**面向对象的有限元软件设计的若干概念**

**1.Design and evolution of deal.II**

**1.1 设计原则**

* 灵活性：扩展性要好，可使用不同的有限元、不同的线性求解器
* 高水平的接口：为了使用不同的有限元，需要快速、复杂的数据结构。这些数据结构对用户来说是隐藏的，所以用户不用关心自由度控制等细节问题。
* 经济性：有限元算起来一般比较慢，所以程序设计要考虑经济性

还有两个常常不被学术界重视的原则是：

* 安全性
* 文档

选择C++的原因：

* 可获得性
* 标准化
* 数据封装
* 灵活强大的数据类型
* 速度
* 代码内文档

**1.2 编程模式**

**iterators and accessors：**

抽象地讲，triangulation可以看作是一个装有点、线、四边形等的容器，访问其中的元素就好比访问一个list的元素一样。但实际上，上述任何一个对象（点或线等）的数据组成都分布在一些不同的数组和其他容器里。deal.II中iterator实际上不指向任何数据。解引用后它们返回一个称为accessor的对象。这个accessor对象本身不含任何数据，它只是有一些数，用于鉴别它所代表的线或四边形；它是一些函数的集合，这些函数知道怎么获得并操作相关的数据。

在triangulation中cell\_iterator是这样定义的：

**typedef TriaIterator<CellAccessor<dim,spacedim>> Triangulation<dim, spacedim>::cell\_iterator**

而TriaIterator的->操作符是这样定义的：

Accessor \* TriaRawIterator<Accessor>::operator -> ()

{

return &(this->operator\* ());

}

**所以cell\_iterator->set\_flags()就等同于(\*cell).set\_flags()，即CellAccessor.set\_flags()**

即：**TriaIterator就类似一个指针，当解引用的时候，它返回一个CellAccessor类型的对象。**

例如一个典型的为triangulation中的四边形设置标记的函数（accessor的成员函数）是这样的：

void QuadAccessor::set\_user\_flag（）const

{

tria->levels[level]->quads.user\_flags[index] = true;

}

level 和index表示这个accessor对象代表的四边形在有层次结构的triangulation中的地址。

这样就可以实现把底层的数据以复杂的数据结构存放，但又集中在accessor中使用户访问，从而使得顶层用户的操作非常方便（同时保证高效复杂的数据结构）。deal.II库只有很少的部分能获得关于这些数据结构的信息，其他部分，包括所有用户程序，都只使用iterator和accessor来访问这些数据。

**Logical addressing of objects**

一个二维的triangulation可以被看成是由vertices，lines，quadrilateral等组成的；也可以看成是由cells，faces等组成的（无论维数）。在deal.II中采取后面这种视角，称为对偶拓扑dual topology。矩阵的组装、误差估计都是在cell和face上完成的。

一个标记所有cells的循环为：

Triangulation<dim> triangulation;

... //triangulate a domain

Triangulation<dim>::cell\_iterator cell;

for(cell = triangulation.begin(); cell!=triangulation.end(); ++cell)

{

cell->set\_user\_flag();

};

请记住，如果cell解引用的话，得到的是一个accessor，它有上述成员函数来进行想要的操作。

在deal.II中，第一种视角一般用于库的内部操作；第二种视角是提供给用户进行编程的。

**Dimension independent programming**

以这种方式编程，只有当编译器接收到了实在的数据类型后，程序才算形成了。比如用接收的quadrilateral类型来具化了代码中的cell。用C++的模板技术，用逻辑上的数据类型如cells或faces，可以写出不依赖于维数的代码。例如，在上面的代码中，如果dim=2，则标记的对象是quadrilateral；如果dim=3，则对象是cells。实现这种数据类型变化的代码大概是这样的：

class TriaDimensionInfo<2>

{

typedef quad\_iterator cell\_iterator;

typedef line\_iterator face\_iterator;

//...

};

class TriaDimensionInfo<3>

{

typedef hex\_iterator cell\_iterator;

typedef quad\_iterator face\_iterator;

//...

};

template <int dim>

class Triangulation : public TriaDimensionInfo<dim>

{

typedef TriaDimensionInfo<dim>::cell\_iterator cell\_iterator;

typedef TriaDimensionInfo<dim>::face\_iterator face\_iterator;

//...

};

**1.3 History**

下面介绍一下deal.II发展历史中的里程碑事件：

1991-1992：Guido Kanschat和Franz-Theo Suttmeier为了他们的博士论文，开发了最初的有限元代码，最早是PASCAL，后来改用C++语言。

1993之后：DEAL开始

1995-1996：Roland Becker加入，实现了多块网格方法，开发了基于dual solution的后验误差估计。用在了这三个程序员的论文中[4][9][16]，及论文[5]中。直到这时候，代码已经很复杂了，而且没有任何文档。

1997年末：Wolfgang Bangerth开始着手他的论文，他认为着手对这个库进行再设计是及时的。原先只有Q1有限元空间，需要增加；误差估计需要更灵活的数据结构；混合三角/四边形网格的策略被舍弃。从头开始重写deal.II。库的核心——网格和自由度控制——完全是他一个人完成的

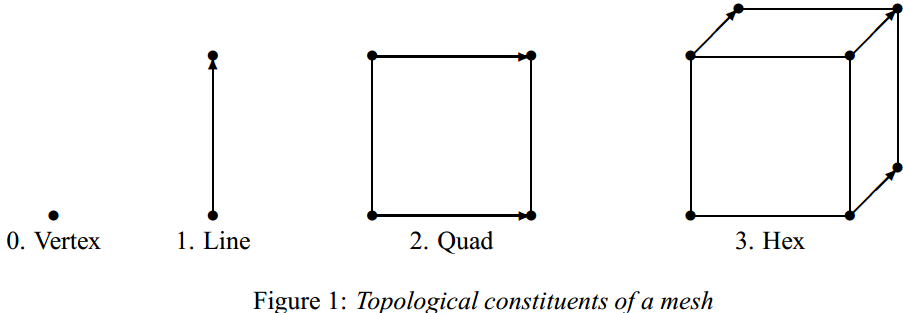
从1998开始：增加了许多工具。

目前，deal.II主要由Wolfgang Bangerth和Guido Kanschat维护。

**2. Grid handling**

因为这个库的目标包括易于编程且相对较快的自适应网格，有层次结构的网格数据结构是唯一选择。其内在思想分为两方面：1）分层的网格描述：使编程更容易 2）分层网格细化：便于实现快速的网格细化算法及多块网格求解等。下面分别介绍：

**2.1 hierarchical cell representation**



上面的每个对象都可以由更低级的对象组成。Quad对象只需存储指针指向四条边，而每条边又存储指针指向两个点。

相比于只存储点坐标的数据结构，这种分层的等级存放方式只多费了很少的内存，但使得计算jump terms，处理悬挂网格等操作变得容易多了。

**2.2** **hierarchical grid refinement**

目前一般有两种自适应细化的方式：

* 非结构网格：在计算了细化指示器之后，完全脱离先前的网格创建一个独立的新网格。可根据指示器，往domain上布散点来实现。例如，使用点云的Voronoi cells来生成新网格。注意，这种方法计算消耗比较大。
* 结构网格：在已有的triangulation上细化那些指示器比较大的，生成有层级结构的网格。这种方式在[15]中被考虑。这种方法存在的问题是必须考虑不相容网格的情况，即相邻网格一边细化了，另一边没细化。对于四边形网格，会出现悬挂点。如果用连续有限元，就必须施加连续性约束条件。

deal.II选择第二种方式。

译者注：参考文献[9]给出了deal.II的前身deal的一些设计概念，下面进行介绍，作为深入理解deal.II的补充材料。

**1. Grid Handling**

triangulation类：

class Triangulation

{

void read(File);

void refine(int levels);

void adaptive\_refine(double tolerence);

Cell\* first\_cell();

Cell\* next\_cell();

Vertex\* first\_vertex();

Vertex\* next\_vertex();

};

triangulation在2D由四边形构成；在3D由六面体构成。我们把这些几何对象抽象成cell。自适应细化需要的cell信息最早在[34]中以非常抽象的方式提出。

Cell类：

class Cell //**这里的Cell在deal.II中被改进成了CellAccessor的概念**

{

int number\_of\_vertices();

int number\_of\_neighbors();

Vertex\* vertex(int nr);

Cell\* neighbor(int nr);

Cell\* father();

Cell\* child(int nr);

int level();

int index();

void refine();

void coarse();

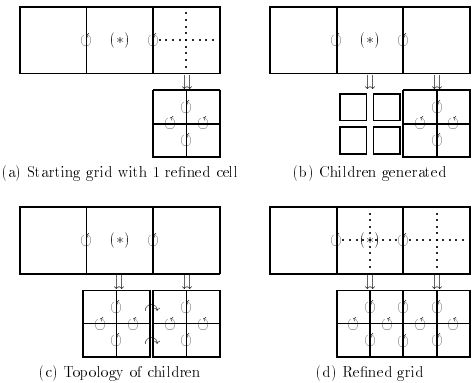
double refinement\_criterion();

};

Triangulation的函数refine()遍历全部cells并令每个cell自身细化。一个正确的triangulation必须有不矛盾的顶点，邻居和父子信息的值，而Cell的refine和coarse函数则保证了这些数据的守恒性。注意这里Cell只是一种抽象，而拓扑信息必须是在更实在的对象上指定的。所以Cell的的refine和coarse函数只是抽象的接口，它们的真正实现是在派生的类比如quadrilateral cell上进行的（在这些更具体的类上我们知道有几个邻居）。

提供了获取顶点，邻居，父子信息的函数接口，它们的实现可以是灵活多样的。比如函数Cell\* child()，可以有两种方式来返回对应的child的指针。它可以存在cell中，内存耗费大点但访问速度快；也可以从存在triangulation的list of cells中重构出来。

我们确保对每个函数而言，所有它操作的对象，在函数返回调用之后，必须处于一种容许的状态中。以下图网格细化过程为例：



以一个局部细化的有效网格开始（a）。cells之间存在联系（拓扑关系），相互性邻居关系用C:\Users\zeng\AppData\Local\Temp\1510749899(1).png表示，”is child of”关系用C:\Users\zeng\AppData\Local\Temp\1510749924(1).png表示。在中间（\*）cell划分为四个子网格之后，拓扑关系就破坏了（b）。为子网格设置好拓扑关系后，我们操作的该cell恢复为有效状态，但这种操作破坏了triangulation的结构：对(\*)的子网格而言，它们关于它们的邻居只有单向的邻居关系C:\Users\zeng\AppData\Local\Temp\1510751098(1).png。最后，邻居网格的拓扑信息也被更新了，最终得到的拓扑关系是有效的（d）。这个过程可使得我们在处理任何一个cell之后停止细化。

**参考文献**

[4] R. Becker. An Adaptive Finite Element Method for the Incompressible Navier-Stokes Equations on Time-dependent Domains. Dissertation, Universitat ¨ Heidelberg, 1995.

[5] R. Becker and R. Rannacher. Weighted a posteriori error control in FE methods. In ENUMATH 95, Paris, September 1995. in [6].

[9] G. Kanschat. Parallel and Adaptive Galerkin Methods for Radiative Transfer Problems.

Dissertation, Universitat ¨ Heidelberg, 1996

[16] F.-T. Suttmeier. Adaptive Finite Element Approximation of Problems in Elasto-Plasticity

Theory. Dissertation, Universitat ¨ Heidelberg, 1996.

[34] W. C. Rheinboldt and C. K. Mesztenyi. On a data structure for adaptive finite element mesh refinements. ACM Trans. Math. Software, 6:166-187, 1980