系数矩阵、右端项矩阵、解矩阵的结束行号。（详见’矩阵输入格式说明’内容，可参考后文的分布式输入示例）

- 数据类型 int
- 取值范围
  - ≥ 0：当前进程的矩阵的结束行号。

  默认值：无
- 注意事项
  - 只有在 MPI 模式且采用分布式矩阵输入的情况下，该参数才生效，即：参数 `method` 需设置为 `Method::DIST_SUPERNODAL`，且参数 `matStorageType` 需设置为 `MatrixStorageType::DISTRIBUTED_CSR`。
  - `localEndId` 需大于 `localBeginId`。

MPI 通讯子 *comm*

参数 `comm` 用于指定 MPI 模式下的 MPI 通讯子

- 数据类型 `MPI_Comm`
- 取值范围 **默认值**：`MPI_COMM_WORLD`

3.1.3 求解接口说明

矩阵输入格式说明

一共有两种矩阵输入格式，即 `MatStorageType::CENTRALIZED_CSR` 和 `MatStorageType::DISTRIBUTED_CSR`，均为行主序的存储格式。其中，`MatStorageType::CENTRALIZED_CSR` 表示所有矩阵集中在主进程中，而 `MatStorageType::DISTRIBUTED_CSR` 表示每个进程只存储部分矩阵。所以，多线程运行的情况下，只能选用 `MatStorageType::CENTRALIZED_CSR`，而在分布式运行（MPI 模式）下，既可以选用 `MatStorageType::CENTRALIZED_CSR`，也可以选用 `MatStorageType::DISTRIBUTED_CSR`。

集中式矩阵输入格式（`MatStorageType::CENTRALIZED_CSR`）

该求解接口直接使用裸指针传参。**注意：输入的矩阵以行主序（CSR）的形式存储，且当矩阵为对称或共轭矩阵时，只存储其上三角部分**。

系数矩阵 A 或 A 的上三角部分

参数	类型	含义
`n`	输入参数	系数矩阵矩阵行数
`indptr`	输入参数	3 数组 CSR 的行起始数组，长度为 n + 1，且 indptr[n] 必须正确表示非零元数量（以下称 nnz）。
`indices`	输入参数	3 数组 CSR 的列下标数组，长度为 nnz
`values`	输入参数	3 数组 CSR 的值数组，长度为 nnz

稠密右端项矩阵 X 或 B

参数	类型	含义
nrhs	输入参数	右端项的列数
rhs	输入输出参数	n×nrhs 的稠密右端项矩阵，列主序存储，leading dimension 必须为 n。调用前：右端项矩阵 B；调用后：解矩阵 X

分布式矩阵输入格式（MatStorageType::DISTRIBUTED_CSR）

在 MPI 分布式环境下，若每个进程只输入部分矩阵，则需指定矩阵的输入格式为 MatrixStorageType::DISTRIBUTED_CSR。在这种格式下，矩阵被按行切分为多个连续的子矩阵，每个子矩阵属于一个 MPI 进程。每个进程的子矩阵的起始行号和结束行号分别由参数 localBeginId 和 localEndId 指定。每个 MPI 进程中，子矩阵的存储格式为行主序格式（CSR）。

注意事项：

- 右端项矩阵的输入范围、解矩阵的输出范围与系数矩阵的输入范围一致。例如，若进程 1 负责输入系数矩阵的 $[10, 20)$ 行，则进程 1 也负责输入右端项矩阵的 $[10, 20)$ 行，输出解矩阵的 $[10, 20)$ 行。
- 相邻子矩阵间不能重叠。例如，若进程 0 负责 $[0, 10)$ 行，进程 1 负责 $[8, 18)$ 行，即进程 0 和进程 1 均输入矩阵的第 8 行和第 9 行，则程序将报错。

如下为分布式矩阵输入的 1 个例子：

对于对称矩阵 A：$A = \begin{bmatrix} 6 & -1 & * & -3 & * \\ -1 & 5 & * & * & * \\ * & * & 11 & 5 & 4 \\ -3 & * & 5 & 10 & * \\ * & * & 4 & * & 5 \end{bmatrix}$

以 MatrixStorageType::DISTRIBUTED_CSR 格式将 A 存储于 2 个 MPI 进程，其中，进程 0 存储行 $[0, 3)$，进程 1 存储行 $[3, 5)$。

则对于进程 0，矩阵 A_0 为：$A_0 = \begin{bmatrix} 6 & -1 & * & -3 & * \\ -1 & 5 & * & * & * \\ * & * & 11 & 5 & 4 \end{bmatrix}$

则进程 0 输入系数矩阵对应的参数如下（注意：对称矩阵只需要输入上三角部分）：

入参	取值
n	5
indptr	(0, 3, 4, 7)
indices	(0, 1, 3, 1, 2, 3, 4)
values	(6, -1, -3, 5, 11, 5, 4)

对于进程 1，矩阵 A_1 为：$A_1 = \begin{bmatrix} -3 & * & 5 & 10 & * \\ * & * & 4 & * & 5 \end{bmatrix}$

则进程 1 输入系数矩阵对应的参数如下（注意：对称矩阵只需要输入上三角部分）：

入参	取值
n	5
indptr	(0, 1, 2)
indices	(3, 4)
values	(10, 5)

返回值说明

可能的返回值及含义有：

```
enum class RetCode : int32_t
{
    SUCCESS = 0,                    // 求解成功
    STRUCT_ZERO_DIAG = 1,           //␣
↪求解失败：指定了正定矩阵，但输入矩阵结构上有零对角元
    NUMERIC_ZERO_DIAG = 2,          //␣
↪求解失败：指定了正定矩阵，但输入矩阵数值上有零对角元
    NONZERO_IMAGINARY_DIAG = 3, //␣
↪求解失败，指定了共轭对称正定矩阵，但输入矩阵的对角元的虚部不为0
    INPUT_NOT_UPPER_CSR = 4,     // 求解失败：指定了对称/
↪共轭对称矩阵，但输入矩阵不是上三角行主序的存储方式
    INPUT_NUMERIC_SINGULAR = 5, // 求解失败：矩阵数值上是奇异的
    INVALID_RHS = 6,                //␣
↪求解失败：矩阵的第i行全零，但右端项的第i行非零，方程组无解
    NON_POSITIVE_DEFINITE = 7,  //␣
↪求解失败：指定了正定矩阵，但求解过程中判定矩阵非正定
    INTERNAL_ERROR = 8,             // 求解失败：求解器内部错误
};
```

3.1.4 使用示例

使用超节点法求解实对称正定方程组，其中：

系数矩阵为 $A = \begin{bmatrix} 4 & & 10 & \\ & 1 & & 6 \\ 10 & & 34 & \\ & 6 & & 52 \end{bmatrix}$，右端项为 $B = \begin{bmatrix} 34 & 78 \\ 26 & 30 \\ 112 & 240 \\ 220 & 244 \end{bmatrix}$，解矩阵应为 $X = \begin{bmatrix} 1 & 7 \\ 2 & 6 \\ 3 & 5 \\ 4 & 4 \end{bmatrix}$

单机多线程版使用示例

```
int main(int argc, char *argv[])
{
    using namespace NCS::DIRECT;

    /* ---------------------  设置系数矩阵及右端项 ---------------------- */
    // 行主序的矩阵存储格式，且只存储上三角
    int n = 4;
    std::vector<int> ptr = {0, 2, 4, 5, 6};
    std::vector<int> ids = {0, 2, 1, 3, 2, 3};
    std::vector<double> matValues = {4, 10, 1, 6, 34, 52};
    // 列主序的右端项
    int nrhs = 2; // 2个右端项
    std::vector<double> rhsValues = {34, 26, 112, 220, 78, 30, 240, 244};

    /* ------------------------  设置求解参数 -------------------------- */
    DirectOptions options;
    options.matType = MatType::REAL_SYMMETRIC_POSITIVE_DEFINITE; //␣
↪实数对称正定矩阵
    options.method = Method::SUPERNODAL;  // 使用超节点法求解
    options.logLevel = LogLevel::ERROR;   // 设置日志等级
    options.numThreads = 8;  // 使用8线程求解

    /* ---------------------  创建求解对象的指针 ----------------------- */
    auto solver = CreateDirectSolver<double>(options);
    assert(solver != nullptr);
```

(续下页)

(接上页)

```
    /* ---------------------------- 求解 ------------------------------ */
    RetCode retCode = solver->Solve(n, ptr.data(), ids.data(), matValues.data(),␣
↪nrhs, rhsValues.data());

    /* ---------------------- 检查求解结果 --------------------------- */
    assert(retCode == RetCode::SUCCESS); // 检查是否求解成功
    std::vector<double> solution = {1, 2, 3, 4, 7, 6, 5, 4};
    for (int i = 0; i < n * nrhs; ++i) {
        assert(rhsValues[i] == solution[i]);
    }

    return 0;
}
```

在 visual studio 中调用示例：

```
#include <vector>
#include <cassert>
#include "NCSDirectSolverWrapper.h"

int main() {
        using namespace NCS::DIRECT;
        cDirectOptions coptions;
        coptions.matType = NCS::MatType::REAL_SYMMETRIC_POSITIVE_DEFINITE;
        coptions.method = Method::SUPERNODAL;
        coptions.numThreads = 8;
        coptions.logLevel = LogLevel::WARN;
        coptions.dataType = DataType::DOUBLE;


        //使用 C 接口创建 NCSDirectSolver 实例
        DirectSolverWrapper* directSolverWrapper = DirectSolver_new(coptions);

        // 使用 C 接口调用 Solver 方法
        int n = 4;
        std::vector<int> ptr = { 0, 2, 4, 5, 6 };
        std::vector<int> ids = { 0, 2, 1, 3, 2, 3 };
        std::vector<double> matValues = { 4, 10, 1, 6, 34, 52 };
        int nrhs = 2;
        std::vector<double> rhsValues = { 34, 26, 112, 220, 78, 30, 240, 244 };

        /* ---------------------------- 求解 ------------------------------ */
        RetCode retCode = DirectSolver_double(SolverOP::Solve, directSolverWrapper,
↪ n, ptr.data(), ids.data(), matValues.data(), nrhs, rhsValues.data());

        /* ---------------------- 检查求解结果 --------------------------- */
        assert(retCode == RetCode::SUCCESS); // 检查是否求解成功
        std::vector<double> solution = { 1, 2, 3, 4, 7, 6, 5, 4 };
        for (int i = 0; i < n * nrhs; ++i) {
                assert(rhsValues[i] == solution[i]);
        }

        DirectSolver_delete(directSolverWrapper);
        return 0;
}
```

对于复数版本的调用，把 std::vector 改为 std::vector<cComplex_double> 即可。

MPI 分布式版使用示例

使用分布式超节点法、2 进程求解：

集中式输入示例

只在主进程输入完整矩阵。

```
int main(int argc, char *argv[])
{
    using namespace NCS::DIRECT;

    /* ------------------------- 初始化MPI ------------------------- */
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    /* --------------------- 设置系数矩阵及右端项 ---------------------- */
    int n = 4, nrhs = 2;
    std::vector<int> ptr, ids;
    std::vector<double> matValues, rhsValues;
    // 只在主进程输入完整的系数矩阵及右端项矩阵
    if (rank == 0) {
        ptr = {0, 2, 4, 5, 6};
        ids = {0, 2, 1, 3, 2, 3};
        matValues = {4, 10, 1, 6, 34, 52};
        rhsValues = {34, 26, 112, 220, 78, 30, 240, 244};
    }

    /* ------------------------  设置求解参数 ------------------------- */
    DirectOptions options{
        .matType = MatType::REAL_SYMMETRIC_POSITIVE_DEFINITE,
        .method = Method::DIST_SUPERNODAL,
        .matStorageType = MatrixStorageType::CENTRALIZED_CSR,
    };

    /* --------------------  创建求解对象的指针 ----------------------- */
    auto solver_ = CreateDirectSolver<double>(options);
    assert(solver != nullptr);

    /* ---------------------------- 求解 ----------------------------- */
    RetCode retCode = solver_->Solve(n, ptr.data(), ids.data(), matValues.data(),␣
↪nrhs, rhsValues.data());

    /* ------------------------ 检查求解结果 --------------------------- */
    assert(retCode == RetCode::SUCCESS); // 检查是否求解成功
    // 只在主进程输出完整的解矩阵
    if (rank == 0) {
        std::vector<double> solution = {1, 2, 3, 4, 7, 6, 5, 4};
        for (int i = 0; i < nrhs * n; ++i) {
            assert(solution[i] == rhsValues[i]);
        }
    }

    /* ------------------------- 结束MPI ---------------------------- */
    MPI_Finalize();

    return 0;
}
```

在 visual studio 中调用示例：

```
int main(int argc, char *argv[]){
        using namespace NCS::DIRECT;

        /* ------------------------- 初始化MPI ------------------------- */
        MPI_Init(&argc, &argv);
        int rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        /* -------------------- 设置系数矩阵及右端项--------------------- */
        int n = 4, nrhs = 2;
        std::vector<int> ptr, ids;
        std::vector<double> matValues, rhsValues;
        // 只在主进程输入完整的系数矩阵及右端项矩阵
        if (rank == 0) {
                ptr = { 0, 2, 4, 5, 6 };
                ids = { 0, 2, 1, 3, 2, 3 };
                matValues = { 4, 10, 1, 6, 34, 52 };
                rhsValues = { 34, 26, 112, 220, 78, 30, 240, 244 };
        }

        cDirectOptions coptions;
        coptions.matType = NCS::MatType::REAL_SYMMETRIC_POSITIVE_DEFINITE;
        coptions.method = Method::DIST_SUPERNODAL;
        coptions.matStorageType = MatrixStorageType::CENTRALIZED_CSR;
        coptions.dataType = DataType::DOUBLE;


        //使用 C 接口创建 NCSDirectSolver 实例
        DirectSolverWrapper* solver = DirectSolver_new(coptions);
        assert(solver != nullptr);

        /* ---------------------------- 求解 ---------------------------- */
        RetCode retCode = DirectSolver_double(SolverOP::Solve, solver, n, ptr.
↪data(), ids.data(), matValues.data(), nrhs, rhsValues.data());
        assert(retCode == RetCode::SUCCESS);

        /* ---------------------- 检查求解结果 ---------------------------- */
        assert(retCode == RetCode::SUCCESS); // 检查是否求解成功
        // 只在主进程输出完整的解矩阵
        if (rank == 0) {
                std::vector<double> solution = { 1, 2, 3, 4, 7, 6, 5, 4 };
                for (int i = 0; i < nrhs * n; ++i) {
                        assert(solution[i] == rhsValues[i]);
                }
        }

        DirectSolver_delete(solver);
        MPI_Finalize();

        return 0;
}
```

对于复数版本的调用，把 std::vector 改为 std::vector<cComplex_double> 即可。

分布式输入示例

进程 0 负责输入 $[0,2)$ 行，进程 1 负责输入 $[2,4)$。

```
int main(int argc, char *argv[])
{
    using namespace NCS::DIRECT;

    /* ------------------------- 初始化MPI -------------------------- */
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int worldSize;
    MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
    assert(worldSize == 2);

    /* -------------------- 设置系数矩阵及右端项 --------------------- */
    int n = 4, nrhs = 2;
    std::vector<int> ptr, ids;
    std::vector<double> matValues, rhsValues;
    // 分布式输入，每个进程输入部分矩阵
    if (rank == 0) {
        ptr = {0, 2, 4};
        ids = {0, 2, 1, 3};
        matValues = {4, 10, 1, 6};
        rhsValues = {34, 26, 78, 30};
    } else {
        ptr = {0, 1, 2};
        ids = {2, 3};
        matValues = {34, 52};
        rhsValues = {112, 220, 240, 244};
    }

    /* ------------------------  设置求解参数 -------------------------- */
    DirectOptions options{
        .matType = MatType::REAL_SYMMETRIC_POSITIVE_DEFINITE,
        .method = Method::DIST_SUPERNODAL,
        .matStorageType = MatrixStorageType::DISTRIBUTED_CSR, //␣
↪分布式矩阵输入及输出
    };
    if (rank == 0) {
        // 进程0负责[0,2)行
        options.localBeginId = 0;
        options.localEndId = 2;
    } else {
        // 进程1负责[2,4)行
        options.localBeginId = 2;
        options.localEndId = 4;
    }

    /* --------------------  创建求解对象的指针 ------------------------ */
    auto solver_ = CreateDirectSolver<double>(options);
    assert(solver != nullptr);

    /* ---------------------------- 求解 ------------------------------ */
    RetCode retCode = solver_->Solve(n, ptr.data(), ids.data(), matValues.data(),␣
↪nrhs, rhsValues.data());

    /* ------------------------ 检查求解结果 --------------------------- */
    assert(retCode == RetCode::SUCCESS); // 检查是否求解成功
    // 每个进程只输出部分解向量
    std::vector<double> solution;
```

(续下页)

(接上页)

```
    if (rank == 0) {
        solution = {1, 2, 7, 6};  // 进程0只输出解向量的[0,2)行
    } else {
        solution = {3, 4, 5, 4};  // 进程1只输出解向量的[2,4)行
    }
    for (int i = 0; i < nrhs; ++i) {
        int domainSize = options.localEndId - options.localBeginId;
        for (int j = 0; j < domainSize; ++j) {
            assert(solution[i * domainSize + j], rhsValues[i * domainSize + j]);
        }
    }

    /* -------------------------- 结束MPI ---------------------------- */
    MPI_Finalize();

    return 0;
}
```

在 visual studio 中调用示例：

```
int main(int argc, char *argv[])
{
        using namespace NCS::DIRECT;

        /* -------------------------- 初始化MPI ---------------------------- */
        MPI_Init(&argc, &argv);
        int rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        int worldSize;
        MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
        assert(worldSize == 2);

        /* --------------------- 设置系数矩阵及右端项 ---------------------- */
        int n = 4, nrhs = 2;
        std::vector<int> ptr, ids;
        std::vector<double> matValues, rhsValues;
        // 分布式输入，每个进程输入部分矩阵
        if (rank == 0) {
                ptr = { 0, 2, 4 };
                ids = { 0, 2, 1, 3 };
                matValues = { 4, 10, 1, 6 };
                rhsValues = { 34, 26, 78, 30 };
        }
        else {
                ptr = { 0, 1, 2 };
                ids = { 2, 3 };
                matValues = { 34, 52 };
                rhsValues = { 112, 220, 240, 244 };
        }

        /* ----------------------  设置求解参数 ------------------------- */
        cDirectOptions coptions;
        coptions.matType = NCS::MatType::REAL_SYMMETRIC_POSITIVE_DEFINITE;
        coptions.method = Method::DIST_SUPERNODAL;
        coptions.matStorageType = MatrixStorageType::DISTRIBUTED_CSR;
        coptions.dataType = DataType::DOUBLE;
        if (rank == 0) {
                // 进程0负责[0,2)行
                coptions.localBeginId = 0;
                coptions.localEndId = 2;
        }
```

(续下页)

(接上页)

```
        else {
                // 进程1负责[2,4)行
                coptions.localBeginId = 2;
                coptions.localEndId = 4;
        }

        //使用 C 接口创建 NCSDirectSolver 实例
        DirectSolverWrapper* solver = DirectSolver_new(coptions);
        assert(solver != nullptr);

        /* ----------------------------- 求解 ----------------------------- */
        RetCode retCode = DirectSolver_double(SolverOP::Solve, solver, n, ptr.
↪data(), ids.data(), matValues.data(), nrhs, rhsValues.data());

        /* ------------------------ 检查求解结果 ---------------------------- */
        assert(retCode == RetCode::SUCCESS); // 检查是否求解成功
        // 每个进程只输出部分解向量
        std::vector<double> solution;
        if (rank == 0) {
                solution = { 1, 2, 7, 6 };  // 进程0只输出解向量的[0,2)行
        }
        else {
                solution = { 3, 4, 5, 4 };  // 进程1只输出解向量的[2,4)行
        }
        for (int i = 0; i < nrhs; ++i) {
                int domainSize = coptions.localEndId - coptions.localBeginId;
                for (int j = 0; j < domainSize; ++j) {
                        assert(solution[i * domainSize + j], rhsValues[i *␣
↪domainSize + j]);
                }
        }

        DirectSolver_delete(solver);
        /* --------------------------- 结束MPI ----------------------------- */
        MPI_Finalize();

        return 0;
}
```

对于复数版本的调用，把 std::vector 改为 std::vector<cComplex_double> 即可。

3.2 线性迭代法求解接口说明

3.2.1 线性迭代法接口整体说明

线性迭代法对用户提供的接口封装在 NCSIterativeSolver.h 中，其具体的实现为：

```
template<typename Scalar>
class NCSIterativeSolver {
public:
     NCSIterativeSolver();

     ~NCSIterativeSolver();

     /* 参数设置接口 */
     void SetOption(const std::string& opt, const std::string& value);

     /* 从文件接收参数的接口 */
```

(续下页)

(接上页)

```
    void SetFromFile(const std::string& filename);

    /* 端到端求解接口 */
    int Solve(const Scalar* data, const int* colIdx, const int* rowPtr,
              const int dim, const Scalar* rhs, const Scalar* iniSol,
              Scalar* sol, MatType matType = MatType::REAL_GENERAL, std::string␣
↪name = "");

    /* 迭代残差返回接口 */
    double GetResNorm() const;

    /* 真实残差返回接口 */
    double GetTrueResNorm() const;

    /* 迭代次数返回接口 */
    int GetIter() const;
}
```

NCSIterativeSolver.h 是一个模板类，模板参数为 Scalar，具体的取值为：

- `double`: 实数双精度
- `std::complex<double>`: 复数双精度类型

针对不同模板参数，创建求解对象实例的具体实现分别为：

- `NCSIterativeSolver<double> linearRealSolver`: 实数求解对象
- `NCSIterativeSolver<std::complex<double>> linearComplexSolver`: 复数求解对象

针对不同类型的矩阵输入，求解接口 Solve() 存在两种具体的实现：

- `Solve()`: 针对串行矩阵输入求解
- `SolveMpi()`: 针对分布式并行矩阵输入求解

而对于 visual studio 用户需要调用封装在 NCSIterativeSolverWrapper.h 中的 C 接口，其具体的实现为：

```
#include "NCSIterativeSolver.h"

namespace NCS::KSP {

enum class DataType : int32_t { DOUBLE = 0, COMPLEX_DOUBLE = 2, };

enum class Get : int32_t {
  GetResNorm0 = 0, GetResNorm = 1, GetRelResNorm = 2, GetTrueResNorm0 = 3,
  GetTrueResNorm = 4, GetRelTrueResNorm = 5, GetTrueAbsResP = 6, GetTrueRelResP =7,
};

extern "C" {
struct DLLEXPORT NCSIterativeSolverWrapper;

struct cComplex_double {
  double real;
  double imag;
};
/* 创建LinearSolver实例 */
DLLEXPORT NCSIterativeSolverWrapper* LinearSolver_new(DataType dataType);
/* 参数设置接口 */
DLLEXPORT void LinearSolverSetOption(DataType dataType, NCSIterativeSolverWrapper*␣
↪wrapper,
                                     const char* copt, const char* cvalue);
/* 从文件接收参数的接口 */
DLLEXPORT void LinearSolverSetFromFile(DataType dataType,␣
```

(续下页)

(接上页)

```
↪NCSIterativeSolverWrapper* wrapper, const char* cfilename);
/* AIHelper接口 */
DLLEXPORT void LinearSolverSetAIHelper(DataType dataType,␣
↪NCSIterativeSolverWrapper* wrapper,
                                       AI4SOLVER::HelperType helpetype =␣
↪AI4SOLVER::HelperType::KSP_PC_RECOMMENDER);

DLLEXPORT void LinearSolverSetFieldSplitIndexSet(DataType dataType,␣
↪NCSIterativeSolverWrapper* wrapper,
                                                 const int* splitIS, int␣
↪splitDim, bool reset = false);
/* double类型端到端求解接口 */
DLLEXPORT int LinearSolverSolve_double(DataType dataType,␣
↪NCSIterativeSolverWrapper* wrapper,
    const double* data, const int* colIdx, const int* rowPtr, int dim, const␣
↪double* rhs,
    const double* iniSol, double* sol, MatType matType = MatType::REAL_GENERAL,␣
↪char* cname = nullptr);
/* 复数double类型端到端求解接口 */
DLLEXPORT int LinearSolverSolve_complex_double(DataType dataType,␣
↪NCSIterativeSolverWrapper* wrapper,
    const cComplex_double* cdata, const int* colIdx, const int* rowPtr, int dim,␣
↪const cComplex_double* crhs,
    const cComplex_double* ciniSol, cComplex_double* csol, MatType matType =␣
↪MatType::REAL_GENERAL, char* cname = nullptr);
/* 计算结果返回接口 */
DLLEXPORT double LinearSolverGet(DataType dataType, NCSIterativeSolverWrapper*␣
↪wrapper, Get getOption);
/* 迭代次数返回接口 */
DLLEXPORT int LinearSolverGetIter(DataType dataType, NCSIterativeSolverWrapper*␣
↪wrapper);
/* 删除LinearSolver实例 */
DLLEXPORT void LinearSolver_delete(NCSIterativeSolverWrapper* wrapper);
}
}

由上面的代码可以看出,线性迭代法共为用户提供三类种类型的接口:参数设置接口、端到端求解接口、以及结

## 接口概览

### 参数设置接口
```c++
void SetOption(std::string& opt, std::string& value)
```

- 通过命令行逐条配置的方式设置参数，其中-opt 表示参数的名字, -value 表示参数的值。

```
void SetFromFile(std::string& filename)
```

- 通过文件读写的方式进行设置参数，其中-filename 表示配置文件名/路径。

```
void SetFieldSplitIndexSet(const int* splitIS, const int splitDim, const bool␣
↪reset = false)
```

- 专门用于 FieldSplit 预处理子的通用切分方式，其中数组-splitIS 包含当前 split 对应的全局行/列号，-splitDim 为数组-splitIS 的维度，-reset 表示是否重新设置。
```

端到端求解接口

```
int Solve(Scalar* data, int* colIdx, int* rowPtr, int dim, Scalar* rhs,Scalar*␣
↪iniSol, Scalar* sol,
          MatType matType = MatType::REAL_GENERAL, std::string name = "")
```

- 输入方程组信息，包括 CSR 格式存储的系数矩阵 (data, colIdx, rowPtr)，矩阵维度 (dim)，右端项 (rhs)，初始解 (iniSol)，获取最终解 (sol) 以及求解状态 (返回值)。此外，也可以指定矩阵元素类型 (matType) 和问题名称 (name)。

结果返回接口

```
int GetIter()
```

- 获取求解过程的迭代次数。

```
double GetResNorm0()
```

- 获取当前近似解对应的的初始残差范数。

```
double GetTrueResNorm0()
```

- 获取当前近似解对应的真实的初始残差范数。

```
double GetResNorm()
```

- 获取当前近似解对应的的残差范数。

```
double GetTrueResNorm()
```

- 获取当前近似解对应的真实的残差范数。

```
double GetRelResNorm()
```

- 获取当前近似解对应的相对残差范数。

```
double GetRelTrueResNorm()
```

- 获取当前近似解对应的真实的相对残差范数。

```
double GetResNorm()
```

- 获取当前近似解对应的迭代残差。

```
double GetTrueResNorm()
```

- 获取当前近似解对应的真实残差。

3.2.2 参数设置接口说明

线性迭代法参数众多，其中不乏一些需要特别注意的参数，这些参数必须由用户手动设置。而另外一些参数则不必受此局限，因为它们已经存在预设默认值。

- 迭代法和预处理子选型：Krylov 迭代法种类丰富，且通常与同样多样的预处理子结合使用，因此这两个参数是必须根据用户的使用场景通过`--ksp_type` 和`--pc_type` 手动设置的；
- 线性方程组的求解模态：Krylov 迭代法支持共享内存与分布式并行两种求解模式，需由用户配置`--global_dimension` 参数进行选定（该参数除明确矩阵的全局维度外，还隐式表明当前方程组的求解模式为分布式并行，如用户想要进行共享内存求解，则无需设置该参数）；

- 矩阵输入方式：特别针对分布式求解场景，用户须使用参数--is_rank0_input_only 明确矩阵是分布式输入还是仅由主进程（RANK 0）统一输入。对于分布式矩阵输入的场景，用户还需额外设置不同进程对应的行起始全局索引--cur_start_index。

本章将介绍两种参数配置接口用于设定迭代法和预处理子相关参数，一种是利用 SetOption() 接口，按照参数名和参数值的方式逐个设置，另一种是利用 SetFromFile() 接口从文本读入，一次性设置所有参数，下面分别对这两种接口进行说明。

```
void SetOption(const std::string& opt, const std::string& value)
```

利用 SetOption() 接口，用户可通过指定参数名和对应参数值的格式设置具体算法的相关参数。以下是简单的示例，

```
NCSIterativeSolver<double> linearSolver;
/******** Krylov 迭代法参数设置 ********/
linearSolver.SetOption("--ksp_type", "cg"); // 迭代法类型（必设参数）
linearSolver.SetOption("--ksp_max_iteration", "1000"); // 最大迭代次数（可选参数）
linearSolver.SetOption("--ksp_relative_tol", "1e-6"); //␣
↪相对迭代残差阈值（可选参数）

/******** 预处理子参数设置 ********/
linearSolver.SetOption("--pc_type", "jacobi"); // 预处理子类型（必设参数）
linearSolver.SetOption("--pc_jacobi_type", "diagonal"); //␣
↪Jacobi预处理子亚型（可选参数）

/******** 分布式求解参数设置 ********/
linearSolver.SetOption("--is_rank0_input_only", false); //␣
↪是否由RANK0唯一输入全量矩阵（分布式求解必设参数）
linearSolver.SetOption("--global_dimension", "1e6"); //␣
↪待求解方程的数量，也即全局矩阵的维度（分布式求解必设参数）
linearSolver.SetOption("--cur_start_index", "1024"); //␣
↪RANK对应矩阵的行起始全局索引（分布式求解时分布式矩阵输入场景必设参数）
```

所有线性迭代法 & 预处理子关联参数说明详见章节设置参数具体说明。

```
void SetFromFile(const std::string& filename)
```

利用 SetFromFile() 接口，用户可以直接从文本文件读取参数，代码示例：

```
NCSIterativeSolver<double> linearSolver;
std::string parasFile = "xxx/paras_file.txt";
linearSolver.SetFromFile(parasFile);
```

参数文本具体的内容格式是一行对应一个参数的设置，形式为 参数名 opt + 空格 whitespace + 参数值 value，具体的 opt 、value 与 SetOption() 接口设置保持一致，文本内容示例为如下，

```
--ksp_type cg
--ksp_max_iteration 1000
--ksp_relative_tol 1e-6
--pc_type jacobi
--pc_jacobi_type diagonal
```

3.2.3 端到端求解接口说明

迭代法对外只提供一个求解接口 `Solve()`，统一了共享内存与分布式计算的求解入口（由分布式求解模式必需参数区分）。

```
int Solve(const Scalar* data, const int* colIdx, const int* rowPtr, const int dim,␣
↪const Scalar* rhs, const Scalar* iniSol, Scalar* sol,
          MatType matType = MatType::REAL_GENERAL, std::string name = "")
```

该接口共有 9 个输入项，分别为：

```
const Scalar* data
```

- 表示 CSR 三元组存储系数矩阵非零元素的数组；

```
const int* colIdx
```

- 表示 CSR 三元组存储非零元列索引的数组；

```
const int* rowPtr
```

- 表示 CSR 三元组存储每行非零元个数的数组；

```
const int dim
```

- 表示方程组的维数。需要注意的是，面对分布式矩阵输入的情况，该参数应设置为当前进程 Rank 对应矩阵切片的局部维度，而非全局维度；

```
const Scalar* rhs
```

- 表示方程组右端项；

```
const Scalar* iniSol
```

- 表示方程组初始解，由用户提供，如果为空则算法会默认初始解为 0；

```
Scalar* sol
```

- 表示方程组求解得到的最终结果；

```
MatType matType
```

- 表示矩阵类型，也即 Scalar 类型，可选参数，默认为 REAL_GENERAL，取值范围包括：REAL_GENERAL = 0, REAL_SYMMETRIC_POSITIVE_DEFINITE = 1, REAL_SYMMETRIC_INDEFINITE = 2, COMPLEX_GENERAL = 3, COMPLEX_SYMMETRIC = 4, COMPLEX_HERMITIAN_POSITIVE_DEFINITE = 5, COMPLEX_HERMITIAN_INDEFINITE = 6；

```
std::string name
```

- 表示问题名称，可选参数，默认为空。

除上述 9 项输入外，迭代求解结束后，该接口会返回一个类型为 `int` 的状态码，用以反映求解成功与失败的原因，具体的状态码说明如下：

```
1    // 求解成功，收敛条件为相对残差范数达到阈值
2    // 求解成功，收敛条件为绝对残差范数达到阈值
3    // 求解成功，在设定的次数内收敛
4    // 解成功，算法“幸运中断”
-1   // 求解失败，达到最大迭代次数
-2   // 求解失败，求解过程发散
```

(续下页)

(接上页)

```
-3   // 求解失败，预条件子或者矩阵奇异
-4   // 求解失败，预条件子非正定
-5   // 求解失败，矩阵非正定
-6   // 求解失败，预条件子设置失败
-99  // 求解失败，矩阵和右端项与选择的算法不兼容
-101 // 求解失败，缺少必须设定的参数，如迭代算法选型
```

3.2.4 求解返回结果说明

求解完成后，可以获取求解的结果、残差和迭代次数等信息，具体接口如下说明：

```
int GetIter()
```

- 获取求解过程涉及的迭代次数

```
double GetResNorm0()
```

- 获取迭代计算的初始残差范数，该残差为由递推公式计算得到的初始残差，可能与初始的真实残差存在偏离

```
double GetTrueResNorm0()
```

- 获取真实的初始残差范数，即 $||b - Ax0||$

```
double GetResNorm()
```

- 获取迭代计算的残差范数，该残差为由递推公式计算得到的残差，可能与真实残差存在偏离

```
double GetTrueResNorm()
```

- 获取真实的残差范数，即 $||b - Ax||$

```
double GetRelResNorm()
```

- 获取迭代计算的相对残差范数，也即”残差范数/初始残差范数”

```
double GetRelTrueResNorm()
```

- 获取真实的相对残差范数，也即”真实的残差范数/真实的初始残差范数”

3.2.5 设置参数具体说明

本章是对 `SetOption()` 参数设置接口的具体说明，将给出全量线性迭代法 & 预处理子相关参数的配置方式。

迭代法参数说明

创建 Krylov 迭代求解算法

```
--ksp_type value
```

由于 Krylov 迭代法种类众多，且每种算法针对的系数矩阵类型各异，很难设置一种默认的算法适配所有的问题，需由客户通过`--ksp_type` 参数指定选用的迭代算法。当前数值计算求解器支持的算法算法选型及其适用的矩阵类型、配置方式如下所述，

- Conjugate Gradient（CG）共轭梯度法：面向 Hermitian 正定矩阵，`value` 参数配置值为 `cg`

- Generalized Minimal Residual（GMRES）广义最小残差法：面向一般矩阵，`value` 参数配置值为 `gmres`
- Minimal Residual（MINRES）极小残差法：面向 Hermitian 不定矩阵，`value` 参数配置值为 `minres`
- Biconjugate Gradient Stablized（BICGSTAB）稳定双共轭梯度法：面向一般大规模矩阵，`value` 参数配置值为 `bicgstab`
- Generalized Product Bi-Conjugate Gradient（GPBICG）基于双共轭梯度的广义乘积型方法：面向一般大规模矩阵，`value` 参数配置值为 `gpbicg`。
- Minimal Residual (MINRES) 极小残差法：面向 Hermitian（厄米）矩阵，尤其适用于非正定的情况下，旨在最小化残差的二范数，`value` 参数配置值为 `minres`。
- Minimal Residual with Quadratic Programming (MINRESQLP) 极小残差二次规划法：扩展自 MINRES，面向线性约束下的 Hermitian 矩阵问题，结合了二次规划方法进行进一步优化，特别适用于需要满足某些约束条件的情况下，`value` 参数配置值为 `minresqlp`。
- preonly （仅对预处理配置生效）：在形如后续提到的 pc_fs_additive_fieldsplit_i_ksp_type 等预条件二级参数的设定中，`preonly` 意味着该求解器只会执行预处理操作，而不执行后续的 Krylov 迭代。

Krylov 迭代法公共参数概览

下表对各个具体算法模块的参数名及其对应的取值进行说明。首先给出的是线性迭代公共参数概览

日志关联参数

```
--ksp_loglevel value
```

- 表示日志打印类型
- `value` 取值类型为 `int`，范围为 `2-info` 等级，`1-warning` 等级，`0-error` 等级，默认值是 `2`

```
--ksp_logpath value
```

- 表示日志文件的存储路径
- `value` 取值类型为 `String`，如“`/home/user/`”

求解模态关联参数

```
--ksp_threads value
```

- 表示系统运行的线程数
- `value` 取值类型为 `int`，如选用 `8` 线程。默认值是 `1` 对应单线程

```
--global_dimension value
```

- 表示线性方程组的全局维度（等于每个 RANK 的局部维度之和），对应待求解问题的方程总数
- `value` 取值类型为 `int`，取值范围为 $(0, 2147483647]$
- 备注：用户如需要分布式求解，则必须设置此参数，否则禁止设置该参数，这是因为软件内核会根据该参数自动判断求解模态是共享内存还是分布式

```
--is_rank0_input_only value
```

- 表示用户是否仅由 RANK0（主进程）执行矩阵输入（如是，则分布式矩阵切分仅支持 `naive` 切分）

- value 取值类型为 bool，默认值是 false
- 备注：我们为用户提供两种截然不同的矩阵输入模式，分别为集中式输入和分布式输入，分布式输入主要适用于局部进程资源受限的场景。内核正是利用该参数进行两种模式的区分的。当 value 取值为 true 时则对应集中式输入，反之则对应分布式矩阵输入，即每个进程负责自己纳管的局部矩阵信息

```
--cur_start_index value
```

- 表示当前进程对应的矩阵切片首行的全局索引（使用该参数时，默认为分布式输入矩阵，--is_rank0_input_only 需设置成为 false）
- value 取值类型为 int，取值范围为 $(0, 2147483647]$
- 备注：该参数仅在用户选择分布式求解模式且矩阵信息分布式提供时（即--is_rank0_input_only value 设置为 false）生效。在此条件下，每个进程仅负责局部的矩阵数据，因此用户自然地需要提供局部数据对应的全局索引给到不同进程，以便每个进程将自己与全局系统关联起来。举例来说，待求问题包含 10 个方程，由 3 个进程执行分布式求解。若进程 1 负责输入系数矩阵的 0~2 行，进程 2 负责输入 3~5 行，进程 3 负责输入 6~9 行，则需作如下配置：

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
NCSIterativeSolver<double> linearSolver;

if (rank == 0) {
    linearSolver.SetOption("--cur_start_index", "0");
} else if (rank == 1) {
    linearSolver.SetOption("--cur_start_index", "3");
} else {
    linearSolver.SetOption("--cur_start_index", "6");
}
```

由上述代码可以看出，除要求客户输入全局索引外，我们还要求每个进程纳管的矩阵数据在全局维度上是连续的，这是方便算法内核可以结合当前进程的起始索引与求解接口输入的局部矩阵维度，计算出本地矩阵每个元素的全局行索引。

迭代终止准则关联参数

```
--ksp_max_iteration value
```

- 表示线性迭代法运行的最大迭代次数限制
- value 取值类型为 int，取值范围为 $(0, 2147483647]$，默认值是 10000
- 备注：当算法执行到该参数规定的上限仍未收敛时，将直接返回-1 状态码，用于表示算法由于计算步数超限而发散

```
--ksp_absolute_tol value
```

- 表示迭代求解终止准则关于绝对残差收敛判断的阈值上限
- value 取值类型为 double，取值范围为 $(0, 1.7e + 308)$，默认值是 1e-99

```
--ksp_relative_tol value
```

- 表示迭代求解终止准则关于相对残差收敛判断的阈值上限
- value 取值类型为 double，取值范围为 $(0, 1.7e + 308)$，默认值是 1e-11

```
--ksp_happy_break_tol value
```

- 表示迭代法提前终止运算的阈值

- value 取值类型为 double，取值范围为 $(0, 1.7e+308)$，默认值是 1e-30
- 备注：迭代法可能会在计算过程中发现了一个非常好的解，这时迭代法就可以提前终止，因为已经找到了满足要求的解，这种情况被称为 happy break

```
--ksp_pc_side value
```

- 表示预处理子的作用方向
- value 取值类型为 int，范围为 0-左，1-右，2-分裂, 3-无, 默认值为 0

```
--ksp_true_res_check value
```

- 表示当前算法求解使用的迭代终止准则是否考虑真实残差
- value 取值类型为 bool，范围为 false-迭代终止判断仅考虑递推残差，true-迭代终止判断同时考虑真实与递推残差
- 备注：真实残差的概念与迭代残差相对，用于表示当前迭代解 x_k 与真实解之间的差距，由计算公式 $trueRes = b - Ax_k$ 表述。

  关于真实残差的计算涉及 SpMV，相较于迭代残差一般的 AXPY 计算更为耗时，因此 Krylov 方法通常利用计算效率更高的迭代残差进行收敛性判断而非其对应的真实值。当矩阵不太病态时，算法数值稳定性通常较好，迭代残差是真实残差的一个良好估计。但如果我们处理的问题比较病态或迭代算法/预处理子选型不好时，两者的数值差别往往很大，这就使得我们仅对迭代残差做迭代终止检查的有效性大打折扣。

  因此，如果用户通过--ksp_true_res_check true 使能真实残差检查，我们则在迭代残差检查的基础上补充真实残差的校验，仅当两者都满足预设条件时，方可认为算法收敛。考虑到真实残差的计算时间长本较高，仅在迭代残差满足条件后，才对真实残差进行校验（这样的校验方式是相对合理的，因为通常情况下迭代残差不满足收敛检查时，真实残差亦不满足）

```
--ksp_norm_type value
```

- 表示迭代残差范数的计算方式
- value 取值类型为 string，范围为 preconditioned-计算预处理后的残差的范数，unpreconditioned-计算不作预处理的原始残差的范数，默认值是 preconditioned
- 备注：对于一般迭代法（无预处理子）求解场景，我们将残差范数简单定义为 L2-Norm。而对于带预处理子场景，定义选项按参数设置分为

  1. Preconditioned L2-Norm：预处理后的残差向量的范数，按 $resNorm = ||M * residual||$ 计算，其中 M 为预处理矩阵
  2. Unpreconditioned L2-Norm：不执行预处理的原始残差残差向量的范数，即 $resNorm = ||residual||$

```
--ksp_relative_res_deno_type value
```

- 表示迭代终止准则依赖的相对残差计算时的分母选择
- value 取值类型为 string，范围为 rhs-右端项范数，iniRes-初始残差范数，minimum-前两者中范数较小的一个，默认值是 rhs
- 备注：相对迭代残差表示第 k 次迭代步计算的迭代残差相对于初始残差的比率，定义为 $relRes = resNorm_k/resNorm_0, k = 0, 1, ...$，其中，$||resNorm_k||$ 表示第 k 次迭代时的残差范数，特别地，$||resNorm_0||$ 表示初始残差范数。与通常理解的初始残差由 $b - Ax_0$ 表示不同的是，我们补充支持使用右端项作为初始残差的选项。用户可通过--ksp_relative_res_deno_type 参数接口指定初始残差的三种不同选择，分别为：

  1. iniRes：$resNorm_0 = ||b - Ax_0||$，其中 x_0 为非零初始解
  2. rhs：$resNorm_0 = ||b||$
  3. minimum：$resNorm_0 = min(||b - Ax_0||, ||b||)$

当初始解输入为 0 向量时，三种方式的初始残差统一为右端项

```
--ksp_divergence_tol value
```

- 表示迭代求解终止准则关于相对残差发散判断的阈值下限
- value 取值类型为 double，取值范围为 $(0, 1.7e+308)$，默认值是 1e5
- 备注：迭代求解的发散检查仅关注相对残差。当系统执行到某一次迭代过程时，发现当前计算的相对迭代残差较大时，即 $relRes >= tol_{dr}$, tol_{dr} 由该参数外部设定，则认为针对当前求解问题，用户选用的迭代法无法收敛（或称为发散）

```
--ksp_conv_criteria_type value
```

- 表示迭代终止准则使用的残差收敛判断条件
- value 取值类型为 String，范围为 only-仅考虑绝对迭代残差，both-同时考虑绝对和相对残差，默认值 both
- 备注：关于迭代残差的收敛终止条件也有两种不同的模式
  1. 仅通过绝对迭代残差判断算法是否已经收敛，检查准则如下：$resNorm <= tol_{ar}$，其中，tol_{ar} 是绝对迭代残差收敛阈值，由用户通过参数--ksp_absolute_tol 外部设定
  2. 同时参考相对残差和绝对残差，两种类型的残差只要其中一个的范数满足对应的收敛条件，也就是 $resNorm <= tol_{ar}$ 或者 $relRes <= tol_{rr}$，即认为求解算法收敛，其中，tol_{rr} 是相对迭代残差收敛阈值，由用户通过参数--ksp_relative_tol 外部设定
  3. 特别地，如果用户意图仅通过相对残差进行收敛性判断，只需设置 $$tol_{ar}$$ 为很小的值或 0

```
--ksp_monitor value
```

- 表示是否开启每一迭代步的迭代残差的监测功能
- value 取值类型为 bool，默认为 false

```
--ksp_monitor_true_residual value
```

- 表示是否开启每一迭代步的迭代残差和真实残差的监测功能
- value 取值类型为 bool，默认为 false

```
--ksp_print_final_infos value
```

- 表示是否打印求解结束后的信息，包括求解状态、迭代步数、残差等
- value 取值类型为 bool，默认为 false

端到端数据处理

```
--update_flag value
```

在处理某些特定问题时，例如时间依赖的偏微分方程（PDE）问题，在前处理网格保持不变的情况下，组装出的矩阵的非零元模式也不会发生变化。在这种情况下，求解器只需在首次调用时对矩阵结构进行设置，并将该结构存储在求解器的特定数据结构中。之后的调用只需要更新矩阵的值，而不需要重新设置矩阵结构，从而节省了矩阵构建的时间。

- value 参数的类型为 bool，默认为 false。
- 当 value 为 true 时，求解器的行为如下：
  1. 在第一个时间步，检查矩阵结构是否已经设置。如果尚未设置，求解器将执行矩阵结构的初始化，并将其存储以供后续使用。

2. 对于后续的时间步，若矩阵结构已经初始化，则跳过矩阵结构的设置步骤，直接进入赋值更新阶段。

- 当 `value` 为 `false` 时，每次调用求解器时矩阵的构建和赋值操作将同时进行，矩阵结构和数值更新将耦合执行。

这种机制的主要目的是提高效率，避免在每个时间步重新构建矩阵结构，从而减少不必要的计算开销。

不同迭代法特化参数详细说明

CG 算法关联参数

```
--ksp_cg_dot_type value
```

- 表示针对复数问题的求解方式
- `value` 取值类型为 `int`，范围为 `0`-共轭对称，`1`-复数对称，默认是 `0`

GMRES 算法关联参数

```
--ksp_gmres_restart_num value
```

- 表示 GMRES 算法执行重启操作的内循环最大迭代步数
- `value` 取值类型为 `int`，范围为 `0` 至`--ksp_max_iteration` 参数设置的算法最大迭代步数，默认值是 `30`

```
--ksp_gmres_orthogo_type value
```

- 表示 GMRES 执行 Krylov 子空间基的正交化运算的方式
- `value` 取值类型为 `int`，范围为 `0`—Classical GS，`1`-Modified GS，`2`-Householder，默认值是 `0`

```
--ksp_gmres_refine value
```

- 表示是否对 GMRES 依赖的 Gram-Schmidt 正交过程做正交优化
- `value` 取值类型为 `bool`，默认值为 `false`
- 此处，我们特别地对`--ksp_gmres_refine` 进一步说明，该参数为 GMRES 算法的正交优化参数，即针对数值不稳定的矩阵，算法会额外做一次正交化来保证结果的正交性。

BiCGSTAB 算法关联参数

```
--ksp_bistabl_l value
```

- 表示算法中涉及的多项式级数
- `value` 取值类型为 `int`，默认值是 `3`

GPBICG 算法关联参数

```
--ksp_gpbicg_l value
```

- 表示 GPBICG 算法参数 l
- value 取值类型为 int，默认值是 1

```
--ksp_gpbicg_m value
```

- 表示 GPBICG 算法参数 m
- value 取值类型为 int，默认值是 0

预处理子参数说明

创建预处理子

```
--pc_type value
```

当前支持的预处理子类型如下：

注意：由于--pc_type 是必需参数，因此如若真实使用时况并不需要预处理子参与，用户须手动设置--pc_type none。

Jacobi 预处理子关联参数

JACOBI 预处理子是最简单的预处理子，包含三种获取 jacobi 向量方法，默认的 diagonal 为直接取矩阵对角元作为预处理子。这种预处理技术没有额外构造花费，易于实施。

```
--pc_jacobi_type
```

- 设置 jacobi 类型, 默认值为 diagonal，串行可选值为 diagonal, rowmax, rowsum，分布式当前仅支持 diagonal

SOR 预处理子关联参数

SOR 预处理子由系数矩阵对角元和严格下三角矩阵构成：$M = D/\omega + L$，其中为 ω 为松弛参数，L 为 A 的下三角矩阵。其中包含 SOR，SSOR，GS，Eisenstat 四种类型，当前仅支持串行求解场景：

- GS 类型为 $\omega = 1$ 的情况
- SSOR 类型由系数矩阵对角元、严格上三角、严格下三角矩阵构成：$M = (D/\omega + U)(D/\omega + L)$，其中 U 为 A 的上三角矩阵
- Eisenstat 类型针对双边预处理子的应用情况进行化简，减少浮点运算

```
--pc_sor_type
```

- 设置 sor 类型，默认值为 ssor，可选值为 sor, ssor, eisenstat

```
--pc_sor_omega
```

- 设置松弛系数，默认值为 1.0，取值范围建议在 $(0, 2)$ 之间

```
--pc_sor_shift
```

- 设置偏移系数，在矩阵对角元为 0 的情况下，可以设置不为零的参数进行调整，默认值为 1.0，取值范围无限制

对于 SOR，SSOR，GS，Eisenstat 四种算法，设置方法分别为：

- SOR：`--pc_type sor`，`--pc_sor_type sor`，`--pc_sor_omega x`，其中 `x` 不为 1，比如 0.1
- GS：`--pc_type sor`，`--pc_sor_type sor`
- SSOR：`--pc_type sor`
- Eisenstat：`--pc_type sor`，`--pc_sor_type eisenstat`

Block Jacobi 预处理子关联参数

Block Jacobi 预处理子选取系数矩阵的分块对角矩阵，将大问题转化为多个子模块，并在子模块中采用其他预处理子求解

```
--pc_bjacobi_sub_pc_type
```

- 设置子模块内部预处理子类型，默认值为 `ilu`，串行和分布式建议取值为 `icc` 和 `ilu`。

```
--pc_bjacobi_block_num
```

- 设置子模块数量，类型为非负整数，串行默认值为 1，分布式默认值为进程数。当块数量大于矩阵行数时会取 1 进行计算（相当于不分块）。

不完全 Cholesky 分解预处理子关联参数

不完全 Cholesky 分解预处理子 (ICC) 适用于对称正定矩阵 A，目标是对其进行近似的 Cholesy 分解: $A \approx LL^T$，其中 L 是稀疏下三角矩阵。

ICC0 /ICCK

静态 ICC，只根据矩阵的结构信息丢弃元素。其中 ICC0 是一种不引入任何 fill-in, 保留原来 A 的稀疏特征的算法。当前仅支持串行求解场景。

```
--pc_icc_factor_level
```

- 设置分解层数限制，类型为非负整数，默认值为 `0`，其中 0 相当于 ICC0，大于 0 的值相当于 ICCK

```
--pc_icc_scale_type
```

- 设置 scale 类型，默认值为 `none`，可选值为 `none`，`rss`

```
--pc_icc_shift_type
```

- 设置 shift 类型，默认值为 `none`，可选值为 `none`，`nonzero`，`positive_definite`

不完全 LU 分解预处理子关联参数

不完全 LU 分解预处理子 (ILU) 适用于任意矩阵 A，目标是对其进行近似的 LU 分解：$A \approx LU$，其中 L 是稀疏下三角矩阵，U 是稀疏上三角矩阵。

ILU0 /ILUK

静态 ILU，只根据矩阵的结构信息丢弃元素。其中 ILU0 是一种不引入任何 fill-in, 保留原来 A 的稀疏特征的算法。当前仅支持串行求解场景。

```
--pc_ilu_factor_level
```

- 设置分解层数限制，类型为非负整数，默认值为 `0`，其中 0 相当于 ILU0，大于 0 的值相当于 ILUK

```
--pc_ilu_scale_type
```

- 设置 scale 类型，默认值为 `none`，可选值为 `none`，`ras`，`rss`，`hybrid`

```
--pc_ilu_shift_type
```

- 设置 shift 类型，默认值为 `none`，可选值为 `none`，`nonzero`，`positive_definite`

ILUT

ILUT 是一种根据分解过程中得到的元素大小进行丢弃的算法。当前仅支持串行求解场景。

```
--pc_ilut_drop_tol
```

- 设置值丢弃阈值,，类型为非负实数，默认值为 `1e-5`，取值范围建议在 $(0, 10)$ 之间

```
--pc_ilut_max_fillin
```

- 设置 fill-in 在 L 和 U 的个数限制，类型为非负整数，默认值为 `20`，特殊情况：当设置为-1 时表示没有个数限制

```
--pc_ilut_scale_type
```

- 设置 scale 类型，默认值为 `norm_inf`，可选值为 `none`，`norm_inf`，`norm_1`，`ras`，`rss`，`hybrid`

```
--pc_ilut_shift_type
```

- 设置 shift 类型，默认值为 `none`，可选值为 `none`，`nonzero`，`positive_definite`

ILUTP

会在每一步分解后将最大元素和当前对角元置换，耗时比 ILUT 更长，但分解会更稳定。当前仅支持串行求解场景

```
--pc_ilutp_drop_tol
```

- 设置值丢弃阈值,，类型为非负实数，默认值为 `1e-5`，取值范围建议在 $(0, 10)$ 之间

```
--pc_ilutp_max_fillin
```

- 设置 fill-in 在 L 和 U 的个数限制，类型为非负整数，默认值为 `20`，特殊情况：当设置为-1 时表示没有个数限制

```
--pc_ilutp_pivot_tol
```

- 设置主元转置的阈值，类型为非负实数，表示主元转置仅对满足大于 $|val| * tol$ 的对角元生效

```
--pc_ilutp_mbloc
```

- 设置主元转置的范围，类型为非负整数，默认值为 0(无范围限制)，如果是一个正整数，则表示只会在当前大小为 mloc 的块范围内寻找最大主元

ILUC

具有更丰富的丢弃策略，耗时比 ILUT 略高，适合非正定矩阵。当前仅支持串行求解场景

```
--pc_iluc_drop_tol
```

- 设置值丢弃阈值,，类型为非负实数，默认值为 1e-5 , 取值范围建议在 $(0, 10)$ 之间

```
--pc_iluc_max_fillin
```

- 设置 fill-in 在 L 和 U 的个数限制，类型为非负整数，默认值为 20，特殊情况：当设置为-1 时表示没有个数限制

```
--pc_iluc_compensate
```

- 设置是否对分解后的上三角矩阵对角元进行补偿，默认值为 false，可选值为 true，false

```
--pc_iluc_drop_val_type_l
--pc_iluc_drop_val_type_u
```

- 针对下/上三角矩阵的基于值的丢弃策略类型，默认值为 2，可选值为 0,1,2,3。其中：0 表示元素当 val < drop_tol 会被丢弃；1 表示和 ILUT 类似的丢弃策略；2 表示基于条件数估计的丢弃策略；3 是使用基于更精确的条件数估计的丢弃策略。

DILU

分布式 ILU 预处理子

```
--pc_dilu_bj
```

- 是否 block jacobi 模式，默认值为 false，可选值为 true，false。当为 true 时表示没有 schur 矩阵

```
--pc_dilu_schur_pc_type
```

- 设置 schur 矩阵求解的 pc 类型，当前仅支持 ilut

```
--pc_dilu_schur_ksp_type
```

- 设置 schur 矩阵求解的迭代法类型，默认值为 gmres，可选值为 gmres，bicgstab

```
--pc_dilu_schur_max_iter
```

- 设置 schur 矩阵求解最大迭代次数，类型为正整数，默认值为 5，取值范围建议在 50 以内

```
--pc_dilu_pre_ordering
```

- 设置每个进程的矩阵重排方法，默认值为 0，可选值为 0,1,2。其中 0 表示无重排，1 表示 RCM 重排，2 表示 MWM(Maximum Weighted Matching) 重排

```
--pc_dilu_inner_ordering
```

- 设置每个进程的 inner 矩阵重排方法，默认值为 0，可选值为 0,1。其中 0 表示无重排，1 表示 RCM 重排

AMG 分解预处理子关联参数

当前版本支持 Classical AMG 和 Agg-based AMG 两种类型

```
--pc_type amg      //Classical AMG
--pc_type amg_agg  //Agg-based AMG
```

通用参数

“xx”需要根据当前采用的 AMG 类型替换为“amg”或者“amg_agg”

```
--pc_xx_cycle_type
```

- 设置循环类型，默认值为 multv，串行可选值为 additive,multv，分布式可选值为 additive,multv，推荐值 multv。

```
--pc_xx_smoother_type
```

- 设置 smoother 类型，默认值为 cheby，串行可选值为 jacobi, sor, cheby，分布式可选值为 jacobi, sor, cheby，对于对称正定矩阵，推荐值为 cheby，否则，推荐值为 sor。

```
--pc_xx_smoother_max_ite
```

- 设置 smoother 迭代次数（前光滑子和后光滑子），类型为非负整数，默认值为 2，对于 Chebyshev 光滑，推荐值为 1，对于 SOR 光滑，推荐值为 1。

```
--pc_xx_num_lev_max
```

- 设置最大深度，类型为正整数，默认值为 10，经典 AMG 推荐值为 20，agg. AMG 推荐值为 10。

```
--pc_xx_coarse_eq_lim
```

- 设置最粗层矩阵大小阈值，类型为正整数，默认值为 50，推荐值为 50。

```
--pc_xx_thresh
```

- 设置丢弃小的非零元的相对阈值，类型为实数，默认值为 0，经典 AMG 推荐值为 0.25，agg. AMG 推荐值为 0。

```
--pc_xx_jacobi_interp
```

- 设置插值算子的 jacobi 迭代光滑迭代次数，类型为非负整数，默认值为 0。对于不能收敛的状况，尝试使用 jacobi interpolation，即将参数“–pc_amg_jacobi_interp”设置为大于 0 的数值。经典 AMG 推荐值为 0。agg. AMG 推荐值为 1。

```
--pc_xx_interp_is_trunc
```

- 设置是否截断插值矩阵，默认值为 true，可选值为 true，false。当 pc_amg_jacobi_interp=0 时没有影响。

```
--pc_xx_interp_trunc_num
```

- 设置插值矩阵执行截断时每行最大非零元数，类型为非负整数，默认值为 4，建议不要设置为 0。当 pc_amg_jacobi_interp=0 时没有影响。

```
--pc_xx_interp_trunc_tol
```

- 设置丢弃插值矩阵非零小值的容差，类型为非负实数，默认值为 0。当 pc_amg_jacobi_interp=0 时没有影响。

```
--pc_xx_coarsen_type
```

- 设置粗化算法类型，默认值为 mis，可选值为 rs，mis，pmis，经典 AMG 推荐值为 pmis 或者 mis，agg. AMG 推荐值为 mis。

```
--pc_xx_restrict_type
```

- 通过插值算子，计算限制算子的方法。取值为 transpose 和 conjtranspose，默认值为 transpose，推荐值为 transpose。

```
--pc_xx_restrict_storage_type
```

- 设置是否显式地存储限制算子。取值为 explicit 和 onthefly，默认值为 explicit，推荐值为 onthefly。

```
--pc_xx_assumed_sym_type
```

- 设置对称类型。取值为 general,symmetric,hermitian，默认值为 general，推荐值为 symmetric。

```
--pc_verbose value
```

- 设置是否打印打印额外的 AMG 预条件信息，包括每一层的信息，比如时间等。value 参数的类型为 0 或 1。默认为 0。推荐需要 Debug AMG 预处理子的时候设置为 1。

Classical AMG

```
--pc_amg_interp_type
```

- 设置插值类型，默认值为 direct，串行可选值为 direct，extplusi，multipass，分布式仅支持 direct。默认值为 direct，推荐值为 direct。

```
--pc_amg_diag_dom_ratio
```

- 设置强度函数应用时的对角占优行检查比率，1 表示不做检查。类型为非负实数，默认值 0.9，推荐值 0.9。

```
--pc_amg_coarsen_strength_type
```

- 粗化算法中，用于生成强连接矩阵的强度函数类型。取值为 both,neg,posneg,hybrid，默认值为 hybrid，推荐值为 both 或者 posneg。

```
--pc_amg_coarsen_strength_type
```

- 插值算法中，用于生成强连接矩阵的强度函数类型。取值为 both,neg,posneg,hybrid，默认值为 hybrid，推荐值为 both 或者 posneg。

Agg-based AMG

结构问题推荐使用 Agg-based AMG，并设置 block_size 为结构问题维度

```
--pc_amg_agg_agg_type
```

- 设置聚集类型，当前仅支持 aggressive

```
--pc_amg_agg_coarsen_type
```

- 设置粗化类型，当前支持 mis 和 pmis

```
--pc_amg_agg_prolong_type
```

- 设置扩展类型，当前仅支持 agg

```
--pc_amg_agg_num_aggressive_layers
```

- 设置聚集层数，类型为正整数，默认值为 1，推荐值为 1。

```
--pc_amg_agg_block_size
```

- 设置网格维度，默认值为 1，可选值为 1,2,3，推荐值为 1。

```
--pc_amg_agg_assumed_sym_type
```

- 设置输入矩阵的对称性，默认值为 general，可选值为 general,symmetric,hermitian。当设置 symmetric 时，将不会计算 strength 和 graph 过程中的转置。

```
--pc_amg_agg_assumed_restrict_storage_type
```

- 设置输入矩阵的对称性，默认值为 explicit，可选值为 explicit,onthefly。当设置 explicit 时，将显式计算和保存限制矩阵为插值矩阵的转置；而设置 onthefly 时，将不会显式计算和保存限制矩阵，而仅使用转置算子来参与计算，从而节省内存开销。

```
--pc_amg_agg_scale_correction
```

- 设置是否执行放缩校正。取值为 false 和 true，默认值为 false，推荐值为 false。

```
--pc_amg_agg_pre_strength_sym_type
```

- 设置计算强度函数前是否对称化矩阵。取值为 general 和 symmetrize，默认值为 symmetrize，推荐值为 symmetrize。

FieldSplit 预处理子关联参数

当前版本支持 Additive、Multiplicative 和 Schur 三种类型的 FieldSplit 块预处理子。当前仅支持串行求解场景。

```
--pc_type fs_additive          //Additive
--pc_type fs_multiplicative    //Multiplicative
--pc_type fs_schur             //Schur
```

通用参数

“xx” 占位符将被替换为"--pc_type" 设置的具体预处理子，即为 fs_additive、fs_multiplicative 和 fs_schur 其中之一

```
--pc_xx_input_mat_format
```

- 设置待求解矩阵的组装方式，用户需要指定值，可选值为 large、small、llarge、general

```
--pc_xx_general_type
```

- 当组装方式设为 general 时，可以通过设置该选项为 large、small、llarge，来模拟相应的矩阵组装方式，它是通过直接设置 Index Set 来实现的

```
--pc_xx_local_index_set
```

- 对于 general 组装方式，设置的 Index Set 是否是切分后的局部 index set，可选值为 `true`、`false`，默认值为 `true`

```
--pc_xx_need_proc_comm
```

- 对于 general 组装方式，设置切分过程中是否会涉及通信，可选值为 `true`、`false`，默认值为 `true`

```
--pc_xx_outer_splits
```

- 设置矩阵切分方式，用户需要指定值，类型为以半角逗号分割的正整数，比如`"1,1,1"`。同时整数之和应为矩阵求解的场变量数，且矩阵的行 (列) 数应为该和的整数倍。需要注意的是，当采用 Schur 预处理子时，只允许将矩阵切分为 2 个 splits。
- 举例：对于包含 2 个速度分量和 1 个压力分量的 2D CFD 问题，若耦合总刚矩阵的行列按照`"u1, u2, ..., un; v1, v2, ..., vn; p1, p2, ..., pn"` 或`"u1, v1, w1; ...; un, vn, wn; p1, p2, ..., pn"` 顺序排列，则可设`"--pc_xx_input_mat_format large --pc_xx_input_mat_format 2,1"`；若总刚矩阵的行列按照`"u1, v1, p1, u2, v2, p2, ..., un, vn, pn"` 顺序排列，则可设`"--pc_xx_input_mat_format small --pc_xx_input_mat_format 2,1"`；若每个进程中均按照`"u1, u2, ..., un; v1, v2, ..., vn; p1, p2, ..., pn"` 或`"u1, v1, w1; ...; un, vn, wn; p1, p2, ..., pn"` 顺序排列，则可设`"--pc_xx_input_mat_format llarge --pc_xx_input_mat_format 2,1"`；此外，用户也可以通过`"--pc_xx_input_mat_format general"` 设置通用的切分方式，但这需要用户通过 `SetFieldSplitIndexSet()` 接口指定每个 split 对应的全局行/列号。
- 针对`"--pc_xx_input_mat_format general"` 的情况，要求用户在执行 `Solve()` 接口之前，依次用 `SetFieldSplitIndexSet(splitIS, splitDim)` 接口指定 fieldsplit_0、fieldsplit_1、fieldsplit_2 等所有 split 对应的全局行/列号；当需要重新设置时，需要在设置 fieldsplit_0 时使用 `SetFieldSplitIndexSet(splitIS, splitDim, true)` 接口，其它 fieldsplit_x 使用 `SetFieldSplitIndexSet(splitIS, splitDim)` 接口。

```
--pc_xx_fieldsplit_ksp_type
```

- 设置求解所有 splits 矩阵的迭代法，默认值为 `gmres`，可选值参考迭代法参数说明-创建 Krylov 迭代法求解算法章节

```
--pc_xx_fieldsplit_i_ksp_type
```

- 设置求解第 i 个 split 矩阵的迭代法，默认值为 `gmres`，可选值参考迭代法参数说明-创建 Krylov 迭代法求解算法章节

```
--pc_xx_fieldsplit_pc_type
```

- 设置求解所有 splits 矩阵的预处理子，默认值为 `jacobi`，可选值参考预处理子参数说明-创建预处理子章节

```
--pc_xx_fieldsplit_i_pc_type
```

- 设置求解第 i 个 split 矩阵的预处理子，默认值为 `jacobi`，可选值参考预处理子参数说明-创建预处理子章节

Additive

除通用参数外，没有额外需要设置的参数。

Multiplicative

```
--pc_fs_multiplicative_multiplicative_type
```

- 设置迭代方式，默认值为 `symmetric`，可选值为 `forward`, `backward`, `symmetric`

Schur

Schur 预处理子只允许将矩阵切分为 2 个 splits，即`"--pc_fs_schur_outer_splits"` 后只能跟两个以半角逗号分隔的正整数，如`"2,1"`，`"3,1"`，`"2,2"`

```
--pc_fs_schur_schur_fact_type
```

- 设置矩阵近似分解类型，默认值为 `full`，可选值为 `diag`, `lower`, `upper`, `full`

```
--pc_fs_schur_schur_pre_type
```

- 设置 Schur 矩阵所需要的预处理矩阵，默认值为 `a11`，可选值为 `a11`, `self`, `user`, `selfp`, `full`

```
--pc_fs_schur_inner_ksp_type
```

- 设置求解 Schur 矩阵 $(A_{00} - A_{10}A_{00}^{-1}A_{01})$ 中 A_{00}^{-1} 的迭代法类型，默认值为 `gmres`，可选值参考迭代法参数说明-创建 Krylov 迭代法求解算法章节

```
--pc_fs_schur_inner_pc_type
```

- 设置求解 Schur 矩阵 $(A_{00} - A_{10}A_{00}^{-1}A_{01})$ 中 A_{00}^{-1} 的预处理子类型，默认值为 `jacobi`，可选值参考预处理子参数说明-创建预处理子章节

```
--pc_fs_schur_fieldsplit_i_diag_dom_ratio
```

- 类型为非负实数

AMG 特化参数

```
--pc_xx_fieldsplit_i_cycle_type              参          见--pc_amg_cycle_type。
--pc_xx_fieldsplit_i_smoother_type           参       见--pc_amg_smoother_type。
--pc_xx_fieldsplit_i_smoother_max_ite        参    见--pc_amg_smoother_max_ite。
--pc_xx_fieldsplit_i_num_lev_max             参         见--pc_amg_num_lev_max。
--pc_xx_fieldsplit_i_coarse_eq_lim           参       见--pc_amg_coarse_eq_lim。
--pc_xx_fieldsplit_i_thresh 参见--pc_amg_thresh。--pc_xx_fieldsplit_i_diag_dom_ratio
参              见--pc_amg_diag_dom_ratio。--pc_xx_fieldsplit_i_jacobi_interp
参      见--pc_amg_jacobi_interp。--pc_xx_fieldsplit_i_coarsen_type        参
见--pc_amg_coarsen_type。--pc_xx_fieldsplit_i_coarsen_strength_type        参
见--pc_amg_coarsen_strength_type。--pc_xx_fieldsplit_i_interp_strength_type
参   见--pc_amg_interp_strength_type。--pc_xx_fieldsplit_i_restrict_type    参
见--pc_amg_restrict_type。--pc_xx_fieldsplit_i_restrict_storage_type       参
见--pc_amg_restrict_storage_type。--pc_xx_fieldsplit_i_assumed_sym_type
参              见--pc_amg_assumed_sym_type。--pc_xx_fieldsplit_i_interp_type
参      见--pc_amg_interp_type。--pc_xx_fieldsplit_i_interp_is_trunc        参
见--pc_amg_interp_is_trunc。--pc_xx_fieldsplit_i_interp_trunc_num          参
```

见--pc_amg_interp_trunc_num。--pc_xx_fieldsplit_i_interp_trunc_tol 参见--pc_amg_interp_trunc_tol。--pc_verbose 参见--pc_verbose。

agg. AMG 特化参数

--pc_fs_schur_fieldsplit_i_cycle_type 参见--pc_amg_agg_cycle_type。
--pc_fs_schur_fieldsplit_i_smoother_type 参见--pc_amg_agg_smoother_type。
--pc_fs_schur_fieldsplit_i_smoother_max_ite 参见--pc_amg_agg_smoother_max_ite。
--pc_fs_schur_fieldsplit_i_num_lev_max 参见--pc_amg_agg_num_lev_max。
--pc_fs_schur_fieldsplit_i_coarse_eq_lim 参见--pc_amg_agg_coarse_eq_lim。
--pc_fs_schur_fieldsplit_i_thresh 参见--pc_amg_agg_thresh。
--pc_fs_schur_fieldsplit_i_jacobi_interp 参见--pc_amg_agg_jacobi_interp。
--pc_fs_schur_fieldsplit_i_coarsen_type 参见--pc_amg_agg_coarsen_type。
--pc_fs_schur_fieldsplit_i_block_size 参见--pc_amg_agg_block_size。
--pc_fs_schur_fieldsplit_i_restrict_type 参见--pc_amg_agg_restrict_type。
--pc_fs_schur_fieldsplit_i_restrict_storage_type 参见--pc_amg_agg_restrict_storage_type。
--pc_fs_schur_fieldsplit_i_assumed_sym_type 参见--pc_amg_agg_assumed_sym_type。
--pc_fs_schur_fieldsplit_i_scale_correction 参见--pc_amg_agg_scale_correction。
--pc_fs_schur_fieldsplit_i_agg_type 参见--pc_amg_agg_agg_type。
--pc_fs_schur_fieldsplit_i_num_aggressive_layers 参见--pc_amg_agg_num_aggressive_layers。
--pc_fs_schur_fieldsplit_i_pre_strength_sym_type 参见--pc_amg_agg_pre_strength_sym_type。
--pc_verbose 参见--pc_verbose。

3.2.6 样例

串行样例

```
#include <NCSIterativeSolver.h>
using std::string;
using namespace NCS::KSP;
int main(int argc, char *argv[])
{
    /* -------------------- 设置系数矩阵及右端项 ---------------------- */
    // A matrix
    // [1 0 1]
    // [0 2 1]
    // [2 0 1]
    double data[6] = {1., 1., 2., 1., 2., 1.};
    int colIdx[6] = {0, 2, 1, 2, 0, 2};
    int rowPtr[4] = {0, 2, 4, 6};
    // right hand side
    double b[3] = {2., 3., 3.};
    double iniSol[3] = {0., 0., 0.};
    double sol[3];

    /* ----------------------  设置求解参数 ------------------------- */
    NCSIterativeSolver<double> linearSolver;
    linearSolver.SetOption("--ksp_type", "gmres"); // ksp type, mandatory para
    linearSolver.SetOption("--pc_type", "jacobi"); // pc type, mandatory para
    linearSolver.SetOption("--ksp_max_iteration", "10000");
    linearSolver.SetOption("--pc_jacobi_type", "diagonal");

    /* --------------------------- 求解 ---------------------------- */
    auto status = linearSolver.Solve(data, colIdx, rowPtr, 3, b, iniSol, sol);
    int iterNum = linearSolver.GetIter();
    auto res = linearSolver.GetTrueResNorm();
    return 0;
}
```

使用 visual studio 的用户，需要调用相应的 C 接口，具体样例如下：

```
#include <string>
#include <NCSIterativeSolverWrapper.h>

using std::string;
using namespace NCS::KSP;

int main(int argc, char *argv[]) {

    auto dataType = DataType::DOUBLE;
        /* -------------------- 设置系数矩阵及右端项---------------------- */
        // A matrix
        // [1 0 1]
        // [0 2 1]
        // [2 0 1]
        double data[6] = { 1., 1., 2., 1., 2., 1. };
        int colIdx[6] = { 0, 2, 1, 2, 0, 2 };
        int rowPtr[4] = { 0, 2, 4, 6 };
        // right hand side
        double b[3] = { 2., 3., 3. };
        double iniSol[3] = { 0., 0., 0. };
        double sol[3];

        //使用 C 接口创建 NCSIterativeSolver 实例
        NCSIterativeSolverWrapper* linearSolverWrapper = LinearSolver_
↪new(DataType::DOUBLE);
        /* ----------------------  设置求解参数 ------------------------- */
        LinearSolverSetOption(dataType, linearSolverWrapper, "--ksp_type", "gmres
↪");
        LinearSolverSetOption(dataType, linearSolverWrapper, "--pc_type", "jacobi
↪");
        LinearSolverSetOption(dataType, linearSolverWrapper, "--ksp_max_iteration",
↪ "10000");
        LinearSolverSetOption(dataType, linearSolverWrapper, "--pc_jacobi_type",
↪"diagonal");

        /* ---------------------------- 求解------------------------------ */
        auto status = LinearSolverSolve_double(dataType, linearSolverWrapper,
                                                                              ␣
↪data, colIdx, rowPtr, 3, b, iniSol, sol);
        int iterNum = LinearSolverGetIter(dataType, linearSolverWrapper);
        auto res = LinearSolverGet(dataType, linearSolverWrapper,␣
↪Get::GetTrueResNorm);
        auto res2 = LinearSolverGet(dataType, linearSolverWrapper,␣
↪Get::GetResNorm);

        LinearSolver_delete(linearSolverWrapper);

        return 0;
}
```

对于复数情形，dataType 取 DataType::COMPLEX_DOUBLE，输入参数由 double 类型改为 cComplex_double 结构体，然后调用接口相应的复数版本即可。

MPI 分布式求解示例

使用分布式超节点法、2 进程求解：

集中式输入示例

只在主进程输入完整矩阵。

```
#include <NCSIterativeSolver.h>
using std::string;
using namespace NCS::KSP;
int main(int argc, char *argv[])
{
    /* ------------------------- 初始化MPI ------------------------- */
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    /* -------------------- 设置系数矩阵及右端项 --------------------- */
    std::vector<int> rowPtr, colIdx;
    std::vector<double> data, rhsValues, iniSol, sol;
    // 只在主进程输入完整的系数矩阵及右端项矩阵
    // A = [1 0 1]
    //     [0 2 1]
    //     [2 0 1]
    if (rank == 0) {
        rowPtr = {0, 2, 4, 6};
        colIdx = {0, 2, 1, 2, 0, 2};
        data = {1., 1., 2., 1., 2., 1.};
        rhsValues = {2., 3., 3.};
        iniSol = {0., 0., 0.};
    }
    sol.resize(3);

    /* ----------------------  设置求解参数 ------------------------- */
    // ksp&pc 设置
    NCSIterativeSolver<double> linearSolver;
    linearSolver.SetOption("--ksp_type", "gmres"); // ksp type, mandatory para␣
↪必填参数
    linearSolver.SetOption("--pc_type", "jacobi"); // pc type, mandatory para␣
↪必填参数
    linearSolver.SetOption("--ksp_max_iteration", "10000");
    linearSolver.SetOption("--pc_jacobi_type", "diagonal");

    // 分布式求解参数
    linearSolver.SetOption("--is_rank0_input_only", "true"); //␣
↪是否仅有rank0输入矩阵, mandatory para必填参数
    linearSolver.SetOption("--global_dimension", "3"); // global dimension,␣
↪mandatory para

    /* ---------------------------- 求解 ----------------------------- */
    auto status = linearSolver.Solve(data.data(), colIdx.data(), rowPtr.data(), 3,␣
↪rhsValues.data(), iniSol.data(), sol.data());

    /* ---------------------------- 输出求解结果 ----------------------------- */
    if (rank == 0) {
        for (int i = 0; i < 3; ++i) {
            std::cout << sol[i] << "\t";
        }
        std::cout << std::endl;
        std::cout << "求解状态为: " << status << std::endl;
```

(续下页)

(接上页)

```
        std::cout << "迭代次数为: " << linearSolver.GetIter() << std::endl;
        std::cout << "求解残差: " << linearSolver.GetResNorm() << std::endl;
        std::cout << "真实残差: " << linearSolver.GetTrueResNorm() << std::endl;
    }

    /* ------------------------- 结束MPI ---------------------------- */
    MPI_Finalize();

    return 0;
}
```

使用 visual studio 的用户，需要调用相应的 C 接口，具体样例如下：

```
#include <string>
#include <iostream>
#include <mpi.h>
#include <NCSIterativeSolverWrapper.h>
using std::string;
using namespace NCS::KSP;

int main(int argc, char *argv[])
{
        auto dataType = DataType::DOUBLE;
        /* ------------------------- 初始化MPI ---------------------------- */
        MPI_Init(&argc, &argv);
        int rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        /* -------------------- 设置系数矩阵及右端项--------------------- */
        std::vector<int> rowPtr, colIdx;
        std::vector<double> data, rhsValues, iniSol, sol;
        // 只在主进程输入完整的系数矩阵及右端项矩阵
        // A = [1 0 1]
        //     [0 2 1]
        //     [2 0 1]
        if (rank == 0) {
                rowPtr = { 0, 2, 4, 6 };
                colIdx = { 0, 2, 1, 2, 0, 2 };
                data = { 1., 1., 2., 1., 2., 1. };
                rhsValues = { 2., 3., 3. };
                iniSol = { 0., 0., 0. };
        }
        sol.resize(3);

        /* ----------------------- 设置求解参数------------------------- */
        // ksp&pc 设置
        NCSIterativeSolverWrapper* linearSolverWrapper = LinearSolver_
↪new(dataType);
        LinearSolverSetOption(dataType, linearSolverWrapper, "--ksp_type", "gmres
↪");
        LinearSolverSetOption(dataType, linearSolverWrapper, "--pc_type", "jacobi
↪");
        LinearSolverSetOption(dataType, linearSolverWrapper, "--ksp_max_iteration",
↪ "10000");
        LinearSolverSetOption(dataType, linearSolverWrapper, "--pc_jacobi_type",
↪"diagonal");
        LinearSolverSetOption(dataType, linearSolverWrapper, "--is_rank0_input_only
↪", "true");
        LinearSolverSetOption(dataType, linearSolverWrapper, "--global_dimension",
↪"3");
```

(续下页)

(接上页)

```
        /* ---------------------------- 求解 ----------------------------- */
        auto status = LinearSolverSolve_double(dataType, linearSolverWrapper,
↪data.data(), colIdx.data(), rowPtr.data(), 3,
↪rhsValues.data(), iniSol.data(), sol.data());

        /* ---------------------------- 输出求解结果----------------------------
↪*/
        if (rank == 0) {
                for (int i = 0; i < 3; ++i) {
                        std::cout << sol[i] << "\t";
                }
                int iterNum = LinearSolverGetIter(dataType, linearSolverWrapper);
                auto res = LinearSolverGet(dataType, linearSolverWrapper,
↪Get::GetTrueResNorm);
                auto res2 = LinearSolverGet(dataType, linearSolverWrapper,
↪Get::GetResNorm);
                std::cout << std::endl;
                std::cout << "求解状态为: " << status << std::endl;
                std::cout << "迭代次数为: " << iterNum << std::endl;
                std::cout << "求解残差: " << res2 << std::endl;
                std::cout << "真实残差: " << res << std::endl;
        }

        LinearSolver_delete(linearSolverWrapper);
        /* ---------------------------- 结束MPI ----------------------------- */
        MPI_Finalize();

        return 0;
}
```

对于复数情形，dataType 取 DataType::COMPLEX_DOUBLE，输入参数由 double 类型改为 cComplex_double 结构体，然后调用接口相应的复数版本即可。

分布式输入示例

进程 0 负责输入 $[0,2)$ 行，进程 1 负责输入 $[2,4)$。

```
int main(int argc, char *argv[])
{
    using namespace NCS::KSP;

    /* ------------------------- 初始化MPI ---------------------------- */
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int worldSize;
    MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
    assert(worldSize == 2);

    /* -------------------- 设置系数矩阵及右端项 ---------------------- */
    int dim;
    std::vector<int> rowPtr, colIdx;
    std::vector<double> data, rhsValues, iniSol, sol;
    if (rank == 0) {
        rowPtr = {0, 2, 4};
        colIdx = {0, 2, 1, 2};
        data = {1., 1., 2., 1.};
        rhsValues = {2., 3.};
```

(续下页)

(接上页)

```
        iniSol = {0., 0.};
    } else {
        rowPtr = {0, 2};
        colIdx = {0, 2};
        data = {2., 1.};
        rhsValues = {3.};
        iniSol = {0.};
    }
    sol.resize(3);

    /* ------------------------  设置求解参数 ------------------------- */
    NCSIterativeSolver<double> linearSolver;

    // ksp&pc 关键参数设置
    linearSolver.SetOption("--ksp_type", "gmres"); // ksp type, mandatory para
    linearSolver.SetOption("--pc_type", "jacobi"); // pc type, mandatory para
    linearSolver.SetOption("--ksp_max_iteration", "10000");
    linearSolver.SetOption("--pc_jacobi_type", "diagonal");

    // 分布式求解设置参数
    linearSolver.SetOption("--global_dimension", "3"); // global dimension,␣
↪mandatory para
    linearSolver.SetOption("--is_rank0_input_only", "false"); // distributed␣
↪matrix input, mandatory para

    if (rank == 0) {
        // 进程0负责[0,2)行
        dim = 2;
        linearSolver.SetOption("--cur_start_index", "0"); // start row index,␣
↪mandatory para
    } else {
        // 进程1负责[2,3)行
        dim = 1;
        linearSolver.SetOption("--cur_start_index", "2"); // start row index,␣
↪mandatory para
    }

    /* ---------------------------- 求解 ----------------------------- */
    auto status =
            linearSolver.Solve(data.data(), colIdx.data(), rowPtr.data(), dim,␣
↪rhsValues.data(), iniSol.data(), sol.data());

    /* ---------------------------- 输出求解结果 ----------------------------- */
    if (rank == 0) {
        for (int i = 0; i < 3; ++i) {
            std::cout << sol[i] << "\t";
        }
        std::cout << std::endl;
        std::cout << "求解状态为: " << status << std::endl;
        std::cout << "迭代次数为: " << linearSolver.GetIter() << std::endl;
        std::cout << "求解残差: " << linearSolver.GetResNorm() << std::endl;
        std::cout << "真实残差: " << linearSolver.GetTrueResNorm() << std::endl;
    }

    /* ---------------------------- 结束MPI ----------------------------- */
    MPI_Finalize();

    return 0;
}
```

使用 visual studio 的用户，需要调用相应的 C 接口，具体样例如下：

```
#include <string>
#include <iostream>
#include <assert.h>
#include <mpi.h>
#include <NCSIterativeSolverWrapper.h>

using std::string;
using namespace NCS::KSP;
int main(int argc, char *argv[])
{
        auto dataType = DataType::DOUBLE;
        /* ------------------------ 初始化MPI ------------------------ */
        MPI_Init(&argc, &argv);
        int rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        int worldSize;
        MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
        assert(worldSize == 2);

        /* -------------------- 设置系数矩阵及右端项-------------------- */
        int dim;
        std::vector<int> rowPtr, colIdx;
        std::vector<double> data, rhsValues, iniSol, sol;
        if (rank == 0) {
                rowPtr = { 0, 2, 4 };
                colIdx = { 0, 2, 1, 2 };
                data = { 1., 1., 2., 1. };
                rhsValues = { 2., 3. };
                iniSol = { 0., 0. };
        } else {
                rowPtr = { 0, 2 };
                colIdx = { 0, 2 };
                data = { 2., 1. };
                rhsValues = { 3. };
                iniSol = { 0. };
        }
        sol.resize(3);

        /* ----------------------- 设置求解参数------------------------ */
        // ksp&pc 设置
        NCSIterativeSolverWrapper* linearSolverWrapper = LinearSolver_
↪new(dataType);
        LinearSolverSetOption(dataType, linearSolverWrapper, "--ksp_type", "gmres
↪");
        LinearSolverSetOption(dataType, linearSolverWrapper, "--pc_type", "jacobi
↪");
        LinearSolverSetOption(dataType, linearSolverWrapper, "--ksp_max_iteration",
↪ "10000");
        LinearSolverSetOption(dataType, linearSolverWrapper, "--pc_jacobi_type",
↪"diagonal");
        // 分布式求解设置参数
        LinearSolverSetOption(dataType, linearSolverWrapper, "--global_dimension",
↪"3");      // global dimension, mandatory para
        LinearSolverSetOption(dataType, linearSolverWrapper, "--is_rank0_input_only
↪", "false");    // // distributed matrix input, mandatory para

        if (rank == 0) {
                // 进程0负责[0,2)行
                dim = 2;
                LinearSolverSetOption(dataType, linearSolverWrapper, "--cur_start_
↪index", "0");     // start row index,mandatory para
        } else {
```

(续下页)

(接上页)

```
                // 进程1负责[2,3)行
                dim = 1;
                LinearSolverSetOption(dataType, linearSolverWrapper, "--cur_start_
↪index", "2");   // start row index,mandatory para
        }

        /* ---------------------------- 求解 ---------------------------- */
        auto status = LinearSolverSolve_double(dataType, linearSolverWrapper,
                data.data(), colIdx.data(), rowPtr.data(), dim, rhsValues.data(),␣
↪iniSol.data(), sol.data());

        /* ---------------------------- 输出求解结果----------------------------␣
↪*/
        if (rank == 0) {
                for (int i = 0; i < 3; ++i) {
                        std::cout << sol[i] << "\t";
                }
                int iterNum = LinearSolverGetIter(dataType, linearSolverWrapper);
                auto res = LinearSolverGet(dataType, linearSolverWrapper,␣
↪Get::GetTrueResNorm);
                auto res2 = LinearSolverGet(dataType, linearSolverWrapper,␣
↪Get::GetResNorm);
                std::cout << std::endl;
                std::cout << "求解状态为: " << status << std::endl;
                std::cout << "迭代次数为: " << iterNum << std::endl;
                std::cout << "求解残差: " << res2 << std::endl;
                std::cout << "真实残差: " << res << std::endl;
        }

        LinearSolver_delete(linearSolverWrapper);
        /* ---------------------------- 结束MPI ---------------------------- */
        MPI_Finalize();

        return 0;
}
```

对于复数情形，dataType 取 DataType::COMPLEX_DOUBLE，输入参数由 double 类型改为 cComplex_double 结构体，然后调用接口相应的复数版本即可。

3.3 非线性迭代法求解接口说明

3.3.1 非线性迭代法接口整体说明

非线性迭代法对用户提供的接口封装在 NCSNonLinearSolver.h 中，其具体的实现为：

```
/* 残差函数指针 */
template<typename Scalar>
using TargetFunPtr = void (*)(const Scalar* x, int n, Scalar*& residual);

/* Jacobian 矩阵函数指针 */
template<typename Scalar>
using JacobianFunPtr = void (*)(const Scalar* x, int n, int*& ptr, int*& ids,␣
↪Scalar*& values);

/* 非线性求解类 */
template<typename Scalar>
class NCSNonLinearSolver {
public:
```

(续下页)

(接上页)

```
    /* 默认构造函数 */
    NCSNonLinearSolver() = default;
    /* 初始解设置接口 */
    void SetInitSol(int n, const Scalar* sol);
    /* 求解接口 */
    SolveStateEnum Solve(int n, Scalar* sol);
    /* 残差函数设置接口 */
    bool SetTargetFun(TargetFunPtr<Scalar> targetFunPtr);
    /* Jacobian 矩阵函数设置接口 */
    bool SetJacobianFun(JacobianFunPtr<Scalar> jacobianFunPtr);
    /* 参数配置接口 */
    void SetOption(const std::string& opt, const std::string& value);
    /* 通过文件配置参数接口 */
    void SetFromFile(const std::string& filename);
    /* 迭代残差返回接口 */
    double GetResNorm() const;
    /* 获取迭代步数 */
    int GetIter() const;
}
```

这是一个模板类，模板参数为 Scalar，具体的取值为：

- `double`：实数双精度
- `std::complex<double>`：复数双精度

针对不同模板参数，创建求解对象实例的具体实现分别为，

```
NCSNonLinearSolver<double> nonLinearRealSolver;                 // 实数求解对象
NCSNonLinearSolver<std::complex<double>> nonLinearComplexSolver; // 复数求解对象
```

对于使用 visual studio 的用户，提供的 C 接口封装在 NCSNonLinearSolverWrapper.h 中，其具体的实现为：

```
#include "NCSNonLinearSolver.h"

namespace NCS::CNLS {

enum class DataType : int32_t { DOUBLE = 0, COMPLEX_DOUBLE = 2, };

extern "C" {

struct DLLEXPORT NCSNonLinearSolverWrapper;

struct cComplex_double {
  double real;
  double imag;
};
/* 创建新的NonLinearSolver实例 */
DLLEXPORT NCSNonLinearSolverWrapper* NonLinearSolver_new(DataType dataType);
 /* 参数配置接口 */
DLLEXPORT void NonLinearSolverSetOption(DataType dataType,␣
↪NCSNonLinearSolverWrapper* wrapper,
  const char* copt, const char* cvalue);
/* 通过文件配置参数接口 */
DLLEXPORT void NonLinearSolverSetFromFile(DataType dataType,␣
↪NCSNonLinearSolverWrapper* wrapper,
  const char* cfilename);

DLLEXPORT void NonLinearSolverSetReportMethod(DataType dataType,␣
↪NCSNonLinearSolverWrapper* wrapper,
  void* cprogressFuncPtr);
/* double类型求解接口 */
```

(续下页)

(接上页)

```
DLLEXPORT SolveStateEnum NonLinearSolverSolve_double(DataType dataType,␣
↪NCSNonLinearSolverWrapper* wrapper,
  int n, double* sol);
/* 复数double类型求解接口 */
DLLEXPORT SolveStateEnum NonLinearSolverSolve_complex_double(DataType dataType,␣
↪NCSNonLinearSolverWrapper* wrapper,
  int n, cComplex_double* sol);

DLLEXPORT int NonLinearSolverAsynchCall_double(DataType dataType,␣
↪NCSNonLinearSolverWrapper* wrapper,
  NCSCommandEnum command, int n, double* sol);

DLLEXPORT int NonLinearSolverAsynchCall_complex_double(DataType dataType,␣
↪NCSNonLinearSolverWrapper* wrapper,
  NCSCommandEnum command, int n, cComplex_double* sol);
/* double类型残差函数设置接口 */
DLLEXPORT bool NonLinearSolverSetTargetFun_double(DataType dataType,␣
↪NCSNonLinearSolverWrapper* wrapper,
  TargetFunPtr<double> targetFunPtr);

DLLEXPORT bool cNonLinearSolverSetTargetFun_complex_double(DataType dataType,␣
↪NCSNonLinearSolverWrapper* wrapper,
  void* ctargetFunPtr);
/* double类型Jacobian 矩阵函数设置接口 */
DLLEXPORT bool NonLinearSolverSetJacobianFun_double(DataType dataType,␣
↪NCSNonLinearSolverWrapper* wrapper,
  JacobianFunPtr<double> jacobianFunPtr);

DLLEXPORT bool cNonLinearSolverSetJacobianFun_complex_double(DataType dataType,␣
↪NCSNonLinearSolverWrapper* wrapper,
  void* cjacobianFunPtr);
/* double类型初始解设置接口 */
DLLEXPORT void NonLinearSolverSetInitSol_double(DataType dataType,␣
↪NCSNonLinearSolverWrapper* wrapper,
  int n, const double* sol);
/* 复数double类型初始解设置接口 */
DLLEXPORT void NonLinearSolverSetInitSol_complex_double(DataType dataType,␣
↪NCSNonLinearSolverWrapper* wrapper,
  int n, const cComplex_double* sol);
/* 获取迭代步数 */
DLLEXPORT int NonLinearSolverGetIter(DataType dataType, NCSNonLinearSolverWrapper*␣
↪wrapper);
/* 迭代残差返回接口 */
DLLEXPORT double NonLinearSolverGetResNorm(DataType dataType,␣
↪NCSNonLinearSolverWrapper* wrapper);
/* 删除已经创建的NonLinearSolver实例 */
DLLEXPORT void NonLinearSolver_delete(NCSNonLinearSolverWrapper* wrapper);

}
 /* 复数double类型残差函数设置接口 */
bool NonLinearSolverSetTargetFun_complex_double(DataType dataType,␣
↪NCSNonLinearSolverWrapper* wrapper,
  TargetFunPtr<std::complex<double>> targetFunPtr)
{
  // void (*ctargetFunPtr) = targetFunPtr;
  void* ctargetFunPtr = (void*)targetFunPtr;
  return cNonLinearSolverSetTargetFun_complex_double(dataType, wrapper,␣
↪ctargetFunPtr);
};
 /* 复数double类型Jacobian 矩阵函数设置接口 */
bool NonLinearSolverSetJacobianFun_complex_double(DataType dataType,␣
```

(续下页)

(接上页)

```
↪NCSNonLinearSolverWrapper* wrapper,
  JacobianFunPtr<std::complex<double>> jacobianFunPtr)
{
  void* cjacobianFunPtr = (void*)jacobianFunPtr;
  return cNonLinearSolverSetJacobianFun_complex_double(dataType, wrapper,␣
↪cjacobianFunPtr);
};
}
```

非线性迭代法一共对用户提供以下的几个接口，

接口类型 1：参数设置接口

参数	接口参数说明
void SetOption(const std::string& opt, const std::string& value)	-opt：参数的名字；-value：参数的值
void SetFromFile(const std::string& filename)	-filename：文件名/路径

示例

```
SetOption("--nls_max_iteration", "10000")
SetFromFile(filename)
```

接口类型 2：残差函数设置接口

参数	接口参数说明
bool SetTargetFun(TargetFunPtr<Scalar> targetFunPtr)	-targetFunPtr：残差函数指针；-返回是否设置成功

示例

```
SetTargetFun(&F)
-F 为非线性残差函数
```

接口类型 3：Jacobian 矩阵函数设置接口

参数	接口参数说明
bool SetJacobianFun(JacobianFunPtr<Scalar> jacobianFunPtr)	-jacobianFunPtr：jacobian 矩阵函数指针；-返回是否设置成功

示例

```
SetJacobianFun(&J)
-J为非线性Jacobian矩阵函数
```

接口类型 4：初始解设置接口

参数	接口参数说明
void SetInitSol(int n, const Scalar* sol)	-n：解的维度；-sol：[输入] 初始解向量

接口类型 5：求解接口

参数	接口参数说明
SolveStateEnum Solve(int n, Scalar* sol)	-n: 解的维度；-sol：[输出] 最终解 -返回求解的状态

接口类型 6：结果返回接口

参数	接口参数说明
int GetIter()	-返回迭代次数
double GetResNorm()	-返回求解残差

由上表可以看出，非线迭代法一共对用户提供六类接口：参数设置接口、残差函数设置接口、Jacobian 矩阵函数设置接口、初始解设置接口、求解接口以及结果返回接口，下面各章分别展开这六类接口的详细说明。

3.3.2 参数设置接口说明

参数设置接口有两个，一个是按照参数名和参数值的方式逐个设置（void SetOption(const std::string& opt, const std::string& value)），另一个是直接从文本读入（void SetFromFile(const std::string& filename)），一次性设置所有参数，下面分别进行说明。如果用户不设置参数，那么算法将采用默认的参数进行求解。

void SetOption(const std::string& opt, const std::string& value)

其中参数 `opt` 对应所设置参数的名字，`value` 对应参数的值。`opt` 的名字是“`--nlsT`”，这些参数不是必须设置的，算法会给定这些参数默认值。

```
NCSNonLinearSolver <double> nonLinearSolver;

/******** Set parameters for nonlinear solver ********/
nonLinearSolver.SetOption("--nls_line_search_type", "FullStep");
nonLinearSolver.SetOption("--nls_fullStep_length_value", "1.0");
nonLinearSolver.SetOption("--nls_linear_solver_type", "Supernodal");
nonLinearSolver.SetOption("--nls_direction_type", "Newton");
nonLinearSolver.SetOption("--nls_max_iteration", "1000");
nonLinearSolver.SetOption("--nls_tol", "1e-6");
nonLinearSolver.SetOption("--nls_rtol", "5e-3");
...
/******** Set log ********/
nonLinearSolver.SetOption("--nls_log_path", "/home/nonlinear/nonlinear.log");
nonLinearSolver.SetOption("—nls_log_level", "0");
```

非线性迭代法参数说明

线搜索:

参数设置接口 opt	value 示例	类型	默认值
`--nls_line_search_type`	`"FullStep"`	string	`"Backtrack"`

- 说明：线性搜索类型（不区分大小写）`"FullStep"` `"Backtrack"` `"MoreThuente"` `"Polynomial"`

FullStep 步长:

参数设置接口 opt	说明	value 示例	类型	默认值
`--nls_fullStep_length_value`	FullStep 线搜索的步长值	1.0	double	1.0

方向:

参数设置接口 opt	说明	value 示例	类型	默认值
`--nls_direction_type`	方向类型（不区分大小写）`"Newton"`	`"Newton"`	string	`"Newton"`

线性求解类型:

参数设置接口 opt	value 示例	类型	默认值
—nls_linear_solver_type	`"Supernodal"`	string	`"Supernodal"`

- 说明：线性求解器类型（不区分大小写）`"Supernodal"` `"KSP"`

注：当线性求解器类型为 KSP 时，支持 KSP 参数设置，详细参考“线性迭代法接口说明文档”。

迭代控制:

参数设置接口 opt	说明	value 示例	类型	默认值
`--nls_tol`	残差的绝对误差阈值	1e-6	double	1e-6
`--nls_rtol`	残差的相对误差阈值	1e-3	double	1e-3
`--nls_max_iteration`	最大迭代次数阈值，到达最大步数停止迭代	1000	int	10000

日志控制:

日志等级

参数设置接口 opt	value 示例	类型	默认值
`--nls_log_level`	1	int	1

- 说明：日志等级类型
  - -1: OFF
  - 0: ERROR
  - 1: WARN
  - 2: INFO

打印统计结果

参数设置接口 opt	value 示例	类型	默认值
`--nls_force_print_results`	1	bool	0

- 说明：打印迭代次数、残差、求解状态、求解时间等统计数据。

日志输出路径

参数设置接口 opt	value 示例	类型	默认值
`--nls_log_path`	`"/home/nonlinear.log"`	string	`"nonlinear.log"`

线程数设置:

参数设置接口 opt	说明	value 示例	类型	默认值
`--nls_threads`	线程数	4	int	1

分布式设置:

参数设置接口 opt	说明	类型	默认值
`--nls_mpi_run`	分布式开关，开启/关闭（1/0)	bool	0
`--nls_mpi_centralized_input`	解的集中式输入开关，开启/关闭（1/0)	bool	1
`--nls_global_dimension`	非线性方程组的全局维度	int	
`--nls_begin_row_index`	不同进程对应的 Jacobian 矩阵行起始全局索引	int	

void SetFromFile(const std::string& filename)

SetFromFile 接口直接从文本文件读取参数，代码示例：

```
NCSNonLinearSolver <double> nonLinearSolver;

/******** Set parameters for nonlinear solver ********/
std::string parasFile = "xxx/paras_file.txt";
nonLinearSolver.SetFromFile(parasFile);
```

具体的文本文档格式为一行对应一个参数的设置，形式为参数名 opt 空格后加上 value，具体的参数名 opt 与 value 与 2.1 中的接口保持一致，文本文档示例为如下：

```
--nls_log_level 2
--nls_line_search_type FullStep
--nls_fullStep_length_value 1.0
--nls_linear_solver_type Supernodal
--nls_direction_type Newton
--nls_max_iteration 1000
--nls_tol 1e-6
--nls_rtol 5e-3
--nls_threads 6
```

3.3.3 残差函数设置接口说明

bool SetTargetFun(TargetFunPtr<Scalar> targetFunPtr)

函数说明

该函数用于设置残差函数。

参数说明

参数	类型	含义
Scalar	模板参数	取值为 double 或者 std::complex
targetFunPtr	TargetFunPtr<Scalar>	残差函数指针

返回值说明

该函数返回值为 bool 类型，true 表示设置成功，false 表示设置失败。

函数指针 TargetFunPtr<Scalar> 类型说明

残差函数指针 TargetFunPtr<Scalar> 具体定义如下：

TargetFunPtr<Scalar> = void (*)(const Scalar* x, int n, Scalar*& residual)

表 3.1 函数指针 TargetFunPtr<Scalar> 参数说明

参数	类型	含义
Scalar	模板参数	取值为 double 或者 std::complex
x	输入参数	当前解向量
n	输入参数	解向量的维度
residual	输出参数	残差向量

注：对于指针 residual 指向的内存空间，非线性求解器不对其 delete 和 new 等操作，只会进行内容拷贝操作。

3.3.4 Jacobian 矩阵函数设置接口说明

bool SetJacobianFun(JacobianFunPtr<Scalar> jacobianFunPtr)

函数说明

该函数用于设置 Jacobian 矩阵函数

参数说明

参数	类型	含义
Scalar	模板参数	取值为 double 或者 std::complex
jacobianFunPtr	JacobianFunPtr<Scalar>	Jacobian 矩阵函数指针

返回值说明

该函数返回值为 bool 类型，true 表示设置成功，false 表示设置失败。

函数指针 JacobianFunPtr<Scalar> 类型说明

Jacobian 矩阵函数指针 JacobianFunPtr<Scalar> 具体数据类型如下

JacobianFunPtr<Scalar> = void (*)(const Scalar* x, int n, int*& ptr, int*& ids, Scalar*& values)

其中，Jacobian 矩阵使用 CSR 三元组表示法。

表 4.1 函数指针 JacobianFunPtr<Scalar> 参数说明

参数	类型	含义
Scalar	输入参数	模板参数，数据类型, 取值为 double 或者 std::complex
x	输入参数	当前解向量
n	输入参数	解向量的维度
ptr	输出参数	矩阵每行的起始位置（Jacobian 矩阵使用 CSR 三元组表示法）, 类型为 int*&，长度为 n+1
ids	输出参数	矩阵每个元素的列号（Jacobian 矩阵使用 CSR 三元组表示法），类型为 int*&，长度为 ptr[n+1]
values	输出参数	矩阵每个元素的值（Jacobian 矩阵使用 CSR 三元组表示法）, 类型为 Scalar*&, 长度 ptr[n+1]

注：对于指针 ptr、ids 和 values 指向的内存空间，非线性求解器不对其 delete 和 new 等操作，只会进行内容拷贝操作。

3.3.5 求解接口说明

SolveStateEnum Solve(int n , Scalar* sol)

函数说明

该函数为非线性迭代法的求解接口。

参数说明

参数	类型	含义
Scalar	模板参数	数据类型, 取值为 double 或者 std::complex
n	输入参数	解向量的维度，数据类型为 int
sol	输出参数	解向量，数据类型为 Scalar*

返回值说明

求解结束后，该接口会返回一个类型为 SolveStateEnum 的状态码，状态码的说明见下表。

表 5.1 非线性迭代法求解状态码列表

值	求解状态码的值	含义
Success	0	求解成功
Unknown	1	求解失败，未知原因
UnConverged	2	求解失败，不收敛
Converged	3	求解成功，收敛
Failed	4	求解失败
MaxCountExceed	5	求解失败，到达最大迭代次数

3.3.6 求解结果返回结果说明

求解完成后，可以获取求解的结果、残差和迭代次数等信息，具体接口如下说明。

int GetIter()

- 获取迭代次数

double GetResNorm()

- 获取迭代计算的残差范数

3.3.7 分布式接口说明

启用分布式计算需要设置参数--nls_mpi_run 为 1，详见参数说明章节。分布式输入有两种形式，一种为集中式输入，另一种为分布式输入。两者都需要设置全局维度--nls_global_dimension。

集中式输入

集中式输入仅需在主进程 (rank0) 定义 Jacobian 矩阵函数及残差函数，可参见章末的分布式示例代码。如果用集中式输入方式，Jacobian 矩阵和残差将只在主进程中计算，但线性方程组求解为分布式计算。计算完后，解只会在主进程中输出。

分布式输入

分布式输入需在各进程中定义局部的 Jacobian 矩阵函数及残差函数，可参见章末的分布式示例代码。

若要使用分布式输入，需要设置参数--nls_mpi_centralized_input 为 false。在每个进程中定义局部维度，局部的初始解，局部 Jacobian 矩阵函数（切分方式为行切分）及局部残差函数，局部 Jacobian 矩阵的全局起始行索引--nls_begin_row_index。程序通过局部的 Jacobian 矩阵函数计算局部的 Jacobian 矩阵，存储方式为 CSR 格式。如:

```
    if (rank == 0) {
        dim = 3;
        initSol = {1.1, 1.2, 1.3};
        solver.SetOption("--nls_begin_row_index", "0");
        solver.SetJacobianFun(&SimpleNonlinearFunctors::JacobianFunTest5Rank0);
        solver.SetTargetFun(&SimpleNonlinearFunctors::TargetFunTest5Rank0);
        solver.SetInitSol(dim, initSol.data());
    } else {
        dim = 1;
        initSol = {1.4};
        solver.SetOption("--nls_begin_row_index", "3");
        solver.SetJacobianFun(&SimpleNonlinearFunctors::JacobianFunTest5Rank1);
        solver.SetTargetFun(&SimpleNonlinearFunctors::TargetFunTest5Rank1);
        solver.SetInitSol(dim, initSol.data());
    }
```

计算完后，解会在不同进程中输出，其维度与输入设置的局部维度一致。

3.3.8 单机多线程版求解示例

```
#include <iostream>

#include <NCSNonLinearSolver.h>

using std::string;
using ScalarType = double;
using namespace NCS::CNLS;

std::vector<ScalarType> residualVec;
std::vector<int> ptrVec;
std::vector<int> idsVec;
std::vector<ScalarType> valuesVec;
```

(续下页)

(接上页)

```
/**
 * 定义非线性函数:
 * x^2 + y = 0
 * x - y = 0
 * 初始解: (0.5, 0.0)
 * 解: (0.0, 0.0)
 */
void TargetFunTest(const ScalarType* x, int n, ScalarType*& residual)
{
    residualVec.resize(2);
    residualVec[0] = x[0] * x[0] + x[1];
    residualVec[1] = x[0] - x[1];
    residual = residualVec.data();
}
/**
 * 定义Jacobian 矩阵函数:
 * 2x 1
 * 1 -1
 */
void JacobianFunTest(const ScalarType* x, int n, int*& ptr, int*& ids, ScalarType*&
↪ values)
{
    ptrVec.resize(3);
    idsVec.resize(4);
    valuesVec.resize(4);
    ptrVec[0] = 0;
    ptrVec[1] = 2;
    ptrVec[2] = 4;
    ptr = ptrVec.data();
    idsVec[0] = 0;
    idsVec[1] = 1;
    idsVec[2] = 0;
    idsVec[3] = 1;
    ids = idsVec.data();
    valuesVec[0] = 2 * x[0];
    valuesVec[1] = 1;
    valuesVec[2] = 1;
    valuesVec[3] = -1;
    values = valuesVec.data();
}

int main(int argc, char* argv[])
{
    // 创建solver 实例
    NCSNonLinearSolver<ScalarType> solver;
    // 非线性问题维数
    int dim = 2;
    // 初始解向量
    auto* initSol = new ScalarType[dim];
    auto* sol = new ScalarType[dim];
    // 解向量初始化
    initSol[0] = 0.5;
    initSol[1] = 0.0;

    // 求解参数配置
    solver.SetOption("--nls_max_iteration", "10000");
    solver.SetOption("--nls_tol", "1e-6");
    solver.SetOption("--nls_direction_type", "Newton");
    solver.SetOption("--nls_line_search_type", "FullStep");
    solver.SetOption("--nls_linear_solver_type", "Supernodal");
```

(续下页)

(接上页)

```
    // 设置残差计算函数
    solver.SetTargetFun(&TargetFunTest);
    // 设置Jacobian 矩阵计算函数
    solver.SetJacobianFun(&JacobianFunTest);
    // 设置初始解
    solver.SetInitSol(dim, initSol);
    // 求解
    auto status = solver.Solve(dim, sol);
    // 输出求解结果
    std::cout << "求解返回码为: " << int(status) << std::endl;
    for (int i = 0; i < dim; ++i) {
        std::cout << sol[i] << "\t";
    }
    std::cout << std::endl;
    delete[] sol;
    delete[] initSol;
    return 0;
}
```

使用 visual studio 的用户调用相应 C 接口，示例如下：

```
#include <string>
#include <iostream>

#include <NCSNonLinearSolverWrapper.h>

using std::string;
using ScalarType = double;
using namespace NCS::CNLS;

std::vector<ScalarType> residualVec;
std::vector<int> ptrVec;
std::vector<int> idsVec;
std::vector<ScalarType> valuesVec;

/**
* 定义非线性函数:
* x^2 + y = 0
* x - y = 0
* 初始解: (0.5, 0.0)
* 解: (0.0, 0.0)
*/
void TargetFunTest(const ScalarType* x, int n, ScalarType*& residual)
{
        residualVec.resize(2);
        residualVec[0] = x[0] * x[0] + x[1];
        residualVec[1] = x[0] - x[1];
        residual = residualVec.data();
}

/**
* 定义Jacobian 矩阵函数:
* 2x 1
* 1 -1
*/
void JacobianFunTest(const ScalarType* x, int n, int*& ptr, int*& ids, ScalarType*&
↪ values)
{
        ptrVec.resize(3);
        idsVec.resize(4);
        valuesVec.resize(4);
```

(续下页)

(接上页)

```
        ptrVec[0] = 0;
        ptrVec[1] = 2;
        ptrVec[2] = 4;
        ptr = ptrVec.data();
        idsVec[0] = 0;
        idsVec[1] = 1;
        idsVec[2] = 0;
        idsVec[3] = 1;
        ids = idsVec.data();
        valuesVec[0] = 2 * x[0];
        valuesVec[1] = 1;
        valuesVec[2] = 1;
        valuesVec[3] = -1;
        values = valuesVec.data();
}

int main(int argc, char* argv[]) {

    auto dataType = DataType::DOUBLE;
        //使用 C 接口创建 nonLinearSolver 实例
        NCSNonLinearSolverWrapper* nonlinearSolverWrapper = NonLinearSolver_
↪new(dataType);
        // 非线性问题维数
        int dim = 2;
        // 初始解向量
        auto* initSol = new ScalarType[dim];
        auto* sol = new ScalarType[dim];
        // 解向量初始化
        initSol[0] = 0.5;
        initSol[1] = 0.0;

        // 求解参数配置
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_max_
↪iteration", "10000");
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_tol",
↪"1e-6");
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_
↪direction_type", "Newton");
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_line_
↪search_type", "FullStep");
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_linear_
↪solver_type", "Supernodal");
        // 设置残差计算函数
        NonLinearSolverSetTargetFun_double(dataType, nonlinearSolverWrapper, &
↪TargetFunTest);
        // 设置Jacobian 矩阵计算函数
        NonLinearSolverSetJacobianFun_double(dataType, nonlinearSolverWrapper, &
↪JacobianFunTest);
        // 设置初始解
        NonLinearSolverSetInitSol_double(dataType, nonlinearSolverWrapper, dim,␣
↪initSol);
         // 求解
        auto status = NonLinearSolverSolve_double(dataType, nonlinearSolverWrapper,
↪ dim, sol);
        // 输出求解结果
        std::cout << "求解返回码为: " << int(status) << std::endl;
        for (int i = 0; i < dim; ++i) {
                std::cout << sol[i] << "\t";
        }
        std::cout << std::endl;
```

(续下页)

(接上页)

```
        NonLinearSolver_delete(nonlinearSolverWrapper);
        delete[] sol;
        delete[] initSol;
        return 0;
}
```

3.3.9 MPI 分布式版求解示例

3.3.10 集中式输入示例

```
#include <cmath>
#include <iostream>

#include "NCSNonLinearSolver.h"
#include "mpi.h"

using ScalarType = double;

std::vector<ScalarType> residualVec;
std::vector<int> ptrVec;
std::vector<int> idsVec;
std::vector<ScalarType> valuesVec;

/**
 * non linear function (Wood, symmetric):
 * f(1) = -200 * x(1) * (x(2) - x(1)^2) - (1 - x(1)) = 0
 * f(2) = 100 * (x(2) - x(1)^2) + 20.2 * (x(2) - 1) + 19.8 * (x(4) - 1) = 0
 * f(3) = -180 * x(3) * (x(4) - x(3)^2) - (1 - x(3)) = 0
 * f(4) = 90 * (x(4) - x(3)^2) + 20.2 * (x(4) - 1) + 19.8 * (x(2) - 1) = 0
 * init solution: (1.1, 1.2, 1.3, 1.4)
 * solution: (1.0, 1.0, 1.0, 1.0)
 */
void TargetFunTest(const ScalarType* x, int n, ScalarType*& residual)
{
    // Ensure there are exactly four variables, as the function assumes
    if (n != 4) {
        throw std::runtime_error("Target dimension should be 4 ");
    }

    residualVec.resize(4);
    ScalarType temp1 = x[1] - x[0] * x[0];
    ScalarType temp2 = x[3] - x[2] * x[2];

    residualVec[0] = -200.0 * x[0] * temp1 - (1.0 - x[0]);
    residualVec[1] = 100.0 * temp1 + 20.2 * (x[1] - 1.0) + 19.8 * (x[3] - 1.0);
    residualVec[2] = -180.0 * x[2] * temp2 - (1.0 - x[2]);
    residualVec[3] = 90.0 * temp2 + 20.2 * (x[3] - 1.0) + 19.8 * (x[1] - 1.0);

    residual = residualVec.data();
}

/**
 * Jacobian matrix function:
 * -200*(x(2)-3*x(1)^2)+1   -200*x(1)   0                        0
 * -200*x(1)                120.2       0                        19.8
 * 0                        0           -180*(x(4)-3*x(3)^2)+1   -180*x(3)
 * 0                        19.8        -180*x(3)                110.2
 */
void JacobianFunTest(const ScalarType* x, int n, int*& ptr, int*& ids, ScalarType*&
```

(续下页)

(接上页)

```
↪ values)
{
    // Ensure there are exactly four variables, as the function assumes
    if (n != 4) {
        throw std::runtime_error("Jacobian dimension should be 4 ");
    }

    // Clear previous data if any
    ptrVec.clear();
    idsVec.clear();
    valuesVec.clear();

    // Allocate enough space for non-zero values
    ptrVec.resize(n + 1, 0);
    idsVec.resize(10);
    valuesVec.resize(10);

    ptrVec = {0, 2, 5, 7, 10};
    idsVec = {0, 1, 0, 1, 3, 2, 3, 1, 2, 3};
    valuesVec[0] = -200.0 * (x[1] - 3.0 * x[0] * x[0]) + 1.0;
    valuesVec[1] = -200.0 * x[0];
    valuesVec[2] = -200.0 * x[0];
    valuesVec[3] = 120.2;
    valuesVec[4] = 19.8;
    valuesVec[5] = -180.0 * (x[3] - 3.0 * x[2] * x[2]) + 1.0;
    valuesVec[6] = -180.0 * x[2];
    valuesVec[7] = 19.8;
    valuesVec[8] = -180.0 * x[2];
    valuesVec[9] = 110.2;

    ptr = ptrVec.data();
    ids = idsVec.data();
    values = valuesVec.data();
}

/**
 * For MPI centralized input, TargetFun and JacobianFun are only defined on the␣
↪master process
 */
void TargetFunTestNull(const double* x, int n, double*& residual) { residual =␣
↪nullptr; }

/**
 * For MPI centralized input, TargetFun and JacobianFun are only defined on the␣
↪master process
 */
void JacobianFunTestNull(const double* x, int n, int*& ptr, int*& ids, double*&␣
↪values)
{
    ptr = nullptr;
    ids = nullptr;
    values = nullptr;
}

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    int rk;
    int np;
    MPI_Comm_rank(MPI_COMM_WORLD, &rk);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
```

(续下页)

(接上页)

```
    std::cout << "rank: " << rk << ", nproc: " << np << '\n';
    MPI_Barrier(MPI_COMM_WORLD);

    // 创建solver 实例
    NCS::CNLS::NCSNonLinearSolver<ScalarType> solver;
    // 非线性问题维数
    int dim = 4;
    // 初始解向量
    auto* initSol = new ScalarType[dim]{1.1, 1.2, 1.3, 1.4};
    auto* sol = new ScalarType[dim];
    // 求解参数配置
    solver.SetOption("--nls_max_iteration", "10000");
    solver.SetOption("--nls_tol", "1e-6");
    solver.SetOption("--nls_direction_type", "Newton");
    solver.SetOption("--nls_line_search_type", "BackTrack");
    solver.SetOption("--nls_linear_solver_type", "supernodal");
    solver.SetOption("--nls_matrix_type_symmetry", "cholesky");  // lu, ldl,␣
↪cholesky
    solver.SetOption("--nls_log_level", "0");                    // Turn Off all␣
↪logs
    solver.SetOption("--nls_force_print_results", "1");          // The final␣
↪results were hard printing

    solver.SetOption("--nls_mpi_run", "1");  // Parameter to turn ON/OFF MPI␣
↪objects (This is bool value (0/1))
    solver.SetOption("--nls_mpi_centralized_input", "true");
    solver.SetOption("--nls_global_dimension", "4");


    if (rk == 0) {
        // 设置Jacobian 矩阵计算函数
        solver.SetJacobianFun(&JacobianFunTest);
        // 设置残差计算函数
        solver.SetTargetFun(&TargetFunTest);
        // 设置初始解
        solver.SetInitSol(dim, initSol);
    } else {
        solver.SetJacobianFun(&JacobianFunTestNull);
        solver.SetTargetFun(&TargetFunTestNull);
        solver.SetInitSol(0, nullptr);
    }

    // 求解
    auto status = solver.Solve(dim, sol);

    // 输出求解结果
    std::cout << "rank: " << rk << ", 求解返回码为: " << int(status) << std::endl;
    for (int i = 0; i < dim; ++i) {
        std::cout << sol[i] << "\t";
    }
    std::cout << std::endl;

    delete[] sol;
    delete[] initSol;
    MPI_Finalize();
    return 0;
}
```

使用 visual studio 的用户调用相应 C 接口，示例如下：

```
#include <string>
```

(续下页)

(接上页)

```
#include <iostream>
#include <cmath>

#include <NCSNonLinearSolverWrapper.h>
#include "mpi.h"

using std::string;
using ScalarType = double;
using namespace NCS::CNLS;

std::vector<ScalarType> residualVec;
std::vector<int> ptrVec;
std::vector<int> idsVec;
std::vector<ScalarType> valuesVec;

/**
 * non linear function (Wood, symmetric):
 * f(1) = -200 * x(1) * (x(2) - x(1)^2) - (1 - x(1)) = 0
 * f(2) = 100 * (x(2) - x(1)^2) + 20.2 * (x(2) - 1) + 19.8 * (x(4) - 1) = 0
 * f(3) = -180 * x(3) * (x(4) - x(3)^2) - (1 - x(3)) = 0
 * f(4) = 90 * (x(4) - x(3)^2) + 20.2 * (x(4) - 1) + 19.8 * (x(2) - 1) = 0
 * init solution: (1.1, 1.2, 1.3, 1.4)
 * solution: (1.0, 1.0, 1.0, 1.0)
 */
void TargetFunTest(const ScalarType* x, int n, ScalarType*& residual)
{
        // Ensure there are exactly four variables, as the function assumes
        if (n != 4) {
                throw std::runtime_error("Target dimension should be 4 ");
        }
        residualVec.resize(4);
        ScalarType temp1 = x[1] - x[0] * x[0];
        ScalarType temp2 = x[3] - x[2] * x[2];
        residualVec[0] = -200.0 * x[0] * temp1 - (1.0 - x[0]);
        residualVec[1] = 100.0 * temp1 + 20.2 * (x[1] - 1.0) + 19.8 * (x[3] - 1.0);
        residualVec[2] = -180.0 * x[2] * temp2 - (1.0 - x[2]);
        residualVec[3] = 90.0 * temp2 + 20.2 * (x[3] - 1.0) + 19.8 * (x[1] - 1.0);
        residual = residualVec.data();
}

/**
 * Jacobian matrix function:
 * -200*(x(2)-3*x(1)^2)+1   -200*x(1)   0                       0
 * -200*x(1)                120.2       0                       19.8
 * 0                        0           -180*(x(4)-3*x(3)^2)+1  -180*x(3)
 * 0                        19.8        -180*x(3)               110.2
 */
void JacobianFunTest(const ScalarType* x, int n, int*& ptr, int*& ids, ScalarType*&
↪ values)
{
        // Ensure there are exactly four variables, as the function assumes
        if (n != 4) {
                throw std::runtime_error("Jacobian dimension should be 4 ");
        }
        // Clear previous data if any
        ptrVec.clear();
        idsVec.clear();
        valuesVec.clear();
        // Allocate enough space for non-zero values
        ptrVec.resize(n + 1, 0);
        idsVec.resize(10);
```

(续下页)

(接上页)

```
        valuesVec.resize(10);
        ptrVec = { 0, 2, 5, 7, 10 };
        idsVec = { 0, 1, 0, 1, 3, 2, 3, 1, 2, 3 };
        valuesVec[0] = -200.0 * (x[1] - 3.0 * x[0] * x[0]) + 1.0;
        valuesVec[1] = -200.0 * x[0];
        valuesVec[2] = -200.0 * x[0];
        valuesVec[3] = 120.2;
        valuesVec[4] = 19.8;
        valuesVec[5] = -180.0 * (x[3] - 3.0 * x[2] * x[2]) + 1.0;
        valuesVec[6] = -180.0 * x[2];
        valuesVec[7] = 19.8;
        valuesVec[8] = -180.0 * x[2];
        valuesVec[9] = 110.2;
        ptr = ptrVec.data();
        ids = idsVec.data();
        values = valuesVec.data();
}

/**
* For MPI centralized input, TargetFun and JacobianFun are only defined on the␣
↪aster process
*/

void TargetFunTestNull(const double* x, int n, double*& residual) {
        residual = nullptr;
}
/**
* For MPI centralized input, TargetFun and JacobianFun are only defined on the?
,↪master process
*/
void JacobianFunTestNull(const double* x, int n, int*& ptr, int*& ids, double*&␣
↪values)
{
        ptr = nullptr;
        ids = nullptr;
        values = nullptr;
}

int main(int argc, char* argv[])
{
        MPI_Init(&argc, &argv);
        int rk;
        int np;
        MPI_Comm_rank(MPI_COMM_WORLD, &rk);
        MPI_Comm_size(MPI_COMM_WORLD, &np);
        std::cout << "rank: " << rk << ", nproc: " << np << '\n';
        MPI_Barrier(MPI_COMM_WORLD);

        //使用 C 接口创建 nonLinearSolver 实例
        auto dataType = DataType::DOUBLE;
        NCSNonLinearSolverWrapper* nonlinearSolverWrapper = NonLinearSolver_
↪new(dataType);
        // 非线性问题维数
        int dim = 4;
        // 初始解向量
        auto* initSol = new ScalarType[dim]{ 1.1, 1.2, 1.3, 1.4 };
        auto* sol = new ScalarType[dim];
        // 求解参数配置
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_max_
↪iteration", "10000");
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_tol",
```

(续下页)

(接上页)

```
↪"1e-6");
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_
↪direction_type", "Newton");
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_line_
↪search_type", "BackTrack");
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_linear_
↪solver_type", "Supernodal");
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_matrix_
↪type_symmetry", "cholesky");   // lu, ldl,cholesky
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_log_level
↪", "0");   // Turn Off all logs
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_force_
↪print_results", "1");   // The final results were hard printing
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_mpi_run",
↪ "1");   // Parameter to turn ON/OFF MPI objects(This is bool value(0 / 1))
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_mpi_
↪centralized_input", "true");
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_global_
↪dimension", "4");

        if (rk == 0) {
                // 设置Jacobian 矩阵计算函数
                NonLinearSolverSetJacobianFun_double(dataType,␣
↪nonlinearSolverWrapper, &JacobianFunTest);
                // 设置残差计算函数
                NonLinearSolverSetTargetFun_double(dataType,␣
↪nonlinearSolverWrapper, &TargetFunTest);
                // 设置初始解
                NonLinearSolverSetInitSol_double(dataType, nonlinearSolverWrapper,␣
↪dim, initSol);
        }else {
                NonLinearSolverSetTargetFun_double(dataType,␣
↪nonlinearSolverWrapper, &TargetFunTestNull);
                NonLinearSolverSetJacobianFun_double(dataType,␣
↪nonlinearSolverWrapper, &JacobianFunTestNull);
                NonLinearSolverSetInitSol_double(dataType, nonlinearSolverWrapper,␣
↪0, nullptr);
        }

        // 求解
        auto status = NonLinearSolverSolve_double(dataType, nonlinearSolverWrapper,
↪ dim, sol);

        // 输出求解结果
        std::cout << "rank: " << rk << ", 求解返回码为: " << int(status) <<␣
↪std::endl;
        for (int i = 0; i < dim; ++i) {
                std::cout << sol[i] << "\t";
        }
        std::cout << std::endl;

        NonLinearSolver_delete(nonlinearSolverWrapper);
        delete[] sol;
        delete[] initSol;
        MPI_Finalize();
        return 0;

}
```

3.3.11 分布式输入示例

```
#include <cmath>
#include <iostream>

#include "NCSNonLinearSolver.h"
#include "mpi.h"

using ScalarType = double;

std::vector<ScalarType> residualVec;
std::vector<int> ptrVec;
std::vector<int> idsVec;
std::vector<ScalarType> valuesVec;

/**
 * non linear function (Wood, symmetric):
 * f(1) = -200 * x(1) * (x(2) - x(1)^2) - (1 - x(1)) = 0 rank0
 * f(2) = 100 * (x(2) - x(1)^2) + 20.2 * (x(2) - 1) + 19.8 * (x(4) - 1) = 0
 * f(3) = -180 * x(3) * (x(4) - x(3)^2) - (1 - x(3)) = 0
 * ----------------------------------------------------------------------------
 * f(4) = 90 * (x(4) - x(3)^2) + 20.2 * (x(4) - 1) + 19.8 * (x(2) - 1) = 0 rank1
 * init solution: (1.1, 1.2, 1.3, 1.4)
 * solution: (1.0, 1.0, 1.0, 1.0)
 */
void TargetFunTestRank0(const ScalarType* x, int n, ScalarType*& residual)
{
    if (n != 3) {
        throw std::runtime_error("Local dimension should be 3 on rank0!");
    }

    residualVec.resize(n);
    ScalarType temp1 = x[1] - x[0] * x[0];
    ScalarType temp2 = x[3] - x[2] * x[2];

    residualVec[0] = -200.0 * x[0] * temp1 - (1.0 - x[0]);
    residualVec[1] = 100.0 * temp1 + 20.2 * (x[1] - 1.0) + 19.8 * (x[3] - 1.0);
    residualVec[2] = -180.0 * x[2] * temp2 - (1.0 - x[2]);

    residual = residualVec.data();
}

void TargetFunTestRank1(const ScalarType* x, int n, ScalarType*& residual)
{
    if (n != 1) {
        throw std::runtime_error("Local dimension should be 1 on rank1!");
    }

    residualVec.resize(n);
    ScalarType temp2 = x[3] - x[2] * x[2];

    residualVec[0] = 90.0 * temp2 + 20.2 * (x[3] - 1.0) + 19.8 * (x[1] - 1.0);

    residual = residualVec.data();
}

/**
 * Jacobian matrix function:
 * -200*(x(2)-3*x(1)^2)+1   -200*x(1)   0                    0
```

(续下页)

(接上页)

```
 * -200*x(1)                120.2       0                        19.8              
↪   rank0
 * 0                        0           -180*(x(4)-3*x(3)^2)+1   -180*x(3)
 * ------------------------------------------------------------------------------
↪----------
 * 0                        19.8        -180*x(3)                110.2             
↪   rank1
 */
void JacobianFunTestRank0(const ScalarType* x, int n, int*& ptr, int*& ids,␣
↪ScalarType*& values)
{
    if (n != 3) {
        throw std::runtime_error("Local dimension should be 3 on rank0!");
    }

    // Clear previous data if any
    ptrVec.clear();
    idsVec.clear();
    valuesVec.clear();

    // Allocate enough space for non-zero values
    ptrVec.resize(n + 1);
    idsVec.resize(7);
    valuesVec.resize(7);

    ptrVec = {0, 2, 5, 7};
    idsVec = {0, 1, 0, 1, 3, 2, 3};
    valuesVec[0] = -200.0 * (x[1] - 3.0 * x[0] * x[0]) + 1.0;
    valuesVec[1] = -200.0 * x[0];
    valuesVec[2] = -200.0 * x[0];
    valuesVec[3] = 120.2;
    valuesVec[4] = 19.8;
    valuesVec[5] = -180.0 * (x[3] - 3.0 * x[2] * x[2]) + 1.0;
    valuesVec[6] = -180.0 * x[2];

    ptr = ptrVec.data();
    ids = idsVec.data();
    values = valuesVec.data();
}

void JacobianFunTestRank1(const ScalarType* x, int n, int*& ptr, int*& ids,␣
↪ScalarType*& values)
{
    if (n != 1) {
        throw std::runtime_error("Local dimension should be 1 on rank1!");
    }

    // Clear previous data if any
    ptrVec.clear();
    idsVec.clear();
    valuesVec.clear();

    // Allocate enough space for non-zero values
    ptrVec.resize(n + 1);
    idsVec.resize(7);
    valuesVec.resize(7);

    ptrVec = {0, 3};
    idsVec = {1, 2, 3};
    valuesVec[0] = 19.8;
    valuesVec[1] = -180.0 * x[2];
```

(续下页)

(接上页)

```
    valuesVec[2] = 110.2;

    ptr = ptrVec.data();
    ids = idsVec.data();
    values = valuesVec.data();
}

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    int rk;
    int np;
    MPI_Comm_rank(MPI_COMM_WORLD, &rk);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    std::cout << "rank: " << rk << ", nproc: " << np << '\n';
    MPI_Barrier(MPI_COMM_WORLD);

    // 创建solver 实例
    NCS::CNLS::NCSNonLinearSolver<ScalarType> solver;
    // 非线性问题维数
    int dim;
    // 初始解向量
    std::vector<ScalarType> sol;
    std::vector<ScalarType> initSol;
    // 求解参数配置
    solver.SetOption("--nls_max_iteration", "10000");
    solver.SetOption("--nls_tol", "1e-6");
    solver.SetOption("--nls_direction_type", "Newton");
    solver.SetOption("--nls_line_search_type", "BackTrack");
    solver.SetOption("--nls_linear_solver_type", "supernodal");
    solver.SetOption("--nls_matrix_type_symmetry", "cholesky");  // lu, ldl,␣
↪cholesky
    solver.SetOption("--nls_log_level", "0");                    // Turn Off all␣
↪logs
    solver.SetOption("--nls_force_print_results", "1");          // The final␣
↪results were hard printing

    solver.SetOption("--nls_mpi_run", "1");  // Parameter to turn ON/OFF MPI␣
↪objects (This is bool value (0/1))
    solver.SetOption("--nls_mpi_centralized_input", "false");
    solver.SetOption("--nls_global_dimension", "4");

    if (rk == 0) {
        dim = 3;  // local size
        initSol = {1.1, 1.2, 1.3};
        solver.SetOption("--nls_begin_row_index", "0");
        solver.SetJacobianFun(&JacobianFunTestRank0);
        solver.SetTargetFun(&TargetFunTestRank0);
        solver.SetInitSol(dim, initSol.data());
    } else {
        dim = 1;  // local size
        initSol = {1.4};
        solver.SetOption("--nls_begin_row_index", "3");
        solver.SetJacobianFun(&JacobianFunTestRank1);
        solver.SetTargetFun(&TargetFunTestRank1);
        solver.SetInitSol(dim, initSol.data());
    }
    sol.resize(dim);  // resize the solution to local size

    // 求解
    auto status = solver.Solve(dim, sol.data());
```

(续下页)

(接上页)

```
    // 输出求解结果
    std::cout << "rank: " << rk << ", 求解返回码为: " << int(status) << std::endl;
    for (int i = 0; i < dim; ++i) {
        std::cout << sol[i] << "\t";
    }
    std::cout << std::endl;

    MPI_Finalize();
    return 0;
}
```

使用 visual studio 的用户调用相应 C 接口，示例如下：

```
#include <string>
#include <iostream>
#include <cmath>

#include <NCSNonLinearSolverWrapper.h>
#include "mpi.h"

using std::string;
using ScalarType = double;
using namespace NCS::CNLS;

std::vector<ScalarType> residualVec;
std::vector<int> ptrVec;
std::vector<int> idsVec;
std::vector<ScalarType> valuesVec;

/**
 * non linear function (Wood, symmetric):
 * f(1) = -200 * x(1) * (x(2) - x(1)^2) - (1 - x(1)) = 0                          ␣
↪ rank0
 * f(2) = 100 * (x(2) - x(1)^2) + 20.2 * (x(2) - 1) + 19.8 * (x(4) - 1) = 0
 * f(3) = -180 * x(3) * (x(4) - x(3)^2) - (1 - x(3)) = 0
 * ------------------------------------------------------------------------------
↪----------
 * f(4) = 90 * (x(4) - x(3)^2) + 20.2 * (x(4) - 1) + 19.8 * (x(2) - 1) = 0       ␣
↪ rank1
 * init solution: (1.1, 1.2, 1.3, 1.4)
 * solution: (1.0, 1.0, 1.0, 1.0)
 */
void TargetFunTestRank0(const ScalarType* x, int n, ScalarType*& residual)
{
        if (n != 3) {
                throw std::runtime_error("Local dimension should be 3 on rank0!");
        }
        residualVec.resize(n);
        ScalarType temp1 = x[1] - x[0] * x[0];
        ScalarType temp2 = x[3] - x[2] * x[2];
        residualVec[0] = -200.0 * x[0] * temp1 - (1.0 - x[0]);
        residualVec[1] = 100.0 * temp1 + 20.2 * (x[1] - 1.0) + 19.8 * (x[3] - 1.0);
        residualVec[2] = -180.0 * x[2] * temp2 - (1.0 - x[2]);
        residual = residualVec.data();
}

void TargetFunTestRank1(const ScalarType* x, int n, ScalarType*& residual)
{
        if (n != 1) {
                throw std::runtime_error("Local dimension should be 1 on rank1!");
```

(续下页)

(接上页)

```
        }
        residualVec.resize(n);
        ScalarType temp2 = x[3] - x[2] * x[2];
        residualVec[0] = 90.0 * temp2 + 20.2 * (x[3] - 1.0) + 19.8 * (x[1] - 1.0);
        residual = residualVec.data();
}
/**
 * Jacobian matrix function:
 * -200*(x(2)-3*x(1)^2)+1   -200*x(1)   0                        0
 * -200*x(1)                120.2       0                        19.8
↪   rank0
 * 0                        0           -180*(x(4)-3*x(3)^2)+1   -180*x(3)
 * ------------------------------------------------------------------------------
↪----------
 * 0                        19.8        -180*x(3)                110.2
↪   rank1
 */
void JacobianFunTestRank0(const ScalarType* x, int n, int*& ptr, int*& ids,␣
↪ScalarType*& values)
{
        if (n != 3) {
                throw std::runtime_error("Local dimension should be 3 on rank0!");
        }
        // Clear previous data if any
        ptrVec.clear();
        idsVec.clear();
        valuesVec.clear();
        // Allocate enough space for non-zero values
        ptrVec.resize(n + 1);
        idsVec.resize(7);
        valuesVec.resize(7);
        ptrVec = { 0, 2, 5, 7 };
        idsVec = { 0, 1, 0, 1, 3, 2, 3 };
        valuesVec[0] = -200.0 * (x[1] - 3.0 * x[0] * x[0]) + 1.0;
        valuesVec[1] = -200.0 * x[0];
        valuesVec[2] = -200.0 * x[0];
        valuesVec[3] = 120.2;
        valuesVec[4] = 19.8;
        valuesVec[5] = -180.0 * (x[3] - 3.0 * x[2] * x[2]) + 1.0;
        valuesVec[6] = -180.0 * x[2];
        ptr = ptrVec.data();
        ids = idsVec.data();
        values = valuesVec.data();
}

void JacobianFunTestRank1(const ScalarType* x, int n, int*& ptr, int*& ids,␣
↪ScalarType*& values)
{
        if (n != 1) {
                throw std::runtime_error("Local dimension should be 1 on rank1!");
        }
        // Clear previous data if any
        ptrVec.clear();
        idsVec.clear();
        valuesVec.clear();
        // Allocate enough space for non-zero values
        ptrVec.resize(n + 1);
        idsVec.resize(7);
        valuesVec.resize(7);
        ptrVec = { 0, 3 };
        idsVec = { 1, 2, 3 };
```

(续下页)

(接上页)

```
        valuesVec[0] = 19.8;
        valuesVec[1] = -180.0 * x[2];
        valuesVec[2] = 110.2;
        ptr = ptrVec.data();
        ids = idsVec.data();
        values = valuesVec.data();
}

int main(int argc, char* argv[])
{
        MPI_Init(&argc, &argv);
        int rk;
        int np;
        MPI_Comm_rank(MPI_COMM_WORLD, &rk);
        MPI_Comm_size(MPI_COMM_WORLD, &np);
        std::cout << "rank: " << rk << ", nproc: " << np << '\n';
        MPI_Barrier(MPI_COMM_WORLD);

        //使用 C 接口创建 nonLinearSolver 实例
        auto dataType = DataType::DOUBLE;
        NCSNonLinearSolverWrapper* nonlinearSolverWrapper = NonLinearSolver_
↪new(dataType);
        // 非线性问题维数
        int dim;
        // 初始解向量
        std::vector<ScalarType> sol;
        std::vector<ScalarType> initSol;
        // 求解参数配置
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_max_
↪iteration", "10000");
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_tol",
↪"1e-6");
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_
↪direction_type", "Newton");
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_line_
↪search_type", "BackTrack");
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_linear_
↪solver_type", "Supernodal");
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_matrix_
↪type_symmetry", "cholesky");   // lu, ldl,cholesky
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_log_level
↪", "0");   // Turn Off all logs
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_force_
↪print_results", "1");   // The final results were hard printing
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_mpi_run",
↪ "1");   // Parameter to turn ON/OFF MPI objects(This is bool value(0 / 1))
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_mpi_
↪centralized_input", "false");
        NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_global_
↪dimension", "4");

        if (rk == 0) {
                dim = 3; // local size
                initSol = { 1.1, 1.2, 1.3 };
                NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_
↪begin_row_index", "0");
                NonLinearSolverSetTargetFun_double(dataType,␣
↪nonlinearSolverWrapper, &TargetFunTestRank0);
                NonLinearSolverSetJacobianFun_double(dataType,␣
↪nonlinearSolverWrapper, &JacobianFunTestRank0);
                NonLinearSolverSetInitSol_double(dataType, nonlinearSolverWrapper,␣
```

(续下页)

(接上页)

```
↪dim, initSol.data());
        } else {
                dim = 1; // local size
                initSol = { 1.4 };
                NonLinearSolverSetOption(dataType, nonlinearSolverWrapper, "--nls_
↪begin_row_index", "3");
                NonLinearSolverSetTargetFun_double(dataType,␣
↪nonlinearSolverWrapper, &TargetFunTestRank1);
                NonLinearSolverSetJacobianFun_double(dataType,␣
↪nonlinearSolverWrapper, &JacobianFunTestRank1);
                NonLinearSolverSetInitSol_double(dataType, nonlinearSolverWrapper,␣
↪dim, initSol.data());
        }
        sol.resize(dim); // resize the solution to local size

        // 求解
        auto status = NonLinearSolverSolve_double(dataType, nonlinearSolverWrapper,
↪ dim, sol.data());

        // 输出求解结果
        std::cout << "rank: " << rk << ", 求解返回码为: " << int(status) <<␣
↪std::endl;
        for (int i = 0; i < dim; ++i) {
                std::cout << sol[i] << "\t";
        }
        std::cout << std::endl;

        NonLinearSolver_delete(nonlinearSolverWrapper);
        MPI_Finalize();
        return 0;
}
```

使用 visual studio 的用户调用相应 C 接口时，对于复数的情况，TargetFun 和 JacobianFun 中的 ScalarType 还是使用 C++ 的 std::complex 不变，但是其它地方的 ScalarType 需要替换成 cComplex_double，然后调用求解器相对应的复数接口。

3.4 特征值求解接口说明

3.4.1 特征值接口整体说明

特征值求解器目前支持标准和广义特征值问题求解：

$$Ax = \lambda Bx$$

当 B 为单位矩阵 I 时，求解标准特征值问题。目前特征值求解器支持实数/复数、Hermitian/Non-Hermitian、共享内存/分布式求解。

特征值求解器对用户提供的接口封装在 NCSEigenSolver.h 中：

```
template <typename Scalar>
class NCSEigenSolver {
public:
    NCSEigenSolver();

    ~NCSEigenSolver();

    /* 参数配置接口 */
    void SetOption(const std::string& opt, const std::string& value);
```

(续下页)

(接上页)

```
    /* 参数文件配置接口 */
    void SetFromFile(const std::string& filename);

    /* 求解接口 */
    int Solve(const int dim, const Scalar* aValue, const int* aColIdx, const int*␣
↪aRowPtr,
              const Scalar* bValue = nullptr, const int* bColIdx = nullptr, const␣
↪int* bRowPtr = nullptr);

    /* 特征值获取接口 */
    void GetEigenValue(const int idx, Scalar* valReal, Scalar* valImg);

    /* 特征向量获取接口 */
    void GetEigenVector(const int idx, Scalar* vecReal, Scalar* vecImg);

    /* 特征值-特征向量获取接口 */
    void GetEigenPair(const int idx, Scalar* valReal, Scalar* valImg, Scalar*␣
↪vecReal, Scalar* vecImg);

    /* 已收敛特征值个数获取接口 */
    int GetConverged() const;

    /* 迭代误差获取接口 */
    double GetResNorm(const int idx = 0) const;

    /* 真实误差获取接口 */
    double GetTrueResNorm(const int idx = 0) const;

    /* 迭代次数获取接口 */
    int GetIter() const;
```

模板参数为 `Scalar`, 目标支持的取值为:

- 实数: `double`
- 复数: `std::complex<double>`

可以支撑用户针对不同数据类型创建类及其实例。

当使用 visual studio 时，提供的接口封装在 `NCSEigenSolverWrapper.h` 中:

```
#include "NCSEigenSolver.h"

namespace NCS::EIG {
enum class DataType : int32_t { DOUBLE = 0, COMPLEX_DOUBLE = 2, };
extern "C" {
struct DLLEXPORT NCSEigenSolverWrapper;
struct cComplex_double {
  double real;
  double imag;
};
/* 创建EigenSolver求解器实例 */
DLLEXPORT NCSEigenSolverWrapper* EigenSolver_new(DataType dataType);
/* 参数配置接口 */
DLLEXPORT void EigenSolverSetOption(DataType dataType, NCSEigenSolverWrapper*␣
↪wrapper,
    const char* copt, const char* cvalue);
/* 参数文件配置接口 */
DLLEXPORT void EigenSolverSetFromFile(DataType dataType, NCSEigenSolverWrapper*␣
↪wrapper,
    const char* cfilename);
 /* double类型求解接口 */
```

(续下页)

(接上页)

```
DLLEXPORT int EigenSolverSolve_double(DataType dataType, NCSEigenSolverWrapper*␣
↪wrapper,
    const int dim, const double* aValue, const int* aColIdx, const int* aRowPtr,
    const double* bValue = nullptr, const int* bColIdx = nullptr, const int*␣
↪bRowPtr = nullptr);
 /* 复数double类型求解接口 */
DLLEXPORT int EigenSolverSolve_complex_double(DataType dataType,␣
↪NCSEigenSolverWrapper* wrapper,
    const int dim, const cComplex_double* aValue, const int* aColIdx, const int*␣
↪aRowPtr,
    const cComplex_double* bValue = nullptr, const int* bColIdx = nullptr, const␣
↪int* bRowPtr = nullptr);
 /* double类型特征值获取接口 */
DLLEXPORT void EigenSolverGetEigenValue_double(DataType dataType,␣
↪NCSEigenSolverWrapper* wrapper,
    const int idx, double* valReal, double* valImg);
 /* 复数double类型特征值获取接口 */
DLLEXPORT void EigenSolverGetEigenValue_complex_double(DataType dataType,␣
↪NCSEigenSolverWrapper* wrapper,
    const int idx, cComplex_double* valReal, cComplex_double* valImg);
 /* double类型特征向量获取接口 */
DLLEXPORT void EigenSolverGetEigenVector_double(DataType dataType,␣
↪NCSEigenSolverWrapper* wrapper,
    const int idx, double* vecReal, double* vecImg);
 /* 复数double类型特征向量获取接口 */
DLLEXPORT void EigenSolverGetEigenVector_complex_double(DataType dataType,␣
↪NCSEigenSolverWrapper* wrapper,
    const int idx, cComplex_double* vecReal, cComplex_double* vecImg);
 /* double类型特征值-特征向量获取接口 */
DLLEXPORT void EigenSolverGetEigenPair_double(DataType dataType,␣
↪NCSEigenSolverWrapper* wrapper,
    const int idx, double* valReal, double* valImg, double* vecReal, double*␣
↪vecImg);
 /* 复数double类型特征值-特征向量获取接口 */
DLLEXPORT void EigenSolverGetEigenPair_complex_double(DataType dataType,␣
↪NCSEigenSolverWrapper* wrapper,
    const int idx, cComplex_double* valReal, cComplex_double* valImg,
    cComplex_double* vecReal, cComplex_double* vecImg);
 /* 已收敛特征值个数获取接口 */
DLLEXPORT int EigenSolverGetConverged(DataType dataType, NCSEigenSolverWrapper*␣
↪wrapper);
 /* 迭代误差获取接口 */
DLLEXPORT double EigenSolverGetResNorm(DataType dataType, NCSEigenSolverWrapper*␣
↪wrapper,
  const int idx = 0);
/* 真实误差获取接口 */
DLLEXPORT double EigenSolverGetTrueResNorm(DataType dataType,␣
↪NCSEigenSolverWrapper* wrapper,
  const int idx = 0);
/* 迭代次数获取接口 */
DLLEXPORT int EigenSolverGetIter(DataType dataType, NCSEigenSolverWrapper*␣
↪wrapper);
/* 删除EigenSolver求解器实例 */
DLLEXPORT void EigenSolver_delete(NCSEigenSolverWrapper* wrapper);
}
}
```

注意：visual studio 中使用的接口，string 类型替换为 char[]，complex 替换为 cComple_double。

接口可大致分为三类：

- 参数配置接口

- 求解接口
- 结果获取接口

3.4.2 参数配置接口说明

特征值求解提供了两种参数配置的方式：

- `SetOption` 通过提供参数名及参数值逐个设置。
- `SetFromFile` 通过提供配置文件，在配置文件中按照格式设置参数名及其参数值。

下面进行接口说明。详细参数说明参考参数详细说明。

SetOption

```
void SetOption(const std::string& opt, const std::string& value);
```

参数说明

参数	类型	含义	可选值
opt	string	参数名称	参考参数详细说明
value	string	参数值	参考参数详细说明

示例

```
NCSEigenSolver<double> solver;

solver.SetOption("--eps_nev", "20"); // 特征值个数
solver.SetOption("--eps_which", "TM"); // 求解离目标值模最近的特征值
solver.SetOption("--eps_tol", "1.0e-8"); // 迭代阈值
```

SetFromFile

```
void SetFromFile(const std::string& filename);
```

参数说明

参数	类型	含义
filename	string	参数文件路径

参数文件示例

```
--eps_nev 20
--eps_which TM
--eps_tol 1.0e-8
```

3.4.3 求解接口说明

问题求解

```
int Solve(const int dim, const Scalar* aValue, const int* aColIdx, const int*␣
↪aRowPtr,
              const Scalar* bValue = nullptr, const int* bColIdx = nullptr, const␣
↪int* bRowPtr = nullptr);
```

此接口用于特征值问题的求解，支持标准特征值问题及广义特征值问题。矩阵的格式为 CSR 格式。

- aRowPtr: 行偏移量数组起始地址，数组长度为 dim + 1, aRowPtr[dim] 需保存非零元个数（nnz）
- aColIdx: 列下标数组起始地址，数组长度为 nnz
- aValue: 值数组起始地址，数组长度为 nnz

如果提供 bValue, bColIdx, 及 bRowPtr, 则代表求解广义特征值问题。

参数说明

表 1 求解接口参数说明

参数	类型	含义
dim	int	矩阵的维度
aValue	Scalar*	A 矩阵 CSR 格式的值数组起始地址
aColIdx	int*	A 矩阵 CSR 格式的列下标数组起始地址
aRowPtr	int*	A 矩阵 CSR 格式的行偏移量数组起始地址
bValue	Scalar*	B 矩阵 CSR 格式的值数组起始地址或空指针
bColIdx	int*	B 矩阵 CSR 格式的列下标数组起始地址或空指针
bRowPtr	int*	B 矩阵 CSR 格式的行偏移量数组起始地址或空指针

MPI 分布式特殊说明

当调用分布式求解时，根据矩阵是 集中式输入或 分布式输入，接口参数说明如下：

集中式矩阵输入

集中式矩阵输入时，主进程 0 的参数说明与 表 1 一致，其余进程，仅提供 dim 的值，其他参数均为空指针 nullptr。另外，如下方参数详细说明所示，需要设置 --eps_distributed 为 true。

分布式矩阵输入

矩阵分布式输入时，矩阵按行切分为多个连续的子矩阵，每个子矩阵属于一个 MPI 进程。每个 MPI 进程中，子矩阵的存储格式为行主序格式（CSR），dim 为矩阵总维度，其他参数为本地数据对应的指针地址。另外，对于求解器选项参数，如下方参数详细说明所示，分布式矩阵输入需要设置以下参数：

表 2 分布式输入情形相关参数

参数	值
--eps_distributed	true
--eps_master_input_only	false
--eps_local_begin_row	当前进程子矩阵的起始行号
--eps_local_size	当前进程子矩阵的行数

注意

- 对称或 Hermitian 矩阵，建议全量矩阵 CSR 输入。
- 对称或 Hermitian 矩阵，若上三角 CSR 输入，需要设置参数 --eps_fill_packed 为 true，且仅共享内存求解支持，分布式暂不支持。

3.4.4 求解结果获取接口说明

已收敛个数

```
int GetConverged() const;
```

获取特征值求解过程中收敛的特征值个数。

迭代次数

```
int GetIter() const;
```

获取求解过程的迭代次数。

特征值

```
void GetEigenValue(const int idx, Scalar* valReal, Scalar* valImg);
```

按照用户设置的求解类型，获取第 idx 个特征值。

参数	类型	含义
idx	int	索引值
valReal	Scalar*	第 idx 个特征值实数部分地址
valImg	Scalar*	第 idx 个特征值虚数部分地址

特征向量

```
void GetEigenVector(const int idx, Scalar* vecReal, Scalar* vecImg);
```

按照用户设置的求解类型，获取第 idx 个特征值对应的特征向量。

参数	类型	含义
idx	int	索引值
vecReal	Scalar*	第 idx 个特征向量实数部分的起始地址
vecImg	Scalar*	第 idx 个特征向量虚数部分的起始地址

MPI 分布式特殊说明

调用 MPI 分布式时，矩阵的输入方式影响特征向量的返回方式：

- 如果矩阵是集中式输入，特征向量仅返回给主进程
- 如果矩阵是分布式输入，特征向量仅将子矩阵对应行的向量部分返回给对应进程。例如，进程 1 负责输入矩阵的 [10,20) 行，则进程 1 获取特征向量的 [10,20) 行

该规则也适用于下方 GetEigenPair 接口中特征向量部分。

特征值-特征向量对

```
void GetEigenPair(const int idx, Scalar* valReal, Scalar* valImg, Scalar* vecReal,␣
↪Scalar* vecImg);
```

按照用户设置的求解类型，获取第 idx 个特征值及对应的特征向量。

参数	类型	含义
idx	int	索引值
valReal	Scalar*	第 idx 个特征值实数部分地址
valImg	Scalar*	第 idx 个特征值虚数部分地址
vecReal	Scalar*	第 idx 个特征向量实数部分的起始地址
vecImg	Scalar*	第 idx 个特征向量虚数部分的起始地址

迭代残差

```
double GetResNorm(const int idx = 0) const;
```

获取第 idx 个特征值-特征向量对的迭代过程残差

真实残差

```
double GetTrueResNorm(const int idx = 0) const;
```

获取第 idx 个特征值-特征向量对的真实相对残差。

3.4.5 参数详细说明

SetOption 中参数名称及对应参数值的类型均为 string。此章节详细介绍各个参数。枚举类型的参数会在枚举型参数介绍进一步介绍。

参数介绍

分布式参数

参数	说明	可选值	默认值
--eps_distributed	是否调用 MPI 分布式	布尔	“false”
--eps_master_input_only	矩阵是否集中式输入	布尔	“true”
--eps_local_begin_row	分布式输入时，当前进程负责子矩阵的起始行号	整数	“0”
--eps_local_size	分布式输入时，当前进程负责子矩阵的行数	整数	“0”

基本参数

参数	说明	可选值	默认值
--eps_thread	线程个数	整数	“1”
--eps_nev	待求解特征值个数	整数	“1”
--eps_which	待求解的特征值类型	枚举	“TM”
--eps_problem_type	特征值问题类型	枚举	“ghep”
--eps_target	目标值	实数	“0.0”
--eps_tol	阈值	实数	“1.0e-8”
--eps_max_it	最大迭代次数	整数	“100”
--eps_convergence_test	收敛判断准则	枚举	“relative”
--eps_fill_packed	对称矩阵是否仅输入上三角 CSR	布尔	“false”

算法参数

参数	说明	可选值	默认值
--eps_type	特征值算法	枚举	“krylovschur”
--eps_ncv	子空间维度	整数	无, 内部可自动计算
--eps_deflation	启用收缩策略	布尔	“true”

谱变换参数

参数	说明	可选值	默认值
--eps_st_type	谱变换类型	枚举	“sinvert”
--eps_st_shift	位移值	实数	默认与 --eps_target 相同
--eps_st_pc_type	矩阵分解算法	枚举	“lu”

向量基组运算参数

参数	说明	可选值	默认值
`--eps_vs_orthog_type`	向量组的正交化算法	枚举	“cgs”

日志等级

参数	说明	可选值	默认值
`--eps_loglevel`	日志等级	整数	“1”

- 说明：日志等级类型
  - -1: OFF
  - 0: ERROR
  - 1: WARN
  - 2: INFO

枚举型参数介绍

which

值	描述
LM	模最大的特征值
SM	模最小的特征值
LR	实数部分最大的特征值
SR	实数部分最小的特征值
TM	离目标值模最近的特征值
TR	离目标值实数最近的特征值

problem_type

值	描述
hep	标准 `Hermitian` 特征值问题
ghep	广义 `Hermitian` 特征值问题
nhep	标准 `Non-Hermitian` 特征值问题
gnhep	广义 `Non-Hermitian` 特征值问题

convergence_test

值	描述
absolute	绝对残差收敛准则、非真实残差
relative	相对残差收敛准则、非真实残差

type

值	描述
krylovschur	krylov-schur 算法
arnoldi	隐式重启 Arnoldi

st_type

值	描述
shift	shift，针对求解 `LM` 和 `LR` 问题
sinvert	shift and invert，针对求解 `TM`,`SM`,`SR`, 和 `TR` 问题

st_pc_type

值	描述
cholesky	矩阵 cholesky 分解
ldl	矩阵 ldl 分解，仅支持共享内存
lu	矩阵 lu 分解，仅支持共享内存

vs_orthog_type

值	描述
cgs	classical Gram-Schmidt
mgs	modified Gram-Schmidt

3.4.6 参考使用样例

$$A = \begin{bmatrix} 2 & 1 & & \\ 1 & 4 & & 1 \\ & & 8 & 2 \\ & 1 & 2 & 10 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & & & \\ & 2 & & \\ & & 3 & \\ & & & 4 \end{bmatrix}$$

单机多线程版使用示例

```
#include <vector>

#include "NCSEigenSolver.h"

int main(int argc, char* argv[])
{
    /* Create matrix A and B */
    int dim = 4;

    double aValue[10] = {2.0, 1.0, 1.0, 4.0, 1.0, 8.0, 2.0, 1.0, 2.0, 10.0};
    int aColIdx[10] = {0, 1, 0, 1, 3, 2, 3, 1, 2 ,3};
    int aRowPtr[5] = {0, 2, 5, 7, 10};

    double bValue[4] = {1.0, 2.0, 3.0, 4.0};
    int bColIdx[4] = {0, 1, 2, 3};
    int bRowPtr[5] = {0, 1, 2, 3, 4};

    /* Solve the problem */
    NCS::EIG::NCSEigenSolver<double> solver;

    /* Set options */
    solver.SetOption("--eps_thread", "8");
    solver.SetOption("--eps_nev", "2");
    solver.SetOption("--eps_which", "TM");
    solver.SetOption("--eps_type", "krylovschur");
    solver.SetOption("--eps_st_type", "sinvert");
    solver.SetOption("--eps_st_pc_type", "ldl");

    /* Solve */
    int status = solver.Solve(dim, aValue, aColIdx, aRowPtr, bValue, bColIdx,␣
↪bRowPtr);

    /* Extract result */
    int nConv = solver.GetConverged();
    int its = solver.GetIter();

    /** Eigenvalue **/
    int nev = nConv;
    std::vector<double> eigReal, eigImg;
    eigReal.resize(nev);
    eigImg.resize(nev);
    double* pEigReal = eigReal.data();
    double* pEigImg = eigImg.data();
    for (int i = 0; i < nev; i++) {
        solver.GetEigenValue(i, pEigReal+i, pEigImg+i);
    }

    /** eigenvectors **/
    std::vector<double> vecReal, vecImg;
    vecReal.resize(dim);
    vecImg.resize(dim);
    double *pVecReal = vecReal.data();
    double *pVecImg = vecImg.data();
    solver.GetEigenVector(0, pVecReal, pVecImg);

    /** residual **/
    std::vector<double> residual;
    residual.resize(nev);
    for (int i = 0; i < nev; i++) {
        residual[i] = solver.GetResNorm(i);
```

(续下页)

(接上页)

```
    }

    /** true residual **/
    std::vector<double> trueResidual;
    trueResidual.resize(nev);
    for (int i = 0; i < nev; i++) {
        trueResidual[i] = solver.GetTrueResNorm(i);
    }

    return 0;
}
```

visual studio 用户采用 C 接口，使用示例如下：

```
#include <vector>
#include "NCSEigenSolverWrapper.h"

using namespace NCS::EIG;

int main(int argc, char* argv[])
{
        auto dataType = DataType::DOUBLE;
        /* Create matrix A and B */
        int dim = 4;
        double aValue[10] = { 2.0, 1.0, 1.0, 4.0, 1.0, 8.0, 2.0, 1.0, 2.0, 10.0 };
        int aColIdx[10] = { 0, 1, 0, 1, 3, 2, 3, 1, 2 ,3 };
        int aRowPtr[5] = { 0, 2, 5, 7, 10 };
        double bValue[4] = { 1.0, 2.0, 3.0, 4.0 };
        int bColIdx[4] = { 0, 1, 2, 3 };
        int bRowPtr[5] = { 0, 1, 2, 3, 4 };

        //使用 C 接口创建 nonLinearSolver 实例
        NCSEigenSolverWrapper* eigenSolverWrapper = EigenSolver_new(dataType);

        /* Set options */
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_thread", "8");
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_nev", "2");
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_which", "TM");
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_type",
↪"krylovschur");
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_st_type",
↪"sinvert");
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_st_pc_type", "ldl
↪");

        /* Solve */
        int status = EigenSolverSolve_double(dataType, eigenSolverWrapper, dim,␣
↪aValue, aColIdx, aRowPtr, bValue, bColIdx, bRowPtr);

        /* Extract result */
        int nConv = EigenSolverGetConverged(dataType, eigenSolverWrapper);
        int its = EigenSolverGetIter(dataType, eigenSolverWrapper);

        /** Eigenvalue **/
        int nev = nConv;
        std::vector<double> eigReal, eigImg;
        eigReal.resize(nev);
        eigImg.resize(nev);
        double* pEigReal = eigReal.data();
        double* pEigImg = eigImg.data();
        for (int i = 0; i < nev; i++) {
```

(续下页)

(接上页)

```
                EigenSolverGetEigenValue_double(dataType, eigenSolverWrapper, i,␣
↪pEigReal + i, pEigImg + i);
        }

        /** eigenvectors **/
        std::vector<double> vecReal, vecImg;
        vecReal.resize(dim);
        vecImg.resize(dim);
        double *pVecReal = vecReal.data();
        double *pVecImg = vecImg.data();
        EigenSolverGetEigenVector_double(dataType, eigenSolverWrapper, 0, pVecReal,
↪ pVecImg);

        /** residual **/
        std::vector<double> residual;
        residual.resize(nev);
        for (int i = 0; i < nev; i++) {
                residual[i] = EigenSolverGetResNorm(dataType, eigenSolverWrapper,␣
↪i);
                std::cout << "residual" << "[" << i << "] : " << residual[i] <<␣
↪std::endl;
        }

        /** true residual **/
        std::vector<double> trueResidual;
        trueResidual.resize(nev);
        for (int i = 0; i < nev; i++) {
                trueResidual[i] = EigenSolverGetTrueResNorm(dataType,␣
↪eigenSolverWrapper, i);
                std::cout << "trueResidual" << "[" << i << "] : " <
↪<trueResidual[i] << std::endl;
        }

        EigenSolver_delete(eigenSolverWrapper);

        return 0;
}
```

MPI 分布式版使用示例

使用 2 进程求解

集中式输入示例

```
#include <vector>
#include <mpi.h>

#include "NCSEigenSolver.h"

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    /* Create matrix A and B */
    int dim = 4;
    std::vector<double> aValue, bValue;
```

(续下页)

(接上页)

```
    std::vector<int> aColIdx, aRowPtr, bColIdx, bRowPtr;
    if (rank == 0) {
        aValue = {2.0, 1.0, 1.0, 4.0, 1.0, 8.0, 2.0, 1.0, 2.0, 10.0};
        aColIdx = {0, 1, 0, 1, 3, 2, 3, 1, 2 ,3};
        aRowPtr = {0, 2, 5, 7, 10};

        bValue = {1.0, 2.0, 3.0, 4.0};
        bColIdx = {0, 1, 2, 3};
        bRowPtr = {0, 1, 2, 3, 4};
    }

    /* Solve the problem */
    NCS::EIG::NCSEigenSolver<double> solver;

    /* Set options */
    solver.SetOption("--eps_distributed", "true");
    solver.SetOption("--eps_nev", "2");
    solver.SetOption("--eps_which", "TM");
    solver.SetOption("--eps_type", "krylovschur");
    solver.SetOption("--eps_st_type", "sinvert");
    solver.SetOption("--eps_st_pc_type", "cholesky");

    /* Solve */
    int status = solver.Solve(dim, aValue.data(), aColIdx.data(), aRowPtr.data(),
                              bValue.data(), bColIdx.data(), bRowPtr.data());

    /* Extract result */
    int nConv = solver.GetConverged();
    int its = solver.GetIter();

    /** Eigenvalue **/
    int nev = nConv;
    std::vector<double> eigReal, eigImg;
    eigReal.resize(nev);
    eigImg.resize(nev);
    double* pEigReal = eigReal.data();
    double* pEigImg = eigImg.data();
    for (int i = 0; i < nev; i++) {
        solver.GetEigenValue(i, pEigReal+i, pEigImg+i);
    }

    /** eigenvectors **/
    std::vector<double> vecReal, vecImg;
    if (rank == 0) {
        vecReal.resize(dim);
        vecImg.resize(dim);
    }
    double *pVecReal = vecReal.data();
    double *pVecImg = vecImg.data();
    solver.GetEigenVector(0, pVecReal, pVecImg);

    /** residual **/
    std::vector<double> residual;
    residual.resize(nev);
    for (int i = 0; i < nev; i++) {
        residual[i] = solver.GetResNorm(i);
    }

    /** true residual **/
    std::vector<double> trueResidual;
    trueResidual.resize(nev);
```

(续下页)

(接上页)

```
    for (int i = 0; i < nev; i++) {
        trueResidual[i] = solver.GetTrueResNorm(i);
    }

    MPI_Finalize();

    return 0;
}
```

visual studio 用户采用 C 接口，使用示例如下：

```
#include <vector>
#include <mpi.h>

#include "NCSEigenSolverWrapper.h"

using namespace NCS::EIG;
int main(int argc, char* argv[])
{
        auto dataType = DataType::DOUBLE;
        MPI_Init(&argc, &argv);
        int rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        /* Create matrix A and B */
        int dim = 4;
        std::vector<double> aValue, bValue;
        std::vector<int> aColIdx, aRowPtr, bColIdx, bRowPtr;
        if (rank == 0) {
                aValue = { 2.0, 1.0, 1.0, 4.0, 1.0, 8.0, 2.0, 1.0, 2.0, 10.0 };
                aColIdx = { 0, 1, 0, 1, 3, 2, 3, 1, 2 ,3 };
                aRowPtr = { 0, 2, 5, 7, 10 };
                bValue = { 1.0, 2.0, 3.0, 4.0 };
                bColIdx = { 0, 1, 2, 3 };
                bRowPtr = { 0, 1, 2, 3, 4 };
        }

        //使用 C 接口创建 EigenSolver 实例
        NCSEigenSolverWrapper* eigenSolverWrapper = EigenSolver_new(dataType);

        /* Set options */
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_distributed",
↪"true");
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_nev", "2");
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_which", "TM");
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_type",
↪"krylovschur");
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_st_type",
↪"sinvert");
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_st_pc_type",
↪"cholesky");

        /* Solve */
        int status = EigenSolverSolve_double(dataType, eigenSolverWrapper,
                  dim, aValue.data(), aColIdx.data(), aRowPtr.data(),
                  bValue.data(), bColIdx.data(), bRowPtr.data());

        /* Extract result */
        int nConv = EigenSolverGetConverged(dataType, eigenSolverWrapper);
        int its = EigenSolverGetIter(dataType, eigenSolverWrapper);
```

(续下页)

(接上页)

```
        /** Eigenvalue **/
        int nev = nConv;
        std::vector<double> eigReal, eigImg;
        eigReal.resize(nev);
        eigImg.resize(nev);
        double* pEigReal = eigReal.data();
        double* pEigImg = eigImg.data();
        for (int i = 0; i < nev; i++) {
                EigenSolverGetEigenValue_double(dataType, eigenSolverWrapper, i,␣
↪pEigReal + i, pEigImg + i);
        }

        /** eigenvectors **/
        std::vector<double> vecReal, vecImg;
        if (rank == 0) {
                vecReal.resize(dim);
                vecImg.resize(dim);
        }
        double *pVecReal = vecReal.data();
        double *pVecImg = vecImg.data();
        EigenSolverGetEigenVector_double(dataType, eigenSolverWrapper,
                                         0, pVecReal, pVecImg);

        /** residual **/
        std::vector<double> residual;
        residual.resize(nev);
        for (int i = 0; i < nev; i++) {
                residual[i] = EigenSolverGetResNorm(dataType, eigenSolverWrapper,␣
↪i);
        }

        /** true residual **/
        std::vector<double> trueResidual;
        trueResidual.resize(nev);
        for (int i = 0; i < nev; i++) {
                trueResidual[i] = EigenSolverGetTrueResNorm(dataType,␣
↪eigenSolverWrapper, i);
        }

        EigenSolver_delete(eigenSolverWrapper);
        MPI_Finalize();

        return 0;
}
```

分布式输入示例

```
#include <vector>

#include <mpi.h>

#include "NCSEigenSolver.h"

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

(续下页)

(接上页)

```
    /* Create matrix A and B */
    int dim = 4;
    std::vector<double> aValue, bValue;
    std::vector<int> aColIdx, aRowPtr, bColIdx, bRowPtr;
    int localSize = 2;
    int rowBegin;
    if (rank == 0) {
        rowBegin = 0;
        aValue = {2.0, 1.0, 1.0, 4.0, 1.0};
        aColIdx = {0, 1, 0, 1, 3};
        aRowPtr = {0, 2, 5};

        bValue = {1.0, 2.0};
        bColIdx = {0, 1};
        bRowPtr = {0, 1, 2};
    }
    if (rank == 1) {
        rowBegin = 2;
        aValue = {8.0, 2.0, 1.0, 2.0, 10.0};
        aColIdx = {2, 3, 1, 2 ,3};
        aRowPtr = {0, 2, 5};

        bValue = {3.0, 4.0};
        bColIdx = {2, 3};
        bRowPtr = {0, 1, 2};
    }

    /* Solve the problem */
    NCS::EIG::NCSEigenSolver<double> solver;

    /* Set options */
    solver.SetOption("--eps_distributed", "true");
    solver.SetOption("--eps_master_input_only", "false");
    solver.SetOption("--eps_local_begin_row", std::to_string(rowBegin));
    solver.SetOption("--eps_local_size", std::to_string(localSize));

    solver.SetOption("--eps_nev", "2");
    solver.SetOption("--eps_which", "TM");
    solver.SetOption("--eps_type", "krylovschur");
    solver.SetOption("--eps_st_type", "sinvert");
    solver.SetOption("--eps_st_pc_type", "cholesky");

    /* Solve */
    int status = solver.Solve(dim, aValue.data(), aColIdx.data(), aRowPtr.data(),
                              bValue.data(), bColIdx.data(), bRowPtr.data());

    /* Extract result */
    int nConv = solver.GetConverged();
    int its = solver.GetIter();

    /** Eigenvalue **/
    int nev = nConv;
    std::vector<double> eigReal, eigImg;
    eigReal.resize(nev);
    eigImg.resize(nev);
    double* pEigReal = eigReal.data();
    double* pEigImg = eigImg.data();
    for (int i = 0; i < nev; i++) {
        solver.GetEigenValue(i, pEigReal+i, pEigImg+i);
    }
```

(续下页)

(接上页)

```
    /** eigenvectors **/
    std::vector<double> vecReal, vecImg;
    vecReal.resize(localSize);
    vecImg.resize(localSize);
    double *pVecReal = vecReal.data();
    double *pVecImg = vecImg.data();
    solver.GetEigenVector(0, pVecReal, pVecImg);

    /** residual **/
    std::vector<double> residual;
    residual.resize(nev);
    for (int i = 0; i < nev; i++) {
        residual[i] = solver.GetResNorm(i);
    }

    /** true residual **/
    std::vector<double> trueResidual;
    trueResidual.resize(nev);
    for (int i = 0; i < nev; i++) {
        trueResidual[i] = solver.GetTrueResNorm(i);
    }

    MPI_Finalize();

    return 0;
}
```

visual studio 用户采用 C 接口，使用示例如下：

```
#include <vector>
#include <mpi.h>

#include "NCSEigenSolverWrapper.h"

using namespace NCS::EIG;
int main(int argc, char* argv[])
{
        auto dataType = DataType::DOUBLE;
        MPI_Init(&argc, &argv);
        int rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        /* Create matrix A and B */
        int dim = 4;
        std::vector<double> aValue, bValue;
        std::vector<int> aColIdx, aRowPtr, bColIdx, bRowPtr;
        int localSize = 2;
        int rowBegin;
        if (rank == 0) {
                rowBegin = 0;
                aValue = { 2.0, 1.0, 1.0, 4.0, 1.0 };
                aColIdx = { 0, 1, 0, 1, 3 };
                aRowPtr = { 0, 2, 5 };
                bValue = { 1.0, 2.0 };
                bColIdx = { 0, 1 };
                bRowPtr = { 0, 1, 2 };
        }
        if (rank == 1) {
                rowBegin = 2;
                aValue = { 8.0, 2.0, 1.0, 2.0, 10.0 };
                aColIdx = { 2, 3, 1, 2 ,3 };
```

(续下页)

(接上页)

```
                aRowPtr = { 0, 2, 5 };
                bValue = { 3.0, 4.0 };
                bColIdx = { 2, 3 };
                bRowPtr = { 0, 1, 2 };
        }

        //使用 C 接口创建 EigenSolver 实例
        NCSEigenSolverWrapper* eigenSolverWrapper = EigenSolver_new(dataType);

        /* Set options */
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_distributed",
↪"true");
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_master_input_only
↪", "false");
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_local_begin_row",
↪ std::to_string(rowBegin).c_str());
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_local_size",␣
↪std::to_string(localSize).c_str());
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_nev", "2");
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_which", "TM");
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_type",
↪"krylovschur");
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_st_type",
↪"sinvert");
        EigenSolverSetOption(dataType, eigenSolverWrapper, "--eps_st_pc_type",
↪"cholesky");

        /* Solve */
        int status = EigenSolverSolve_double(dataType, eigenSolverWrapper,
                    dim, aValue.data(), aColIdx.data(), aRowPtr.data(),
                    bValue.data(), bColIdx.data(), bRowPtr.data());

        /* Extract result */
        int nConv = EigenSolverGetConverged(dataType, eigenSolverWrapper);
        int its = EigenSolverGetIter(dataType, eigenSolverWrapper);

        /** Eigenvalue **/
        int nev = nConv;
        std::vector<double> eigReal, eigImg;
        eigReal.resize(nev);
        eigImg.resize(nev);
        double* pEigReal = eigReal.data();
        double* pEigImg = eigImg.data();
        for (int i = 0; i < nev; i++) {
                EigenSolverGetEigenValue_double(dataType, eigenSolverWrapper,
                                                i, pEigReal + i, pEigImg + i);
        }

        /** eigenvectors **/
        std::vector<double> vecReal, vecImg;
        vecReal.resize(localSize);
        vecImg.resize(localSize);
        double *pVecReal = vecReal.data();
        double *pVecImg = vecImg.data();
        EigenSolverGetEigenVector_double(dataType, eigenSolverWrapper,
                                         0, pVecReal, pVecImg);

        /** residual **/
        std::vector<double> residual;
        residual.resize(nev);
        for (int i = 0; i < nev; i++) {
```

(续下页)

(接上页)

```
                residual[i] = EigenSolverGetResNorm(dataType, eigenSolverWrapper,␣
↪i);
        }

        /** true residual **/
        std::vector<double> trueResidual;
        trueResidual.resize(nev);
        for (int i = 0; i < nev; i++) {
                trueResidual[i] = EigenSolverGetTrueResNorm(dataType,␣
↪eigenSolverWrapper, i);
        }

        EigenSolver_delete(eigenSolverWrapper);
        MPI_Finalize();

        return 0;
}
```

对于复数版本的调用，根据接口的参数类型要求把 double 改为 cComplex_double，string 类型改为 char[]，auto dataType = DataType::DOUBLE 改为 auto dataType = DataType::COMPLEX_DOUBLE。

3.5 AI 辅助求解接口说明

3.5.1 AI 辅助求解接口整体说明

AI 辅助求解对用户提供的接口封装在 NCSAIHelper.h 中，其具体的实现为：

```
template <typename Scalar>
class NCSAIHelper {
public:
    NCSAIHelper() = default;
    virtual ~NCSAIHelper() = default;

    /* 初始化接口 */
    virtual bool Init() = 0;

    /* 参数设置接口，用于初始化前，在调用Init之后参数才会生效 */
    virtual void SetOption(const std::string& op, const std::string& value) = 0;

    /* 参数更改接口，用于使用途中，会使参数立即生效*/
    virtual void ChangeOption(const std::string& op, const std::string& value) = 0;

    /* 用于迭代法求解辅助的接口 */
    virtual bool RunForIterative(const std::string& name, NCS::MatType matType,␣
↪const Scalar* data, const int* colIdx,
                                 const int* rowPtr, int dim, const Scalar* rhs,␣
↪const Scalar* iniSol) = 0;

    virtual HelperType Type() = 0;
};

// 根据AI辅助类型，创建相应helper的工厂方法
template <typename Scalar>
std::shared_ptr<NCSAIHelper<Scalar>> CreateAIHelper(HelperType type);
```

模板参数 Scalar 表示浮点数的类型，目前支持如下类型：

- `double`：实数双精度

3.5.2 AI 辅助求解类型说明

类型用于创建相应的 helper，当前只支持 KSP 和 PC 智能选参

```
enum class HelperType : int32_t { KSP_PC_RECOMMENDER = 0 };
```

3.5.3 AI 辅助求解参数说明

3.5.4 KSP&PC 智能选参使用样例

通用使用样例

当前 KSP&PC 智能选参仅支持实数串行求解场景，步骤如下：

1. 初始化智能选参

```
std::shared_ptr<NCSAIHelper<double>> recommender = CreateAIHelper<double>
↪(HelperType::KSP_PC_RECOMMENDER);
recommender->Init();
```

2. 线性迭代法求解器设置智能选参模块，可重复使用，比如：

```
NCSIterativeSolver<double> linearSolver1;
linearSolver1.SetAIHelper(recommender);
NCSIterativeSolver<double> linearSolver2;
linearSolver2.SetAIHelper(recommender);
```

3. 线性迭代法求解器求解时，可通过设置矩阵类型，提升智能选参效果。默认矩阵类型为 NCS::MatType::REAL_GENERAL

```
linearSolver.Solve(data, colIdx, rowPtr, 3, b, iniSol, sol); //非对称矩阵
linearSolver.Solve(data, colIdx, rowPtr, 3, b, iniSol, sol, NCS::MatType::REAL_
↪SYMMETRIC_POSITIVE_DEFINITE); //对称矩阵
```

参考代码样例如下，建议设置 ksp_type 和 pc_type，以避免智能选参失败而导致求解程序中止：

```
#include <NCSAIHelper.h>

using std::string;
using namespace NCS::KSP;
using namespace NCS::AI4SOLVER;
int main(int argc, char *argv[])
{
    /* -------------------- 初始化KSP&PC智能选参 --------------------- */
    std::shared_ptr<NCSAIHelper<double>> recommender = CreateAIHelper<double>
↪(HelperType::KSP_PC_RECOMMENDER);
    recommender->Init();

    /* -------------------- 设置系数矩阵及右端项 --------------------- */
    // A matrix
    // [1 0 1]
    // [0 2 1]
    // [2 0 1]
    double data[6] = {1., 1., 2., 1., 2., 1.};
    int colIdx[6] = {0, 2, 1, 2, 0, 2};
    int rowPtr[4] = {0, 2, 4, 6};
    // right hand side
    double b[3] = {2., 3., 3.};
    double iniSol[3] = {0., 0., 0.};
    double sol[3];
```

(续下页)

(接上页)

```
    /* ----------------------  设置求解参数 ------------------------- */
    NCSIterativeSolver<double> linearSolver;
    linearSolver.SetAIHelper(recommender);
    linearSolver.SetOption("--ksp_type", "gmres"); // ksp type, 建议设置
    linearSolver.SetOption("--pc_type", "jacobi"); // pc type, 建议设置
    linearSolver.SetOption("--ksp_max_iteration", "10000");
    linearSolver.SetOption("--pc_jacobi_type", "diagonal");

    /* ---------------------------- 求解 ----------------------------- */
    auto status = linearSolver.Solve(data, colIdx, rowPtr, 3, b, iniSol, sol);
    int iterNum = linearSolver.GetIter();
    auto res = linearSolver.GetTrueResNorm();
    return 0;
}
```

使用 visual studio 的用户调用 C 接口，参考代码样例如下：

```
#include <iostream>
#include <NCSIterativeSolverWrapper.h>

using std::string;
using namespace NCS::KSP;
using namespace NCS::AI4SOLVER;
int main(int argc, char *argv[])
{
        auto dataType = DataType::DOUBLE;
        /* -------------------- 设置系数矩阵及右端项--------------------- */
        // A matrix
        // [1 0 1]
        // [0 2 1]
        // [2 0 1]
        double data[6] = { 1., 1., 2., 1., 2., 1. };
        int colIdx[6] = { 0, 2, 1, 2, 0, 2 };
        int rowPtr[4] = { 0, 2, 4, 6 };
        // right hand side
        double b[3] = { 2., 3., 3. };
        double iniSol[3] = { 0., 0., 0. };
        double sol[3];

        /* ----------------------- 设置求解参数------------------------- */
        //使用 C 接口创建 NCSIterativeSolver 实例
        NCSIterativeSolverWrapper* linearSolverWrapper = LinearSolver_
↪new(dataType);
        // 初始化KSP&PC智能选参
        LinearSolverSetAIHelper(dataType, linearSolverWrapper, HelperType::KSP_PC_
↪RECOMMENDER);
        LinearSolverSetOption(dataType, linearSolverWrapper, "--ksp_type", "gmres
↪");
        LinearSolverSetOption(dataType, linearSolverWrapper, "--pc_type", "jacobi
↪");
        LinearSolverSetOption(dataType, linearSolverWrapper, "--ksp_max_iteration",
↪ "10000");
        LinearSolverSetOption(dataType, linearSolverWrapper, "--pc_jacobi_type",
↪"diagonal");

        /* ---------------------------- 求解----------------------------- */
        auto status = LinearSolverSolve_double(dataType, linearSolverWrapper,
                data, colIdx, rowPtr, 3, b, iniSol, sol);
        int iterNum = LinearSolverGetIter(dataType, linearSolverWrapper);
        auto res = LinearSolverGet(dataType, linearSolverWrapper,␣
```

(续下页)

(接上页)

```
↪Get::GetTrueResNorm);
        auto res2 = LinearSolverGet(dataType, linearSolverWrapper,␣
↪Get::GetResNorm);

        LinearSolver_delete(linearSolverWrapper);
        return 0;
}
```

使用历史信息的使用样例

1. 设置可以使用历史推荐的参数

```
recommender->SetOption("--use_history", "true"); //初始化前
recommender->ChangeOption("--use_history", "true"); //使用中设置
```

2. 初次求解时，使用名称标识符保存矩阵推荐参数。当标识符为空字符串时，将不会触发该功能。

```
linearSolver.Solve(data, colIdx, rowPtr, 3, b, iniSol, sol, NCS::MatType::REAL_
↪GENERAL, "Ux");
linearSolver.Solve(data, colIdx, rowPtr, 3, b, iniSol, sol, NCS::MatType::REAL_
↪SYMMETRIC_POSITIVE_DEFINITE, "p");
linearSolver.Solve(data, colIdx, rowPtr, 3, b, iniSol, sol, NCS::MatType::REAL_
↪GENERAL, ""); //结果不会保存
```

   后续求解时，如果输入同样名称的矩阵，将会直接启用历史推荐参数

3. 需要取消，或者需要刷新历史参数时，可以通过参数更改接口设置

```
recommender->ChangeOption("--use_history", "false"); //取消设置
```

   注意：如果继续使用原来的名称标识符求解，历史参数会被覆盖直到再次启用相关设置，请结合实际情况使用

```
linearSolver.Solve(data, colIdx, rowPtr, 3, b, iniSol, sol, NCS::MatType::REAL_
↪GENERAL, "Ux"); //原来的Ux推荐参数将会被覆盖
linearSolver.Solve(data, colIdx, rowPtr, 3, b, iniSol, sol, NCS::MatType::REAL_
↪GENERAL, ""); //原来的Ux推荐参数不会被覆盖
```

3.6 SDK 快速开始

3.6.1 迭代法

C++ 接口

二进制可执行文件

在 bin 文件夹下提供了如下可执行文件 interface-iterative-ncs-serial、interface-iterative-ncs 和 interface-iterative-ncs-dist-input 三个可执行程序。

参数说明

- `--mtxF/--rhsF`：对应矩阵和右端项文件。
- `--mtxFs/--rhsFs`：对应矩阵和右端项文件，分布式输入，注意分布式文件输入要求每个文件的矩阵从全局索引开始。
- `--partitioning_type`：表示输入数据的类型，可选 `naive`（默认值）和 `self`。`naive` 表示分布式数据输入，`self` 表示主进程输入数据。预留接口，为后续可执行程序的合并为一。
- 其余参数：–ksp 和–pc 前缀开头等其余参数，不加额外说明，都是算法参数，算法参数含义可查看接口文档章节 *线性迭代法求解接口说明*。

集中式数据格式

下面所使用到的数据格式如下：

AA.mtx ——3 行 3 列，6 个非零元

3 3 6 1 1 1 1 3 1 2 2 2 2 3 1 3 1 2 3 3 1

bb.mtx——3 行 1 列

3 1 2 3 3

分布式式数据格式 1

demo_mtxs_2_mpi.txt——以路径形式给定多段数据 A 输入

./A1.mtx ./A2.mtx

A1.mtx——3 行 3 列，4 个非零元

3 3 4 1 1 1 1 3 1 2 2 2 2 3 1

A2.mtx——3 行 3 列，2 个非零元

3 3 2 3 1 2 3 3 1

demo_mtxs_2_mpi_b.txt——以路径形式给定多段数据 b 输入

./A1b.mtx ./A2b.mtx

A1b.mtx 2 行 1 列

2 1 2 3

A2b.mtx 1 行 1 列

1 1 3

interface-iterative-ncs-serial

串行和共享内存版本 NCSolver 端到端程序，示例使用方式如下：

串行使用：

```
interface-iterative-ncs-serial  --mtxF AA.mtx --rhsF bb.mtx --ksp_type bicgstab --
↪ksp_threads 1  --pc_type amg_agg --pc_amg_agg_pre_strength_sym_type symmetric --
↪pc_amg_agg_assumed_sym_type symmetric --pc_amg_agg_restrict_storage_type␣
↪onthefly --pc_amg_agg_num_aggressive_layers 1 --pc_amg_agg_thresh 0.0 --pc_amg_
↪agg_coarsen_type mis --pc_amg_agg_jacobi_interp 1 --pc_amg_agg_num_lev_max 10 --
↪pc_amg_agg_coarse_eq_lim 50 --pc_eig_est_type svd --pc_amg_agg_cycle_type multv -
↪-pc_amg_agg_smoother_type sor --pc_amg_agg_smoother_max_ite 1 --pc_amg_agg_
↪jacobi_damping_omega 1.2 --update_flag false
```

共享内存使用：

```
export OMP_NUM_THREADS=2;interface-iterative-ncs-serial  --mtxF AA.mtx --rhsF bb.
↪mtx --ksp_type bicgstab --ksp_threads 1  --pc_type amg_agg --pc_amg_agg_pre_
↪strength_sym_type symmetric --pc_amg_agg_assumed_sym_type symmetric --pc_amg_agg_
↪restrict_storage_type onthefly --pc_amg_agg_num_aggressive_layers 1 --pc_amg_agg_
↪thresh 0.0 --pc_amg_agg_coarsen_type mis --pc_amg_agg_jacobi_interp 1 --pc_amg_
↪agg_num_lev_max 10 --pc_amg_agg_coarse_eq_lim 50 --pc_eig_est_type svd --pc_amg_
↪agg_cycle_type multv --pc_amg_agg_smoother_type sor --pc_amg_agg_smoother_max_
↪ite 1 --pc_amg_agg_jacobi_damping_omega 1.2 --update_flag false
```

interface-iterative-ncs

集中式数据输入 MPI 版本 NCSolver 端到端程序，示例使用方式如下：

```
mpirun -np 2 interface-iterative-ncs  --mtxF AA.mtx --rhsF bb.mtx --ksp_type␣
↪bicgstab --ksp_threads 1  --pc_type amg_agg --pc_amg_agg_pre_strength_sym_type␣
↪symmetric --pc_amg_agg_assumed_sym_type symmetric --pc_amg_agg_restrict_storage_
↪type onthefly --pc_amg_agg_num_aggressive_layers 1 --pc_amg_agg_thresh 0.0 --pc_
↪amg_agg_coarsen_type mis --pc_amg_agg_jacobi_interp 1 --pc_amg_agg_num_lev_max␣
↪10 --pc_amg_agg_coarse_eq_lim 50 --pc_eig_est_type svd --pc_amg_agg_cycle_type␣
↪multv --pc_amg_agg_smoother_type sor --pc_amg_agg_smoother_max_ite 1 --pc_amg_
↪agg_jacobi_damping_omega 1.2 --update_flag false --partitioning_type naive
```

interface-iterative-ncs-dist-input

分布式数据输入 MPI 版本 NCSolver 端到端程序，示例使用方式如下：

```
mpirun -np 2 interface-iterative-ncs-dist-input  --mtxFs demo_mtxs_2_mpi.txt --
↪rhsFs demo_mtxs_2_mpi_b.txt --ksp_type bicgstab --ksp_threads 1  --pc_type amg_
↪agg --pc_amg_agg_pre_strength_sym_type symmetric --pc_amg_agg_assumed_sym_type␣
↪symmetric --pc_amg_agg_restrict_storage_type onthefly --pc_amg_agg_num_
↪aggressive_layers 1 --pc_amg_agg_thresh 0.0 --pc_amg_agg_coarsen_type mis --pc_
↪amg_agg_jacobi_interp 1 --pc_amg_agg_num_lev_max 10 --pc_amg_agg_coarse_eq_lim␣
↪50 --pc_eig_est_type svd --pc_amg_agg_cycle_type multv --pc_amg_agg_smoother_
↪type sor --pc_amg_agg_smoother_max_ite 1 --pc_amg_agg_jacobi_damping_omega 1.2 --
↪update_flag false --partitioning_type self
```

数据 IO

矩阵和右端项读入和结果写出接口。

开发中……

数据生成

特定要求的矩阵生成和缺失右端项时的右端项生成。

开发中……

数据修改

解耦非零元位置 pattern 和数值，提供数值独立修改功能，使嵌入时间发展问题更加灵活。
开发中……

数据批量计算

方程组求解问题，批量计算。
开发中……

Python 接口

开发中……

Java 接口

开发中……

3.6.2 直接法

开发中……

3.6.3 特征值问题

开发中……

CHAPTER 4

CHANGELOG

...