deal.II learning note

1.deal.II介绍

deal.II是一个开放源代码的有限元程序库，是一个C++软件库，支持创建有限元代码以及开放的用户和开发人员社区。

deal.II 是一个开源的有限元法求解器，支持大规模并行计算，自适应网格。采用C++编写，实现优雅。其文档完整丰富，文档共有三个级别，由浅入深：

- tutorial：一系列的教学程序，共有64步教学步骤，通过tutorial的学习可以对dealii有整体的认识。
- manual：对每个类以及相应的函数的介绍。适合用于查询类与函数的具体用法。
- Modules：介绍了实现某一个功能需要用到的一系列类与函数，比如 Sparsity patterns 介绍了存储稀疏矩阵相关的内容。

本deal.II学习笔记基于deal.II官方文档，学长的学习笔记和Wolfgang Bangerth's lectures。

2.deal.II安装

3.deal.II示例学习

3.1.Step-1

相关链接：deal.II step-1 tutorial program

step-1的源码位于**examples/step-1**目录，进入该目录并输入如下命令进行编译（对于其它的step项目同样需要进行如下的编译）：

```
cmake .         # 生成makefile，会查找deal.II库
make            # 生成可执行文件
make run        # 执行可执行文件，也会编译代码，所以第二句代码可忽略
```

如`cmake .`报错，如果此命令无法找到deal.II库，则需要使用以下命令提供安装路径：

```
cmake -DDEAL_II_DIR=/path/to/installed/deal.II .
```

内容说明

step-1的主要内容是网格生成。网格有两种，一是方形网格，二是环形网格，并展示了网格迭代细化的过程。

有限元程序的三个过程：网格的Triangulation类型的对象；调用GridGenerator函数以生成网格；并在相关迭代器的所有单元上循环；

示例程序

加载头文件

首先需要加载头文件，最为重要的是 Triangulation 类，其用于生成单元，Triangulation<1>表示一维单元，以此类推，可以表示二维单元和三维单元，对于边界单元，Triangulation<1,2>表示面边界上的曲线单元，Triangulation<2,3>表示体边界上的曲面单元。

```
#include <deal.II/grid/tria.h>
```

加载以下两个与在单元格和（或）面上的循环的头文件，实现单元存取和迭代：

```
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/grid/tria_iterator.h>
```

生成具有一点过标准形状的网格：

```
#include <deal.II/grid/grid_generator.h>
```

网格的格式化输出：

```
#include <deal.II/grid/grid_out.h>
```

C++输入输出：

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

载入库cmath（数学运算库）：

```
#include <cmath>    // for std::sqrt and std::fabs
```

最后加上命名空间dealii，确保调用的函数来自dealii库。

```
using namespace dealii;
```

创建第一个网格（正方形网格）

首先申明二维单元triangulation，然后设置单元为立方体形状，并对单元进行四次全局细化，每次细化一分为四，总共将产生$4^4 = 256$个单位网格。最终将网格绘制成eps图像输出（或其它图像格式，如*.svg）。

```
void first_grid()
{
  Triangulation<2> triangulation;
  GridGenerator::hyper_cube(triangulation);
  triangulation.refine_global(4);
  std::ofstream out("grid-1.eps");
  GridOut       grid_out;
  grid_out.write_eps(triangulation, out);
  std::cout << "Grid written to grid-1.eps" << std::endl;
}
```

创建第二个网格（环形网格）

同样首先申明二维单元triangulation，然后设置中心点，内外圆的半径，并用hyper_shell函数生成网格，可以通过此函数自动调整圆周单元的数量，最后一个参数显式设置为10，即内外圈分为10个环带网格。

```
void second_grid()
{
  Triangulation<2> triangulation;
  const Point<2> center(1, 0);
  const double   inner_radius = 0.5, outer_radius = 1.0;
  GridGenerator::hyper_shell(
    triangulation, center, inner_radius, outer_radius, 10);
```

> 默认情况下，triangulation假定所有边界均为直线，为保证区域边界是弯曲的，引入一个manifold indicator，如果没有manifold描述与特定的manifold indicator关联，则表示产生笔直边缘的manifold。

然后通过五次（自定义）循环细化网格：

```
for (unsigned int step = 0; step < 5; ++step)
  {
```

> triangulation的单元格并不是数组方式储存，而是以迭代器（iterator）的方式进行储存，包含指针，迭代器的基本操作见C++学习笔记。
>
> active cell是未进一步细化的单元，并且是可以标记为进一步细化的单元，即已激活的网格，所以遍历的是最细的网格。

遍历所有active cell：

```
for (auto it = triangulation.active_cell_iterators().begin();
     it != triangulation.active_cell_iterators().end();
     ++it)
  {
    auto cell = *it;
    // Then a miracle occurs...
  }
```

上面的写法需要声明循环的初始与结束条件，在此还可以采用更为简便的方式：

```
for (auto &cell : triangulation.active_cell_iterators())
  {
```

获取单元的维度：

```
    for (const auto v : cell->vertex_indices())
     {
```

在此细化的策略为：如果网格是靠近圆环内边沿的网格，即有节点在圆环内径上，那么就加密这个网格。由于计算机机内小数表示有误差，因此不能直接判断等于，而是差值绝对值小于一个极小数，该极小数设置为等于环的内半径的10^{-6}倍。

```
      const double distance_from_center =
        center.distance(cell->vertex(v));  // 网格节点
      if (std::fabs(distance_from_center - inner_radius) <=
          1e-6 * inner_radius)
        {
          cell->set_refine_flag();  // 标记为active cell
          break;
        }
    }
}
```

已经标记了所有要细化的单元，接下来执行网格的稀疏和细化函数即可：

```
  triangulation.execute_coarsening_and_refinement();
}
```

最后输出网格的图像：

```
  std::ofstream out("grid-2.eps");
  GridOut       grid_out;
  grid_out.write_eps(triangulation, out);
  std::cout << "Grid written to grid-2.eps" << std::endl;
}
```

main函数

在main()调用之前生成两个网格的函数即可。

```
int main()
{
  first_grid();
  second_grid();
}
```

3.2.Step-2

相关链接：[deal.II step-2 tutorial program]

内容说明

在Step-1中创建了网格之后，Step-2将介绍如何在此网格上定义自由度。 在此示例中，我们将使用最低阶（Q_1）有限元，其自由度与网格的顶点相关。 在后面的示例将介绍高阶单元，在高阶有限元中，自由度不必再与顶点关联，而是可以与边缘，面或体相关联。

在有限元方法中，自由度（degree of freedom）有两个存在略微差异的意思：

- 将有限元解表示为形状函数的线性组合，即$u_h(x) = \Sigma_{j=0}^{N-1} U_j \varphi_j(x)$。式中，$U_j$是展开系数向量，其值未知，所以称之为未知量或自由度。
- 有限元问题的数学解释为，寻找一个满足某些方程组的有限维函数$u_h \in V_h$，如对所有的测试函数$\varphi_h \in V_h$均满足$a(u_h, \varphi_h) = (f, \varphi_h)$。首先需要选择空间$V_h$的一组基，是一组形状函数，特别选择由有限元函数描述的基元，这些函数通常是定义在网格和单元上的。所以，自由度即计算出的空间V_h的基函数数目，在deal.II中，类`DoFHandler`提供V_h基数目的计算，提供了自由度相关的功能。

  基于类`DoFHandler`，所以在deal.II中定义自由度变得很简单，需要做的即创建一个有限元对象，使用`DoFHandler::disturbute_dofs`函数进行有限元对象的自由度定义，能够从中获取局部的自由度和形状函数的下标等等，这些是确定系统矩阵的大小以及将单个单元的组成矩阵复制到全局矩阵时需要的信息。

稀疏性

稀疏性是有限元方法的显着特征之一，稀疏性有助于求解有数以万计自由度的矩阵问题。

在定义了自由度之后，便需要根据微分方程，计算相应的线性方程组，在Step-3中会讲解相应的过程。有限元法中非常重要的一点：有限元法线性方程组中的矩阵是稀疏的，即大多数系数为0。

更精确地说，如果矩阵中每行/列非零项的数量由一个与总自由度数量无关的数字所限制，则矩阵是稀疏的。例如，求解拉普拉斯方程的有限差分近似的简单5点法得到的解为稀疏矩阵，因为每行非零项的数量为5，与矩阵的总大小无关。对于更复杂的问题（例如，Step-22的斯托克斯问题），尤其是在三维中，每行的数字数目可能为数百。 但是重要的一点是，该数目与问题的整体大小无关：如果优化（精细）网格，则每行非零数的最大数目保持不变。

假设一个有N行的矩阵，其每行具有特定的非零数目上限，需要$O(N)$个存储位置进行存储（空间复杂度），而矩阵向量乘法也仅需要$O(N)$个操作（时间复杂度）。显然，如果矩阵不是稀疏的，通过$O(N)$个操作求解完成是不可能的，因为非稀疏矩阵中的项数必须为$O(N^s)$，且其中$s > 1$，会进行$O(N^s)$次运算，这还是使用了多重网格法这种非常专业的求解器情况下。

在有限元方法中，稀疏性源于有限元形状函数是在单个单元上局部定义的，而不是全局定义的，并且局部微分算子仅与相邻的单元的形状函数有关，体现在矩阵中就是非零项，而非相邻项则为零，从而导致了矩阵的稀疏，即，未知量i，j所在的单元如果没有毗邻，则在最终的矩阵中$A_{ij} = A_{ji} = 0$。

自由度计算

默认情况下，`DoFHandler`类随机处理网格上的自由度。 因此，稀疏矩阵也没有针对任何特定目的进行优化，即不是最优的。

对于大多数算法，对自由度进行编号的确切方式并不重要。如CG算法。但是也有一些算法对自由度的排布较为敏感，如SSOR、LU分解、Cholesky分解等。因此，deal.II中也包含了可以重新排列未知量的工具：`FRenumbering`。这个函数相比与随机排布有一定的优化。

示例程序

加载头文件

同Step-1一样，首先导入相应的头文件：

```
#include <deal.II/grid/tria.h>
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/grid/tria_iterator.h>
#include <deal.II/grid/grid_generator.h>
```

引入处理自由度的库：

```
#include <deal.II/dofs/dof_handler.h>
```

接下来引入与单元有关的库，在这个库种，包含了各种维度，各种阶次的拉格朗日单元的实现：

```
#include <deal.II/fe/fe_q.h>
```

引入处理自由度的工具：

```
#include <deal.II/dofs/dof_tools.h>
```

引入稀疏矩阵可视化网格的类：

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

引入中间稀疏模式结构：

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

对自由度进行重新排列：

```
#include <deal.II/dofs/dof_renumbering.h>
```

文件输出：

```
#include <fstream>
```

导入deal.II名字空间：

```
using namespace dealii;
```

网格生成

使用的是Step-1程序生成环形网格，并已经过数步网格优化,可以返回根据参数生成的网格。

```
void make_grid(Triangulation<2> &triangulation)
{
  const Point<2> center(1, 0);
  const double   inner_radius = 0.5, outer_radius = 1.0;
  GridGenerator::hyper_shell(
    triangulation, center, inner_radius, outer_radius, 5);
  for (unsigned int step = 0; step < 3; ++step)
    {
      for (auto &cell : triangulation.active_cell_iterators())
        for (const auto v : cell->vertex_indices())
          {
```

```
            const double distance_from_center =
              center.distance(cell->vertex(v));
            if (std::fabs(distance_from_center - inner_radius) <=
                1e-6 * inner_radius)
              {
                cell->set_refine_flag();
                break;
              }
          }
      triangulation.execute_coarsening_and_refinement();
    }
}
```

自由度处理DoFHandler

至此，已创建了一个网格，且节点和单元的几何信息和拓扑信息已基本明确。为了使用数值算法，我们需要将节点（或线，单元）的信息与自由度相关联，以便后续生成有限元的矩阵和向量。deal.II中`DoFHandler`类就是来完成这部分工作的。但是，在执行此操作之前，首先需要建立有限元单元来描述与对象相关联的自由度。衍生类`FE_Q`能够描述拉格朗日有限单元，它的构造函数采用一个参数来说明元素的多项式，如参数1表示形状函数为1阶多项式，假设参数为3，则将创建一盒有限单元，每个顶点具有一个自由度，每条线具有两个自由度，并且在单元内具有四个自由度。

首先需要建立一个有限单元，并用`DoFHandler`进行自由度处理：

```
void distribute_dofs(DoFHandler<2> &dof_handler)
{
  const FE_Q<2> finite_element(1);
  dof_handler.distribute_dofs(finite_element);
```

网格的每一个节点都有一个形状函数。求解拉普拉斯方程时，得到的矩阵中非零元素a_{ij}为每对相邻(i,j)形状函数梯度的积分（求和）。`DoFHandler :: distribute\_dofs()`对顶点进行了或多或少的随机编号，因此矩阵中非零项的模式有些参差不齐，需要进一步的处理。

1. 首先需要一个数据结构来储存非零元素的位置，那么便可以根据这个存储下来的“稀疏模式”来存储矩阵中的值。`SparsityPattern`类可以完成位置的储存，其缺点在于当我们直接使用时，需要预估每一行（列）的最大非零项数目（预估值）。对于二维空间，可信的预估值可通过函数

   `DoFHandler::max\_couplings\_between\_dofs()`获得；
   对于三维空间，预估值一般都偏大，会造成储存空间的浪费，为防止这种浪费，引入`DynamicSparsityPattern`类型来创建内部数据空间便于无过载地转换成`SparsityPattern`类，其初始化需要给出矩阵的大小。

   ```
   DynamicSparsityPattern dynamic_sparsity_pattern(dof_handler.n_dofs(),
                                                   dof_handler.n_dofs());
   ```

2. 随后得到非零元素的位置：

   ```
   DoFTools::make_sparsity_pattern(dof_handler,dynamic_sparsity_pattern;
   ```

3. 创建实际的稀疏模式，以后可以将其用于矩阵：

   ```
   SparsityPattern sparsity_pattern;
   sparsity_pattern.copy_from(dynamic_sparsity_pattern);
   ```

4. 将结果写入.svg文件中，其中矩阵中的每个非零项都与图像中的红色正方形相对应。

```
std::ofstream out("sparsity_pattern1.svg");
sparsity_pattern.print_svg(out);
}
```

> 查看.svg文件，可以注意到稀疏模式是对称的。 这不足为奇，因为我们没有给`DoFTools :: make_sparsity_pattern`任何非对称矩阵信息；
>
> 还将注意到，它具有几个不同的区域，这是由于编号从最稀疏的单元网格开始，然后移动到更精细的单元网格。

自由度重新排布

在之前程序生成的稀疏模式中，非零项离对角线很远，这对某些算法（如LU分解或者GS迭代）的使用是不利的，所以需要通过重新排布非零项来进行优化。

(i,j)为矩阵中的非零项，只有单元i和j相邻时，其值不为0，为了使非零项聚集在对角线附近，所以需要调整相邻的形状函数的索引号相差不大。（最优化问题）

这可以通过简单的前行算法来实现：从给定的顶点开始，并为其指定索引0。 然后，对其邻点进行连续编号，使其索引接近原始索引（1，2，3，...）。 然后，对邻点的邻点（如果尚未编号）进行编号，以此类推。

示例中采用的是由Cuthill和McKee提出的算法，在代码中调用`DoFRenumbering::Cuthill_McKee`即可：

```
void renumber_dofs(DoFHandler<2> &dof_handler)
{
  DoFRenumbering::Cuthill_McKee(dof_handler);
  DynamicSparsityPattern dynamic_sparsity_pattern(dof_handler.n_dofs(),
                                                  dof_handler.n_dofs());
  DoFTools::make_sparsity_pattern(dof_handler, dynamic_sparsity_pattern);
  SparsityPattern sparsity_pattern;
  sparsity_pattern.copy_from(dynamic_sparsity_pattern);
  std::ofstream out("sparsity_pattern2.svg");
  sparsity_pattern.print_svg(out);
}
```

值得注意的是，`DoFRenumbering`类还提供了许多其他算法来对自由度进行重新编号。 例如，所有非零项都在矩阵的下三角或上三角中，但对于对称稀疏模式，这当然是无法实现的，但是在某些涉及输运方程的特殊情况下是可行的，如从流入边界沿流线到流出边界。

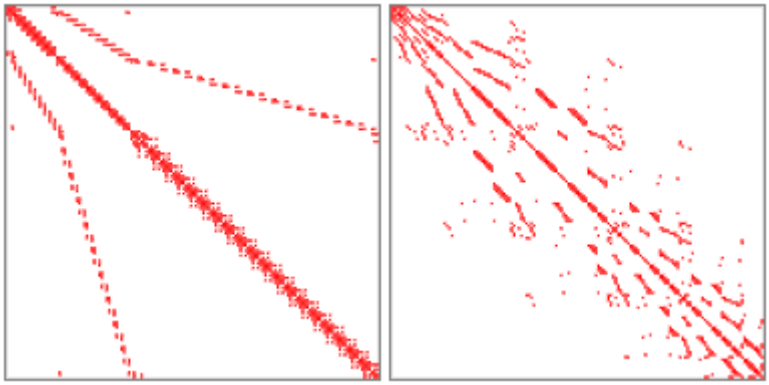
主函数

最后，用主函数来调用各函数，形成最终的程序：

```
int main()
{
  Triangulation<2> triangulation;
  make_grid(triangulation);
  DoFHandler<2> dof_handler(triangulation);
  distribute_dofs(dof_handler);
  renumber_dofs(dof_handler);
}
```

结果

该程序运行后产生了两种稀疏模式。可以通过在Web浏览器中打开.svg文件来可视化它们。



左侧图片中的不同区域（左上角和其他部分网格区域）表示网格的不同细化级别上的自由度。右侧图片的稀疏模式在重新编号后非零项离对角线更近了。

拓展

首先，可以尝试更改单元的阶数，比如改成3或5，即`distribute_dofs`函数的参数设置为3或5。

或者，如果想要观察网格细化对矩阵的影响，将局部加密的步数（细化次数）改为4。

最后，还可以尝试`DoFRenumbering`名字空间中的其他的排布算法。

在可视化的方式上，可以采用`gnuplot`，在源代码函数 `distribute_dofs()` 和`renumber_dofs()`中将`print_svg()`改为`print_gnuplot()`。

```
examples/step-2> gnuplot
        G N U P L O T
        Version 3.7 patchlevel 3
        last modified Thu Dec 12 13:00:00 GMT 2002
        System: Linux 2.6.11.4-21.10-default
        Copyright(C) 1986 - 1993, 1998 - 2002
        Thomas Williams, Colin Kelley and many others
        Type `help` to access the on-line reference manual
        The gnuplot FAQ is available from
        http://www.gnuplot.info/gnuplot-faq.html
        Send comments and requests for help to <info-gnuplot@dartmouth.edu>
        Send bugs, suggestions and mods to <bug-gnuplot@dartmouth.edu>
Terminal type set to 'x11'
gnuplot> set style data points
gnuplot> plot "sparsity_pattern.1"
```

另一种基于GNUPLOT的做法是尝试打印出节点的位置和编号的网格。需要包括GridOut和MappingQ1的头文件。此代码是：

```
std::ofstream out("gnuplot.gpl");
out << "plot '-' using 1:2 with lines, "
    << "'-' with labels point pt 2 offset 1,1"
    << std::endl;
GridOut().write_gnuplot (triangulation, out);
out << "e" << std::endl;
const int dim = 2;
```

```
std::map<types::global_dof_index, Point<dim> > support_points;
DoFTools::map_dofs_to_support_points (MappingQ1<dim>(),
                                      dof_handler,
                                      support_points);
DoFTools::write_gnuplot_dof_support_point_info(out,
                                               support_points);
out << "e" << std::endl;
```

为了查看.gpl文件，需要在命令行输入

```
gnuplot -p gnuplot.gpl
```

3.3.Step-3

相关链接：deal.II step-3 tutorial program

内容说明

有限元法简介

前面的示例中建立了网格并涉及了稀疏矩阵，在Step-3中讲使用有限元法对零边界和右值不为0的泊松方程：

$$
\begin{aligned}
-\Delta u &= f \quad in \Omega; \\
u &= 0 \quad on\ \partial\Omega;
\end{aligned}
$$

计算域为方形域，$\Omega = [-1,1]^2$。在本实例中仅考虑$f(x) = 1$的情况，将在Step-4中将会考虑更加复杂的情况。

在有限元法中，为得到近似解u，需要对方程进行近似处理（有限尺度近似）。引入测试函数φ，对方程进行整个域里的积分：

$$-\int_\Omega \varphi \Delta u = \int_\Omega \varphi f.$$

由格林第一公式可将上式化为：

$$\int_\Omega \nabla\varphi \cdot \nabla u - \int_{\partial\Omega} \varphi \boldsymbol{n} \cdot \nabla u = \int_\Omega \varphi f.$$

> 格林第一公式的证明：
>
> 假设$u(x,y,z), v(x,y,z) \in C^2(\bar{\Omega})$，设$F = u\nabla v$根据高斯公式有
>
> $$\iiint_\Omega \nabla F = \iiint_\Omega \nabla \cdot (u\nabla v) = \iint_{\partial\Omega} u\nabla v \cdot \boldsymbol{n}$$
>
> 积分中的项$\nabla \cdot (u\nabla v) = \nabla u \nabla v + u\Delta v$，代入上式中，即可得到格林第一公式：
>
> $$\iiint_\Omega u\Delta v = \iint_{\partial\Omega} u\frac{\partial v}{\partial n} - \iiint_\Omega \nabla u \cdot \nabla v.$$

测试函数也需要满足边界条件，在边界上$\varphi = 0$，从而得到弱化的方程

$$(\nabla\varphi, \nabla u) = (\varphi, f),$$

其中通用符号$(a,b) = \int_\Omega ab$。所以问题求解就变成了寻找合适的函数u，使得对于任意的测试函数，都要满足以上的方程。

通过计算机来寻找这样的函数u是很难的，所以可以寻找其近似展开

$u_h(x) = \Sigma_j U_j \varphi_j(x)$

其中，U_j为未知的展开式系数（自由度），$\varphi_i(x)$为有限元的形状函数。定义形状函数，需要以下步骤：

- 定义网格。在Step-1以及Step-2中已经讨论了相应的操作。
- 定义有限单元。在deal.II中，一维为$[0,1]$，二维为$[0,1]^2$，三维为$[0,1]^3$，即均为单位长度的线、面、体。在Step-2中，我们定义了一个`FE_Q<2>`类型的对象，表示二维拉格朗日单元，通过支撑点插值得到形状函数。最简单的就是`FE_Q<2>(1)`，表示线性二维拉格朗日单元。
- `DoFHandler`类枚举得到网格上的所有自由度，并以有限元对象对应的参考单元描述为基础。
- 通过映射从参考单元上的有限元类定义的形状函数中获得实单元上的形状函数。

通过以上步骤，我们得到了一系列的形状函数φ_i，可以定义离散问题的求解形式：找到一个函数u_h，即找到上面提到的展开系数U_j，使得

$$(\nabla\varphi_i, \nabla u_h) = (\varphi_i, f), \qquad i = 0 \ldots N-1.$$

在C或者C++中，下标都是从0开始的。上述方程可以写成一个线性方程组

$$\begin{aligned}(\nabla\varphi_i, \nabla u_h) =& (\nabla\varphi_i, \nabla[\Sigma_j U_j \varphi_j]) \\ =& \Sigma_j (\nabla\varphi_i, \nabla[U_j \varphi_j]) \\ =& \Sigma_j (\nabla\varphi_i, \nabla\varphi_j) U_j.\end{aligned}$$

因此，问题变成了求解$AU = F$，其中，$A_{ij} = (\nabla\varphi_i, \nabla\varphi_j), F_i = (\varphi_i, f)$.

测试函数左乘或右乘

测试函数右乘，可以得到$U^T A = F^T$，其中F^T为列向量，对方程进行转置有$A^T U = F$，其中，$A^T = A$是对称矩阵。不过在大多数情况下A并不是对称矩阵，在此只是为了说明左乘和右乘的问题对此进行简化。大多数情况下在方程左乘测试函数只是习惯，无论左乘右乘最终的结果都是正确的。

计算矩阵和右端向量

对于矩阵和右端向量的求解，需要知道以下几点：

- 线性方程的A为稀疏矩阵，U和F为列向量，下文的示例程序讲介绍如何求解线性方程。
- 在有限元方法中，这通常是使用正交方法来代替积分，即积分被每个单元上一组点上的加权和所代替。 也就是说，首先将Ω上的积分拆分为所有单元上的积分

  $$\begin{aligned}A_{ij} =& (\nabla\varphi_i, \nabla\varphi_j) = \sum_{K\in T} \int_K \nabla\varphi_i \cdot \nabla\varphi_j, \\ F_i =& (\varphi_i, f) = \sum_{K\in T} \int_K \varphi_i f,\end{aligned}$$

  讲积分拆为各单元积分之和：

  $$\begin{aligned}A_{ij}^K =& \int_K \nabla\varphi_i \cdot \nabla\varphi_j \approx \sum_q \nabla\varphi_i(x_q^K) \cdot \nabla\varphi_j(x_q^K) w_q^K, \\ F_i^K =& \int_K \varphi_i f \approx \sum_q \varphi_i(x_q^K) f(x_q^K) w_q^K,\end{aligned}$$

  其中，x_q^K是单元格K上的第q个正交点，w_q^K是第q个正交权重。
- 首先，同形状函数映射一样从参考单元进行映射，通过隐式地使用`MappingQ1`类来描述x_q^K和w_q^K，或从`Mapping`中派生出的其它函数进行映射。参考单元上的位置和权重由从`Quadrature`类派生类进行描述。通常数值积分的值可以等于理论积分的值，因为所有需要积分的项均为多项式，在此一般用高斯积分，在deal.II中由`QGauss`类实现。

- 然后，利用`FEValues`来计算单元网格K上的$\varphi_i(x_q^K)$。它需要一个有限元对象来描述参考单元格上的φ，一个正交对象来描述正交点和权重， 一个映射对象（或隐式采用MappingQ1类），并在位于K上的正交点上，提供实单元K上的形状函数的值和导数以及积分所需的各种其他信息。

`FEValues`是合成线性方程组的核心过程。可以这样看待这一过程：类`FiniteElement`以及衍生的类描述了形状函数，即保证了每个点上都有相应的函数值。需要形函数是因为我们需要对函数进行积分。但是，计算机只能处理离散信息，所以需要使用数值积分。由`Mapping`类匹配实际单元与参考单元，由`Quadrature`类提供数值积分点，通过对函数值加权求和得到数值积分的值。更为简单地说，我们需要在有限个点上拥有形函数值以及导数、高斯积分权重以及法向量等，便可以求解这个问题。`FEValues`类便是整合这些信息的类。

这三个类的工作也可以通过程序实现，并整合相应的信息。但是deal.II中的`FEValues`提供了大量优化，实现简洁、快速。

最后采用线性求解器求解线性方程组，并通过`DataOut`类输出结果。

关于程序实现

虽然这是用有限元法实现的最简单的问题，但是在示例程序中展示了有限元法程序的基本结构，这也可以作为求解其它问题的一个基模板，程序中类的结构如下：

```
class Step3
{
  public:
    Step3 ();
    void run ();
  private:
    void make_grid ();
    void setup_system ();
    void assemble_system ();
    void solve ();
    void output_results () const;
    Triangulation<2>     triangulation;
    FE_Q<2>              fe;
    DoFHandler<2>        dof_handler;
    SparsityPattern      sparsity_pattern;
    SparseMatrix<double> system_matrix;
    Vector<double>       solution;
    Vector<double>       system_rhs;
};
```

成员变量：需要网格`Triangulation`类以及自由度分配`DoFHandler`类的实例对象、以及一个有限元对象来描述形函数。接着还需要与线性代数有关的对象：包括矩阵、右端向量、解向量以及一个描述稀疏模式的矩阵。与此相反，仅在组成线性方程过程中才需要`FEValues`类，便只在相关的函数中声明，将其创建为本地对象，并在使用结束后销毁它。

成员函数：这些函数构成了基本结构，后续教程将会用到。

- `make_grid()`：创建对象储存网格，在之后的教程中，还会具有处理边界条件、几何等相关信息的功能。
- `setup_system()`：设置求解问题所需的所有数据结构。它会初始化`DoFHandler`对象，并且处理与求解线性方程组的所有类。该函数独立于`make_grid()`函数，是因为后续的自适应细化网格中，就会循环调用该函数（Step-6），而且创建网格并进行设置只需要在程序开头运行一次`make_grid()`。
- `assemble_system()`：计算线性方程组中的系数矩阵和右端向量。

- `solve()`:求解线性方程组。由于目前的矩阵较为简单，所以`solve()`函数的实现也较为简单。但是随着问题逐渐复杂，求解器的设计也会相应变得复杂（在之后的Step-20、Step-22以及Step-31会涉及到）。
- `output_results()`：对线性方程组的解U进行后处理。如，可以将解输出为特定格式便于分析计算。

上述的函数都是与公共函数`run()`结合在一起的，封装这些函数到`run()`中是为了对之进行改动，而不用影响类之外的代码。

变量类型

deal.II通过名称空间`types`定义了许多整数类型。在本程序中，可以看到`types::global_dof_index`的类型变量，该类型是为了表示自由度的数目（全局变量），即`DoFHandler`在整个网格上得到的未知量个数。对于目前的程序，可能得到DoF的总数为几千到几百万个。因此，需要一个足够大的数来存储DoF的数目。

`unsigned int`的范围为0到40亿，在deal.II 7.3版本之前，`types::global_dof_index`都是这个类型。不过，现在deal.II已经支持非常大规模的计算，超过了40亿的范围。因此对于`types::global_dof_index`做了设定，默认类型为`unsigned int`，如果超过表示范围，那么类型更改为`unsigned long long int`。

也是有出于储存的目的：代码中的一个使用数据类型`types::global_dof_index`的对象，其所表示的数量实际上是全局DoF索引。而如果将这个值赋值给其它的类型变量，如`types::global_dof_id`可能会出现错误，尽管两者的数据类型均为`unsigned int`。

在通常的情况下，出现某种类型的变量就说明有相应的组装过程中，比如创建一个二维Q_1单元，那么在局部会有一个4×4的矩阵，随后需要将这个矩阵添加到总矩阵中。那么便需要知道局部坐标下的点在总体坐标下的下标是多少，实现的方式如下：

```
cell->get_dof_indices (local_dof_indices);
```

其中，下标`local_dof_indices`声明方式如下：

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

示例程序

结果

拓展

3.4.Step-4

相关链接：[deal.II step-4 tutorial program]

内容说明

示例程序

结果

拓展