有限元编程指北

作者： W. Huang

时间： Jan 10, 2024

目录

第 1 章 deal.II 的安装与使用

本教程在 Ubuntu 22.04 上亲测可用。

1.1 安装教程

首先，用 apt 安装一些必要的包（已安装的包可以跳过）：

```
sudo apt install gcc
sudo apt install g++
sudo apt install gfortran
sudo apt install make
sudo apt install cmake
sudo apt install libopenmpi-dev
sudo apt install libboost-dev
sudo apt install libnetcdf-dev
sudo apt install libmatio-dev
sudo apt install libhdf5-dev
sudo apt install libmetis-dev
```

点此下载 Trilinos 13.4.0。下载完成后解压，创建文件 `install.sh`，将以下内容写入：

```
cd Trilinos-trilinos-release-13-4-0
mkdir build
cd build

cmake                                       \
-DTrilinos_ENABLE_Amesos=ON                 \
-DTrilinos_ENABLE_Epetra=ON                 \
-DTrilinos_ENABLE_EpetraExt=ON              \
-DTrilinos_ENABLE_Ifpack=ON                 \
-DTrilinos_ENABLE_AztecOO=ON                \
-DTrilinos_ENABLE_Sacado=ON                 \
-DTrilinos_ENABLE_SEACAS=ON                 \
-DTrilinos_ENABLE_Teuchos=ON                \
-DTrilinos_ENABLE_MueLu=ON                  \
-DTrilinos_ENABLE_ML=ON                     \
-DTrilinos_ENABLE_ROL=ON                    \
-DTrilinos_ENABLE_Tpetra=ON                 \
-DTrilinos_ENABLE_COMPLEX_DOUBLE=ON         \
-DTrilinos_ENABLE_COMPLEX_FLOAT=ON          \
-DTrilinos_ENABLE_Zoltan=ON                 \
-DTrilinos_VERBOSE_CONFIGURE=OFF            \
-DTPL_ENABLE_MPI=ON                         \
-DBUILD_SHARED_LIBS=ON                      \
-DCMAKE_VERBOSE_MAKEFILE=OFF                \
-DCMAKE_BUILD_TYPE=RELEASE                  \
-DCMAKE_INSTALL_PREFIX:PATH=$HOME/share/trilinos \
../
```

```
make install -j8
```

保存，然后打开终端，执行：

```
bash install.sh
```

需要等待大约半小时，在等待的过程中可以先把接下来的 p4est 安装好。

点此下载 p4est 2.8.5.5。下载完成后解压，创建文件 `p4est-setup.sh`，点击打开脚本链接，复制全部内容，将其写入刚刚创建的文件并保存。现在打开终端，运行：

```
chmod u+x p4est-setup.sh
./p4est-setup.sh p4est-2.8.5.5-9ddbb.tar.gz $HOME/share/p4est
```

点此下载 deal.II 9.5.1。解压，并进入解压后的文件夹。等 Trilinos 安装完成后，打开终端，执行：

```
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/usr/local \
-DDEAL_II_WITH_MPI=ON        \
-DDEAL_II_WITH_METIS=ON      \
-DDEAL_II_WITH_UMFPACK=ON    \
-DDEAL_II_WITH_P4EST=ON      \
-DP4EST_DIR=$HOME/share/p4est \
-DDEAL_II_WITH_TRILINOS=ON -DTRILINOS_DIR=$HOME/share/trilinos/ \
../
sudo make install -j8
```

安装完成后可以执行测试（也可以不测试）：

```
make test
```

1.2 使用方法

从附件的任何一个代码文件夹里拷贝一份 `CMakeLists.txt` 出来，放到你的代码目录中。假设你的代码文件为 `yourcode.cc`，打开刚刚拷贝的文件，将里面的第 6 行改为：

```
set(TARGET "yourcode")
```

保存后打开终端，依次执行：

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

即可编译你的程序，然后输入

```
./yourcode
```

即可运行。

第 2 章 椭圆方程

2.1 变分形式推导

考虑齐次 Neumann 边界条件的椭圆方程

$$\begin{cases} -\Delta u + u = f, & \text{in } \Omega, \\ \mathbf{n} \cdot \nabla u = 0, & \text{on } \partial\Omega. \end{cases} \tag{2.1}$$

与测试函数作积分，得到

$$(-\Delta u + u, \phi) = (f, \phi). \tag{2.2}$$

通过分部积分降一阶，并将齐次 Neumann 边界条件代入，得到

$$(\nabla u, \nabla \phi) + (u, \phi) = (f, \phi). \tag{2.3}$$

从而得到有限元空间中的变分形式

$$(\nabla u_h, \nabla \phi_h) + (u_h, \phi_h) = (f, \phi_h), \quad \forall \phi_h \in V_h. \tag{2.4}$$

对应的稀疏线性系统为：

$$A\mathbf{u} = \mathbf{f}, \tag{2.5}$$

其中

$$A(i,j) = (\nabla \phi_i, \nabla \phi_j) + (\phi_i, \phi_j), \quad i,j = 1, ..., N. \tag{2.6}$$

$$\mathbf{f}(i) = (f, \phi_i), \quad i = 1, ..., N. \tag{2.7}$$

$\phi_1, ..., \phi_N$ 是所有的基函数。

这一章中，我们采用精确解 $u(x,y) = \cos(\pi x)\cos(\pi y)$ 进行测试，右端项 $f = 2\pi^2 u$。

2.2 编程指北 1：最基础的求解技术

2.2.1 框架

首先引入必要的头文件：

```
#include <deal.II/grid/tria.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/grid_out.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/fe/fe_simplex_p.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/fe/mapping_fe.h>
#include <deal.II/dofs/dof_renumbering.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/function.h>
#include <deal.II/numerics/vector_tools.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/lac/vector.h>
#include <deal.II/lac/full_matrix.h>
#include <deal.II/lac/sparse_matrix.h>
```

```
#include <deal.II/lac/dynamic_sparsity_pattern.h>
#include <deal.II/lac/solver_cg.h>
#include <deal.II/lac/precondition.h>
#include <deal.II/numerics/data_out.h>
using namespace dealii;
```

定义精确解与右端项：

```
class Solution : public Function<2>{
public:
  double value(const Point<2> & p, const unsigned int component = 0) const{
    return cos(M_PI*p[0]) * cos(M_PI*p[1]);
  };

  virtual Tensor<1, 2>
  gradient(const Point<2> & p, const unsigned int component = 0) const{
    Tensor<1, 2> grad;
    grad[0] = -M_PI * sin(M_PI*p[0]) * cos(M_PI*p[1]);
    grad[1] = -M_PI * cos(M_PI*p[0]) * sin(M_PI*p[1]);
    return grad;
  };
};

double f(const Point<2> &p){
  return (2*M_PI*M_PI) * cos(M_PI*p[0]) * cos(M_PI*p[1]);
}
```

接下来我们定义有限元方法的类。有限元算法的主要步骤为：生成网格、初始化稀疏线性系统、装配刚度矩阵与右端项、求解稀疏线性系统、输出结果。在这些步骤中，有一些重要的东西，它们应该作为成员变量：网格、有限元系统（这里我们选择 $\mathcal{P}_k$ 元系统 FE_SimplexP）、自由度处理器、稀疏模式（即标记稀疏矩阵非零元素位置的工具）、稀疏矩阵、右端项向量、解向量。我们再引入一个 level 变量，表示网格的加密层级。最终，我们需要向用户提供一个构造函数和一个“一键运行”的函数。所以我们的类定义如下：

```
class Elliptic{
public:
  Elliptic(const int &level);
  void run();

private:
  void make_grid();
  void setup_system();
  void assemble_system();
  void solve();
  void output_results() const;

  int level;
  Triangulation<2> triangulation;
  FE_SimplexP<2> fe;
  DoFHandler<2>  dof_handler;
```

```
  SparsityPattern   sparsity_pattern;
  SparseMatrix<double> system_matrix;

  Vector<double> solution;
  Vector<double> system_rhs;
};
```

在构造函数中，我们需要把 $\mathcal{P}_k$ 元系统的 k 定下来，这里我们选择线性元，即 $k=1$。我们还需要把自由度处理器与网格绑定。同时我们为用户提供一个参数，用于输入网格加密层级。

```
Elliptic::Elliptic(const int &level)
: level(level),
  fe(1),
  dof_handler(triangulation)
{}
```

现在，让我们先把“一键运行”的函数给写了：

```
void Elliptic::run()
{
  make_grid();
  setup_system();
  assemble_system();
  solve();
  output_results();
}
```

2.2.2 生成网格

接下来我们依次实现有限元算法的每一个步骤。第一步是生成网格，我们采用规则的三角网格，并进行 level 次加密。

```
void Elliptic::make_grid()
{
  GridGenerator::subdivided_hyper_cube_with_simplices(triangulation, 1);
  triangulation.refine_global(level);
}
```

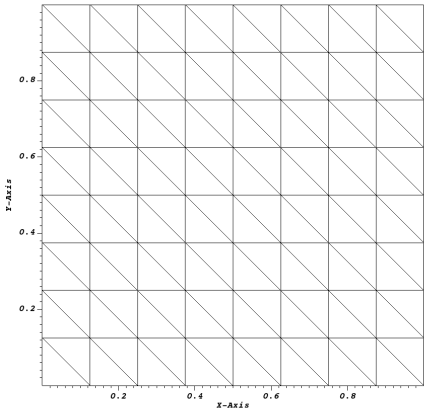

图 2.1: level=3 时生成的网格

2.2.3 初始化稀疏线性系统

接下来是初始化稀疏线性系统。首先需要将自由度处理器与有限元系统绑定，自由度处理器会根据你的有限元系统生成所需的自由度，例如在线性元系统中，每个网格顶点就是一个自由度。我们还可以通过 `dof_handler.n_dofs()` 知道自由度的总数。

```
void Elliptic::setup_system()
{
  dof_handler.distribute_dofs(fe);
  std::cout << "DoF Nunbers: " << dof_handler.n_dofs() << std::endl;
```

在 deal.II 中，稀疏矩阵的初始化需要一个“稀疏模式”，它用于标记稀疏矩阵中所有可能的非零元素位置，例如在线性元中，当自由度 i 与 j 对应的顶点相邻时，我们就说这两个自由度是**关联**的，矩阵的 (i,j) 位置就有可能非零，从而应该出现在稀疏模式中。deal.II 为我们提供了自动生成稀疏模式的函数，它生成一个标记了所有关联自由度动态稀疏模式（意味着你仍然可以自己向里面加一些东西，但这里不需要），然后我们将它拷贝为静态稀疏模式。

```
  DynamicSparsityPattern dynamic_sparsity_pattern(dof_handler.n_dofs(),
                                                  dof_handler.n_dofs());
  DoFTools::make_sparsity_pattern(dof_handler, dynamic_sparsity_pattern);
  sparsity_pattern.copy_from(dynamic_sparsity_pattern);
```

接下来可以根据稀疏模式来初始化稀疏矩阵了，同时我们也初始化向量。

```
  system_matrix.reinit(sparsity_pattern);

  solution.reinit(dof_handler.n_dofs());
  system_rhs.reinit(dof_handler.n_dofs());
}
```

2.2.4 装配刚度矩阵

接下来是装配刚度矩阵，**重点!!!** 首先我们要决定在每个网格单元上用何种积分公式来计算积分，由于我们需要计算 (ϕ_i, ϕ_j)，它是一个二次函数的积分，因此我们希望积分公式的代数精度至少为 2 阶，因此我们选择 2 点高斯积分公式（具有 3 阶代数精度）。

```
void Elliptic::assemble_system()
{
  QGaussSimplex<2> quadrature_formula(fe.degree + 1);
```

接下来，FEValues 将为你提供每个网格单元中每个积分节点上的信息，在任何一个网格单元中，我们需要知道基函数在每个积分节点上的取值、基函数的梯度在每个积分节点上的取值、积分节点对应的权重、积分节点的位置（用于代入 f 求右端项）。因此我们初始化如下：

```
FEValues<2> fe_values(fe,
                      quadrature_formula,
                      update_values |
                      update_gradients |
                      update_JxW_values |
                      update_quadrature_points);
```

我们装配刚度矩阵的思路是这样的：枚举每一个三角形单元 $\mathcal{K}$，这个三角形单元上的所有自由度为 $1,2,...,m$（例如线性元中为 $1,2,3$，对应于三角形单元三个顶点处的值；二次元中为 $1,2,3,4,5,6$，对应于三角形单元三个顶点和三条边的中点处的值），对应基函数 $\phi_1,\phi_2,...,\phi_m$。我们计算

$$A_{\mathcal{K}}(i,j)=(\nabla\phi_i,\nabla\phi_j)_{\mathcal{K}}+(\phi_i,\phi_j)_{\mathcal{K}},\quad i,j=1,2,...,m. \tag{2.8}$$

这叫 **局部刚度矩阵**，计算 $\mathcal{K}$ 上的数值积分时，我们需要枚举积分节点，求积分节点上的值，并乘上积分节点的权重，然后累加。然后我们需要知道三角形单元上的自由度 $1,2,...,m$ 在所有自由度中的编号 $d_1,d_2,...,d_m$（我们称作**索引数组**），然后将局部刚度矩阵“贡献”到全局刚度矩阵中：

$$A(d_i,d_j)+=A_{\mathcal{K}}(i,j),\quad i,j=1,2,...,m. \tag{2.9}$$

对于右端项，我们计算

$$\mathbf{f}_{\mathcal{K}}(i)=(f,\phi_i)_{\mathcal{K}},\quad i=1,2,...,m. \tag{2.10}$$

这叫 **局部右端项**，同样的，需要枚举积分节点来计算，最后我们将它“贡献”到全局右端项中：

$$\mathbf{f}(d_i)+=\mathbf{f}_{\mathcal{K}}(i),\quad i=1,2,...,m. \tag{2.11}$$

我们已经知道了原理，现在来写代码。首先是确定每个三角单元里的自由度数量（例如线性元为 3），然后定义局部刚度矩阵与局部右端项，然后是索引数组。

```
  const unsigned int dofs_per_cell = fe.n_dofs_per_cell();

  FullMatrix<double> cell_matrix(dofs_per_cell, dofs_per_cell);
  Vector<double>  cell_rhs(dofs_per_cell);

  std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);
```

现在开始枚举三角单元，在每一个三角单元内，首先要清空局部刚度矩阵与局部右端项，然后通过 fe_values 来获取这个三角单元中的信息，并将获取到的积分节点位置信息提取出来。

```
  for (const auto &cell : dof_handler.active_cell_iterators())
    {
      cell_matrix = 0;
      cell_rhs  = 0;
      fe_values.reinit(cell);
      const std::vector<Point<2>> &quadrature_points
                              = fe_values.get_quadrature_points();
```

接下来，枚举积分节点以计算三角形上的积分。首先是计算局部刚度矩阵：

```
    for (const unsigned int q_index : fe_values.quadrature_point_indices())
    {
      for (const unsigned int i : fe_values.dof_indices())
        for (const unsigned int j : fe_values.dof_indices())
          cell_matrix(i, j) +=
            (fe_values.shape_grad(i, q_index) * // grad phi_i(x_q)
            fe_values.shape_grad(j, q_index) + // grad phi_j(x_q)
            fe_values.shape_value(i, q_index) * // phi_i(x_q)
            fe_values.shape_value(j, q_index) // phi_j(x_q)
            ) * fe_values.JxW(q_index);      // 积分节点的权重
```

然后是计算局部右端项：

```
      for (const unsigned int i : fe_values.dof_indices())
      cell_rhs(i) += (fe_values.shape_value(i, q_index) * // phi_i(x_q)
                      f(quadrature_points[q_index]) * // f(x_q)
                      fe_values.JxW(q_index));       // 积分节点的权重
    }
```

现在，让我们来获取索引数组，并将局部刚度矩阵与局部右端项“贡献”给全局：

```
    cell->get_dof_indices(local_dof_indices);

    for (const unsigned int i : fe_values.dof_indices())
      for (const unsigned int j : fe_values.dof_indices())
        system_matrix.add(local_dof_indices[i],
                          local_dof_indices[j],
                          cell_matrix(i, j));

    for (const unsigned int i : fe_values.dof_indices())
      system_rhs(local_dof_indices[i]) += cell_rhs(i);
  }
}
```

2.2.5 求解稀疏线性系统

装配好刚度矩阵之后，我们就可以求解稀疏线性系统了。我们当然是使用迭代法，所以需要一个控制器来确定何时停止迭代，它需要控制最大迭代次数、终止迭代的误差。

```
void Elliptic::solve()
{
  SolverControl solver_control(5000, 1e-6 * system_rhs.l2_norm());
```

我们选择共轭梯度法来求解，如下。PreconditionIdentity() 表示无预优器，以后我们将会使用多重网格作为预优器，这一章暂时不管。

```
  SolverCG<Vector<double>> solver(solver_control);
  solver.solve(system_matrix, solution, system_rhs, PreconditionIdentity());
}
```

2.2.6 输出结果与计算误差

最后是输出结果，没什么好说的：

```
void Elliptic::output_results() const
{
  DataOut<2> data_out;
  data_out.attach_dof_handler(dof_handler);
  data_out.add_data_vector(solution, "solution");
  data_out.build_patches();
  std::ofstream output("solution.vtk");
  data_out.write_vtk(output);
```

然后我们计算解的 L^2 范数误差。在数学上，计算 L^2 范数需要计算一个积分，这个积分需要通过自适应数值积分方法计算，deal.II 为我们提供了计算误差范数的函数，首先计算每个网格单元上的 L^2 误差，然后累积得到全局 L^2 误差，我们只需要提供必要的信息即可，见代码注释。

```
  Vector<double> difference_per_cell(triangulation.n_active_cells());
  VectorTools::integrate_difference(MappingFE<2>(fe), // 自由度到顶点坐标的映射
                                    dof_handler,     // 自由度处理器
                                    solution,        // 数值解向量
                                    Solution(),      // 精确解
                                    difference_per_cell, // 每个单元上的误差结果
                                    QGaussSimplex<2>(fe.degree + 1), // 数值积分公式
                                    VectorTools::L2_norm); // 误差范数
  const double L2_error =
    VectorTools::compute_global_error(triangulation,
                                      difference_per_cell,
                                      VectorTools::L2_norm);
  std::cout << "L2-norm error: " << L2_error << std::endl;
}
```

注意，这样输出 L^2 误差会很精确，但也很慢，网格大的时候几乎是求解耗时的 5 倍，因此不建议使用。其实我们只需要粗略计算 L^2 误差，我们可以用枚举三角单元、枚举积分节点的方式自己写一个函数，这一章我们暂时不写。

2.2.7 主函数

主函数随便写写就好了。

```
int main(int argc, char* argv[])
{
  int level;
  std::cin >> level;
  Elliptic solver(level);
  solver.run();
  return 0;
}
```

完整代码见 `elliptic` 文件夹。

2.3 编程指北 2：代数多重网格

这个小节我们来引入多重网格。deal.II 支持的几何多重网格是基于四边形网格上的 $\mathcal{Q}_k$ 元的，在实际应用中，$\mathcal{Q}_k$ 元很常用也很好用，但在我们这门课程里并不使用。想学习相关内容，请移步 deal.II 官方教程的 step-16。我们介绍一个代数多重网格。首先需要引入头文件

```
#include <deal.II/lac/generic_linear_algebra.h>
```

除了 `solve` 函数，其余部分与之前一样，我们来重写 `solve`。首先需要引入代数多重网格，以及包含多重网格各项的参数结构体：

```
void Elliptic::solve()
{
  TrilinosWrappers::PreconditionAMG preconditioner;
  TrilinosWrappers::PreconditionAMG::AdditionalData additional_data;
```

我们来看多重网格包含了哪些参数。以下是 deal.II 官方文档中给出的定义：

```
TrilinosWrappers::PreconditionAMG::AdditionalData::AdditionalData (  const bool                                  elliptic = true,
                                                                     const bool                                  higher_order_elements = false,
                                                                     const unsigned int                          n_cycles = 1,
                                                                     const bool                                  w_cycle = false,
                                                                     const double                                aggregation_threshold = 1e-4,
                                                                     const std::vector< std::vector< bool > > &  constant_modes = std::vector<std::vector<bool>>(0),
                                                                     const unsigned int                          smoother_sweeps = 2,
                                                                     const unsigned int                          smoother_overlap = 0,
                                                                     const bool                                  output_details = false,
                                                                     const char *                                smoother_type = "Chebyshev",
                                                                     const char *                                coarse_type = "Amesos-KLU"
                                                                   )
```

图 2.2: PreconditionAMG 的各项参数

对于新手而言，我们只建议调整 `higher_order_elements`、`w_cycle`、`smoother_sweeps` 这三个参数。当你使用 $\mathcal{P}_k$ $(k \geq 2)$ 元时，可以设置 `higher_order_elements` 为 `true`；默认的多重网格循环是 V-cycle，可以设置 `w_cycle` 为 `true` 以切换为 W-cycle，通常会比 V-cycle 的效果要好；`smoother_sweeps` 是磨光次数，默认的 2 次通常是最好的，某些情况下 1 可能会好一些。

下面我们来初始化 `preconditioner`：

```
  additional_data.w_cycle = true;
  preconditioner.initialize(system_matrix, additional_data);
```

剩下的部分和以前一样，只要把预优器换成我们的 `preconditioner` 即可。最后可以输出 CG 迭代次数来看看多重网格的效果如何。

```
  SolverControl        solver_control(5000, 1e-6 * system_rhs.l2_norm());
  SolverCG<Vector<double>> solver(solver_control);
  solver.solve(system_matrix, solution, system_rhs, preconditioner);

  std::cout << solver_control.last_step() << " CG steps." << std::endl;
}
```

注意，由于我们使用了 `Trilinos` 的包，它是依赖 `MPI` 的，所以我们要在 `main()` 函数的第一行加上：

```
  Utilities::MPI::MPI_InitFinalize mpi_initialization(argc, argv, 1);
```

完整代码见 `ellipticAMG` 文件夹。

第 3 章 Stokes 方程

3.1 变分形式推导

考虑如下形式的 Stokes 方程：

$$\begin{cases} -\Delta \mathbf{u} + \nabla p = \mathbf{f}, & \text{in } \Omega, \\ \nabla \cdot \mathbf{u} = 0, & \text{in } \Omega, \\ \mathbf{u} = 0, & \text{on } \partial\Omega. \end{cases} \tag{3.1}$$

内积上测试函数 $(\mathbf{v}, q)^T$，并将齐次边界条件代入，得到：

$$\begin{aligned} &\left(\begin{pmatrix} -\Delta \mathbf{u} + \nabla p \\ \nabla \cdot \mathbf{u} \end{pmatrix}, \begin{pmatrix} \mathbf{v} \\ q \end{pmatrix} \right)_\Omega \\ =&(-\Delta \mathbf{u} + \nabla p, \mathbf{v})_\Omega + (\nabla \cdot \mathbf{u}, q)_\Omega \\ =&(\nabla \mathbf{u}, \nabla \mathbf{v})_\Omega + (p, \nabla \cdot \mathbf{v})_\Omega + (\nabla \cdot \mathbf{u}, q)_\Omega. \end{aligned}$$

所以有变分形式：

$$(\nabla \mathbf{u}, \nabla \mathbf{v})_\Omega + (p, \nabla \cdot \mathbf{v})_\Omega + (\nabla \cdot \mathbf{u}, q)_\Omega = (\mathbf{f}, \mathbf{v}), \quad \forall \mathbf{v} \in H_0^1(\Omega), q \in L_0^2(\Omega). \tag{3.2}$$

我们记有限元空间 $W_h = V_h \times Q_h$，其中 $V_h = [\mathcal{P}_2]^d$（d 是空间维数），$Q_h = \mathcal{P}_1$，记 V_h 的基函数为 $\phi_1, ..., \phi_N$（向量函数），Q_h 的基函数为 $\psi_1, ..., \psi_M$，那么 W_h 的基函数为：

$$\Phi_i = \begin{pmatrix} \phi_i, \\ 0 \end{pmatrix}, \quad i = 1, ..., N; \tag{3.3}$$

$$\Phi_{N+i} = \begin{pmatrix} \mathbf{0}, \\ \psi_i \end{pmatrix}, \quad i = 1, ..., M. \tag{3.4}$$

我们记有限元变分解 $(\mathbf{u}_h, p_h)^T = \sum_i \alpha_i \Phi_i$，并记 $\Phi_{i,\mathbf{u}}$ 为 Φ_i 在 V_h 上的分量，$\Phi_{i,p}$ 为 Φ_i 在 Q_h 上的分量，代入变分形式，可以得到如下形式的线性系统：

$$\mathcal{A}\alpha = \mathbf{b}. \tag{3.5}$$

其中

$$\mathcal{A}(i,j) = (\nabla \Phi_{i,\mathbf{u}}, \nabla \Phi_{j,\mathbf{u}}) + (\nabla \cdot \Phi_{i,\mathbf{u}}, \Phi_{j,p}) + (\Phi_{i,p}, \nabla \cdot \Phi_{j,\mathbf{u}}). \tag{3.6}$$

事实上，我们将上面的式子对于 $i = 1, ..., N$ 和 $i = N+1, ..., M$ 分别写出来，并扔掉等于零的项，可以得到下面这些方程：

$$\sum_{j=1}^{N} (\nabla \Phi_{i,\mathbf{u}}, \nabla \Phi_{j,\mathbf{u}}) \alpha_j + \sum_{j=N+1}^{M} (\nabla \cdot \Phi_{i,\mathbf{u}}, \Phi_{j,p}) \alpha_j = (\mathbf{f}, \Phi_{i,\mathbf{u}}), \quad i = 1, ..., N; \tag{3.7}$$

$$\sum_{j=1}^{N} (\nabla \cdot \Phi_{j,\mathbf{u}}, \Phi_{i,p}) \alpha_j = 0, \quad i = N+1, ..., M; \tag{3.8}$$

所以 (3.5) 是一个对称的鞍点型线性系统，我们可以把它写成下面的分块形式

$$\begin{pmatrix} A & B^T \\ B & O \end{pmatrix} \begin{pmatrix} \alpha_{\mathbf{u}} \\ \alpha_p \end{pmatrix} = \begin{pmatrix} \mathbf{f} \\ \mathbf{0} \end{pmatrix} \tag{3.9}$$

其中

$$A(i,j)=(\nabla\phi_i,\nabla\phi_j),\quad B(i,j)=(\psi_i,\nabla\cdot\phi_j). \tag{3.10}$$

在这一章中，我们均取精确解

$$\mathbf{u}(\mathbf{x})=\pi(\sin^2(\pi x_1)\sin(2\pi x_2),-\sin(2\pi x_1)\sin^2(\pi x_2)), \tag{3.11}$$

$$p(\mathbf{x})=\cos(\pi x_1)\sin(\pi x_2), \tag{3.12}$$

并导出右端项（建议自己用 WolframAlpha 算一下）：

$$f_1(\mathbf{x})=\pi\sin(\pi x_1)\sin(\pi x_2)-2\pi^3(-1+2\cos(2\pi x_1))\sin(2\pi x_2), \tag{3.13}$$

$$f_2(\mathbf{x})=-\pi\cos(\pi x_1)\cos(\pi x_2)+2\pi^3(-1+2\cos(2\pi x_2))\sin(2\pi x_1). \tag{3.14}$$

接下来的两篇编程指北可以独立阅读，从实用性角度出发建议阅读 Uzawa 方法，从完成作业的角度出发建议阅读预优 MinRes 方法。

3.2 编程指北 1：Uzawa 方法

3.2.1 Uzawa 方法简述

Uzawa 方法并不在这门课的编程要求内，但是它实现起来最简单，而且据我测试，它比预优 MinRes 方法、投影法、Schur 分解直接求解法等一众方法都要更快。因此，在这份编程指北中，我们首先介绍这种方法，但不作理论分析。

Uzawa 方法对速度与压强进行轮流更新, 即, 重复如下两步迭代:

(Uzawa-1) $\alpha_{\mathbf{u}}\leftarrow A^{-1}(\mathbf{f}-B^T\alpha_p)$;

(Uzawa-2) $\alpha_p\leftarrow\alpha_p+M_Q^{-1}B\alpha_{\mathbf{u}}$.

上面涉及到两个逆矩阵的计算，代价略大。从实用角度出发，我们更愿意使用非精确的 Uzawa 方法。即: 将上述迭代中的 A^{-1} 用一轮多重网格 W-Cycle 松弛替代（记做 C_W），将 M_Q^{-1} 用 $(\text{diag}\ M_Q)^{-1}$ 替代：

(Inex-Uzawa-1) $\alpha_{\mathbf{u}}\leftarrow\alpha_{\mathbf{u}}+C_W(\mathbf{f}-B^T\alpha_p-A\alpha_{\mathbf{u}})$;

(Inex-Uzawa-2) $\alpha_p\leftarrow\alpha_p+(\text{diag}\ M_Q)^{-1}B\alpha_{\mathbf{u}}$.

可以证明，精确与非精确的 Uzawa 方法具有一样好的收敛性。在实际使用中，非精确 Uzawa 的表现通常要好得多。现在我们开始介绍非精确 Uzawa 方法的编程。

3.2.2 框架

首先引入必要的头文件

```
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/logstream.h>
#include <deal.II/base/function.h>
#include <deal.II/base/utilities.h>
#include <deal.II/base/tensor_function.h>

#include <deal.II/lac/block_vector.h>
#include <deal.II/lac/full_matrix.h>
#include <deal.II/lac/block_sparse_matrix.h>
#include <deal.II/lac/solver_cg.h>
#include <deal.II/lac/precondition.h>
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/lac/generic_linear_algebra.h>
```

```
#include <deal.II/lac/linear_operator.h>
#include <deal.II/lac/packaged_operation.h>

#include <deal.II/grid/tria.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/grid_tools.h>
#include <deal.II/grid/grid_refinement.h>

#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_renumbering.h>
#include <deal.II/dofs/dof_tools.h>

#include <deal.II/fe/fe_simplex_p.h>
#include <deal.II/fe/fe_system.h>
#include <deal.II/fe/fe_values.h>

#include <deal.II/numerics/vector_tools.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/error_estimator.h>

using namespace dealii;

#include <iostream>
#include <fstream>
#include <memory>
```

为了方便，我们引入下面的计时器。注意，传统的 `clock()` 函数无法用于计时，因为 deal.II 运用了大量的并行技术。

```
#include <chrono>

struct CPUTimer
{
  using HRC = std::chrono::high_resolution_clock;
  std::chrono::time_point<HRC> start;
  CPUTimer() { reset(); }
  void reset() { start = HRC::now(); }
  double operator() () const {
    std::chrono::duration<double> e = HRC::now() - start;
    return e.count();
  }
};
```

首先来定义右端项函数与精确解，注意右端项函数是一个向量函数，所以我们需要继承 `TensorFunction`。精确解我们以分量形式返回，前两个分量是速度，第三个分量是压强，见代码。

```
template <int dim>
class RightHandSide : public TensorFunction<1, dim>
{
public:
```

```
  RightHandSide()
    : TensorFunction<1, dim>()
  {}
  virtual Tensor<1, dim> value(const Point<dim> &p) const override;
};


template <int dim>
Tensor<1, dim> RightHandSide<dim>::value(const Point<dim> &p) const
{
  Tensor<1, dim> ans;
  ans[0] = M_PI * sin(M_PI*p[0]) * sin(M_PI*p[1])
           - 2*M_PI*M_PI*M_PI * (-1. + 2.*cos(2*M_PI*p[0])) * sin(2*M_PI*p[1]);
  ans[1] = -M_PI*cos(M_PI*p[0])*cos(M_PI*p[1])
           + 2*M_PI*M_PI*M_PI * (-1. + 2.*cos(2*M_PI*p[1])) * sin(2*M_PI*p[0]);
  return ans;
}


template <int dim>
class TrueSolution : public Function<dim>
{
public:
  TrueSolution()
    : Function<dim>()
  {}
  virtual double value(const Point<dim> &p,
                       const unsigned int component = 0) const override;
};


template <int dim>
double TrueSolution<dim>::value(const Point<dim> &p,
                                const unsigned int component) const
{
  if(component == 0)
    return M_PI * pow(sin(M_PI*p[0]), 2.) * sin(2.*M_PI*p[1]);
  else if(component == 1)
    return -M_PI * pow(sin(M_PI*p[1]), 2.) * sin(2.*M_PI*p[0]);
  else if(component == 2)
    return -cos(M_PI*p[0]) * sin(M_PI*p[1]);
  return 0;
}
```

现在来定义主类。这里我们用了一个模板参数 dim，它可以让我们的代码在二维和三维中通用。主要的过程和椭圆方程一样，只是多了一个用于计算 L^2 误差的函数。另外，注意到我们把稀疏矩阵和向量都改成了分块的版本，这是因为我们的线性系统是分块的。我们还使用到了 FESystem，它可以将一些有限元系统“打包”。最后增加的 constraints 对象是 deal.II 中处理 Dirichlet 型边界条件的标准模块。

```
template<int dim>
```

```
class StokesProblem
{
public:
  StokesProblem(const int &);
  void run();

private:
  void make_grid();
  void setup_system();
  void assemble_system();
  void solve();
  void compute_L2_error();
  void output_results(const unsigned &);

  unsigned int level;

  Triangulation<dim> triangulation;
  FESystem<dim>   fe;
  DoFHandler<dim> dof_handler;

  BlockSparsityPattern  sparsity_pattern;
  BlockSparseMatrix<double> system_matrix;

  BlockVector<double> system_rhs;
  BlockVector<double> solution;
  BlockVector<double> checker;
  BlockVector<double> block_diag_MQ;

  AffineConstraints<double> constraints;
};
```

现在，在构造函数中，我们要声明有限元系统 $[\mathcal{P}_2]^d \times \mathcal{P}_1$，这里用 `^dim` 可以让我们的代码在二维和三维中通用（后文中，未经特殊说明，均默认二维情况，此时我们一共有三个有限元系统）。其余部分和椭圆方程一样，如下。

```
template<int dim>
StokesProblem<dim>::StokesProblem
  (const int &level):
   level(level),
   fe(FE_SimplexP<dim>(2) ^ dim, FE_SimplexP<dim>(1)),
   dof_handler(triangulation)
{}
```

接下来我们还是先把一键运行的函数写了，这次我们加上计时器，以便观察每个步骤分别的耗时，并在最后调用我们自己的函数来计算误差。

```
template<int dim>
void StokesProblem<dim>::run()
{
  make_grid();
  CPUTimer timer;
```

```
  std::cout << "Level: " << level << "---------------------------------" << std::endl;
  std::cout << "Setup..." << std::endl;
  setup_system();

  timer.reset();
  std::cout << "Assembling..." << std::endl;
  assemble_system();
  std::cout << "  " << timer() << "s." << std::endl;

  timer.reset();
  std::cout << "Solving..." << std::endl;
  solve();
  std::cout << "  " << timer() << "s." << std::endl;

  std::cout << "Output results..." << std::endl;
  output_results();

  std::cout << "Computing error..." << std::endl;
  compute_L2_error();
}
```

接下来我们分别实现每一个步骤。第一个步骤 `make_grid` 和椭圆方程是完全一样的，这里不再展示。我们直接来看 `setup_system`。

3.2.3 初始化稀疏线性系统

同样的，首先需要将有限元系统绑定到自由度控制器。接下来我们需要对自由度进行排序，我们需要确保 $\Phi_1, ..., \Phi_N$ 对应的自由度确实是前 N 个自由度，这样最后得到的线性系统才能按我们所预期的那样呈现分块结构。我们需要一个数组来说明第 i 个有限元系统属于第几个分量，我们一共有三个有限元系统，前两个系统属于第 0 个分量（速度分量）、后一个系统属于第 1 个分量（压强分量），因此我们定义一个这样的数组，并将它传入 `component_wise` 函数，这样自由度就能按我们的预期排序。

```
template<int dim>
void StokesProblem<dim>::setup_system()
{
  dof_handler.distribute_dofs(fe);

  std::vector<unsigned> block_component(dim+1, 0);
  block_component[dim] = 1;
  DoFRenumbering::component_wise(dof_handler, block_component);
```

接下来引入处理 Dirichlet 边界条件的标准模块，我们在这个模块中定义速度分量的齐次 Dirichlet 边界条件，这里的 `component_mask(velocities)` 表示该边界条件仅针对速度分量。而 `velocities` 是我们定义的从第 0 个有限元系统开始的一个向量类型系统，即速度分量对应的有限元系统，我们不用关心内部实现。最后需要调用 `close()` 以完成 `constraints` 模块的初始化。当 Dirichlet 边界条件不是齐次时，需要自己写一个边界条件的函数，格式和右端项函数一样。

```
  const FEValuesExtractors::Vector velocities(0);
```

```
constraints.clear();
VectorTools::interpolate_boundary_values(dof_handler,
                                         0,
                                         Functions::ZeroFunction<dim>(),
                                         constraints,
                                         fe.component_mask(velocities));
constraints.close();
```

接下来我们可以输出一下速度分量、压强分量分别有几个自由度

```
auto dofs_per_block = DoFTools::count_dofs_per_fe_block(dof_handler,
                                                        block_component);
const unsigned n_u = dofs_per_block[0];
const unsigned n_p = dofs_per_block[1];

std::cout << "  Number of active cells: " << triangulation.n_active_cells()
          << std::endl
          << "  Total number of cells: " << triangulation.n_cells()
          << std::endl
          << "  Number of DoFs: " << dof_handler.n_dofs()
          << " (" << n_u << '+' << n_p << ')' << std::endl;
```

接下来我们考虑稀疏模式应该如何初始化。默认的初始化方式是这样的：如果自由度 i 与自由度 j 在同一个三角单元中，那么就将 (i,j) 加入稀疏模式。但是，现在事情变了，比如说，如果自由度 i 与自由度 j 分别属于第 0 个有限元系统和第 1 个有限元系统，那么 $(\nabla\Phi_{i,\mathbf{u}},\nabla\Phi_{j,\mathbf{u}})=0$，这样一来 (i,j) 就没必要加入稀疏模式了。事实上，如果我们不把这些没必要的位置加入稀疏模式，我们得到的稀疏模式会比默认的小一半以上！

不难发现，在 Stokes 方程中，对于两个在同一个三角形单元中的自由度 i,j，位置 (i,j) 需要被加入稀疏模式，当且仅当符合下面三个条件之一：

- 两个自由度均属于第 0 个有限元系统；
- 两个自由度均属于第 1 个有限元系统；
- 其中一个自由度属于前两个有限元系统、另一个自由度属于第 2 个有限元系统。

我们需要创建一个表格来表示哪些位置是有必要加入稀疏模式的，然后将表格与 `constraints` 模块传入创建动态稀疏模式的函数中，详见代码。代码中用了一个异或布尔运算，是对上述条件的一个精巧表示，读者可以思考一下。

```
BlockDynamicSparsityPattern dsp(dofs_per_block, dofs_per_block);

Table<2, DoFTools::Coupling> coupling(dim+1, dim+1);
for (unsigned int c = 0; c < dim + 1; ++c)
  for (unsigned int d = 0; d < dim + 1; ++d)
    if ((c==d) ^ (c==dim || d==dim))
      coupling[c][d] = DoFTools::always;
    else
      coupling[c][d] = DoFTools::none;

DoFTools::make_sparsity_pattern(
    dof_handler, coupling, dsp, constraints, false);
sparsity_pattern.copy_from(dsp);
system_matrix.reinit(sparsity_pattern);
```

最后是创建分块的向量。

```
  solution.reinit(dofs_per_block);
  system_rhs.reinit(dofs_per_block);
  checker.reinit(dofs_per_block);
  block_diag_MQ.reinit(dofs_per_block);
}
```

3.2.4 装配刚度矩阵

我们的框架还是：枚举三角网格、计算局部刚度矩阵与局部右端项、贡献给全局。另外，由于我们要计算 $\mathrm{diag}(M_Q)$，所以我们还需要一个局部 $\mathrm{diag}(M_Q)$，我们用向量来存储对角矩阵。下面是我们需要的一些局部变量：

```
template<int dim>
void StokesProblem<dim>::assemble_system()
{
  QGaussSimplex<dim> quadrature_formula(2);
  FEValues<dim> fe_values(fe,
                          quadrature_formula,
                          update_values | update_quadrature_points |
                            update_JxW_values | update_gradients);

  const unsigned int dofs_per_cell = fe.n_dofs_per_cell();
  const unsigned int n_q_points = quadrature_formula.size();

  FullMatrix<double> local_matrix(dofs_per_cell, dofs_per_cell);
  Vector<double>  local_rhs(dofs_per_cell);
  Vector<double>  local_diag_MQ(dofs_per_cell);

  std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);
```

我们需要计算 $\mathbf{f}=(f_1,f_2)$ 在每个积分节点的值，并且存起来，deal.II 将向量函数的返回值定义为一个一阶张量 `Tensor<1,dim>`，所以我们需要一个长度为 `n_q_points` 的一阶张量型数组。

```
  const RightHandSide<dim> right_hand_side;
  std::vector<Tensor<1, dim>> rhs_values(n_q_points, Tensor<1, dim>());
```

接下来是两个神奇的模块，可以帮助我们提取 $\Phi_{i,\mathbf{u}}$ 与 $\Phi_{i,p}$。有了它们，我们就可以用 `fe_values[velocities].shape_value(i,q)` 表示 $\Phi_{i,\mathbf{u}}(x_q)$，以及 `fe_values[pressure].shape_value(i,q)` 表示 $\Phi_{i,p}(x_q)$。当然，$\nabla\Phi_{i,\mathbf{u}}(x_q)$ 和 $\nabla\cdot\Phi_{i,\mathbf{u}}(x_q)$ 也可以用类似的方法表示。

```
  const FEValuesExtractors::Vector velocities(0);
  const FEValuesExtractors::Scalar pressure(dim);
```

接下来是一些局部变量，用于存储 $\nabla\Phi_{i,\mathbf{u}}(x_q),\nabla\cdot\Phi_{i,\mathbf{u}}(x_q),\Phi_{i,\mathbf{u}}(x_q),\Phi_{i,p}(x_q)$（$i$ 取遍三角单元中的所有自由度）

```
  std::vector<Tensor<2, dim>>     grad_phi_u(dofs_per_cell);
  std::vector<double>             div_phi_u(dofs_per_cell);
  std::vector<Tensor<1, dim>>     phi_u(dofs_per_cell);
  std::vector<double>             phi_p(dofs_per_cell);
```

接下来开始枚举三角单元，初始化 fe_values，并计算好右端项函数在这个单元的所有积分节点上的值。

```
for (const auto &cell : dof_handler.active_cell_iterators())
  {
    fe_values.reinit(cell);
    local_matrix  = 0;
    local_diag_MQ = 0;
    local_rhs     = 0;

    right_hand_side.value_list(fe_values.get_quadrature_points(),
                               rhs_values);
```

现在枚举积分节点 x_q，并对所有的自由度 i，将 $\nabla\Phi_{i,\mathbf{u}}(x_q), \nabla\cdot\Phi_{i,\mathbf{u}}(x_q), \Phi_{i,\mathbf{u}}(x_q), \Phi_{i,p}(x_q)$ 计算好。

```
    for (unsigned int q = 0; q < n_q_points; ++q)
      {
        for (unsigned int k = 0; k < dofs_per_cell; ++k)
          {
            grad_phi_u[k] = fe_values[velocities].gradient(k, q);
            div_phi_u[k] = fe_values[velocities].divergence(k, q);
            phi_u[k]     = fe_values[velocities].value(k, q);
            phi_p[k]     = fe_values[pressure].value(k, q);
          }
```

剩下的部分就是按照 (3.6) 式来计算局部的 $\mathcal{A}_{\mathcal{K}}(i,j)$，局部右端项和局部 $\mathrm{diag}(M_Q)$ 的计算无需多言。注意，由于 $\mathcal{A}_{\mathcal{K}}$ 是对称的，我们可以只计算它的下三角部分。

```
        for (unsigned int i = 0; i < dofs_per_cell; ++i)
        {
          for (unsigned int j = 0; j <= i; ++j)
            {
              local_matrix(i, j) +=
                (   grad_phi_u[i][0] * grad_phi_u[j][0] // (1)
                  + grad_phi_u[i][1] * grad_phi_u[j][1]
                  - div_phi_u[i] * phi_p[j]           // (2)
                  - phi_p[i] * div_phi_u[j])          // (3)
                * fe_values.JxW(q);                   // * dx
            }
          local_rhs(i) += phi_u[i]       // phi_u_i(x_q)
                        * rhs_values[q]  // * f(x_q)
                        * fe_values.JxW(q); // * dx
          local_diag_MQ(i) +=
               (phi_p[i] * phi_p[i])
               * fe_values.JxW(q);
        }
      }
```

现在，我们根据对称性来填充 $\mathcal{A}_{\mathcal{K}}$ 的上三角部分

```
    for (unsigned int i = 0; i < dofs_per_cell; ++i)
      for (unsigned int j = i + 1; j < dofs_per_cell; ++j)
        local_matrix(i, j) = local_matrix(j, i);
```

最后，将局部矩阵与向量贡献给全局。这里我们不再手动枚举、贡献，而是直接使用 constraints 模块的内置函数。（注意，当存在 Dirichlet 边界条件时，必须使用这种方式来将局部信息贡献给全局）

```
      cell->get_dof_indices(local_dof_indices);
      constraints.distribute_local_to_global(local_matrix,
                                             local_rhs,
                                             local_dof_indices,
                                             system_matrix,
                                             system_rhs);
      constraints.distribute_local_to_global(local_diag_MQ,
                                             local_dof_indices,
                                             block_diag_MQ);
    }
}
```

3.2.5 求解稀疏线性系统

现在到了激动人心的部分，我们要开始求解线性系统了。首先是初始化多重网格，为了代码简单，我们仍然使用 Trilinos 的代数多重网格。

```
template<int dim>
void StokesProblem<dim>::solve()
{
  TrilinosWrappers::PreconditionAMG::AdditionalData additional_data;
  additional_data.higher_order_elements = true;
  additional_data.w_cycle = true;

  TrilinosWrappers::PreconditionAMG A_preconditioner;
  A_preconditioner.initialize(system_matrix.block(0,0), additional_data);
```

现在，$\mathrm{diag}(M_Q)$ 被存到了分块向量 block_diag_MQ 的压强分量中，我们需要把它提取出来作为一个普通向量，然后将每个元素求倒数得到 $\mathrm{diag}(M_Q)^{-1}$。

```
  auto diag_MQ_inverse = block_diag_MQ.block(1);
  for(auto& p : diag_MQ_inverse) p = 1. / p;
```

接下来是 Uzawa 迭代过程中需要用到的一些临时量

```
  Vector<double> rhs(solution.block(0).size());
  Vector<double> correction_u(solution.block(0).size());
  Vector<double> correction_p(solution.block(1).size());
  int uzawa_steps = 0;
```

接下来我们引入一些“语法糖”，我们把分块矩阵、分块向量的每一个块起一个名字，和我们数学上的表述统一起来，接下来我们就可以像写数学表达式一样写代码了。注意，线性算子只是一个包装，稀疏矩阵和向量之间的乘法运算符是没有被定义的，需要用成员函数 vmult 来实现，但当我们将稀疏矩阵包装成线性算子后，我们就可以用乘法运算符了。

```
  const auto B = linear_operator(system_matrix.block(1,0));
  const auto BT = linear_operator(system_matrix.block(0,1));
  const auto A = linear_operator(system_matrix.block(0,0));
  const auto & f = system_rhs.block(0);
```

```
  auto & u     = solution.block(0);
  auto & p     = solution.block(1);
```

现在正式开始 Uzawa 迭代。第一步是计算 $rhs = \mathbf{f} - B^T\alpha_p - A\alpha_{\mathbf{u}}$，然后对它进行一次多重网格作用得到 $\alpha_{\mathbf{u}}$ 的校正，将它加给 $\alpha_{\mathbf{u}}$。然后需要调用一次 `distribute` 来将 Dirichlet 边界条件赋值给速度分量。

```
  do{
    uzawa_steps++;
    rhs = f - BT * p - A * u;
    A_preconditioner.vmult(correction_u, rhs);
    u += correction_u;
    constraints.distribute(solution);
```

接下来是按 (Inex-Uzawa-2) 来校正 α_p。

```
    correction_p = B * u;
    for(unsigned i = 0; i < p.size(); i++)
      correction_p[i] *= diag_MQ_inverse[i];
    p += correction_p;
```

最后，我们需要计算整个系统的残差，当残差的二范数达到容许误差时，结束迭代。

```
    system_matrix.vmult(checker, solution);
    checker -= system_rhs;
  }while(checker.l2_norm() >= 1e-7*system_rhs.l2_norm());
```

我们可以观察 Uzawa 迭代一共进行了几轮（通常不超过 100 轮，且轮数不随网格加密而增加）。

```
  std::cout << "  " << uzawa_steps << " Uzawa iterations." << std::endl;
}
```

3.2.6 输出结果

我们要把解向量拆分成速度向量分量、压强标量分量来分别输出，所以需要一些声明，即：前两个有限元系统是速度向量分量的一部分、后一个有限元系统是压强标量分量。具体见代码。

```
template <int dim>
void StokesProblem<dim>::output_results()
{
  std::vector<std::string> solution_names(dim, "u");
  solution_names.emplace_back("p");
  std::vector<DataComponentInterpretation::DataComponentInterpretation>
    interpretation(dim,
                   DataComponentInterpretation::component_is_part_of_vector);
  interpretation.push_back(DataComponentInterpretation::component_is_scalar);
```

然后传进 `add_data_vector` 里输出即可。我们传给 `build_patches` 里的参数表示在输出时，对每个三角形单元输出几个点的值；当不传值时，只输出三角形三个顶点的值；当传入 2 时，会额外输出三角形三边中点的值，可以让我们画出的图像更光滑一些。

```
  DataOut<dim> data_out;
  data_out.add_data_vector(dof_handler,
                           solution,
```

```
                        solution_names,
                        interpretation);

  data_out.build_patches(2);

  std::ofstream output(
    "solution" + Utilities::int_to_string(level, 2) + ".vtk");
  data_out.write_vtk(output);
}
```

3.2.7 计算误差

我们这里只计算 L^2 误差，计算公式是：

$$E_{u_1}^2 = \int_\Omega |u_1 - u_{h,1}|^2 \, dx \approx \sum_{\mathcal{K}\in\mathcal{T}} \sum_q |u_1(x_q) - u_{h,1}(x_q)|^2 w_q. \tag{3.15}$$

其中 x_q 是积分节点的坐标，w_q 是积分节点的权重，q 取遍 $\mathcal{K}$ 上的所有积分节点。E_{u_2} 和 E_p 类似。

为了使计算结果能够体现出收敛阶，我们需要一个至少 3 阶代数精度的积分公式，这里我们用了 3 点高斯积分公式，它具有 5 阶代数精度。

代码的思路和装配矩阵一样，但省去了贡献给全局的步骤，如下。

```
template <int dim>
void StokesProblem<dim>::compute_L2_error()
{
  QGaussSimplex<dim> quadrature_formula(3);
  FEValues<dim> fe_values(fe,
                          quadrature_formula,
                          update_values | update_quadrature_points |
                            update_JxW_values);

  const unsigned int n_q_points = quadrature_formula.size();

  const TrueSolution<dim> true_solution;
  std::vector<Tensor<1, dim>> numerical_results_u(n_q_points);
  std::vector<double>     numerical_results_p(n_q_points);

  const FEValuesExtractors::Vector velocities(0);
  const FEValuesExtractors::Scalar pressure(dim);

  double err_u1 = 0.;
  double err_u2 = 0.;
  double err_p = 0.;

  for (const auto &cell : dof_handler.active_cell_iterators())
    {
      fe_values.reinit(cell);
      fe_values[velocities].get_function_values(solution, numerical_results_u);
      fe_values[pressure].get_function_values(solution, numerical_results_p);
      auto q_points = fe_values.get_quadrature_points();
```

```
    for (unsigned int q = 0; q < n_q_points; ++q)
      {
        auto jxw = fe_values.JxW(q);
        err_u1 += pow(true_solution.value(q_points[q], 0) - numerical_results_u[q][0], 2.) * jxw;
        err_u2 += pow(true_solution.value(q_points[q], 1) - numerical_results_u[q][1], 2.) * jxw;
        err_p += pow(true_solution.value(q_points[q], 2) - numerical_results_p[q], 2.) * jxw;
      }
  }

 std::cout << "  u1 error: " << sqrt(err_u1) << std::endl;
 std::cout << "  u2 error: " << sqrt(err_u2) << std::endl;
 std::cout << "  p  error: " << sqrt(err_p) << std::endl << std::flush;
}
```

值得注意的是，我们这样算出的并不是 $||u_1 - u_{h,1}||_{L^2}$，只是一个近似值，但用于计算收敛阶已经足够。要算出误差的精确值，要将每个三角形不断切分成小三角形计算下去，直到三角形足够小或者结果足够可信。精确计算误差的 L^2 范数代价是很大的，甚至比求解的耗时更久，所以我们不推荐这么做。

3.2.8 主函数

主函数随便写写就好了。注意因为我们用了 Trilinos 的代数多重网格，所以要加上一句 MPI 的初始化。

```
int main(int argc, char *argv[]){
 Utilities::MPI::MPI_InitFinalize mpi_initialization(argc, argv, 1);
 int level;
 std::cin >> level;
 StokesProblem<2> stokes(level);
 stokes.run();
 return 0;
}
```

完整代码见文件夹 `stokesUzawa`。

3.3 编程指北 2：预优 MinRes 方法

3.3.1 预优 MinRes 方法介绍

预优 MinRes 方法是这门课的编程要求。而在这门课的作业中，线性系统 (3.9) 被进一步地分块为：

$$\begin{pmatrix} A_1 & O & B_1^T \\ O & A_2 & B_2^T \\ B_1 & B_2 & O \end{pmatrix} \begin{pmatrix} \alpha_{u_1} \\ \alpha_{u_2} \\ \alpha_p \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \mathbf{0} \end{pmatrix} \tag{3.16}$$

这也不难理解，我们只需要把 $\Phi_1, ..., \Phi_M$ 进一步排序为：

$$\begin{pmatrix} \xi_1 \\ 0 \\ 0 \end{pmatrix}, ..., \begin{pmatrix} \xi_s \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \xi_1 \\ 0 \end{pmatrix}, ..., \begin{pmatrix} 0 \\ \xi_s \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ \psi_1 \end{pmatrix}, ..., \begin{pmatrix} 0 \\ 0 \\ \psi_m \end{pmatrix} \tag{3.17}$$

即可，其中 $\xi_1, ..., \xi_s$ 是 $\mathcal{P}_2$ 的基函数，$\psi_1, ..., \psi_m$ 是 $\mathcal{P}_1$ 的基函数。

MinRes 方法是一种求解对称系统的方法，不要求正定性。deal.II 为我们提供了 `SolverMinRes`，它的接口和

椭圆方程中用到的 SolverCG 完全一样。我们要做的事情就是定义预优器。根据作业要求，预优器为：

$$\begin{pmatrix} A_\Delta & & \\ & A_\Delta & \\ & & \operatorname{diag}(M_Q)^{-1} \end{pmatrix}, \tag{3.18}$$

其中 A_Δ 表示针对 $\mathcal{P}_2$ 元上 Poisson 方程的多重网格 V-cycle 的一次作用。

3.3.2 框架

首先引入必要的头文件

```
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/logstream.h>
#include <deal.II/base/function.h>
#include <deal.II/base/utilities.h>
#include <deal.II/base/tensor_function.h>

#include <deal.II/lac/block_vector.h>
#include <deal.II/lac/full_matrix.h>
#include <deal.II/lac/block_sparse_matrix.h>
#include <deal.II/lac/solver_minres.h>
#include <deal.II/lac/precondition.h>
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/lac/generic_linear_algebra.h>

#include <deal.II/grid/tria.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/grid_tools.h>
#include <deal.II/grid/grid_refinement.h>

#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_renumbering.h>
#include <deal.II/dofs/dof_tools.h>

#include <deal.II/fe/fe_simplex_p.h>
#include <deal.II/fe/fe_system.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/fe/mapping_fe.h>

#include <deal.II/numerics/vector_tools.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/error_estimator.h>

using namespace dealii;

#include <iostream>
#include <fstream>
#include <memory>
```

为了方便，我们引入下面的计时器。注意，传统的 clock() 函数无法用于计时，因为 deal.II 运用了大量的并行技术。

```
#include <chrono>

struct CPUTimer
{
  using HRC = std::chrono::high_resolution_clock;
  std::chrono::time_point<HRC> start;
  CPUTimer() { reset(); }
  void reset() { start = HRC::now(); }
  double operator() () const {
    std::chrono::duration<double> e = HRC::now() - start;
    return e.count();
  }
};
```

首先来定义右端项函数与精确解，注意右端项函数是一个向量函数，所以我们需要继承 TensorFunction。精确解我们以分量形式返回，前两个分量是速度，第三个分量是压强，见代码。

```
template <int dim>
class RightHandSide : public TensorFunction<1, dim>
{
public:
  RightHandSide()
    : TensorFunction<1, dim>()
  {}
  virtual Tensor<1, dim> value(const Point<dim> &p) const override;
};


template <int dim>
Tensor<1, dim> RightHandSide<dim>::value(const Point<dim> &p) const
{
  Tensor<1, dim> ans;
  ans[0] = M_PI * sin(M_PI*p[0]) * sin(M_PI*p[1])
           - 2*M_PI*M_PI*M_PI * (-1. + 2.*cos(2*M_PI*p[0])) * sin(2*M_PI*p[1]);
  ans[1] = -M_PI*cos(M_PI*p[0])*cos(M_PI*p[1])
           + 2*M_PI*M_PI*M_PI * (-1. + 2.*cos(2*M_PI*p[1])) * sin(2*M_PI*p[0]);
  return ans;
}


template <int dim>
class TrueSolution : public Function<dim>
{
public:
  TrueSolution()
    : Function<dim>()
  {}
```

```
  virtual double value(const Point<dim> &p,
                       const unsigned int component = 0) const override;
};


template <int dim>
double TrueSolution<dim>::value(const Point<dim> &p,
                                const unsigned int component) const
{
  if(component == 0)
    return M_PI * pow(sin(M_PI*p[0]), 2.) * sin(2.*M_PI*p[1]);
  else if(component == 1)
    return -M_PI * pow(sin(M_PI*p[1]), 2.) * sin(2.*M_PI*p[0]);
  else if(component == 2)
    return -cos(M_PI*p[0]) * sin(M_PI*p[1]);
  return 0;
}
```

现在来定义主类。这里我们用了一个模板参数 dim，它可以让我们的代码在二维和三维中通用。主要的过程和椭圆方程一样，只是多了一个用于计算 L^2 误差的函数。另外，注意到我们同时有一个分块稀疏矩阵和稀疏矩阵，这两个矩阵的内容是完全一样的，只是存储方式不一样，在后文中我们将会知道为什么要用两种方式存储同一个矩阵。我们还使用到了 FESystem，它可以将一些有限元系统“打包”。最后增加的 constraints 对象是 deal.II 中处理 Dirichlet 型边界条件的标准模块。

```
template <int dim>
class StokesProblem
{
public:
  StokesProblem(const unsigned&);
  void run();

private:
  void setup_dofs();
  void assemble_system();
  void solve();
  void output_results() const;
  void refine_mesh();
  void compute_L2_error();

  int level;
  Triangulation<dim> triangulation;
  FESystem<dim>   fe;
  DoFHandler<dim> dof_handler;

  std::vector<types::global_dof_index> dofs_per_block;

  AffineConstraints<double> constraints;

  BlockVector<double> block_diag_MQ;
```

```
  Vector<double> solution;
  Vector<double> system_rhs;

  BlockSparsityPattern  sparsity_pattern;
  BlockSparseMatrix<double> system_matrix;

  SparsityPattern   full_sparsity_pattern;
  SparseMatrix<double> full_system_matrix;
};
```

现在，在构造函数中，我们要声明有限元系统 $[\mathcal{P}_2]^d \times \mathcal{P}_1$，这里用 `^dim` 可以让我们的代码在二维和三维中通用（后文中，未经特殊说明，均默认二维情况，此时我们一共有三个有限元系统）。其余部分和椭圆方程一样，如下。

```
template<int dim>
StokesProblem<dim>::StokesProblem
  (const int &level):
   level(level),
   fe(FE_SimplexP<dim>(2) ^ dim, FE_SimplexP<dim>(1)),
   dof_handler(triangulation)
{}
```

接下来我们还是先把一键运行的函数写了，这次我们加上计时器，以便观察每个步骤分别的耗时，并在最后调用我们自己的函数来计算误差。因为生成网格太简单了，我们这次就不单独写一个函数，而是直接写在 `run()` 里面。

```
template <int dim>
void StokesProblem<dim>::run()
{
  GridGenerator::subdivided_hyper_cube_with_simplices(triangulation, 1);
  triangulation.refine_global(level);
  CPUTimer timer;

  std::cout << "Level " << level;
  std::cout << " ------------------------------------------" << std::endl;
  setup_dofs();

  timer.reset();
  std::cout << "   Assembling...  " << std::flush;
  assemble_system();
  std::cout << timer() << "s" << std::endl;

  timer.reset();
  std::cout << "   Solving..." << std::endl << std::flush;
  solve();
  std::cout << "   Solved in " << timer() << "s" << std::endl;

  std::cout << "   Outputing...  " << std::endl << std::flush;
  output_results();
```

```
  std::cout << "   Computing L2 error...   " << std::endl << std::flush;
  compute_L2_error();
}
```

3.3.3 初始化稀疏线性系统

同样的，首先需要将有限元系统绑定到自由度控制器。接下来我们需要对自由度根据 (3.17) 式进行排序，这样最后得到的线性系统才能按我们所预期的那样呈现 (3.16) 式的分块结构。事实上，`component_wise` 函数默认的排序方式就符合我们的要求。

```
template <int dim>
void StokesProblem<dim>::setup_dofs()
{
  dof_handler.distribute_dofs(fe);
  DoFRenumbering::component_wise(dof_handler);
```

接下来引入处理 Dirichlet 边界条件的标准模块，我们在这个模块中定义速度分量的齐次 Dirichlet 边界条件，这里的 `component_mask(velocities)` 表示该边界条件仅针对速度分量。而 `velocities` 是我们定义的从第 0 个有限元系统开始的一个向量类型系统，即速度分量对应的有限元系统，我们不用关心内部实现。最后需要调用 `close()` 以完成 `constraints` 模块的初始化。当 Dirichlet 边界条件不是齐次时，需要自己写一个边界条件的函数，格式和右端项函数一样。

```
  constraints.clear();
  const FEValuesExtractors::Vector velocities(0);
  VectorTools::interpolate_boundary_values(dof_handler,
                                           0,
                                           Functions::ZeroFunction<dim>(),
                                           constraints,
                                           fe.component_mask(velocities));
  constraints.close();
```

接下来我们可以获取一下每个有限元系统分别有几个自由度，即 (3.17) 式中的 s, s, m。

```
  dofs_per_block = DoFTools::count_dofs_per_fe_block(dof_handler);
```

接下来我们考虑稀疏模式应该如何初始化。默认的初始化方式是这样的：如果自由度 i 与自由度 j 在同一个三角单元中，那么就将 (i, j) 加入稀疏模式。但是，现在事情变了，比如说，如果自由度 i 与自由度 j 分别属于第 0 个有限元系统和第 1 个有限元系统，那么 $(\nabla\Phi_{i,\mathbf{u}}, \nabla\Phi_{j,\mathbf{u}}) = 0$，这样一来 (i, j) 就没必要加入稀疏模式了。事实上，如果我们不把这些没必要的位置加入稀疏模式，我们得到的稀疏模式会比默认的小一半以上！

不难发现，在 Stokes 方程中，对于两个在同一个三角形单元中的自由度 i, j，位置 (i, j) 需要被加入稀疏模式，当且仅当符合下面三个条件之一：

- 两个自由度均属于第 0 个有限元系统；
- 两个自由度均属于第 1 个有限元系统；
- 其中一个自由度属于前两个有限元系统、另一个自由度属于第 2 个有限元系统。

我们需要创建一个表格来表示哪些位置是有必要加入稀疏模式的，然后将表格与 `constraints` 模块传入创建动态稀疏模式的函数中，详见代码。代码中用了一个异或布尔运算，是对上述条件的一个精巧表示，读者可以思考一下。

```
  Table<2, DoFTools::Coupling> coupling(dim + 1, dim + 1);
  for (unsigned int c = 0; c < dim + 1; ++c)
```

```
    for (unsigned int d = 0; d < dim + 1; ++d)
      if ((c==d) ^ (c==dim || d==dim))
        coupling[c][d] = DoFTools::always;
      else
        coupling[c][d] = DoFTools::none;

  BlockDynamicSparsityPattern dsp(dofs_per_block, dofs_per_block);
  DoFTools::make_sparsity_pattern(
    dof_handler, coupling, dsp, constraints, false);
  sparsity_pattern.copy_from(dsp);
  system_matrix.reinit(sparsity_pattern);
```

接下来我们还要初始化另一个稀疏矩阵，它和分跨稀疏矩阵的内容完全一样，只是以非分块的形式存储，所以还是用刚才的 Table 来创建即可。

```
  DynamicSparsityPattern fulldsp(dof_handler.n_dofs(),
                                 dof_handler.n_dofs());
  DoFTools::make_sparsity_pattern(
    dof_handler, coupling, fulldsp, constraints, false);
  full_sparsity_pattern.copy_from(fulldsp);
  full_system_matrix.reinit(full_sparsity_pattern);
```

最后是创建向量以及分块向量。其中 $\mathrm{diag}(M_Q)$ 采用分块向量存储是因为，当我们在装配矩阵时将局部 $\mathrm{diag}(M_Q)$ 贡献给全局，它会自然地存储在向量的压强分量位置，而分块向量可以让我们很方便地取出压强分量。

```
  system_rhs.reinit(dof_handler.n_dofs());
  solution.reinit(dof_handler.n_dofs());
  block_diag_MQ.reinit(dofs_per_block);
}
```

3.3.4 装配刚度矩阵

我们的框架还是：枚举三角网格、计算局部刚度矩阵与局部右端项、贡献给全局。另外，由于我们要计算 $\mathrm{diag}(M_Q)$，所以我们还需要一个局部 $\mathrm{diag}(M_Q)$，我们用向量来存储对角矩阵。下面是我们需要的一些局部变量：

```
template <int dim>
void StokesProblem<dim>::assemble_system()
{
  QGaussSimplex<dim> quadrature_formula(2);

  FEValues<dim> fe_values(fe,
                          quadrature_formula,
                          update_values | update_quadrature_points |
                            update_JxW_values | update_gradients);

  const unsigned int dofs_per_cell = fe.n_dofs_per_cell();

  const unsigned int n_q_points = quadrature_formula.size();

  FullMatrix<double> local_matrix(dofs_per_cell, dofs_per_cell);
```

```
Vector<double>  local_rhs(dofs_per_cell);
Vector<double>  local_block_diag_MQ(dofs_per_cell);

std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);
```

我们需要计算 $\mathbf{f} = (f_1, f_2)$ 在每个积分节点的值，并且存起来，deal.II 将向量函数的返回值定义为一个一阶张量 `Tensor<1,dim>`，所以我们需要一个长度为 `n_q_points` 的一阶张量型数组。

```
const RightHandSide<dim> right_hand_side;
std::vector<Tensor<1, dim>> rhs_values(n_q_points, Tensor<1, dim>());
```

接下来是两个神奇的模块，可以帮助我们提取 $\Phi_{i,\mathbf{u}}$ 与 $\Phi_{i,p}$。有了它们，我们就可以用 `fe_values[velocities].shape_value(i,q)` 表示 $\Phi_{i,\mathbf{u}}(x_q)$，以及 `fe_values[pressure].shape_value(i,q)` 表示 $\Phi_{i,p}(x_q)$。当然，$\nabla\Phi_{i,\mathbf{u}}(x_q)$ 和 $\nabla\cdot\Phi_{i,\mathbf{u}}(x_q)$ 也可以用类似的方法表示。

```
const FEValuesExtractors::Vector velocities(0);
const FEValuesExtractors::Scalar pressure(dim);
```

接下来是一些局部变量，用于存储 $\nabla\Phi_{i,\mathbf{u}}(x_q), \nabla\cdot\Phi_{i,\mathbf{u}}(x_q), \Phi_{i,\mathbf{u}}(x_q), \Phi_{i,p}(x_q)$ （i 取遍三角单元中的所有自由度）

```
std::vector<Tensor<2, dim>>     grad_phi_u(dofs_per_cell);
std::vector<double>             div_phi_u(dofs_per_cell);
std::vector<Tensor<1, dim>>     phi_u(dofs_per_cell);
std::vector<double>             phi_p(dofs_per_cell);
```

接下来开始枚举三角单元，初始化 `fe_values`，并计算好右端项函数在这个单元的所有积分节点上的值。

```
for (const auto &cell : dof_handler.active_cell_iterators())
  {
    fe_values.reinit(cell);
    local_matrix      = 0;
    local_block_diag_MQ = 0;
    local_rhs         = 0;

    right_hand_side.value_list(fe_values.get_quadrature_points(),
                               rhs_values);
```

现在枚举积分节点 x_q，并对所有的自由度 i，将 $\nabla\Phi_{i,\mathbf{u}}(x_q), \nabla\cdot\Phi_{i,\mathbf{u}}(x_q), \Phi_{i,\mathbf{u}}(x_q), \Phi_{i,p}(x_q)$ 计算好。

```
    for (unsigned int q = 0; q < n_q_points; ++q)
      {
        for (unsigned int k = 0; k < dofs_per_cell; ++k)
          {
            grad_phi_u[k] = fe_values[velocities].gradient(k, q);
            div_phi_u[k] = fe_values[velocities].divergence(k, q);
            phi_u[k]     = fe_values[velocities].value(k, q);
            phi_p[k]     = fe_values[pressure].value(k, q);
          }
```

剩下的部分就是按照 (3.6) 式来计算局部的 $\mathcal{A}_{\mathcal{K}}(i,j)$，局部右端项和局部 $\mathrm{diag}(M_Q)$ 的计算无需多言。注意，由于 $\mathcal{A}_{\mathcal{K}}$ 是对称的，我们可以只计算它的下三角部分。

```
        for (unsigned int i = 0; i < dofs_per_cell; ++i)
        {
          for (unsigned int j = 0; j <= i; ++j)
            {
              local_matrix(i, j) +=
                (   grad_phi_u[i][0] * grad_phi_u[j][0]
                  + grad_phi_u[i][1] * grad_phi_u[j][1] // (1)
                  - div_phi_u[i] * phi_p[j]             // (2)
                  - phi_p[i] * div_phi_u[j])            // (3)
                * fe_values.JxW(q);                     // * dx
            }
          local_block_diag_MQ(i) +=
               (phi_p[i] * phi_p[i]) // (4)
               * fe_values.JxW(q); // * dx
          local_rhs(i) += phi_u[i]       // phi_u_i(x_q)
                        * rhs_values[q]  // * f(x_q)
                        * fe_values.JxW(q); // * dx
        }
      }
```

现在，我们根据对称性来填充 $\mathcal{A}_K$ 的上三角部分

```
    for (unsigned int i = 0; i < dofs_per_cell; ++i)
      for (unsigned int j = i + 1; j < dofs_per_cell; ++j)
        local_matrix(i, j) = local_matrix(j, i);
```

最后，将局部矩阵与向量贡献给全局。这里我们不再手动枚举、贡献，而是直接使用 constraints 模块的内置函数。（当存在 Dirichlet 边界条件时，必须使用这种方式来将局部信息贡献给全局）注意到我们要把 $\mathcal{A}_K$ 贡献两次，一次是给分块方式存储的稀疏矩阵，另一次是给非分块方式存储的稀疏矩阵。

```
    cell->get_dof_indices(local_dof_indices);
    constraints.distribute_local_to_global(local_matrix,
                                           local_rhs,
                                           local_dof_indices,
                                           full_system_matrix,
                                           system_rhs);
    constraints.distribute_local_to_global(local_matrix,
                                           local_dof_indices,
                                           system_matrix);
    constraints.distribute_local_to_global(local_block_diag_MQ,
                                           local_dof_indices,
                                           block_diag_MQ);
  }
}
```

3.3.5 定义预优器

预优器是一个拥有成员函数 vmult 的对象，该成员函数用于描述预优器的作用，原型如下：

```
void vmult(Vector<double> &dst, const Vector<double> &src) const;
```

它有两个参量，以第二个常引用参量 src 作为输入，经过预优器作用后，以第一个引用参量 dst 作为输出。

根据作业要求，预优器应该对第一速度分量、第二速度分量分别施以一次多重网格 V-cycle 作用，对压强分量施以一次 $\mathrm{diag}(M_Q)^{-1}$ 的作用。这就要求我们能够提取分量，但是传进来的 src 不是分块向量，不能直接提取分量，所以我们要把它先转换成分块向量。

```
void PreconditionStokes::vmult(Vector<double> &dst, const Vector<double> &src) const
{
  BlockVector<double> block_src;
  BlockVector<double> block_dst;
  block_src.reinit(dofs_per_block);
  block_dst.reinit(dofs_per_block);
  block_src = src;
```

现在对每个分量分别施以对应的作用：

```
  A_preconditioner[0].vmult(block_dst.block(0), block_src.block(0));
  A_preconditioner[1].vmult(block_dst.block(1), block_src.block(1));

  for(unsigned i = 0; i < diag_MQ_inverse.size(); i++)
    block_dst.block(2)[i] = block_src.block(2)[i] * diag_MQ_inverse[i];
```

最后将结果转换回普通向量：

```
  dst = block_dst;
}
```

通过 vmult 函数可以看出，类 PreconditionStokes 的成员变量至少需要包括：

- 多重网格模块 A_preconditioner[2]；
- 存储 $\mathrm{diag}(M_Q)^{-1}$ 的向量 diag_MQ_inverse；
- 初始化分块向量所需的 dofs_per_block。

而初始化这些成员函数需要：分块方式存储的 $\mathcal{A}$、$\mathrm{diag}(M_Q)$、用于说明分块方式的数组。因此，我们可以写出预优器类的定义：

```
class PreconditionStokes
{
public:
  void initialize(const BlockSparseMatrix<double>& system_matrix,
                  const BlockVector<double>& block_diag_MQ,
                  const std::vector<types::global_dof_index>& dofs_per_block_input);
  void vmult(Vector<double> &dst, const Vector<double> &src) const;

private:
  TrilinosWrappers::PreconditionAMG A_preconditioner[2];
  std::vector<types::global_dof_index> dofs_per_block;
  Vector<double> diag_MQ_inverse;
};
```

现在，只剩一个初始化函数需要我们实现。首先将说明分块方式的数组拷贝给成员变量：

```
void PreconditionStokes::initialize(
  const BlockSparseMatrix<double>& system_matrix,
  const BlockVector<double>& block_diag_MQ,
```

```
  const std::vector<types::global_dof_index>& dofs_per_block_input)
{
  dofs_per_block = dofs_per_block_input;
```

接下来初始化 Trilinos 的多重网格模块，开启高阶元参数：

```
  TrilinosWrappers::PreconditionAMG::AdditionalData additional_data;
  additional_data.higher_order_elements = true;

  A_preconditioner[0].initialize(system_matrix.block(0,0), additional_data);
  A_preconditioner[1].initialize(system_matrix.block(1,1), additional_data);
```

最后提取 block_diag_MQ 的压强分量得到 diag(M_Q)，并求逆：

```
  diag_MQ_inverse = block_diag_MQ.block(2);
  for(auto& p : diag_MQ_inverse) p = 1. / p;
}
```

3.3.6 求解稀疏线性系统

有了预优器，求解稀疏线性系统就很简单了。首先是求解控制器，传入最大迭代次数、容忍误差。然后定义一个 MinRes 求解器。

```
template <int dim>
void StokesProblem<dim>::solve()
{
  SolverControl solver_control(50000, 1e-6 * system_rhs.l2_norm());
  SolverMinRes<Vector<double>> minres(solver_control);
```

接下来创建一个 PreconditionStokes 对象，并按我们自己定义的方式传入初始化参数：

```
  PreconditionStokes preconditioner;
  preconditioner.initialize(system_matrix,
                            block_diag_MQ,
                            dofs_per_block);
```

接下来调用 minres.solve() 进行求解。注意，这个成员函数只能接受非分块方式存储的稀疏矩阵和向量，这就是我们之所以用两种方式存储刚度矩阵的原因！

```
  minres.solve(full_system_matrix,
               solution,
               system_rhs,
               preconditioner);
```

然后是将 Dirichlet 边界条件赋值给解：

```
  constraints.distribute(solution);
```

最后可以输出看看 MinRes 总迭代轮数。（通常不超过 100 轮，且轮数不随网格加密而增加。如果将预优器换成 PreconditionIdentity，即无预优，我们将会观察到迭代轮数指数级增加）

```
  std::cout << "    " << solver_control.last_step()
            << " MINRES iterations." << std::endl;
}
```

3.3.7 输出结果

我们要把解向量拆分成速度向量分量、压强标量分量来分别输出，所以需要一些声明，即：前两个有限元系统是速度向量分量的一部分、后一个有限元系统是压强标量分量。具体见代码。

```
template <int dim>
void StokesProblem<dim>::output_results() const
{
  std::vector<std::string> solution_names(dim, "velocity");
  solution_names.emplace_back("pressure");

  std::vector<DataComponentInterpretation::DataComponentInterpretation>
    data_component_interpretation(
      dim, DataComponentInterpretation::component_is_part_of_vector);
  data_component_interpretation.push_back(
    DataComponentInterpretation::component_is_scalar);
```

然后传进 add_data_vector 里输出即可。我们传给 build_patches 里的参数表示在输出时，对每个三角形单元输出几个点的值；当不传值时，只输出三角形三个顶点的值；当传入 2 时，会额外输出三角形三边中点的值，可以让我们画出的图像更光滑一些。

```
  DataOut<dim> data_out;
  data_out.attach_dof_handler(dof_handler);
  data_out.add_data_vector(solution,
                           solution_names,
                           DataOut<dim>::type_dof_data,
                           data_component_interpretation);
  data_out.build_patches(2);

  std::ofstream output(
    "solution" + Utilities::int_to_string(level, 2) + ".vtk");
  data_out.write_vtk(output);
}
```

3.3.8 计算误差

我们这里只计算 L^2 误差，计算公式是：

$$E_{u_1}^2 = \int_\Omega |u_1 - u_{h,1}|^2 \, dx \approx \sum_{\mathcal{K}\in\mathcal{T}} \sum_q |u_1(x_q) - u_{h,1}(x_q)|^2 w_q. \tag{3.19}$$

其中 x_q 是积分节点的坐标，w_q 是积分节点的权重，q 取遍 $\mathcal{K}$ 上的所有积分节点。E_{u_2} 和 E_p 类似。

为了使计算结果能够体现出收敛阶，我们需要一个至少 3 阶代数精度的积分公式，这里我们用了 3 点高斯积分公式，它具有 5 阶代数精度。

代码的思路和装配矩阵一样，但省去了贡献给全局的步骤，如下。

```
template <int dim>
void StokesProblem<dim>::compute_L2_error()
{
  QGaussSimplex<dim> quadrature_formula(3);
  FEValues<dim> fe_values(fe,
                          quadrature_formula,
```

```
                      update_values | update_quadrature_points |
                        update_JxW_values);

  const unsigned int n_q_points = quadrature_formula.size();

  const TrueSolution<dim> true_solution;
  std::vector<Tensor<1, dim>> numerical_results_u(n_q_points);
  std::vector<double>     numerical_results_p(n_q_points);

  const FEValuesExtractors::Vector velocities(0);
  const FEValuesExtractors::Scalar pressure(dim);

  double err_u1 = 0.;
  double err_u2 = 0.;
  double err_p = 0.;

  for (const auto &cell : dof_handler.active_cell_iterators())
    {
      fe_values.reinit(cell);
      fe_values[velocities].get_function_values(solution, numerical_results_u);
      fe_values[pressure].get_function_values(solution, numerical_results_p);
      auto q_points = fe_values.get_quadrature_points();

      for (unsigned int q = 0; q < n_q_points; ++q)
        {
          auto jxw = fe_values.JxW(q);
          err_u1 += pow(true_solution.value(q_points[q], 0) - numerical_results_u[q][0], 2.) * jxw;
          err_u2 += pow(true_solution.value(q_points[q], 1) - numerical_results_u[q][1], 2.) * jxw;
          err_p += pow(true_solution.value(q_points[q], 2) - numerical_results_p[q], 2.) * jxw;
        }
    }

  std::cout << "      u1 error: " << sqrt(err_u1) << std::endl;
  std::cout << "      u2 error: " << sqrt(err_u2) << std::endl;
  std::cout << "      p  error: " << sqrt(err_p) << std::endl << std::flush;
}
```

值得注意的是，我们这样算出的并不是 $||u_1 - u_{h,1}||_{L^2}$，只是一个近似值，但用于计算收敛阶已经足够。要算出误差的精确值，要将每个三角形不断切分成小三角形计算下去，直到三角形足够小或者结果足够可信。精确计算误差的 L^2 范数代价是很大的，甚至比求解的耗时更久，所以我们不推荐这么做。

3.3.9 主函数

主函数随便写写就好了。注意因为我们用了 Trilinos 的代数多重网格，所以要加上一句 MPI 的初始化。

```
int main(int argc, char *argv[])
{
  Utilities::MPI::MPI_InitFinalize mpi_initialization(argc, argv, 1);
  int level;
  std::cin >> level;
```

```
  StokesProblem<2> flow_problem(level);
  flow_problem.run();
  return 0;
}
```

完整代码见文件夹 stokesMinRes。