Hp有限元方法在80年代初由Babuska和Guo提出，作为以下两种方法的替代和深化：1.网格细化（减小网格参数h），2.增加基函数的多项式次数p。 这种方法基于如下研究观察：对于足够光滑的解而言，增加基函数阶数能减小逼近误差。但另一方面，众所周知即便对于性质良好（存在性、唯一性）的椭圆型问题，在临近边界、拐角或系数间断的地区，也无法保证足够高的光滑性；因而，在这些地区仅仅提高多项式次数是无法改善近似结果的，而只能靠细化网格。这两种策略导出了所谓的hp方法，即在解的光滑区，用于逼近的有限元空间自适应地提高其多项式次数；而在解的非光滑区，则自动地减小网格尺寸。在hp方法的第一篇文献中，就已经可以看到，该方法可使误差随自由度减小的关系呈现出指数型衰减而不仅仅是某个负阶数衰减。

为了实现这种方法，我们需要在常规的有限元程序上增加一些东西，特别是要基于我们在step-6之前介绍过的内容。特别地，我们会介绍如下方面：

* 我们要在不同的网格上使用不同的有限元函数，而不是像之前那样，在所有的网格上都采用相同的有限元。
* 在某个网格上，要根据它上面所使用的有限元来分配自由度。对连续有限元方法而言，现在不但要在悬挂网格点处施加约束，即便是一般的相邻网格，也需要施加约束了。
* 我们需要能把网格和界面上的贡献组装到全局的矩阵和右端项。
* 在每一步求解线性系统之后，我们需要分析结果。尤其是希望能计算出某种误差指示器，让我们决定在给定的网格上是该提高次数还是该细化网格。

Deal.II已经能支持上述的大多数功能，我们只需要提供一个程序逻辑，而不需要全面地知道程序是如何实现的。

多数hp功能都在命名空间hp下提供。此外，很多DoFTools和VectorTools命名空间下的函数也接受hp对象作参数。

现在有很多有限元的函数包都实现了hp方法（参见文献hp paper的引用）。但是，大多数的包都只限于：1、二维情况 2、DG方法。尤其是第二条在很大程度上简化了hp方法的代码实现，因为DG方法不要求网格界面间的连续性，因而不需要特殊处理。但是，deal.II实现了最一般的情形，即：同时允许连续或非连续的有限元，且同时适用于一维、二维或三维情况，并自动处理由此而带来的复杂性。尤其是，它能很好地在不同阶次的有限元网格交界面上处理好约束条件。相关的很多算法和数据结构都详细地在hp paper上给出了。

我们希望基于这种一般性的程序实现，能够帮助大家更好地挖掘hp方法的潜力。

**有限元集**

对hp方法而言，不同的网格能选择性地使用不同的有限元函数。为此，deal.II引入了所谓的有限元集（finite element collection），在类 hp::FECollection中实现。实际上有点类似于std::vector<FiniteElement>，但增加了一些修饰且具有更适用的内存管理方法。我们也会使用类似的积分函数集（quadrature collections），以及——尽管在这里不会用到——映射集（mapping collections）。在hp Collections综述中对所有这些类进行了表述。

在这个教程中，使用2-7阶连续型拉格朗日元（2D）或2-5阶（3D）。用到的有限元集可按如下方法创建：

hp::FECollection<dim> fe\_collection;

for (unsigned int degree=2; degree<=max\_degree; ++degree)

fe\_collection.push\_back (FE\_Q<dim>(degree));

**hp::DoFHandler类，联系起网格、有限元和约束**

从step-2起我们可以看到，DoFHandler类是通过在每个顶点、面、边、网格上分配正确数目的自由度，从而在网格（用Triangulation对象描述）和有限元之间建立起联系。在这里情况要复杂一些，因为需要在不同的网格上使用不同的有限元。因而我们需要两个东西：1.一个能处理这种情况的DoFHandler变种， 2、某种方法：能告诉DoFHandler在某个网格上该使用哪个有限元。

第一条已经在hp::DoFHandler类中实现了，与之前把它与单一的网格及单一有限元对象联结不同，在这里DoFHandler对象是与单一的网格和一个有限元集联系的。第二条是通过一个在hp::DoFHandler的网格上的循环来实现的。我们在每个网格上设定有限元编号，从而指定该使用哪个有限元对象。我们称有限元集内的该编号为网格的激活有限元索引（cell’s active FE index），表示在该网格上，该有限元集内此编号对应的有限元对象是激活状态的，而其他有限元对象是非激活状态。大致表示为：

hp::DoFHandler<dim> dof\_handler (triangulation);

for (typename hp::DoFHandler<dim>::active\_cell\_iterator cell = dof\_handler.begin\_active();

cell != dof\_handler.end(); ++cell)

cell->set\_active\_fe\_index (...);

dof\_handler.distribute\_dofs (fe\_collection);

set\_active\_fe\_index()函数的省略号...表示我们还需要有某种策略，用于决定在哪个网格上该使用哪种有限元。这里可看出，这段程序的首行和尾行和非hp情形下的程序是很类似的。

此外，除了在悬挂网格点处施加约束外，对于使用了不同有限元的相邻网格，现在也需要施加约束了。当方法是很类似的，代码几乎一样：

ConstraintMatrix constraints;

DoFTools::make\_hanging\_node\_constraints (dof\_handler,

constraints);

也就是说，[DoFTools::make\_hanging\_node\_constraints](http://www.dealii.org/8.5.0/doxygen/deal.II/group__constraints.html" \l "ga3eaa31a679484e80c193e74e8a967dc8)不但可以处理悬挂网格点约束，也能处理hp型约束。

**基于hp对象组装矩阵和向量**

在hp方法下建立矩阵和向量和非hp的情况是一样的。但组装需要一些思考和技巧。

主要的思路是一样的：我们必须在网格上循环，把局部的贡献组装成为整体。就像在step-3中提到的细节那样，deal.II有一个类FEValues，能够把有限元、映射、以及积分公式融合到一起用于估算真实网格积分点处的基函数值、导数等信息。每当转移到一个新的网格上，需要重新初始化对应的FEValues对象以重新计算需要的信息。

但在hp方法中，在使用了更高阶有限元的网格上需要使用更高阶的积分公式和更高阶的映射。

为了实现这些，deal.II有一个类hp::FEValues。所不同的是，它不是使用单一的有限元和积分公式、映射，它使用了这些东西的集合。用法和常规的FEValues非常类似，即：在网格上的循环是这样的：

hp::FEValues<dim> hp\_fe\_values (mapping\_collection,

fe\_collection,

quadrature\_collection,

update\_values | update\_gradients | update\_q\_points | update\_JxW\_values);

typename hp::DoFHandler<dim>::active\_cell\_iterator

cell = dof\_handler.begin\_active(),

endc = dof\_handler.end();

for (; cell!=endc; ++cell)

{

hp\_fe\_values.reinit (cell,

cell->active\_fe\_index(),

cell->active\_fe\_index(),

cell->active\_fe\_index());

const FEValues<dim> &fe\_values = hp\_fe\_values.get\_present\_fe\_values ();

... // assemble local contributions and copy them into global object

}

在这个教程中，只使用Q1映射，因而hp::FEValues的参数mapping\_collection可以省略。在循环体中，我们首先初始化当前网格的hp::FEValues对象。其第2,3,4个参数表示我们想要在这个网格上使用的积分公式（quadrature）、映射（mapping）、有限元（finite element）对象在它们对应的集合中的索引（编号）。三个参数按这种顺序安排的原因是很多时候当转移到一个新的网格，我们会想要选用不同的积分或映射对象，但往往不经常改变使用的有限元。而它作为最后一个参数就可以省略掉。

这个reinit函数调用做了如下操作：在当前网格下，hp::FEValues对象检查它是否已经针对当前指定的这样一个有限元、积分公式、映射组合方式分配了一个非hp型的FEValues对象（即一般的FEValues对象），如果没有的话，就分配一个满足这种组合的FEValues对象。然后为当前网格重新初始化这个对象以得到需要的值。之后再调用函数hp\_fe\_values.get\_present\_fe\_values()返回这个FEValues对象的引用，用于组装局部贡献。

**一个用于hp细化和光滑度估算的简单指示器**

自适应有限元方法的核心之一是后验性地检查数值解，并用一个指示器来表征哪些网格上误差较大，从而细化对应的网格。在很多其他教程中，我们使用了KellyErrorEstimator类来得到误差估计，当然在其他地方如step-14中我们也讨论了一些更复杂的策略。

在很多情况下，如果只是需要判断细化或不细化，那么问题其实不是困难。但在这里我们实现了hp型的细化，也就是说，我们在面对误差较大的网格时突然有了两种选择：要么是细化该网格，要么是提高对应的基函数多项式次数。如何来判断哪种方式更好，是在写本教材时我们需要解决的核心问题。

简单讲，目前而言在文献上针对这一问题还没有定论。虽然有很多或简单或复杂的策略，但没有一个像KellyErrorEstimator这样被普遍接受的优良的误差指示器。尽管绝大多数方法都默认这样的规则：在解的局部光滑区更适宜提高多项式次数，而在不那么光滑的区域则适合细化网格。**然而问题是：怎么定义局部光滑度，又如何判断何种程度的光滑性就足够适宜使用提高次数的方法了呢？**

在下面，我们提出了一个简单的局部光滑性的指示器。我们将在结果中看到，这个指示器是存在缺陷的，尤其是当考虑悬挂网格点的时候。因为我们并不把下面的方法当做一种完整的方法。而是把它作为一个可以继续研究的思路起点。

**思路**

我们的方法很简单：对于网格K上位于Sobolev空间Hs(K)中的一个函数u(x)，它必须满足条件：

C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\73%ZIKBDPV`Q)UB$$F}(1U4.png

假定网格K是非退化的，即：从单位参考网格到网格K的映射是足够光滑的（regular），上述条件等价于：

C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\1039MP$3LZ2(5)E5({%6%XA.png

C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\38M)5]4%F}Z5T~FN(_1{FEP.png表示u(x)映射回单位网格K^.的函数。然后，我们定义u^的傅里叶级数：

C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\MBS%J6$D8MEGDKPMA`}7CEB.png

其中**k**为傅里叶向量k=(kx,ky)（2D）或k=(kx,ky,kz)（3D），kx,ky,kz=0,2π,4π,…。系数U^**k**可通过指数基的L2正交性获得：

C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\[R2)TN(NG)9F$36(_8}WP68.png

即得到：

C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\~{U8IZAV3ZF@B~YPVR)$BL2.png

然后我们就能写出u^的Hs范数：

C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\9%[(97J(F~V9]9EGOH`6~3W.png

换句话说，如果该范数是有限的（即C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\FMY~XMJ`4Y2T7]X5$3GD)HG.png位于C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\M_H8I6G06W3M7%1_))%{MVG.png空间中），我们需要：

C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\@4~UM0{VM6ID%%KJ7R7}WCO.png

换言之，**如果想要更高的光滑度s，傅里叶系数需要更快地趋于0**。如果你好奇多出来的指数C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\)4]ZB9B%(PA7OGW2HJB$MQR.png是怎么来的：我们利用结论有：如果对任意ϵ>0，数列al =O(l−1−ϵ) ，则∑al<∞。问题是在这里我们不是对单一的变量求和，而是对隐藏在d维空间中的2π的整数倍求和（因为向量分量kx,ky,…）。正如我们在证明上述数列al收敛时用一个线上的全域积分来代替求和一样，我们也可以用一个d维空间里的积分来代替这里的d维求和。现在需要注意到，在|k|和|k|+d|k|距离之间，有最多|k|d−1常数个模态，正和我们把体积元dxdy转换到2πrdr采用了差不多的方式。因而，必须以C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\NGUV(JJPH}S90]Q7~VE$9AI.png的方式衰减的不再是C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\0%GF44]BEMUP6A~T19`ZSDP.png，而是C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\)J7_1~6H4J3@E0]KAD39Q[G.png。比较指数可以得出这个结论。

那么反过来：假设给定函数u^是未知光滑度的。让我们计算其系数C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\X0D}~YD1G2C}9[$FVLMW1G5.png，看看他们衰减得多快。如果他们以C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\L)`W[]3$M7)A46SGOMP[{MW.png的阶次衰减，则我们可以知道这个函数是位于C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\VJ_C{(8GQSRYQ02K[J}`F9G.png的。

**我们需要做什么**

所以为了估计网格K上的光滑度我们需要做什么呢？显然，第一步是计算我们的解的傅里叶级数。在这里我们简化为只计算级数的前几项，这样C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\9BIGSU(6)81[9ODYO_YIQ@D.png，截断到某个项N。顺便说一句，我们希望选择足够大的N，使得我们至少能捕捉到变化最大的基函数的变化。但另一方面，N又不能过大：显然，一个有限元函数，作为多项式，在任意网格上都是位于C∞空间的，因而系数必然在某一个点以指数形式衰减；因为我们想要估测的是这个多项式所逼近的函数的光滑性，而非这个多项式本身，所以必须选取一个合理的截断值N。但无论怎样，计算这个数列不是很困难：由条件：

C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\Y2I6VQM39A8)4PF4J(9N@QD.png

可看出我们可以这样计算系数C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\K`V[P_VD0WOU3VHD`M]B{TK.png：

C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\[LV@D9`QL1U11)M]S8@H(~G.png

这里ui是这个网格上第i个自由度的值。我们可以把它写成矩阵向量点积的形式：

C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\VMU)0W[{BY{IJI2XI%S}I7Y.png

这里：C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\~]X6L(FZIOLC{{0E$BL_OKM.png

给定了一定数量的基函数C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\F9136VOG~NWRNH]D13HV6T1.png和傅里叶模态N，这个矩阵很容易计算。从而计算系数C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\K`V[P_VD0WOU3VHD`M]B{TK.png是个很平凡的工作。为了简化工作，我们将使用FESeries::Fourier类。

下一个任务是必须估算这些系数随|**k**|衰减得有多快。当然，问题在于我们只有有限的系数项。换言之，我们只能做到把一个函数C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\FT6LZLAKCH3EC}H_SR%M826.png拟合到我们的数据点C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\US%R{KT3MJHILTFXKW$6CGL.png上。例如，采用最小二乘法来决定这里的α和μ：

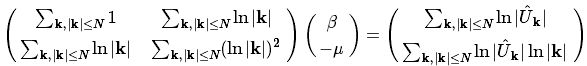
C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\M2N8C[{P7{[_RWLH`%QG772.png

然而，问题在于这样做将会导向一个非线性问题，是我们不愿意看到的。但是，如果我们只是把系数的对数拟合成函数C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\FT6LZLAKCH3EC}H_SR%M826.png的对数的话，就能把问题转换成较简单的情形。像这样：

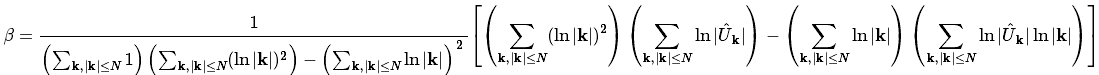
C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\3Y)VDX`(A[WTF87MUK3@I1O.png

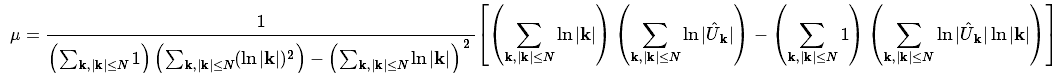
转换为问题： C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\B69YL5K1XVEO8ZT81WD})GL.png，这里β=lnα

最优条件是：C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\8J[S`0$Z`4)[23WMI1GCD06.png，是关于β、μ的线性问题。我们可以把这两个条件写成：



反解出：





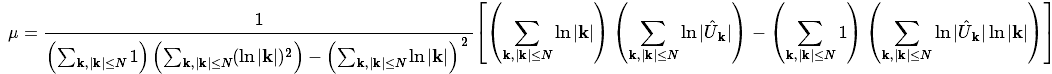
这其实就是线性回归而已，我们可以用FESeries::linear\_regression()来实现。虽然我们对β的值不感兴趣，但上述公式给我们提供了一种计算指数μ的方法，我们可以用来保证C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\`G~3RX(F3WP2H2LRQCPZ{PJ.png位于C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\`O$0IL9A5~%ZXED$GMFY2`T.png中，s=μ-d/2

**补偿各项异性**

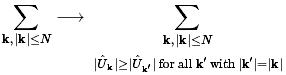
在上面的公式中，我们已经得到了傅里叶系数C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\GP%T7@@`OLQ6Q27KIC90G1R.png。因**k**是个向量，即便对于同样的绝对值|k|，我们也会得到一系列不同的傅里叶系数C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\]~X[FA(UMHIS[AZBT@C2%92.png，对应了不同方向上的傅里叶变换。如果考虑像C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\F(07K}TMVVZAGG_R1C4TJHB.png这样的函数，则我们在x方向上会得到很多大的傅里叶系数值，因为函数在这个方向上不光滑，但在y方向上系数会快速衰减因为在这个方向上函数是光滑的。现在问题来了：如果我们简单地把多项式衰减C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\BE7$8O@]{MT{~G6Y$A7BJ[W.png拟合到所有的傅里叶系数上去，我们会拟合的其实是所有方向上的平均光滑度。这是我们想要的吗？或者说，如果对于所有绝对值相同的**k**仅仅考虑最大的系数C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\]~X[FA(UMHIS[AZBT@C2%92.png是否更好？本质上是由光滑度最差的方向来决定解的光滑度。

这个问题是见仁见智的。如果deal.II能够使用各向异性的有限元函数会很吸引人（也就是说，有限元多项式在不同空间方向上有不同的次数）。但目前而言，deal.II还不具备这种功能。

但无论哪种方式，因为我们仅仅有各向同性的有限元类，我们采用这种看法，即：为了减轻数值计算的负担，我们应该把多项式裁剪到最低的光滑性。那么，我们不会采用上面的公式来计算μ：



我们不得不稍微修改所有的求和项：针对相同的|**k**|，这里的求和仅仅考虑其中比较大的傅里叶系数项，而不是把所有傅里叶的模态都加起来。也就是说，上面所有的求和会用下述求和来替代：



这是我们在程序里会采用的形式。

**关于网格尺寸的问题**

可能有人会问，我们只在参考网格而不是实际网格上计算傅里叶变换究竟存不存在没有问题。毕竟，我们在转换过程中把解乘了一个系数1/h，从而把傅里叶频率改变了h倍。尤其是在相邻网格尺寸相差两倍的情况下更要关注这个问题。我们在接下来的结果还会看到一点原因是，估算的解的光滑性多多少少应该表现为连续函数，但却由于网格尺寸的突变呈现出值的跳跃。因而看起来似乎考虑补偿这种变换带来的问题确实是有道理的。

然而答案是“no”，不需要。在上面的过程中，我们试图找到系数β、μ能使得下述项的平方和为最小：

C:\Users\zeng\AppData\Roaming\Tencent\Users\154462188\QQ\WinTemp\RichOle\J7P{C{E{AR$UQX{B092%9WJ.png

**Hp离散的线性系统的复杂性**

创建稀疏模式

hp方法中的一个问题是基函数多项式的高次性质和大量自由度的约束条件导致了得到的矩阵在某些行有很多非零项。同时，由于在某些区域使用了低阶次的多项式因而得到的矩阵在有些行有很少的非零项。所以，为这些矩阵分配稀疏模式是一个挑战。

Deal.II中大多数程序都使用DoFHandler::make\_sparsity\_pattern函数来分配矩阵的稀疏模式，然后再使用ConstraintMatrix::condense来添加一些必要的项来处理自由度的约束条件。之后使用SparsityPattern::compress来压缩稀疏模式。在step-6中介绍了这个方法并且在大多数教程中使用了这种方法。这种方法需要我们先预估每一行中非零项个数的上限，例如可采用DoFHandler::max\_couplings\_between\_dofs函数来得到。

然后不幸的是，DoFHandler::max\_couplings\_between\_dofs在三维情形和高阶有限元情形下无法有效地预估上限。如果在这些情况下使用的话，它会分配过多的内存，而这些内存几乎全部都会在调用SparsityPattern::compress的时候释放。造成这种低效率的原因是即便绝大多数行本可以是更短的，但DoFHandler::max\_couplings\_between\_dofs只能产生一个数来表示每行的元素个数上限。这种低效是如此严重，以至于为SparsityPattern初始分配的内存超出了实际需要的10倍乃至更多，可能造成内存耗尽（尽管实际上并不需要这么多内存）。

我们已经在step-11和step-18中讨论过了针对这一问题的解决方法。它使用了一种中间对象：DynamicSparsityPattern。这个类使用了一种不同的内存格式用于在矩阵每行的非零项上限不确定的情况下创建稀疏模式，但不适宜使用因为稀疏模式本质上是基于稀疏矩阵的。？？在建立了这个中间对象后，它被复制到一个真正的稀疏模式对象，这个过程是很经济且不会造成内存的多分配。与此有关的典型代码展示在DynamicSparsityPattern类的文档中。这种方式会比直接建立稀疏模式慢一下，但使用的内存是实际需要的。

**消去受约束的自由度**

**带注释的程序：**

**头文件**

这几个头文件在之前案例已经讲过了，这里不赘述。

#include <deal.II/base/quadrature\_lib.h>

#include <deal.II/base/function.h>

#include <deal.II/base/logstream.h>

#include <deal.II/base/utilities.h>

#include <deal.II/lac/dynamic\_sparsity\_pattern.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/solver\_cg.h>

#include <deal.II/lac/precondition.h>

#include <deal.II/lac/constraint\_matrix.h>

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

#include <deal.II/grid/grid\_generator.h>

#include <deal.II/grid/tria\_accessor.h>

#include <deal.II/grid/tria\_iterator.h>

#include <deal.II/grid/grid\_refinement.h>

#include <deal.II/dofs/dof\_accessor.h>

#include <deal.II/dofs/dof\_tools.h>

#include <deal.II/fe/fe\_q.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>

下面这些是新的头文件。前两个用来代替DoFHandler和FEValues。最后一个提供单位网格上的傅里叶变换。

#include <deal.II/hp/dof\_handler.h>

#include <deal.II/hp/fe\_values.h>

#include <deal.II/fe/fe\_series.h>

最后还需要标准C++头文件。第三个用于复数相关的计算。

#include <fstream>

#include <iostream>

#include <complex>

namespace Step27

{

using namespace dealii;

**主类**

主要用到的类和之前的很像，比如step-6中用到的。区别在于这里把refine\_grid和output\_results函数融合到了一起，因为我们还想要输出一些光滑指示器相关的数据。

至于成员变量，我们使用和step-6一样的结构。区别在于有限元、积分函数、面积分对象不再是单个的，而是相应的集合。max\_degree用来代表将要用到的有限元的最高次数。

template <int dim>

class LaplaceProblem

{

public:

LaplaceProblem ();

~LaplaceProblem ();

void run ();

private:

void setup\_system ();

void assemble\_system ();

void solve ();

void create\_coarse\_grid ();

void estimate\_smoothness (Vector<float> &smoothness\_indicators);

void postprocess (const unsigned int cycle);

std::pair<bool,unsigned int> predicate(const TableIndices<dim> &indices);

Triangulation<dim> triangulation;

hp::DoFHandler<dim> dof\_handler;

hp::FECollection<dim> fe\_collection;

hp::QCollection<dim> quadrature\_collection;

hp::QCollection<dim-1> face\_quadrature\_collection;

hp::QCollection<dim> fourier\_q\_collection;

std\_cxx11::shared\_ptr<FESeries::Fourier<dim> > fourier;

std::vector<double> ln\_k;

Table<dim,std::complex<double> > fourier\_coefficients;

ConstraintMatrix constraints;

SparsityPattern sparsity\_pattern;

SparseMatrix<double> system\_matrix;

Vector<double> solution;

Vector<double> system\_rhs;

const unsigned int max\_degree;

};

**方程数据**

接下来，我们定义右端项函数。它对于1d是x+1，对于2d是(x+1)(y+1)，诸如此类。

template <int dim>

class RightHandSide : public Function<dim>

{

public:

RightHandSide () : Function<dim> () {}

virtual double value (const Point<dim> &p, const unsigned int component) const;

};

template <int dim>

double

RightHandSide<dim>::value (const Point<dim> &p, const unsigned int / \*component\* /) const

{

double product = 1;

for (unsigned int d=0; d<dim; ++d)

product \*= (p[d]+1);

return product;

}

**主类的实现**

LaplaceProblem::LaplaceProblem

构造函数是很直观的。它把hp::DoFHandler对象和triangulation联系起来，然后设置多项式最高次数为7（in 1d and 2d）或5（in 3d and higher）。

然后，我们填充有限元集、网格和面积分公式对象。我们以二次元开始，并选取合适的积分公式对象。

最后，初始化FESeries::Fourier对象，它将被用来计算傅里叶级数展开的系数。为了计算参考网格上的积分，除了hp::FECollection，我们还需要提供积分公式hp::QCollection。

为了分配fourier\_coefficients表的大小，我们使用下面的辅助函数。

template <int dim,typename T>

void resize(Table<dim,T> &coeff, const unsigned int N)

{

TableIndices<dim> size;

for (unsigned int d=0; d<dim; d++)

size[d] = N;

coeff.reinit(size);

}

template <int dim>

LaplaceProblem<dim>::LaplaceProblem ()

:

dof\_handler (triangulation),

max\_degree (dim <= 2 ? 7 : 5)

{

for (unsigned int degree=2; degree<=max\_degree; ++degree)

{

fe\_collection.push\_back (FE\_Q<dim>(degree));

quadrature\_collection.push\_back (QGauss<dim>(degree+1));

face\_quadrature\_collection.push\_back (QGauss<dim-1>(degree+1));

}

就像在介绍中描述的那样，我们定义傅里叶向量**k**用来计算傅里叶系数。在二维情况下，我们要计算对应于C:\Users\zeng\AppData\Local\Temp\1510131467(1).png的系数。我们把**k**限定在，也就是C:\Users\zeng\AppData\Local\Temp\1510131690(1).png。N表示使用的有限元多项式的最高次数。FESeries::Fourier类的构造函数中的第一个参数N定义了1D中的系数个数为：Ndim。每个有限元（FiniteElement）的变换矩阵只在第一次hp-自适应细化的时候计算一次。这些工作只需要在参考网格上进行，不需要映射。因而可以预先计算出来。为了计算某个网格上的特定局部自由度的系数，只需要简单的矩阵-向量相乘。

const unsigned int N = max\_degree;

我们需要组装矩阵C:\Users\zeng\AppData\Local\Temp\1510133524(1).png，用它来进行傅里叶展开。必须对每个用到的有限元函数都计算对应的C:\Users\zeng\AppData\Local\Temp\1510133524(1).png。这里面需要用到数值积分规则，在这里我们使用与有限元一样的积分公式，即：循环两点高斯积分，直到C:\Users\zeng\AppData\Local\Temp\1510133825(1).png的最高指数。

QGauss<1> base\_quadrature (2);

QIterated<dim> quadrature (base\_quadrature, N);

for (unsigned int i = 0; i < fe\_collection.size(); i++)

fourier\_q\_collection.push\_back(quadrature);

现在已经准备好创建FESeries::Fourier对象了

fourier = std\_cxx11::make\_shared<FESeries::Fourier<dim> >(N, fe\_collection, fourier\_q\_collection);

我们需要根据模态数N来resize傅里叶系数矩阵

resize(fourier\_coefficients,N);

}

**LaplaceProblem::~LaplaceProblem**

析构函数和step-6中的一样

template <int dim>

LaplaceProblem<dim>::~LaplaceProblem ()

{

dof\_handler.clear ();

}

**LaplaceProblem::setup\_system**

这个函数是step-6的翻版。内部使用的算法有点区别，因为这里使用的dof\_handler是hp类型的对象。

template <int dim>

void LaplaceProblem<dim>::setup\_system ()

{

dof\_handler.distribute\_dofs (fe\_collection);

solution.reinit (dof\_handler.n\_dofs());

system\_rhs.reinit (dof\_handler.n\_dofs());

constraints.clear ();

DoFTools::make\_hanging\_node\_constraints (dof\_handler,

constraints);

VectorTools::interpolate\_boundary\_values (dof\_handler,

0,

ZeroFunction<dim>(),

constraints);

constraints.close ();

DynamicSparsityPattern dsp (dof\_handler.n\_dofs(),

dof\_handler.n\_dofs());

DoFTools::make\_sparsity\_pattern (dof\_handler, dsp, constraints, false);

sparsity\_pattern.copy\_from (dsp);

system\_matrix.reinit (sparsity\_pattern);

}

**LaplaceProblem::assemble\_system**

这个函数把每个网格的局部贡献组装成全局矩阵和右端项。不同点在于我们是使用一个FEValues对象的集合，而且我们必须在把局部贡献复制到全局对象前就已经把受约束的自由度消去了。

另一个稍微复杂的地方在于，因为我们在不同网格上使用不同次数的多项式近似，所以不同网格的局部贡献不是相同大小的矩阵和向量。因而在每次网格循环前需要resize到正确的大小。我们在设计这些类的时候，使其在减小当前矩阵或向量的尺寸不需要释放当前分配的内存，因而resizing的过程只会在前几个循环内要求重新开辟内存。一旦我们发现某个网格上的有限元已经是最高次数了，不会再发生重新开辟内存的情况，因为所有接下来的reinit操作仅仅会设置大小使其匹配当前的内存。这是很重要的特性，因为开辟内存是非常耗时的。所以不能说每次访问一个新的网格就开辟一次新的内存。

template <int dim>

void LaplaceProblem<dim>::assemble\_system ()

{

hp::FEValues<dim> hp\_fe\_values (fe\_collection,

quadrature\_collection,

update\_values | update\_gradients |

update\_quadrature\_points | update\_JxW\_values);

const RightHandSide<dim> rhs\_function;

FullMatrix<double> cell\_matrix;

Vector<double> cell\_rhs;

std::vector<types::global\_dof\_index> local\_dof\_indices;

typename hp::DoFHandler<dim>::active\_cell\_iterator

cell = dof\_handler.begin\_active(),

endc = dof\_handler.end();

for (; cell!=endc; ++cell)

{

const unsigned int dofs\_per\_cell = cell->get\_fe().dofs\_per\_cell;

cell\_matrix.reinit (dofs\_per\_cell, dofs\_per\_cell);

cell\_matrix = 0;

cell\_rhs.reinit (dofs\_per\_cell);

cell\_rhs = 0;

hp\_fe\_values.reinit (cell);

const FEValues<dim> &fe\_values = hp\_fe\_values.get\_present\_fe\_values ();

std::vector<double> rhs\_values (fe\_values.n\_quadrature\_points);

rhs\_function.value\_list (fe\_values.get\_quadrature\_points(),

rhs\_values);

for (unsigned int q\_point=0;

q\_point<fe\_values.n\_quadrature\_points;

++q\_point)

for (unsigned int i=0; i<dofs\_per\_cell; ++i)

{

for (unsigned int j=0; j<dofs\_per\_cell; ++j)

cell\_matrix(i,j) += (fe\_values.shape\_grad(i,q\_point) \*

fe\_values.shape\_grad(j,q\_point) \*

fe\_values.JxW(q\_point));

cell\_rhs(i) += (fe\_values.shape\_value(i,q\_point) \*

rhs\_values[q\_point] \*

fe\_values.JxW(q\_point));

}

local\_dof\_indices.resize (dofs\_per\_cell);

cell->get\_dof\_indices (local\_dof\_indices);

constraints.distribute\_local\_to\_global (cell\_matrix, cell\_rhs,

local\_dof\_indices,

system\_matrix, system\_rhs);

}

}

**LaplaceProblem::solve**

这个函数和之前的例子完全一样。我们仅仅是尝试减小初始的residual一定的因子（等于右端项的l2范数）（try to reduce the initial residual by a certain factor）

template <int dim>

void LaplaceProblem<dim>::solve ()

{

SolverControl solver\_control (system\_rhs.size(),

1e-8\*system\_rhs.l2\_norm());

SolverCG<> cg (solver\_control);

PreconditionSSOR<> preconditioner;

preconditioner.initialize(system\_matrix, 1.2);

cg.solve (system\_matrix, solution, system\_rhs,

preconditioner);

constraints.distribute (solution);

}

**LaplaceProblem::postprocess**

在求解了线性系统之后，我们想要对解进行后处理。这里，我们只估算误差，估算局部光滑度，然后输出图像，最后根据指示器进行h型和p型的网格细化。我们在一个后处理函数里进行这些，因为我们计算误差和光滑度不仅仅用于网格细化，还要把它们以图像的方式输出来。

template <int dim>

void LaplaceProblem<dim>::postprocess (const unsigned int cycle)

{

Let us start with computing estimated error and smoothness indicators, which each are one number for each active cell of our triangulation. For the error indicator, we use the KellyErrorEstimator class as always. Estimating the smoothness is done in the respective function of this class; that function is discussed further down below:

Vector<float> estimated\_error\_per\_cell (triangulation.n\_active\_cells());

KellyErrorEstimator<dim>::estimate (dof\_handler,

face\_quadrature\_collection,

typename FunctionMap<dim>::type(),

solution,

estimated\_error\_per\_cell);

Vector<float> smoothness\_indicators (triangulation.n\_active\_cells());

estimate\_smoothness (smoothness\_indicators);

然后我们想要生成图像输出文件。除了上面的两个估算值，我们还希望结果能显示出用在每个网格上的有限元阶次。

为了实现这个目的，需要用cell->active\_fe\_index()函数把每个网格上的有限元函数标号提取出来。用其结果来询问有限元集，得到该标号对应的有限元的多项式次数。把结果放到一个向量里，向量元素个数等于网格数。DataOut类需要这个向量是小数型的，尽管标号是整数。所以：

{

Vector<float> fe\_degrees (triangulation.n\_active\_cells());

{

typename hp::DoFHandler<dim>::active\_cell\_iterator

cell = dof\_handler.begin\_active(),

endc = dof\_handler.end();

for (; cell!=endc; ++cell)

fe\_degrees(cell->active\_cell\_index())

= fe\_collection[cell->active\_fe\_index()].degree;

}

现在所有数据向量都得到了——解、误差估计、光滑度指示器、有限元函数次数。我们创建一个DataOut对象用于可视化输出。注意DataOut有第二个模板参数（默认为DoFHandler<dim>）用于表示用到的DoFHandler的类型。这里，我们必须使用hp::DoFHandler类。

DataOut<dim,hp::DoFHandler<dim> > data\_out;

data\_out.attach\_dof\_handler (dof\_handler);

data\_out.add\_data\_vector (solution, "solution");

data\_out.add\_data\_vector (estimated\_error\_per\_cell, "error");

data\_out.add\_data\_vector (smoothness\_indicators, "smoothness");

data\_out.add\_data\_vector (fe\_degrees, "fe\_degree");

data\_out.build\_patches ();

最后一步是决定输出文件的文件名，打开文件，把数据写进去（这里，使用vtk格式）：

const std::string filename = "solution-" +

Utilities::int\_to\_string (cycle, 2) +

".vtk";

std::ofstream output (filename.c\_str());

data\_out.write\_vtk (output);

}

这之后，我们真正进行网格细化，同时有h型和p型细化。首先，使用估算的误差来标记需要细化的网格，正如我们之前的教程中的那样：

{

GridRefinement::refine\_and\_coarsen\_fixed\_number (triangulation,

estimated\_error\_per\_cell,

0.3, 0.03);

然后需要分辨出被标记的网格是该用h型还是p型细化。我们在这里使用这样的策略：先检查被标记网格的光滑度指示器，设定一个光滑度阈值，对光滑度高于该阈值的网格进行p型细化。为了实现这点，我们需要先确定所有被标记网格的光滑度的极大值和极小值。先采用一个网格循环来找出所有网格中光滑度的极大值和极小值。然后我们再取极大极小值的平均为该阈值，高于它的网格就用p型细化：

float max\_smoothness = \*std::min\_element (smoothness\_indicators.begin(),

smoothness\_indicators.end()),

min\_smoothness = \*std::max\_element (smoothness\_indicators.begin(),

smoothness\_indicators.end());

{

typename hp::DoFHandler<dim>::active\_cell\_iterator

cell = dof\_handler.begin\_active(),

endc = dof\_handler.end();

for (; cell!=endc; ++cell)

if (cell->refine\_flag\_set())

{

max\_smoothness = std::max (max\_smoothness,

smoothness\_indicators(cell->active\_cell\_index()));

min\_smoothness = std::min (min\_smoothness,

smoothness\_indicators(cell->active\_cell\_index()));

}

}

const float threshold\_smoothness = (max\_smoothness + min\_smoothness) / 2;

有了这，我们可以回过头去，再次在所有网格上循环，对于那些1）标记了需要细化的2）光滑度高于阈值3）我们还有比它当前使用的有限元函数次数更高的有限元的网格，我们增加其多项式次数，并且移除它们上面的网格细化标记。对其他网格，细化标记保留不变：

{

typename hp::DoFHandler<dim>::active\_cell\_iterator

cell = dof\_handler.begin\_active(),

endc = dof\_handler.end();

for (; cell!=endc; ++cell)

if (cell->refine\_flag\_set()

&&

(smoothness\_indicators(cell->active\_cell\_index()) > threshold\_smoothness)

&&

(cell->active\_fe\_index()+1 < fe\_collection.size()))

{

cell->clear\_refine\_flag();

cell->set\_active\_fe\_index (cell->active\_fe\_index() + 1);

}

}

在这部分最后，我们细化网格。这个过程中，被细分的网格的子网格继承了它们的母网格的有限元标号：

triangulation.execute\_coarsening\_and\_refinement ();

}

}

**LaplaceProblem::create\_coarse\_grid**

下面的函数用于创建初始网格。它只适用于2d情况，如果想在别的情况下使用，必须进行相应修改。这个函数实际上是从step-14中借过来的，生成的网格在step-14中就用过了：

template <>

void LaplaceProblem<2>::create\_coarse\_grid ()

{

const unsigned int dim = 2;

static const Point<2> vertices\_1[]

= { Point<2> (-1., -1.),

Point<2> (-1./2, -1.),

Point<2> (0., -1.),

Point<2> (+1./2, -1.),

Point<2> (+1, -1.),

Point<2> (-1., -1./2.),

Point<2> (-1./2, -1./2.),

Point<2> (0., -1./2.),

Point<2> (+1./2, -1./2.),

Point<2> (+1, -1./2.),

Point<2> (-1., 0.),

Point<2> (-1./2, 0.),

Point<2> (+1./2, 0.),

Point<2> (+1, 0.),

Point<2> (-1., 1./2.),

Point<2> (-1./2, 1./2.),

Point<2> (0., 1./2.),

Point<2> (+1./2, 1./2.),

Point<2> (+1, 1./2.),

Point<2> (-1., 1.),

Point<2> (-1./2, 1.),

Point<2> (0., 1.),

Point<2> (+1./2, 1.),

Point<2> (+1, 1.)

};

const unsigned int

n\_vertices = sizeof(vertices\_1) / sizeof(vertices\_1[0]);

const std::vector<Point<dim> > vertices (&vertices\_1[0],

&vertices\_1[n\_vertices]);

static const int cell\_vertices[][GeometryInfo<dim>::vertices\_per\_cell]

= {{0, 1, 5, 6},

{1, 2, 6, 7},

{2, 3, 7, 8},

{3, 4, 8, 9},

{5, 6, 10, 11},

{8, 9, 12, 13},

{10, 11, 14, 15},

{12, 13, 17, 18},

{14, 15, 19, 20},

{15, 16, 20, 21},

{16, 17, 21, 22},

{17, 18, 22, 23}

};

const unsigned int

n\_cells = sizeof(cell\_vertices) / sizeof(cell\_vertices[0]);

std::vector<CellData<dim> > cells (n\_cells, CellData<dim>());

for (unsigned int i=0; i<n\_cells; ++i)

{

for (unsigned int j=0;

j<GeometryInfo<dim>::vertices\_per\_cell;

++j)

cells[i].vertices[j] = cell\_vertices[i][j];

cells[i].material\_id = 0;

}

triangulation.create\_triangulation (vertices,

cells,

SubCellData());

triangulation.refine\_global (3);

}

**LaplaceProblem::run**

这个函数实现了本程序的逻辑结构。参见step-6。

本质上，它包含了自适应过程：在第一个循环中创建稀疏的网格，然后建立线性系统，组装起来，求解，然后对解进行后处理（包括网格细化）；然后再从头开始。

template <int dim>

void LaplaceProblem<dim>::run ()

{

for (unsigned int cycle=0; cycle<6; ++cycle)

{

std::cout << "Cycle " << cycle << ':' << std::endl;

if (cycle == 0)

create\_coarse\_grid ();

setup\_system ();

std::cout << " Number of active cells: "

<< triangulation.n\_active\_cells()

<< std::endl

<< " Number of degrees of freedom: "

<< dof\_handler.n\_dofs()

<< std::endl

<< " Number of constraints : "

<< constraints.n\_constraints()

<< std::endl;

assemble\_system ();

solve ();

postprocess (cycle);

}

}

**LaplaceProblem::estimate\_smoothness**

As described in the introduction, we will need to take the maximum absolute value of fourier coefficients which correspond to k-vector |k|=const. To filter the coefficients Table we will use the FESeries::process\_coefficients() which requires a predicate to be specified. The predicate should operate on TableIndices and return a pair of bool and unsigned int. The latter is the value of the map from TableIndicies to unsigned int. It is used to define subsets of coefficients from which we search for the one with highest absolute value, i.e. l∞-norm. The bool parameter defines which indices should be used in processing. In the current case we are interested in coefficients which correspond to 0<i∗i+j∗j<N∗N and 0<i∗i+j∗j+k∗k<N∗N in 2D and 3D, respectively.

正如在介绍部门描述的，我们需要取|k|=常数的所有**k**向量对应的傅里叶系数的最大值。为了从系数表中过滤出这个系数，我们会使用FESeries::process\_coefficients()函数，它需要一个”谓语”（predicate）。这个谓语应该在TableIndices上进行操作，返回一个bool值和无符号整数。后者是从TableIndicies到无符号整数的映射（map）的值。用它来定义系数表的子集，从这个子集中我们搜索绝对值最大的那个，即l∞范数最大的那个。Bool参数定义应该使用哪个indices。在当前算例中我们只对满足0<i∗i+j∗j<N∗N条件的系数有兴趣。

template <int dim>

std::pair<bool,unsigned int>

LaplaceProblem<dim>::

predicate(const TableIndices<dim> &ind)

{

unsigned int v = 0;

for (unsigned int i = 0; i <dim; i++)

v += ind[i]\*ind[i];

if (v>0 && v < max\_degree\*max\_degree)

return std::make\_pair(true,v);

else

return std::make\_pair(false,v);

}

这最后一个重要的函数实现了用于估算光滑度指数的算法。这在介绍中已经讲过了。

template <int dim>

void

LaplaceProblem<dim>::

estimate\_smoothness (Vector<float> &smoothness\_indicators)

{

因为绝大多数难题都在FESeries::Fourier中解决了，而且我们在构造函数中创建了这个类的对象，剩下需要做的就是使用这个类去计算系数，然后进行线性回归来拟合它们的衰减速度。

需要做的第一件事是在所有网格上循环，然后再网格上进行操作，即进行局部的傅里叶变换，然后估算衰减系数。我们将使用下述数组作为一个在循环中存储局部DoF值的scratch数组：

Vector<double> local\_dof\_values;

然后是循环：

typename hp::DoFHandler<dim>::active\_cell\_iterator

cell = dof\_handler.begin\_active(),

endc = dof\_handler.end();

for (; cell!=endc; ++cell)

{

在循环中，我们先需要得到局部自由度的值（我们会把它们放到local\_dof\_values数组）然后需要用对应的F矩阵乘以这个向量来计算傅里叶变换。使用FESeries::Fourier::calculate()函数来完成这一操作。需要提供local\_dof\_values, cell->active\_fe\_index() 以及一个用于存储系数的Table给这个函数。

local\_dof\_values.reinit (cell->get\_fe().dofs\_per\_cell);

cell->get\_dof\_values (solution, local\_dof\_values);

fourier->calculate(local\_dof\_values,

cell->active\_fe\_index(),

fourier\_coefficients);

下一步是我们希望只用每个|k|对应的最大的傅里叶系数来拟合衰减指数。我们使用FESeries::process\_coefficients()把系数修改为期望的格式。

函数FESeries::process\_coefficients()有一个输入参数为一个coefficients表，两个返回参数predicate值向量的引用和子集norm向量（这两个向量构成一个pair，所以此函数的返回值是pair <std::vector<unsigned int>,std::vector<double> >）。函数计算coefficients表的子集的norm，这个subset是由predicate定义的。

predicate应该返回一对由bool和unsigned int组成的对组pair。Bool用于标记是否使用某个给定的TableIndices；unsigned int is the unrolled value of indices according to which the subsets of coefficients will be formed.

我们只取给定|k|值对应的最大的系数，因而需要使用VectorTools::Linfty\_norm：

std::pair<std::vector<unsigned int>, std::vector<double> > res =

FESeries::process\_coefficients<dim>(fourier\_coefficients,

std\_cxx11::bind(&LaplaceProblem<dim>::predicate,

this,

std\_cxx11::\_1),

VectorTools::Linfty\_norm);

Assert (res.first.size() == res.second.size(),

ExcInternalError());

std::pair中的第一个向量会存储谓语（predicate）的值，即i∗i+j∗j=const。这个向量对所有网格都相同，所以在整个hp循环中，只需要计算相应傅里叶向量|k|的对数一次：

if (ln\_k.size() == 0)

{

ln\_k.resize(res.first.size(),0);

for (unsigned int f = 0; f < ln\_k.size(); f++)

ln\_k[f] = std::log (2.0\*numbers::PI\*std::sqrt(1.\*res.first[f]));

}

我们必须计算系数绝对值的对数，并用于线性回归拟合得到μ：

for (unsigned int f = 0; f < res.second.size(); f++)

res.second[f] = std::log(res.second[f]);

std::pair<double,double> fit = FESeries::linear\_regression(ln\_k,res.second);

最后一步是计算Sobolev指数s=μ−d2，并存放到一个估算值向量中：

smoothness\_indicators(cell->active\_cell\_index()) = -fit.first - 1.\*dim/2;

}

}

}

**The main function**

int main ()

{

try

{

using namespace dealii;

using namespace Step27;

LaplaceProblem<2> laplace\_problem;

laplace\_problem.run ();

}

catch (std::exception &exc)

{

std::cerr << std::endl << std::endl

<< "----------------------------------------------------"

<< std::endl;

std::cerr << "Exception on processing: " << std::endl

<< exc.what() << std::endl

<< "Aborting!" << std::endl

<< "----------------------------------------------------"

<< std::endl;

return 1;

}

catch (...)

{

std::cerr << std::endl << std::endl

<< "----------------------------------------------------"

<< std::endl;

std::cerr << "Unknown exception!" << std::endl

<< "Aborting!" << std::endl

<< "----------------------------------------------------"

<< std::endl;

return 1;

}

return 0;

}

**Results**

在这部分，我们讨论一些结果。更多的计算结果，尤其是拓展到3D，统计计算时长等在hp\_paper中给出。

运行的时候会显示出下述内容:

examples/step-27> make run

============================ Running step-27

Cycle 0:

Number of active cells: 768

Number of degrees of freedom: 3264

Number of constraints : 384

Cycle 1:

Number of active cells: 966

Number of degrees of freedom: 5245

Number of constraints : 936

Cycle 2:

Number of active cells: 1143

Number of degrees of freedom: 8441

Number of constraints : 1929

Cycle 3:

Number of active cells: 1356

Number of degrees of freedom: 12349

Number of constraints : 3046

Cycle 4:

Number of active cells: 1644

Number of degrees of freedom: 18178

Number of constraints : 4713

Cycle 5:

Number of active cells: 1728

Number of degrees of freedom: 22591

Number of constraints : 6095

从结果中可以看到，受约束的网格数约为总自由度数目的20-25%（在三维情况下可高达30%）。这与非hp离散的量级差不多。举个例子，在step-6的最后一步中，有18401个自由度，其中4104个是受约束的。区别在于在step-6中，每个悬挂网格只收到两个相邻自由度的约束，但在hp例子中，受约束的点是收到多得多的自由度的限制。