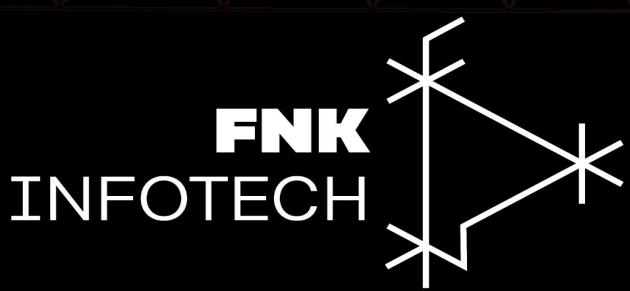


Hans Petter Langtangen
Anders Logg

用PYTHON求解偏微分方程

FEniCS教程第一卷

翻译作者 信吉平 高振



Copyright © 2017 Hans Petter Langtangen & Anders Logg

Creative Commons Attribution 4.0

This book is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, duplication, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

Edition 0.0

Chinese translation by Jiping Xin

Printed in

Contents

前言	5
1 预赛	6
1.1 FEniCS 项目	6
1.2 你会学到什么	6
1.3 使用本教程	7
1.4 获取软件	7
1.5 获取教程示例	9
1.6 背景知识	9
2 基本原理：解决泊松方程	11
2.1 数学问题的制定	11
2.2 FEniCS 实现	15
2.3 解剖方案	17
2.4 膜的变形	25
3 有限元求解器画廊	30
3.1 热方程式	30
3.2 非线性 Poisson 方程	37
3.3 线性弹性方程	40

3.4	Navier–Stokes 方程	45
3.5	平流–扩散–反应方程组	58
4	子域和边界条件	65
4.1	结合 Dirichlet 和 Neumann 条件	65
4.2	设置多个 Dirichlet 条件	67
4.3	定义不同材质的子域	68
4.4	设置多个 Dirichlet, Neumann 和 Robin 条件	73
4.5	用子域生成网格	78
5	扩展：改进 Poisson 求解器	86
5.1	重构 Poisson 求解器	86
5.2	使用线性求解器	90
5.3	高级和低级解算器接口	93
5.4	自由度和功能评估	96
5.5	后处理计算	100
5.6	下一步	111
	参考书目	111



本书简要介绍了有限元素基于流行的 FEniCS 软件库在 Python 中进行编程。FEniCS 可以在 C++ 和 Python 中编程，但本教程专注于 Python 编程，因为这是最简单的并为初学者提供最有效的方法。消化过后本教程中的示例，读者应该能够了解更多从 FEniCS 文档中可以看到众多演示程序与软件和综合 FEniCS 书 *Automated Solution of Differential Equations by the Finite Element Method* [FEniCS]。本教程是开幕式的进一步发展 [FEniCS] 中的章节。

我们感谢 Johan Hake, Kent-Andre Mardal 和 Kristian Valen-Sendstad 在准备第一个时候进行许多有益的讨论本教程的版本为 FEniCS 书 [FEniCS]。我们是特别感谢 Douglas Arnold 非常有价值对文本早期版本的反馈。Øystein Sørensen 指出许多打字错误，并提供了许多有用的意见。许多错误还有 Mauricio Angeles, IdaDrøsdal, Miroslav Kuchta, Hans Ekkehard Plessner, Marie Rognes, Hans Joachim Scroll, Glenn Terje Lines, Simon Funke, Matthew Moelter 和 Magne Nordaas。Ekkehard Ellmann 以及两位匿名评审员一系列建议和改进。特别感谢 Benjamin Kehlet 为他所有的工作与 mshr 工具和快速实施我们对本教程的要求。

注释和更正可以报告为本书 Git 存储库的 emph issue:

<https://github.com/hplgit/fenics-tutorial>

Oslo and Smögen, November 2016

Hans Petter Langtangen, Anders Logg

1. 预赛

1.1	FEniCS 项目	6
1.2	你会学到什么	6
1.3	使用本教程	7
1.4	获取软件	7
1.5	获取教程示例	9
1.6	背景知识	9

1.1 FEniCS 项目

FEniCS 项目是针对的一个研究和软件项目创建自动计算的数学方法和软件数学建模。这意味着创建简单,直观,高效,和用于求解偏微分方程的灵活软件 PDE 使用有限元法。FEniCS 最初创建于 2003 年由研究人员合作开发世界各地的大学和研究机构数量。对于有关 FEniCS 的更多信息和 FEniCS 的最新更新软件和本教程,请访问 FEniCS 网页 <https://fenicsproject.org>。

FEniCS 由许多构建块(软件组件)组成,一起形成 FEniCS 软件: DOLFIN [DOLFIN], FFC [FFC], FIAT [FIAT], UFL [UFL_2014], mshr, 和其他几个。有关概述,请参阅 [FEniCS]。FEniCS 用户很少需要考虑这一点 FEniCS 的内部组织,但即使是临时用户也可以有时会遇到各种 FEniCS 组件的名称,我们简要地说列出 FEniCS 中的组件及其主要角色。DOLFIN 是计算高性能的 C ++ 后端的 FEniCS。DOLFIN 实现数据结构,如网格,函数空间和函数,计算密集型算法如有限元组合和网格细化,以及与线性代数解算器的接口和数据结构如 PETSc。DOLFIN 还实施了 FEniCS C ++ 和 Python 中的解决问题的环境。FFC 是代码 FEniCS(表单编译器)的一代引擎,负责从高级数学生成高效的 C ++ 代码抽象。FIAT 是 FEniCS 的有限元后端,负责生成有限元基函数, UFL 实现用户可以使用的抽象数学语言表达变体问题, mshr 提供 FEniCS 网格生成功能。

1.2 你会学到什么

本教程的目标是演示如何应用有限元解决 FEniCS 中的 PDE。通过一系列例子,我们演示如何:

- 求解线性 PDE(如泊松方程),
- 解决时间依赖型 PDEs(如热方程),
- 求解非线性 PDE,
- 解决时间依赖非线性 PDE 的系统。

重要话题涉及如何设定各种边界条件类型 (Dirichlet, Neumann, Robin), 如何创建网格, 如何定义变量系数, 如何与线性和非线性求解器, 以及如何后处理和可视化解决方案。

我们还将讨论如何最好地构建 Python 代码 PDE 求解器, 如何调试程序, 以及如何利用的测试框架。

1.3 使用本教程

插图的数学保持简单, 更好地关注 FEniCS 功能和语法。这意味着我们大多使用 Poisson 方程和时间依赖扩散方程作为模型问题, 经常与输入数据调整, 使我们得到一个非常简单的解决方案可以通过任何标准有限的精确复制元素方法在均匀的结构化网格中。后一个属性大大简化了实现的验证。我们偶尔会插入一个更为相关的实例来提醒读者从解决一个简单的模型问题到一个 FEniCS 的挑战现实世界的问题往往相当短暂和容易。

使用 FEniCS 来解决 PDE 似乎需要一个彻底的理解有限的抽象数学框架元素方法以及 Python 编程方面的专业知识。不过, 事实证明, 很多用户都可以拿起来有限元素和 Python 编程的基本原理以及本教程。只需继续阅读并尝试例子。你会惊讶于用 PDE 来解决 PDE 的容易程度 FENICS!

1.4 获取软件

使用本教程显然需要访问 FEniCS 软件。FEniCS 是一个复杂的软件库, 本身和到期它对许多依赖于最先进的开源科学的依赖软件库。手动构建 FEniCS 及其所有依赖项从源头可以成为一项艰巨的任务。即使是知道的专家究竟如何配置和构建每个组件, 一个完整的构建可以在字面上需要几个小时! 除了软件的复杂性本身, 还有一层额外的复杂性在多少不同种类的操作系统 (Linux, Mac, Windows) 可能在用户的笔记本电脑或计算服务器上运行不同的要求如何配置和构建软件。

因此, FEniCS 项目提供了预先构建的包安装方便, 快捷, 万无一失。

注意

(FEniCS 下载和安装) 在本教程中, 我们强调安装的两个主要选项 FEniCS 软件: Docker 容器和 Ubuntu 软件包。当时 Docker 容器适用于所有操作系统, 即 Ubuntu 软件包只能在基于 Ubuntu 的系统上工作。注意内置的 FEniCS 虽然基本原理, 目前情况并不适用于 Docker 通过 Docker Jupyter 笔记本选项支持绘图。

FEniCS 也可以使用其他方法安装, 包括 Conda 包装和建筑从源头。有关更多安装选项和关于最简单和最佳安装选项的最新信息 FEniCS, 查看官方 FEniCS 安装说明。这些可以在那里找到

<https://fenicsproject.org/download>

注意

(FEniCS 版本: 2016.2) FEniCS 版本标注为 2016.1, 2016.2, 2017.1 等, 主要数字表示发行年份次数是从 1 开始的计数器。发布次数每年变化不大, 但通常可以预期每次 2–3 次年。本教程是为 FEniCS 准备和测试的版本 2016.2。

1.4.1 使用 Docker 容器进行安装

一个现代化的解决方案，挑战软件安装的多样化软件平台是使用所谓的容器。FEniCS 项目提供定制容器，受控，一致和高性能的软件环境为 FEniCS 节目。FEniCS 容器工作同样好¹ 所有操作系统，包括 Linux, Mac 和 Windows。

要使用 FEniCS 容器，您必须先安装 Docker 平台。Docker 安装很简单，可以使用说明书在

<https://www.docker.com>

一旦你安装了 Docker，只需将以下行复制到终端窗口：

Bash code
1 \$ curl -s https://get.fenicsproject.org | bash

上面的命令将安装程序 fenicsproject 系统。此程序可让您轻松创建 FEniCS 会话（容器）在您的系统：

Bash code
1 \$ fenicsproject run

此命令有几个有用的选项，如轻松切换最新发布的 FEniCS 之间的最新开发版本还有很多。要了解更多信息，请键入 fenicsproject help。FEniCS 可以也可以直接与 Docker 一起使用，但这通常需要键入一个比较复杂的 Docker 命令，例如：

Bash code
1 \$ docker run --rm -ti -v `pwd`:/home/fenics/shared -w
2 /home/fenics/shared quay.io/fenicsproject/stable:current '/bin/bash -l
3 -c "export TERM=xterm; bash -i"

注意

(与 FEniCS 容器共享文件) 当您使用 fenicsproject run 运行 FEniCS 会话时，它会自动共享您当前的工作目录（该目录使用 FEniCS 从中运行 fenicsproject 命令）会话。当 FEniCS 会话启动时，它会自动启动进入一个名为 shared 的目录，它将与之相同您的主机系统上的当前工作目录。这意味着您可以轻松地编辑文件并在 FEniCS 会话中写入数据文件将直接在您的主机系统上访问。它是建议您使用自己喜欢的编辑器编辑程序（如 Emacs 或 Vim），并使用 FEniCS 会话只能运行你的程序。

1.4.2 使用 Ubuntu 软件包进行安装

对于 Ubuntu GNU/Linux 用户，FEniCS 也可以轻松安装标准的 Ubuntu 软件包 manager apt-get。只需复制以下内容线路进入终端窗口：

Bash code
1 \$ sudo add-apt-repository ppa:fenics-packages/fenics
2 \$ sudo apt-get update
3 \$ sudo apt-get install fenics
4 \$ sudo apt-get dist-upgrade

¹与 Linux 上运行 Docker 容器相比，运行在 Mac 和 Windows 上的 Docker 容器具有较小的性能开销。然而，通过使用官方 FEniCS 容器附带的 FEniCS 的高度调优和优化版本，与在 Mac 或 Windows 上的源代码构建 FEniCS 及其依赖相比，性能损失通常很小，并且经常得到补偿。

这将添加 FEniCS 包存档 (PPA) 到您的 Ubuntu 电脑的软件清单，然后安装 FEniCS。它会还将自动安装 FEniCS 的依赖关系。

注意

(**注意旧包装!**) 除了 FEniCS PPA, FEniCS 提供软件也是官方 Ubuntu 存储库的一部分。然而, 这取决于您正在运行的 Ubuntu 的版本, 何时启动发布是针对最新的 FEniCS 版本而创建的官方 Ubuntu 存储库可能包含过时的版本 FENICS。因此, 最好从 FEniCS PPA 安装。

1.4.3 测试您的安装

一旦你安装了 FEniCS, 你应该快速测试一下您的安装正常工作。为此, 请键入以下内容命令在 FEniCS 启用的² terminal:

Bash code

```
1 $ python -c 'import fenics'
```

如果一切顺利, 你应该能够运行这个命令没有任何错误信息 (或任何其他输出)。

1.5 获取教程示例

在本教程中, 您将学习有限元和 FEniCS 编程通过一些示范程序, 演示如何使用有限元法解决特定的 PDE, 如何编程在 FEniCS 中的解算器, 以及如何创建精心设计的 Python 代码以后可以扩大到解决更复杂的问题。所有示例程序可从本书的网页获得

<https://fenicsproject.org/tutorial>。

节目以及此文本的源代码也可以直接从

<https://github.com/hplgit/fenics-tutorial/>

书。

1.6 背景知识

1.6.1 用 Python 编程

虽然你可以通过工作来接受基本的 Python 编程通过本教程中的示例, 您可能需要学习关于 Python 语言的附加资料。一个自然的起点对于初学者来说, 是经典的 Python Tutorial[PythonTutorial], 或者是针对科学计算的教程 [Langtangen_Hellevik_tutorial]。在后者中, 你也会发现指向 Python 中科学计算的其他教程。其中普通书我们推荐一般介绍 Dive into Python[Pilgrim] 以及专注于科学的文本用 Python 进行计算 [Langtangen2008, Langtangen2009a, Kinder_Nelson_2015, Kiusalaas2005, Landau_2015]。

注意

(**Python 版本**) Python 有两个版本, 2 和 3, 这些不兼容。FEniCS 适用于两种版本的 Python。一切本教程中的程序也可以开发, 以便可以运行它们在 Python 2 和 3 之下。

²对于 FEniCS 容器的用户, 这意味着首先运行命令 fenicsproject run。

需要打印的 Python 程序必须然后开始

Bash code

```
1 $ from __future__ import print_function
```

在 Python 2 中启用 Python 3 中的 print 函数在本教程的程序中使用 print 由函数组成电话，如 `print('a: ', a)`。几乎所有的其他结构都是在 Python 2 和 3 中看起来一样的表单。

1.6.2 有限元法

许多好书已经写在有限元法上。该书通常属于两类：摘要方法的数学版本或工程“结构”分析“制定”。FEniCS 严格依赖于这些概念抽象数学博览会。第一作者有书³[Langtangen_Mardal_FEM_2016]一个

<http://hplgit.github.io/fem-book/doc/web/index.html>

在开发中以直观的方式解释有限元法的所有细节，使用 FEniCS 采用的抽象数学公式。

Larson 和 Bengzon 的有限元文本 [Larson_2013] 是我们推荐的有限元法介绍，数学符号与 FEniCS 一致。一本易于阅读的书籍，也为您提供了一个很好的一般背景使用 FEniCS，是 Gockenbach [Gockenbach2006]。这本书由 Donea 而 Huerta [DoneaHuerta2003] 具有相似的风格，但目标是读者对流体流动问题感兴趣。Hughes [Hughes1987] 也被推荐，特别是对于对固体感兴趣的读者力学和热传递应用。

具有工程“结构分析”背景的读者有限元方法的版本可能会找到 Bickford [Bickford1994] 一个有吸引力的桥梁到抽象 FEniCS 建立的数学公式。那些有一个微分方程的弱背景一般应该咨询一个更基础的书，和 Eriksson et al [ErikssonEstepHansboEtAl1996] 是一个很好的选择。在另一方面，FEniCS 用户具有较强的数学背景将欣赏 Brenner 和 Scott [BrennerScott2008] 的文章，Braess [Braess2007]，Ern 和 Guermond [ErnGuermond2004]，Quarteroni 和 Valli [QuarteroniValli1994]，或 Ciarlet [Ciarlet2002]。

³<http://hplgit.github.io/fem-book/doc/web/index.html>

2. 基本原理：解决泊松方程

2.1	数学问题的制定	11
2.2	FEniCS 实现	15
2.3	解剖方案	17
2.4	膜的变形	25

本章的目标是展示泊松方程如何所有 PDE 中最基本的都可以用几行快速解决的 FEniCS 代码。我们介绍最多基本 FEniCS 对象：Mesh, Function, FunctionSpace, TrialFunction, TestFunction。了解如何编写基本的 PDE 求解器，包括如何制定数学变分问题，应用边界条件，调用 FEniCS 求解器和绘图解决方案。

2.1 数学问题的制定

许多关于编程语言的书籍都以 HelloWorld 开头程序。读者好奇知道基本的任务是什么用语言表达，并将文字打印到屏幕上即可这样的一个任务。在有限元方法的世界，PDE，最基本的任务是解决泊松方程。我们的因此对应于古典 HelloWorld 程序解决以下边界值问题：

$$-\nabla^2 u(\mathbf{x}) = f(\mathbf{x}), \quad \mathbf{x} \text{ in } \Omega, \tag{2.1}$$

$$u(\mathbf{x}) = u_D(\mathbf{x}), \quad \mathbf{x} \text{ on } \partial\Omega. \tag{2.2}$$

这里， $u = u(\mathbf{x})$ 是未知函数， $f = f(\mathbf{x})$ 是规定的函数， ∇^2 是 Laplace 算子（通常写为 Δ ）， Ω 是空间域，而 $\partial\Omega$ 是 Ω 的边界。Poisson 问题，包括 PDE $-\nabla^2 u = f$ 和边界条件 $u = u_D$ on $\partial\Omega$ ，是边界值的一个例子问题，必须在之前精确地陈述使用 FEniCS 开始解决它是有意义的。

在具有坐标 x 和 y 的两个空间维度中，我们可以写出来 Poisson 方程为

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x, y). \tag{2.3}$$

未知的 u 现在是两个变量 $u = u(x, y)$ 的函数在二维域 Ω 。

Poisson 方程出现在许多物理环境中，包括热传导，静电，物质扩散，扭转弹性棒，非粘性流体流和水波。而且，方程出现在数值分割策略中更为复杂 PDE 系统，特别是 Navier-Stokes 方程。

解决边界值问题，如 Poisson 方程 FEniCS 包括以下步骤：

1. 识别计算域 (Ω), PDE, 它边界条件和源项 (f)。
2. 将 PDE 重新定义为有限元变分问题。
3. 编写一个定义计算域的 Python 程序, 变分问题, 边界条件和来源条款, 使用相应的 FEniCS 抽象。
4. 调用 FEniCS 来解决边界值问题, 并且可选地, 扩展程序计算衍生量, 如通量和平均值, 以及可视化结果。

现在我们将详细介绍第 2–4 步。的主要特点 FEniCS 是步骤 3 和 4 导致相当短的代码, 而 a 大多数其他 PDE 软件框架中的类似程序都需要更多的代码和技术上困难的编程。

注意

(什么使 FEniCS 有吸引力?) 虽然许多软件框架都非常优雅 HelloWorld 的例子 Poisson 方程式, FEniCS 是我们知道的唯一框架代码保持紧凑和美观, 非常接近数学即使是数学和算法的复杂性从笔记本电脑转移到高性能时会增加计算服务器(集群)。

2.1.1 有限元变分法

FEniCS 是基于有限元法, 它是一般的和高效数学机械的数值解 PDE。有限元方法的出发点是 PDE 以变体形式表达。不熟悉的读者变数问题将会简要介绍一下这个话题在本教程中, 但阅读有限元上的正确书鼓励方法。经验表明, 您可以使用 FEniCS 作为解决 PDE 的工具, 即使没有深入的了解有限元法, 只要你有人帮你将 PDE 作为变分问题。

将 PDE 转化为变分问题的基本方法是将 PDE 乘以函数 v , 整合得到的等式通过域 Ω , 并按部分条款执行整合与二阶导数。函数 v 乘以 PDE 称为测试功能。未知函数 u 为近似被称为试验函数。试用版和测试功能也用于 FEniCS 程序。试用和测试功能属于指定的所谓功能空间功能的属性。

在这种情况下, 我们先乘以 Poisson 方程通过测试函数 v 并集成 Ω :

$$-\int_{\Omega} (\nabla^2 u)v \, dx = \int_{\Omega} fv \, dx. \quad (2.4)$$

我们这里让 dx 表示用于集成的差分元素域 Ω 。我们稍后会让 ds 表示差分在 Ω 的边界上整合的元素。

当我们得出变分公式时, 一个常见的规则就是我们尝试保持 u 和 v 的衍生工具的顺序尽可能小可能。在这里, 我们有一个 u 的二阶空间导数, 这可以转换为 u 和 v 的一阶导数应用零件整合技术。公式读

$$-\int_{\Omega} (\nabla^2 u)v \, dx = \int_{\Omega} \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \, ds, \quad (2.5)$$

其中 $\frac{\partial u}{\partial n} = \nabla u \cdot n$ 是 u 的衍生物向外向正方向 n 边界。

变分配方的另一个特点是测试函数 v 需要在部分消失解决方案 u 的边界已知。在现在问题, 这意味着 $v = 0$ 在整个边界 $\partial\Omega$ 。第二个术语在右边 (2.5) 因此消失。从 (2.4) 和 (2.5) 遵循

$$\int_{\Omega} \nabla u \cdot \nabla v \, dx = \int_{\Omega} fv \, dx. \quad (2.6)$$

如果我们要求该方程适用于所有测试函数 v 一些合适的空间 \hat{V} , 所谓的测试空间, 我们获得一个明确的数学问题, 唯一地决定了解决方案 u 在于一些可能不同的功能空间 V , 所谓的试验空间。我们参考 (2.6) 作为弱形式或变体形式原边界值问题 (2.1)–(2.2)。

适当的陈述我们的变分问题现在如下：找到 $u \in V$ 这样

$$\int_{\Omega} \nabla u \cdot \nabla v \, dx = \int_{\Omega} f v \, dx \quad \forall v \in \hat{V}. \quad (2.7)$$

现在试用和测试空间 V 和 \hat{V} 问题定义为

$$\begin{aligned} V &= \{v \in H^1(\Omega) : v = u_D \text{ on } \partial\Omega\}, \\ \hat{V} &= \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}. \end{aligned}$$

简而言之, $H^1(\Omega)$ 是数学上众所周知的 Sobolev 空间包含函数 v , 使 v^2 和 $|\nabla v|^2$ 有有限积分超过 Ω (基本上意味着函数是连续的)。底层 PDE 的解决方案必须在于函数空间中的衍生物也是连续的, 但是 Sobolev 空间 $H^1(\Omega)$ 允许不连续的函数衍生物。 u 的连续性要求较弱变量语句 (2.7), 作为结果整合部分, 具有很大的实际后果构造有限元函数空间。特别是它允许使用分段多项式函数空间; 即功能通过简单地将多项式函数拼接在一起构成的空间域, 如间隔, 三角形或四面体。

变量问题 (2.7) 是连续的问题：它定义了无穷维的解 u 功能空间 V 。Poisson 方程的有限元法找出变分问题的近似解 (2.7) 替换无限维函数空间 V 和 \hat{V} 通过离散 (有限维) 试验测试空间 $V_h \subset V$ 和 $\hat{V}_h \subset \hat{V}$ 。离散变分问题如下：找到 $u_h \in V_h \subset V$ 这样

$$\int_{\Omega} \nabla u_h \cdot \nabla v \, dx = \int_{\Omega} f v \, dx \quad \forall v \in \hat{V}_h \subset \hat{V}. \quad (2.8)$$

这个变分问题, 连同一个合适的定义函数空间 V_h 和 \hat{V}_h , 唯一地定义我们的近似值 Poisson 方程的数值解 (2.1)。注意边界条件被编码为试验和测试的一部分空间。起初, 数学框架可能看起来很复杂一瞥, 但好消息是有限元变分问题 (2.8) 看起来和连续的一样变分问题 (2.7) 和 FEniCS 可以自动解决变量问题, 如 (2.8)!

注意

(**我们的意思是符号 u 和 V**) 关于变数问题的数学文献写了 u_h 解的离散问题和 u 的解决方案连续问题获得 (几乎) 一对一的关系在一个问题的数学表达与之间相应的 FEniCS 程序, 我们将下拉 h 和使用 u 用于解决离散问题。我们将使用 u_e 来确定解决连续问题, 如果我们需要明确区分两者之间。类似地, 我们将让 V 表示离散有限元素功能空间, 我们寻求我们的解决方案。

2.1.2 抽象有限元变分公式

原来是方便的介绍下列规范变量问题的符号：找到 $u \in V$ 这样

$$a(u, v) = L(v) \quad \forall v \in \hat{V}. \quad (2.9)$$

对于 Poisson 方程, 我们有:

$$a(u, v) = \int_{\Omega} \nabla u \cdot \nabla v \, dx, \quad (2.10)$$

$$L(v) = \int_{\Omega} f v \, dx. \quad (2.11)$$

从数学文献中， $a(u, v)$ 被称为双线性形式和 $L(v)$ 作为线性形式。我们将在每一个线性问题中我们解决，确定与未知 u 的条款并收集他们 $a(u, v)$ ，并且类似地收集只有已知功能的所有术语 $L(V)$ 。 a 和 L 的公式可以直接表达我们的 FEniCS 计划。

为了解决 FEniCS 中的线性 PDE，如 Poisson 方程，用户因此需要执行两个步骤：

- 通过指定选择有限元空间 V 和 \hat{V} 域（网格）和函数空间的类型（多项式）度和类型）。
- 将 PDE 表示为（离散）变分问题：找到 $u \in V$ 所以 $a(u, v) = L(v)$ 为 $v \in \hat{V}$ 。

2.1.3 选择测试问题

泊松问题 (2.1)–(2.2) 有远程功能一般域 Ω 和一般功能 u_D 边界条件和右边的 f 。对于我们的第一个实现我们将需要为 Ω 做出具体的选择， u_D 和 f 。构建一个已知的问题是明智的分析解决方案，使我们可以方便地检查计算解决方案是正确的。低阶多项式的解是主要候选人标准有限元函数空间的度数 r 将完全重现度为 r 的多项式。分段线性元素 ($r = 1$) 能够精确地再现二次方均匀分布网格上的多项式。这个重要的结果可以用于验证我们的实现。我们只是制造一些比如说 2D 中的二次函数作为确切的解决方案

$$u_e(x, y) = 1 + x^2 + 2y^2. \quad (2.12)$$

通过将 (2.12) 插入到 Poisson 方程式中 (2.1)，我们发现 $u_e(x, y)$ 是一个解决方案，以便

$$f(x, y) = -6, \quad u_D(x, y) = u_e(x, y) = 1 + x^2 + 2y^2,$$

只要 u_e 被规定，不管域的形状边界。我们在这里选择，为了简单，该域为单位广场，

$$\Omega = [0, 1] \times [0, 1].$$

这个简单但非常强大的构建测试问题的方法是称为制造解决方案：选择一个简单表达式为精确解，将其插入等式获得右边（源项 f ），然后用公式求解这个右手边使用精确的解决方案作为边界条件，并尝试重现确切的解决方案。

注意

(提示: 尝试使用精确的数值解决方案验证您的代码!) 测试数值方法实现的常用方法是比较数字解决方案具有测试问题的精确解析解得出结论，如果错误是“足够小”，该程序可以工作。不幸的是，不可能知道一个大小为 10^{-5} 的错误 20×20 mesh 的线性元素是预期的准确度数值近似或者如果误差也包含 a 的影响代码中的错误。我们通常都知道数值误差是它的 asymptotic properties，例如它与 h^2 成正比如果 h 是网格中单元格的大小。然后我们比较使用不同的 h -values 来查看网格的错误，看是否渐近行为是正确的。这是一个非常强大的验证技术，并在 [\[F\]5.5.4](#) 部分中详细解释。但是，如果我们有一个测试问题我们知道应该没有近似误差，我们知道 PDE 问题的分析解决方案应该重现机器精度由程序。这就是为什么我们强调这种本教程中的测试问题。通常，元素 degree r 可以重现度为 r 的多项式，所以这样是构建无数值解的起点近似误差

2.2 FEniCS 实现

2.2.1 完整的程序

一个用于解决 Poisson 方程的测试问题的 FEniCS 程序在 2D 中，给定的 Ω , u_D 和 f 的选项可能看起来像如下：

Python code

```

1 from fenics import *
2
3 # Create mesh and define function space
4 mesh = UnitSquareMesh(8, 8)
5 V = FunctionSpace(mesh, 'P', 1)
6
7 # Define boundary condition
8 u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', degree=2)
9
10 def boundary(x, on_boundary):
11     return on_boundary
12
13 bc = DirichletBC(V, u_D, boundary)
14
15 # Define variational problem
16 u = TrialFunction(V)
17 v = TestFunction(V)
18 f = Constant(-6.0)
19 a = dot(grad(u), grad(v))*dx
20 L = f*v*dx
21
22 # Compute solution
23 u = Function(V)
24 solve(a == L, u, bc)
25
26 # Plot solution and mesh
27 plot(u)
28 plot(mesh)
29
30 # Save solution to file in VTK format
31 vtkfile = File('poisson/solution.pvd')
32 vtkfile << u
33
34 # Compute error in L2 norm
35 error_L2 = errornorm(u_D, u, 'L2')
36
37 # Compute maximum error at vertices
38 vertex_values_u_D = u_D.compute_vertex_values(mesh)
39 vertex_values_u = u.compute_vertex_values(mesh)
40 import numpy as np
41 error_max = np.max(np.abs(vertex_values_u_D - vertex_values_u))
42
43 # Print errors
44 print('error_L2 =', error_L2)
45 print('error_max =', error_max)
46
47 # Hold plot
48 import matplotlib.pyplot as plt
49 plot(u)
50 plt.show()

```

该示例程序可以在文件中找到ft01_poisson.py。

https://fenicsproject.org/pub/tutorial/python/vol1/ft01_poisson.py

2.2.2 运行程序

FEniCS 程序必须以纯文本文件的形式提供文本编辑器，如 Atom, Sublime Text, Emacs, Vim 等。有几种方法可以运行 Python 程序ft01_poisson.py:

https://fenicsproject.org/pub/tutorial/python/vol1/ft01_poisson.py

- 使用终端窗口。
- 使用集成开发环境 (IDE)，例如 Spyder。
- 使用 Jupyter 笔记本。

终端窗口。 打开一个终端窗口，移动到包含程序的目录并键入以下命令：

Bash code

```
1 $ python ft01_poisson.py
```

请注意，此命令必须在支持 FEniCS 的终端中运行。对于 FEniCS Docker 容器的用户，这意味着您必须键入启动 FEniCS 会话后使用此命令 fenicsproject run 或 fenicsproject start。

运行上述命令时，FEniCS 将运行程序进行计算近似解 u 。大概的解决方案 u 将是与确切的解决方案 $u_e = u_d$ 相比， L^2 和将打印最大规范。既然我们知道我们的大概解决方案应该在机器内重现精确的解决方案精度，这个错误应该是小的，有的是顺序的 10^{-15} 。如果您的 FEniCS 安装中启用了绘图，那么一个简单的解决方案的窗口将显示为在图[E]2.1中。

Spyder。 许多人喜欢在一个集成的开发环境中工作提供编程编辑器，执行代码的窗口，用于检查对象的窗口等。只需打开文件ft01_poisson.py

https://fenicsproject.org/pub/tutorial/python/vol1/ft01_poisson.py

并按播放按钮运行它。我们参考 Spyder 教程了解更多关于在 Spyder 环境中工作的信息。Spyder 是强烈推荐，如果你习惯在图形工作 MATLAB 环境。

Jupyter notebooks。 笔记本电脑可以将文本和可执行代码混合在一起文档，但您也可以使用它来在网络中运行程序浏览器。从终端窗口运行命令 Jupyter notebook 在 GUI 的右上角找到 New 下拉菜单，在 Python 2 或 3 中选择一个新的 notebook，写入\%load ft01_poisson.py在这这个 notebook 的空白单元格中，然后按 Shift + Enter 执行单元格。然后，将文件ft01_poisson.py

https://fenicsproject.org/pub/tutorial/python/vol1/ft01_poisson.py

加载到 notebook。重新执行 cell (Shift + Enter) 运行程序。您可以将整个程序划分成几个单元格进行检查中间结果：将光标放在要分割的位置 cell 并选择 Edit - Split Cell。对于 FEniCS Docker 的用户图像，运行 fenicsproject notebook 命令并按照说明。要启用绘图，请确保运行该命令 \%matplotlib inline在 notebook 里面。

2.3 解剖方案

我们现在将详细剖析我们的 FEniCS 计划。列出的 FEniCS 程序定义有限元网格，有限元函数空间 V 在此网格上，边界条件为 u （函数 u_D ），和双线性和线性形式 $a(u, v)$ 和 $L(v)$ 。此后，解决方案 u 被计算。在程序结束时，我们比较了数值和确切的解决方案。我们也使用该图来绘制解决方案 plot 命令并将解决方案保存到外部文件后期处理。

2.3.1 重要的第一行

程序的第一行，

Python code

```
1 from fenics import *
```

导入关键类 UnitSquareMesh, FunctionSpace, Function, 等等，从 FEniCS 图书馆。所有 FEniCS 程序通过有限元法解决 PDE 通常从这开始。

2.3.2 生成简单的网格

该声明

Python code

```
1 mesh = UnitSquareMesh(8, 8)
```

在单位平方上定义均匀的有限元网格 $[0, 1] \times [0, 1]$ 。网格由 cells 组成，其中 2D 是三角形有直边。参数 8 和 8 指定正方形应分为 8×8 矩形，每个分为一对三角形。三角形 (cells) 的总数因此变为 128。网格中的顶点总数为 $9 \cdot 9 = 81$ 。在后面的章节中，您将学习如何生成更复杂的网格。

2.3.3 定义有限元函数空间

一旦创建了网格，就可以创建一个有限元的功能空间 V ：

Python code

```
1 V = FunctionSpace(mesh, 'P', 1)
```

第二个参数 'P' 指定元素的类型。这里的元素是 P ，意味着标准的 Lagrange 系列元素。您也可以使用 'Lagrange' 来指定此类型元件。FEniCS 支持所有单体元素族和符号定义在¹有限元素周期表 [ArnoldLogg2014]。

<https://www.femtable.org>

第三个参数 1 指定有限元的程度。在这种情况下，标准的 P_1 linear Lagrange 元素，其中是三角形，节点在三个顶点。一些有限元从业者将此元素称为“线性三角形”。该计算的解决方案 u 将在元素和线性上是连续的每个元素内的 x 和 y 变化。高阶多项式通过增加每个单元的近似值 FunctionSpace 的第三个参数，然后生成函数类型为 P_2, P_3 的空格等等。更改 'DP' 的第二个参数创建一个函数空间不连续的 Galerkin 方法。

¹<https://www.femtable.org>

2.3.4 定义试用和测试功能

在数学中，我们区分试验空间 V 和 \hat{V} 。目前问题的唯一区别是边界条件。在 FEniCS 中，我们没有指定边界条件作为功能空间的一部分，所以这是足够的工作其中一个公共空间 V 用于试用和测试功能程序：

Python code

```
1 u = TrialFunction(V)
2 v = TestFunction(V)
```

2.3.5 定义边界条件

下一步是指定边界条件： $u = u_D$ 上 $\partial\Omega$ 。这是完成的

Python code

```
1 bc = DirichletBC(V, u_D, boundary)
```

u_D 是定义解决方案值的表达式 `boundary` 和 `boundary` 是定义的一个函数（或对象）哪些点属于边界。

$u = u_D$ 类型的边界条件称为 Dirichlet 条件。对于 Poisson 的当前有限元方法问题，他们也被称为必需边界条件，因为它们需要作为审判空间的一部分明确强加（相比之下）被隐含地定义为变分公式的一部分）。自然地，FEniCS 类用于定义 Dirichlet 边界条件命名为 `DirichletBC`。

变量 `u_D` 指的是一个 `Expression` 对象，用于代表数学函数。典型的建筑是

Python code

```
1 u_D = Expression(formula, degree=1)
```

其中 `formula` 是一个包含数学表达式的字符串。公式必须用 C++ 语法编写自动变成一个高效的，编译的 C++ 函数。

注意

(**表达和准确性**) 当定义一个 `Expression` 时，第二个参数 `degree` 是一个指定应该如何处理表达式的参数计算。在每个本地元素上，FEniCS 将内插表达为有限元空间的指定程度。获得最佳（顺序）计算的准确性，通常是一个很好的选择使用与用于审判的空间 V 相同的程度和测试功能。但是，如果使用 `Expression` 表示一个精确的解决方案，用于评估计算的准确性解决方案，更高的程度必须用于表达（一个或两个度以上）。

该表达式可能取决于变量 $x[0]$ 和 $x[1]$ 对应于 x 和 y 坐标。在 3D 中，表达式也可能依赖于对应于 z 的变量 $x[2]$ 坐标。我们选择 $u_D(x, y) = 1 + x^2 + 2y^2$ ，公式字符串可以写为 $1 + x[0]^2 + 2x[1]^2$ ：

Python code

```
1 u_D = Expression('1 + x[0]^2 + 2*x[1]^2', degree=2)
```

我们将学位设置为 2，以便 `u_D` 可能代表确切的二次解决我们的测试问题。

注意

(**字符串表达式必须具有有效的 C++ 语法！**) `Expression` 对象的字符串参数必须遵守 C++ 语法。数学表达式的大多数 Python 语法也是有效的 C++ 语法，但是权力表达式是一个例外：`p**a` 必须写为 C++ 中的 `pow(p, a)`（这也是一个替代的 Python 语法）。以

下数学函数可以直接使用在 C++ 中定义 Expression 对象时的表达式: cos, sin, tan, acos, asin, atan, atan2, cosh, sinh, tanh, exp, frexp, ldexp, log, log10, modf, pow, sqrt, ceil, fabs, floor, and fmod. 此外, 数字 π 可用作符号 pi。所有列出的函数取自 cmath C++ 头文件, 和因此可能请参阅 cmath 的文档了解更多信息各种功能。

If/else 可以使用 C 语法进行内联分支。该功能

$$f(x, y) = \begin{cases} x^2, & x, y \geq 0, \\ 2, & \text{otherwise,} \end{cases}$$

被实现为

Python code

```
1 f = Expression('x[0]>=0 && x[1]>=0 ? pow(x[0], 2) : 2', degree=2)
```

表达式字符串中的参数是允许的, 但是必须通过关键字初始化创建 Expression 对象时的参数。例如, 函数 $f(x) = e^{-\kappa\pi^2 t} \sin(\pi k x)$ 可以编码为

Python code

```
1 f = Expression('exp(-kappa*pow(pi, 2)*t)*sin(pi*k*x[0])', degree=2,
2 kappa=1.0, t=0, k=4)
```

随时可以更新参数:

Python code

```
1 f.t += dt
2 f.k = 10
```

函数 boundary 指定属于哪些点应用边界条件的边界的一部分:

Python code

```
1 def boundary(x, on_boundary):
2     return on_boundary
```

用于标记边界的 boundary 的函数必须返回布尔值:True 如果给定点 x 位于 Dirichlet 上 boundary 和 False 否则。参数on_boundary 是 True 如果 x 在网格的物理边界上, 那么在现在情况下, 我们应该为所有的点返回 True 边界, 我们可以返回on_boundary的提供值。该将为每个离散点调用 boundary 函数网格, 这意味着我们可以定义 u 的边界如果需要, 在域内知道。

考虑 FEniCS 边界规范的一种方法是 FEniCS 会问你 (或者说是函数 boundary 哪个你已经实现了) 某个特定点 x 是否是其中的一部分边界。FEniCS 已经知道该点是否属于实际边界 (域的数学边界) 和善意在变量on_boundary中与您共享此信息。您可以选择使用这些信息 (如我们在这里), 或忽略它完全。

参数on_boundary 也可以省略, 但在这种情况下, 我们需要测试 x 中的坐标值:

Python code

```
1 def boundary(x):
2     return x[0] == 0 or x[1] == 0 or x[0] == 1 or x[1] == 1
```

使用完全匹配测试比较浮点值 == 不是很好的编程实践, 因为小的舍入误差在 x 值的计算中可以进行测试 $x[0] == 1$ 即使 x 位于边界上也变为虚假。一个更好的考验是明确地检查公平的公平

Python code

```

1 tol = 1E-14
2 def boundary(x):
3     return abs(x[0]) < tol or abs(x[1]) < tol \
4         or abs(x[0] - 1) < tol or abs(x[1] - 1) < tol

```

或在 FEniCS 中使用 near 命令:

Python code

```

1 def boundary(x):
2     return near(x[0], 0, tol) or near(x[1], 0, tol) \
3         or near(x[0], 1, tol) or near(x[1], 1, tol)

```

注意

(不要使用 $==$ 来比较实数!) 如果 $x[0]$ 是真实的, 则不应该使用像 $x[0] == 1$ 的比较数, 因为 $x[0]$ 中的舍入误差可能会使测试失败当它在数学上是正确的。考虑以下计算在 Python 中:

Python code

```

1 »> 0.1 + 0.2 == 0.3
2 False
3 »> 0.1 + 0.2
4 0.30000000000000004

```

需要使用公差进行实数比较! 该公差的值取决于所涉及的数字的大小算术运算:

Python code

```

1 »> abs(0.1 + 0.2 - 0.3)
2 5.551115123125783e-17
3 »> abs(1.1 + 1.2 - 2.3)
4 0.0
5 »> abs(10.1 + 10.2 - 20.3)
6 3.552713678800501e-15
7 »> abs(100.1 + 100.2 - 200.3)
8 0.0
9 »> abs(1000.1 + 1000.2 - 2000.3)
10 2.2737367544323206e-13
11 »> abs(10000.1 + 10000.2 - 20000.3)
12 3.637978807091713e-12

```

对于单位大小的数量, 可以使用低至 $3 \cdot 10^{-16}$ 的公差 (实际上, 这个容差在 FEniCS 中被称为常数 DOLFIN_EPS)。否则, 必须使用适当缩放的公差。

2.3.6 定义源术语

在定义双线性和线性形式 $a(u, v)$ 和 $L(v)$ 之前必须指定源术语 f :

Python code

```

1 f = Expression(' -6 ', degree=0)

```

当 f 在域上是恒定的, f 可以更有效地表示为 Constant:

Python code

```

1 f = Constant(-6)

```

2.3.7 定义变分问题

我们现在有了我们需要定义的所有成分变分问题：

Python code

```
1 a = dot(grad(u), grad(v))*dx
2 L = f*v*dx
```

实质上，这两行指定要解决的 PDE。注意 Python 语法之间非常接近的对应关系数学公式 $\nabla u \cdot \nabla v dx$ 和 $f v dx$ 。这个是 FEniCS 的关键优势：变分中的公式配方将直接转换为非常类似的 Python 代码，一个功能这使得容易指定和解决复杂的 PDE 问题。该用于表达弱表单的语言称为 UFL (Unified Form Language)[UFL_2014, FEniCS]，并且是 FEniCS 的组成部分。

注意

(**表达内在产品**) 内部产品 $\int_{\Omega} \nabla u \cdot \nabla v dx$ 可以在 FEniCS 中以各种方式表达。以上，我们用过符号 `dot(grad(u), grad(v))*dx`。点产品在 FEniCS/UFL 计算最后一个索引的总和（收缩）的第一个因素和第二个因素的第一个指标。在这种情况下，这两个因素都是一级（向量）和所以这个总和刚刚超过 ∇u 的一个单一的索引和 ∇v 。计算矩阵的内积（与两个索引），一个必须代替 `dot` 使用函数 `inner`。对于向量，`dot` 和 `inner` 是等效的。

2.3.8 形成和求解线性系统

定义了有限元变分问题和边界条件下，我们现在可以要求 FEniCS 计算解决方案：

Python code

```
1 u = Function(V)
2 solve(a == L, u, bc)
```

请注意，我们首先将变量 `u` 定义为 `TrialFunction` 和用它来表示 `a` 中的未知数。其后，我们重新定义 `u` 成为表示解决方案的 `Function` 对象；即计算的有限元函数 `u`。这个重新定义变量 `u` 在 Python 中是可行的，通常用于 FEniCS 线性问题的应用。`u` 的两种类型的对象指的是从数学角度来看是相等的，因此是对两个对象使用相同的变量名称是自然的。

2.3.9 使用 plot 命令绘制解决方案

一旦解决方案被计算出来，它就可以被可视化 `plot` 命令：

Python code

```
1 import matplotlib.pyplot as plt
2 plot(u)
3 plt.show()
```

注意在 `plot` 命令之后调用函数 `plt.show`。Figure 2.1 显示两个地块。

`plot` 命令对调试和初步科学有用调查。更高级的可视化更好地创建将解决方案导出到文件并使用高级可视化像 ParaView 这样的工具，如下一节所述。

通过在绘图窗口中单击鼠标左键，您可以旋转解决方案，而鼠标右键用于缩放。点鼠标到 Help 文本左下角显示一个列出所有可用的快捷命令。帮助菜单可能或者通过在绘图窗口中键入 `h` 来激活。该 `plot` 命令也接受一些额外的参数，如例如设置绘图窗口的标题：

Python code

```
1 plot(u, title='Finite element solution')
2 plot(mesh, title='Finite element mesh')
```

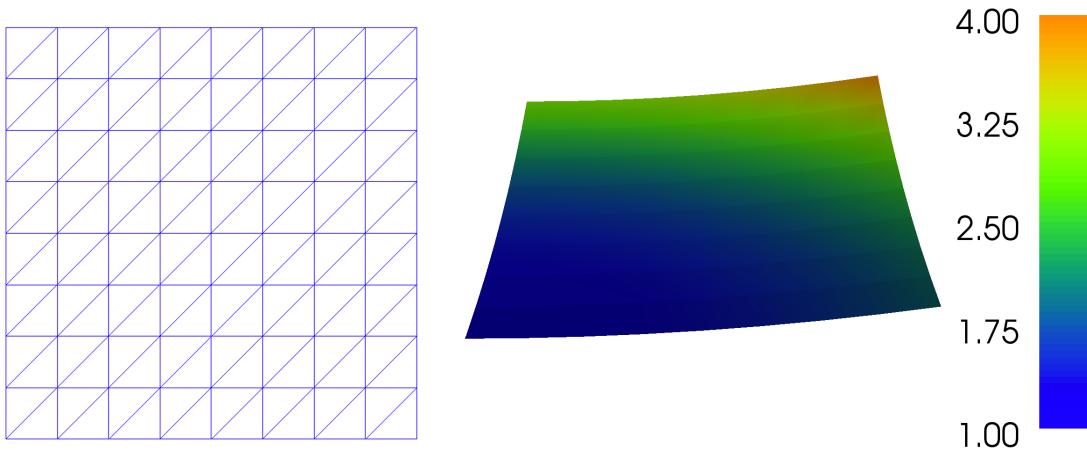


Figure 2.1: 使用内置的 FEniCS 可视化工具 (plot 命令) 创建的网格绘图和 Poisson 问题的解决方案。

有关详细的文档, 请运行命令 `help(plot)` Python 或 `pydoc fenics.plot` 从终端窗口。

注意

(Mac OS X 和 Docker 内置绘图) FEniCS 中的内置绘图可能无法按预期方式工作在 Mac OS X 上运行或在 FEniCS Docker 容器内运行时。FEniCS 支持使用 Mac OS X 上的 `plot` 命令进行绘图但是, 键盘快捷键可能会失败上班。当在 Docker 容器内运行时, 绘图不是因为 Docker 不与您的窗口系统进行交互。对于需要绘图的 Docker 用户, 建议您进行任何工作在 Jupyter/FEniCS notebook 中 (命令 `fenicsproject notebook`) 或依靠 ParaView 或其他外部工具进行可视化。

2.3.10 使用 ParaView 绘制解决方案

简单的 `plot` 命令对于快速可视化是有用的, 但是对于更高级的可视化必须使用外部工具。在这个我们演示如何在 ParaView 中可视化解决方案。

<http://www.paraview.org>²

Paraview 是一个强大的用于可视化标量和矢量场的工具, 包括那些由 FEniCS 计算。

第一步是以 VTK 格式导出解决方案:

Python code

```
1 vtkfile = File('poisson/solution.pvd')
2 vtkfile << u
```

以下步骤演示如何创建解决方案我们在 ParaView 中的 Poisson 问题。得到的情节显示在 Figure 2.2。

1. 启动 ParaView 应用程序。

²<http://www.paraview.org>

2. 单击顶部菜单中的 File–Open...，导航到包含导出解决方案的目录。这应该在 FEniCS Python 程序启动目录下面的名为 poisson 的子目录中。选择文件 solution.pvd，然后单击 OK。
3. 在左侧的 Properties 窗格中单击 Apply。这将提出解决方案。
4. 要制作解决方案的 3D 图，我们将使用 ParaView 的许多 filters 之一。单击顶部菜单中的 Filters–Alphabetical–Warp By Scalar，然后在左侧的 Properties 窗格中单击 Apply。这产生了由解值确定的高度的高架表面。
5. 要在升高的表面上显示原始图表，请单击左侧 Pipeline 浏览器窗格中 solution.pvd 左侧的小眼图标。还可以点击绘图窗口顶部的小 2D 按钮，将可视化文件更改为 3D。这可以通过旋转（鼠标左键）和缩放（Ctrl + 鼠标左键）与图形交互。
6. 要显示有限元网格，请在 Pipeline 浏览器中单击 solution.pvd，导航到 Properties 窗格中的 Representation，然后选择 Surface With Edges。这应该使有限元网格可见。
7. 要更改绘图的宽高比，请单击 Pipeline 浏览器中的 WarpByScalar1，然后导航到 Properties 窗格中的 Scale Factor。将值更改为 0.2，然后单击 Apply。这将改变扭曲情节的规模。我们还在 Properties 窗格的底部单击 Orientation Axis Visibility，以删除绘图窗口左下角的小 3D 轴。你现在应该看到类似于 Figure 2.2 的情节。
8. 最后，要将可视化文件导出到文件中，请单击 File–Save Screenshot...，然后选择一个合适的文件名，如 poisson.png。

有关更多信息，请参阅 ParaView 指南 [Paraview] （免费 PDF 可用），

http://www.paraview.org/Wiki/The_ParaView_Tutorial ParaView tutorial³

和指令 video

<https://vimeo.com/34037236> ParaView 简介⁴。

2.3.11 计算错误

最后，我们计算错误以检查解决方案的准确性。我们通过将有限元解决方案 u 与确切的比较来做到这一点解决方案，在这个例子中恰好与之相同表达 u_D 用于设置边界条件。我们计算错误有两种不同的方式。首先，我们计算 L^2 norm 错误，由... 定义

$$E = \sqrt{\int_{\Omega} (u_D - u)^2 dx}.$$

由于确切的解是二次方程和有限元解是分段线性的，这个错误将是非零。计算此错误在 FEniCS 中，我们简单地写道

1	error_L2 = errornorm(u_D, u, 'L2')	Python code
---	------------------------------------	-------------

³http://www.paraview.org/Wiki/The_ParaView_Tutorial

⁴<https://vimeo.com/34037236>

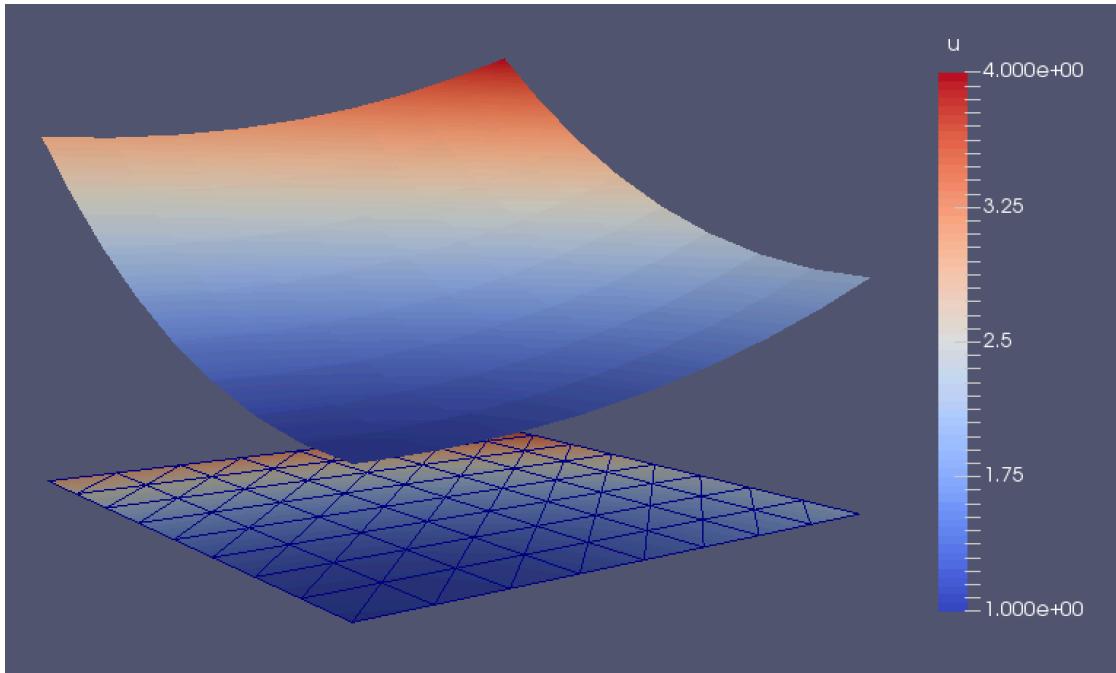


Figure 2.2: 使用 ParaView 创建的网格绘图和 Poisson 问题的解决方案。

`errornorm` 函数也可以计算其他错误规范作为 H^1 标准。在终端窗口中键入 `pydoc fenics.errornorm` 详细信息。

我们还计算所有顶点的误差的最大值有限元网格。如上所述，我们期待这个错误对于该特定示例，对于机器内部的精度为零。至计算顶点的误差，我们先要求 FEniCS 来计算两个 u_D 的值和 u 在所有顶点，然后减去结果：

```

Python code
1 vertex_values_u_D = u_D.compute_vertex_values(mesh)
2 vertex_values_u = u.compute_vertex_values(mesh)
3 import numpy as np
4 error_max = np.max(np.abs(vertex_values_u_D - vertex_values_u))
```

我们在这里使用 `numpy` 的最大和绝对值函数，因为这些对于大型阵列（因子为 30）要高得多，比 Python 的内置 `max` 和 `abs` 函数。

注意

(**如何检查错误消失**) 用不精确（浮点）算术，最大顶点的错误不是零，而应该是一个小数字。该机器精度约为 10^{-16} ，但在有限元中计算，这个大小的舍入误差可能会累积，产生大于 10^{-16} 的错误。实验表明，增加元素数量增加有限元的程度多项式增加误差。对于一个网格 $2 \times (20 \times 20)$ cubic Lagrange 元素（3 级），错误大约是 $2 \cdot 10^{-12}$ ，而对于 128 个线性元素，错误大约是 $2 \cdot 10^{-15}$ 。

2.3.12 检查自由度和顶点值

有限元函数（如 u ）表示为线性组合的基础函数 ϕ_j ，跨越空间 V ：

$$u = \sum_{j=1}^N U_j \phi_j. \quad (2.13)$$

通过在程序中写入 `solve(a == L,u,bc)`, 线性系统将会由 a 和 L 组成, 这个系统是为了解决的值 U_1, \dots, U_N 。值 U_1, \dots, U_N 被称为 degree of freedom(“dofs”) 或 nodal values 为 u 。对于 Lagrange 元素 (和许多其他元素类型) U_j 只是它的值 u 在全局号为 j 的节点上。节点的位置和对于线性的 Lagrange 元素, 单元格顶点重合, 而 for 高阶元素有与之相关联的附加节点小平面, 边缘, 有时也是细胞的内部。

将 u 表示为 Function 对象, 我们可以进行评估 $u(x)$ 在网格中的任何点 x (昂贵的操作!), 或者我们可以在矢量 U 中直接获取所有自由度

Python code

```
1 nodal_values_u = u.vector()
```

结果是一个 Vector 对象, 它基本上是一个封装在使用的线性代数包中使用的矢量对象解决变分问题引起的线性系统。由于我们在 Python 中编程, 所以转换 Vector 对象到标准 numpy 数组进行进一步处理:

Python code

```
1 array_u = nodal_values_u.array()
```

使用 numpy 数组, 我们可以编写类似 MATLAB 的代码来分析数据。索引用方括号完成: `array_u[j]`, 其中 index j 始终从 0 开始。如果解决方案是用分段线性 Lagrange 元素 (P_1), 然后是大小数组 `array_u` 等于顶点的数量, 每个 `array_u[J]` 是网格中某个顶点的值。但是, 学位的自由并不一定按照相同的方式编号的顶点目。(这在[E]5.4.1部分中有详细的讨论)。如果我们想要知道顶点的值, 我们需要调用函数 `u.compute_vertex_values` 此函数返回网格的所有顶点的值与 numpy 数组相同对于网格的顶点编号, 例如:

Python code

```
1 vertex_values_u = u.compute_vertex_values()
```

请注意, 对于 P_1 元素, `array_u` 和 `vertex_values_u` 具有相同的长度并且包含相同的值, 尽管以不同的顺序。

2.4 膜的变形

我们第一个针对 Poisson 方程的 FEniCS 程序简单的测试问题, 我们可以很容易地验证实现。我们现在把注意力转移到身体上相关问题的解决方案有些更令人兴奋的形状。

我们要计算一个二维的偏转 $D(x,y)$, 圆形膜半径为 R , 受加载 p 超过膜。适当的 PDE 模型是

$$-T\nabla^2 D = p \quad \text{in } \Omega = \{(x,y) | x^2 + y^2 \leq R\}. \quad (2.14)$$

这里, T 是膜中的张力 (常数), p 是外部的压力负荷。膜的边界没有偏移, 意味着 $D = 0$ 作为边界条件。本地化的负载可以建模为 Gaussian 函数:

$$p(x,y) = \frac{A}{2\pi\sigma} \exp\left(-\frac{1}{2}\left(\frac{x-x_0}{\sigma}\right)^2 - \frac{1}{2}\left(\frac{y-y_0}{\sigma}\right)^2\right) \quad (2.15)$$

参数 A 是压力的幅度, (x_0, y_0) 本地化的最大点的负载, 和 σ 宽度 p 。我们将把中心 (x_0, y_0) 的压力为 $(0, R_0)$ 为 $0 < R_0 < R$ 。

2.4.1 扩大方程

这个问题有很多物理参数，我们可以受益通过缩放对它们进行分组。让我们介绍无量纲坐标 $\bar{x} = x/R$, $\bar{y} = y/R$ 和无量纲偏移 $w = D/D_c$, 其中 D_c 是特征尺寸偏转。介绍 $\bar{R}_0 = R_0/R$, 我们得到

$$-\frac{\partial^2 w}{\partial \bar{x}^2} - \frac{\partial^2 w}{\partial \bar{y}^2} = \alpha \exp(-\beta^2(\bar{x}^2 + (\bar{y} - \bar{R}_0)^2)) \text{ for } \bar{x}^2 + \bar{y}^2 < 1,$$

哪里

$$\alpha = \frac{R^2 A}{2\pi T D_c \sigma}, \quad \beta = \frac{R}{\sqrt{2}\sigma}.$$

通过适当的缩放, w 及其衍生物的大小统一, 所以缩放 PDE 的左侧是大小统一, 而右侧则有 α 作为其特征尺寸。这表明选择 α 是统一的, 或者是统一的。我们应该在这种特殊情况下, 选择 $\alpha = 4$ 。(也可以找到分析解决方案在缩放坐标并显示最大值偏差 $D(0,0)$ 是 D_c , 如果我们选择 $\alpha = 4$ 来确定 D_c 。) 使用 $D_c = AR^2/(8\pi\sigma T)$ 并放弃我们获得的条缩放的问题

$$-\nabla^2 w = 4 \exp(-\beta^2(x^2 + (y - R_0)^2)), \quad (2.16)$$

在边界上用 $w = 0$ 在单位盘上解决。现在只有两个参数可以改变: 无量纲的程度压力, β 和本地化的压力峰值 $R_0 \in [0,1]$ 。作为 $\beta \rightarrow 0$, 解决方案将接近特殊情况 $w = 1 - x^2 - y^2$ 。

给定一个计算的缩放解 w , 物理偏差可以通过计算

$$D = \frac{AR^2}{8\pi\sigma T} w.$$

我们以前的程序需要进行一些修改才能解决这个新问题。

2.4.2 定义网格

单元磁盘上的网格可以由 mshr 工具创建 FENICS:

Python code

```
1 from mshr import *
2 domain = Circle(Point(0, 0), 1)
3 mesh = generate_mesh(domain, 64)
```

mshr 中的 Circle 形状占据了中心和半径圈作为参数。第二个参数为 generate_mesh 函数指定所需的网格分辨率。单元格大小将是 (大约) 等于域的直径除以解析度。

2.4.3 定义负载

右侧压力功能由 Expression 对象表示。那里是 f 的公式中的两个物理参数表达式字符串和这些参数必须设置它们的值按关键词参数:

Python code

```
1 beta = 8
2 R0 = 0.6
3 p = Expression('4*exp(-pow(beta, 2)*(pow(x[0], 2) + pow(x[1] - R0, 2)))',
4 degree=1, beta=beta, R0=R0)
```

Expression 对象中的坐标总是一个数组 x 与 components $x[0]$, $x[1]$ 和 $x[2]$ 对应 x , y 和 z 。否则我们可以自由地介绍参数的名称这些是通过关键字参数给出的默认值。一切由关键字参数初始化的参数可以随时拥有值修改。例如，我们可以设置

Python code

```
1 p.beta = 12
2 p.R0 = 0.3
```

2.4.4 定义变分问题

变分问题和边界条件与我们的第一个 Poisson 问题，但是我们可能会引用 w 而不是 u primary unknown 和 p 而不是 f 作为右侧功能：

Python code

```
1 w = TrialFunction(V)
2 v = TestFunction(V)
3 a = dot(grad(w), grad(v))*dx
4 L = p*v*dx
5
6 w = Function(V)
7 solve(a == L, w, bc)
```

2.4.5 绘制解决方案

有兴趣的可视化压力 p 以及偏移 w ，以便我们可以检查膜对于的反应压力。我们必须将公式 (Expression) 转换为有限元函数 (Function)。最自然的方法是构造自由度有限元函数从 p 计算。也就是说，我们将 p 内插到函数空间 V ：

Python code

```
1 p = interpolate(p, V)
```

请注意， p 的赋值会破坏以前的 Expression object p ，所以如果有兴趣还可以访问这个对象，返回的 Function 对象必须使用另一个名称通过 `interpolate`。可以绘制两个函数 w 和 p 使用内置绘图命令：

Python code

```
1 plot(w, title='Deflection')
2 plot(p, title='Load')
```

像以前一样，我们还以 VTK 格式导出解决方案 ParaView 中的可视化

Python code

```
1 vtkfile_w = File('poisson_membrane/deflection.pvd')
2 vtkfile_w << w
3 vtkfile_p = File('poisson_membrane/load.pvd')
4 vtkfile_p << p
```

图 2.3 显示可视化的偏移 w 和使用 ParaView 创建的 load p 。

2.4.6 通过域进行曲线图

比较偏转和负载的另一种方法是制作曲线图沿线 $x = 0$ 。这只是定义一组点的问题沿着 y -axis 并评估有限元函数 w 和 p 在这些点上：

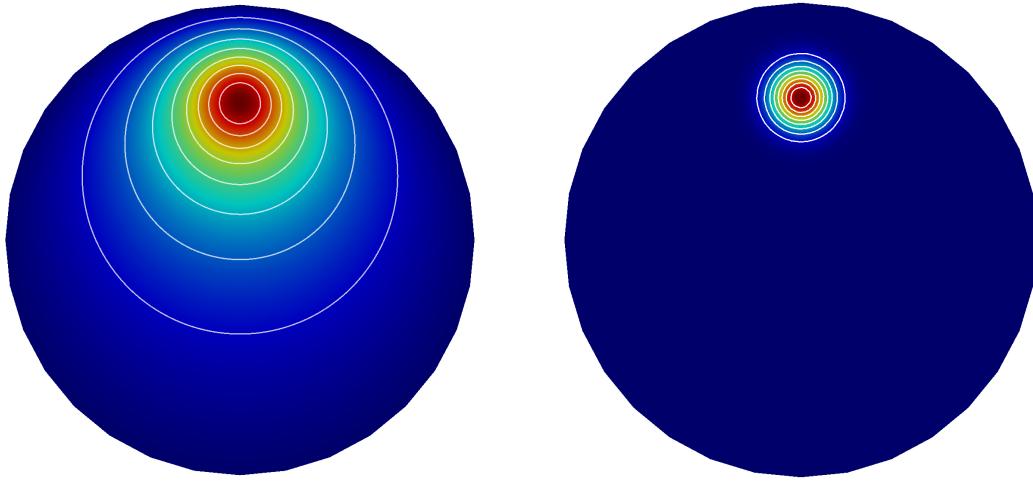


Figure 2.3: 使用 ParaView 创建的膜问题的曲线图 (左) 和负载 (右)。该图使用 10 个等间距等值线作为解值和可选的 jet 色彩映射。

Python code

```

1 # Curve plot along x = 0 comparing p and w
2 import numpy as np
3 import matplotlib.pyplot as plt
4 tol = 0.001 # avoid hitting points outside the domain
5 y = np.linspace(-1 + tol, 1 - tol, 101)
6 points = [(0, y_) for y_ in y] # 2D points
7 w_line = np.array([w(point) for point in points])
8 p_line = np.array([p(point) for point in points])
9 plt.plot(y, 50*w_line, 'k', linewidth=2) # magnify w
10 plt.plot(y, p_line, 'b--', linewidth=2)
11 plt.grid(True)
12 plt.xlabel('$y$')
13 plt.legend(['Deflection ($\times 50$)', 'Load'], loc='upper left')
14 plt.savefig('poisson_membrane/curves.pdf')
15 plt.savefig('poisson_membrane/curves.png')
```

该示例程序可以在文件

https://fenicsproject.org/pub/tutorial/python/vol1/ft02_poisson_membrane.py

中找到。

得到的曲线图如图 2.4 所示。本地化输入 (p) 很大在输出中减弱和平滑 (w)。这反映了一个典型的 Poisson 方程的属性。

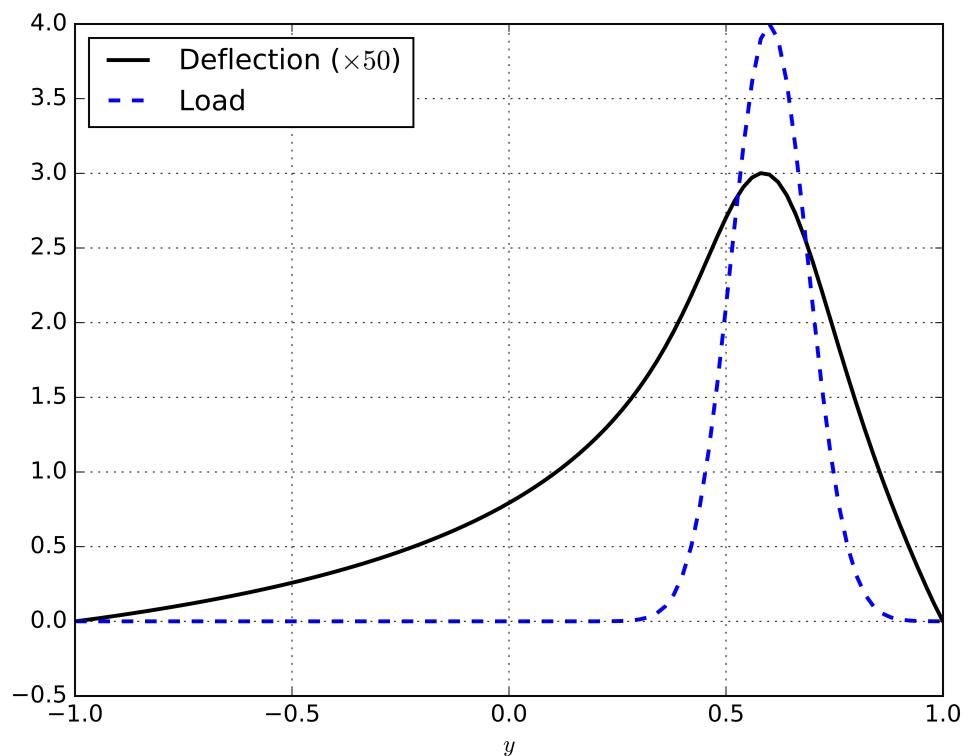


Figure 2.4: 使用 Matplotlib 创建的膜问题的偏转和负载曲线，并沿着 y -axis 对两个函数进行抽样。

3. 有限元求解器画廊

3.1	热方程式	30
3.2	非线性 Poisson 方程	37
3.3	线性弹性方程	40
3.4	Navier–Stokes 方程	45
3.5	平流–扩散–反应方程组	58

本章的目标是展示一系列重要的内容科学与工程学院的 PDEs 可以很快得到解决 FEniCS 代码行。我们从热方程开始，继续用非线性 Poisson 方程，线性方程弹性，Navier-Stokes 方程式，最后看看如何解决非线性对流 - 扩散反应的系统方程。这些问题说明了如何解决时间依赖性问题问题，非线性问题，向量值问题和系统 PDEs。对于每个问题，我们得出变分公式在 Python 中以非常类似的方式表达问题数学。

3.1 热方程式

作为前一章的 Poisson 问题的第一个扩展，我们考虑时间依赖热方程，或时间依赖性扩散方程。这是 Poisson 的自然延伸描述体内热量的固定分布的方程式时间依赖的问题。

我们会看到，通过将时间离散成小时间间隔采用标准时间步法，可以解决热量通过求解一系列变分问题的方程式，很像一个我们遇到的 Poisson 方程。

3.1.1 PDE 问题

我们的时间依赖型 PDE 的模型问题读取

$$\frac{\partial u}{\partial t} = \nabla^2 u + f \quad \text{in } \Omega \times (0, T], \quad (3.1)$$

$$u = u_D \quad \text{on } \partial\Omega \times (0, T], \quad (3.2)$$

$$u = u_0 \quad \text{at } t = 0. \quad (3.3)$$

在这里， u 随空间和时间而变化，例如，如果空间，则 $u = u(x, y, t)$ 域 Ω 是二维的。源函数 f 和边界值 u_D 也可能因空间和时间而异。初始条件 u_0 仅是空间的函数。

3.1.2 变化公式

一个简单的方法来解决时间依赖的 PDEs 有限元方法是首先离散时间导数 a 有限差分近似，产生一个序列固定问题，然后将每个静止问题转化为变分公式

让上标 n 表示时间 t_n 的数量，其中 n 是一个整数计数时间级别。例如， u^n 表示 u 在时间级 n 。时间上的有限差分离散化包括在某个时间级别对 PDE 进行采样，说 t_{n+1} :

$$\left(\frac{\partial u}{\partial t}\right)^{n+1} = \nabla^2 u^{n+1} + f^{n+1}. \quad (3.4)$$

时间导数可以通过差商近似。为了简单和稳定的原因，我们选择一个简单向后的差异:

$$\left(\frac{\partial u}{\partial t}\right)^{n+1} \approx \frac{u^{n+1} - u^n}{\Delta t}, \quad (3.5)$$

其中 Δt 是时间离散参数。 (3.4) 插入 (3.5) 产生

$$\frac{u^{n+1} - u^n}{\Delta t} = \nabla^2 u^{n+1} + f^{n+1}. \quad (3.6)$$

这是我们的时间离散版本的热方程 (3.1) ，所谓的 backward Euler 或 implicit Euler 离散化。

我们可以重新排序 (3.6) 左侧包含未知 u^{n+1} 的条款右侧仅包含计算的条件。结果是假设为 u^{n+1} 的空间（静态）问题的序列 u^n 从以前的时间步长是已知的:

$$u^0 = u_0, \quad (3.7)$$

$$u^{n+1} - \Delta t \nabla^2 u^{n+1} = u^n + \Delta t f^{n+1}, \quad n = 0, 1, 2, \dots \quad (3.8)$$

给定 u_0 ，我们可以解决 u^0, u^1, u^2 等。

(3.8) 的替代方法，可以是方便实施，是收集平等标志一方的所有术语:

$$u^{n+1} - \Delta t \nabla^2 u^{n+1} - u^n - \Delta t f^{n+1} = 0, \quad n = 0, 1, 2, \dots \quad (3.9)$$

我们使用有限元方法来解决 (3.7) 和任一方程式 (3.8) 或 (3.9) 。这个需要将方程式变为弱形式。像往常一样，我们倍增通过一个测试函数 $v \in \hat{V}$ 并将二阶导数合并部分。在 u^{n+1} (这是自然的) 中引入符号 u 程序)，由此产生的弱势形式配方 (3.8) 可以方便地写入标准符号:

$$a(u, v) = L_{n+1}(v),$$

哪里

$$a(u, v) = \int_{\Omega} (uv + \Delta t \nabla u \cdot \nabla v) dx, \quad (3.10)$$

$$L_{n+1}(v) = \int_{\Omega} (u^n + \Delta t f^{n+1}) v dx. \quad (3.11)$$

替代形式 (3.9) 有一个抽象配方

$$F_{n+1}(u; v) = 0,$$

哪里

$$F_{n+1}(u; v) = \int_{\Omega} (uv + \Delta t \nabla u \cdot \nabla v - (u^n + \Delta t f^{n+1})v) dx. \quad (3.12)$$

除了每个时间步长要解决的变化问题外，我们还需要近似初始条件 (3.7) 。这个方程也可以变成 a 变分问题:

$$a_0(u, v) = L_0(v),$$

哪里

$$a_0(u, v) = \int_{\Omega} uv \, dx, \quad (3.13)$$

$$L_0(v) = \int_{\Omega} u_0 v \, dx. \quad (3.14)$$

当解决这个变分问题时, u^0 成为 L^2 将给定的初始值 u_0 投影到有限元中空间。另一种方式是通过内插来构造 u^0 初始值 u_0 ; 也就是说, 如果 $u^0 = \sum_{j=1}^N U_j^0 \phi_j$, 我们只需设置 $U_j = u_0(x_j, y_j)$, 其中 (x_j, y_j) 是坐标节点编号 j 。我们将这两个策略称为计算初始条件通过投影或插值。都在 FEniCS 中通过单一语句轻松计算操作, 使用 project 或 interpolate 函数。最常见的选择是 project, 它计算一个近似值为 u_0 , 但是在一些我们想通过再现来验证代码的应用程序确切的解决方案, 必须使用 interpolate(我们使用这样的测试这里的问题!)。

总之, 我们需要解决以下变分序列计算热方程有限元解的问题: 找到 $u^0 \in V$, 以便 $a_0(u^0, v) = L_0(v)$ 对于所有 $v \in \hat{V}$, 然后找到 $u^{n+1} \in V$ 使得 $a(u^{n+1}, v) = L_{n+1}(v)$ 对于 $v \in \hat{V}$, 或者, 对于 $v \in \hat{V}$ 中的 $F_{n+1}(u^{n+1}, v) = 0$, $n = 0, 1, 2, \dots$ 。

3.1.3 FEniCS 实现

我们的程序需要手动实现时间步长, 但可以依靠 FEniCS 轻松计算 a_0 , L_0 , a 和 L (或 F_{n+1}), 并解决未知数的线性系统。

测试问题 1: 一个已知的分析解决方案。 就像前一章的 Poisson 问题一样, 我们构建一个测试问题, 使其容易确定是否计算正确。既然我们知道我们的一流时间步长方案对于线性函数是精确的, 我们创建一个测试时间线性变化的问题。我们把这与一个空间二次变化。我们因此而来

$$u = 1 + x^2 + \alpha y^2 + \beta t, \quad (3.15)$$

其产生了在节点处的计算值将是的函数确切的, 不管元素的大小和 Δt , 只要网格被均匀分割。插入 (3.15) 转换为热方程 (3.1), 我们发现右边的 f 必须由 $f(x, y, t) = \beta - 2 - 2\alpha$ 给出。边界值是 $u_D(x, y, t) = 1 + x^2 + \alpha y^2 + \beta t$ 和初始值为 $u_0(x, y) = 1 + x^2 + \alpha y^2$ 。

FEniCS 实现。 一个新的编程问题是如何处理不同的功能空间和时间, 如边界条件 $u_D(x, y, t) = 1 + x^2 + \alpha y^2 + \beta t$ 。一个自然的解决方案是使用 FEniCS Expression 与 time t 作为参数, 除了参数 α 和 β :

Python code

```
1 alpha = 3; beta = 1.2
2 u_D = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t',
3 degree=2, alpha=alpha, beta=beta, t=0)
```

Expression 将 x 的组件用作独立的变量, 而 α , β 和 t 是参数。该 time t 可以稍后更新

Python code

```
1 u_D.t = t
```

在这种情况下，沿着整个边界的基本边界条件，以与我们以前实施的相同的方式实施泊松问题的边界条件：

Python code

```

1 def boundary(x, on_boundary):
2     return on_boundary
3
4 bc = DirichletBC(V, u_D, boundary)

```

我们将使用变量 u 作为新的未知 u^{n+1} 时间步长和变量 u_n 对于上一次的 u^n 步。 u_n 的初始值可以通过任一投影来计算或插值 u_0 。由于我们为边界值设置了 $t = 0$ u_D ，我们可以使用 u_D 指定初始条件：

Python code

```

1 u_n = project(u_D, V)
2 # or
3 u_n = interpolate(u_D, V)

```

注意

(投影与内插初始条件) 实际恢复确切的解决方案 (3.15) 到机器精度，这很重要通过内插 u_0 来计算离散的初始条件。这个确保自由度是准确的（机器精度）在 $t = 0$ 。投影导致节点处的近似值。

我们可以根据上述公式定义 a 或 L ，或者我们可能只是定义 F ，并要求 FEniCS 找出哪些术语应该去进入双线性形式 a ，并且应该进入线性形式 L 。后者是方便的，特别是在更复杂的问题，所以我们说明了建设 a 和 L ：

Python code

```

1 u = TrialFunction(V)
2 v = TestFunction(V)
3 f = Constant(beta - 2 - 2*alpha)
4
5 F = u*v*dx + dt*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx
6 a, L = lhs(F), rhs(F)

```

最后，我们在循环中执行时间步长：

Python code

```

1 u = Function(V)
2 t = 0
3 for n in range(num_steps):
4
5     # Update current time
6     t += dt
7     u_D.t = t
8
9     # Solve variational problem
10    solve(a == L, u, bc)
11
12    # Update previous solution
13    u_n.assign(u)

```

在时间步长循环的最后一步，我们分配值变量 u （新计算的解决方案）变量 u_n 包含上一个时间步长的值。必须这样做使用 `assign` 成员函数。如果我们反而尝试做 $u_n = u$ ，我们将设

置 u_n 变量与 u 变量相同这不是我们想要的。(我们需要两个变量, 一个是值在前一个时间步长, 另一个是当前时间的值步。)

注意

(记住用当前时间更新表达式对象!) 在时间循环内, 观察 $u_D.t$ 必须在之前更新 solve 语句强制执行 Dirichlet 条件的计算当前时间步长。一个 Dirichlet 条件定义为 Expression 查找并应用参数的值, 如 t 当它被评估并应用于线性系统时。

上面的时间步进循环不包含任何比较数字和确切的解决方案, 我们必须包括为了验证实施。至于 Poisson 方程式部分 2.3, 我们计算差异在 u 的节点数组和节点数组之间内插精确解的值。这可以做到如下:

Python code

```

1 u_e = interpolate(u_D, V)
2 error = np.abs(u_e.vector().array() - u.vector().array()).max()
3 print('t = %.2f: error = %.3g' % (t, error))

```

对于 Poisson 示例, 我们使用了这个函数 `compute_vertex_values` 提取功能值顶点。这里我们举例说明一种提取方法顶点值, 通过调用函数 `vector` 返回自由度向量。对于 P_1 功能空间, 这个矢量的自由度将等于通过调用获得的顶点值数组 `compute_vertex_values`, 虽然可能有不同的顺序。

解决热方程的完整程序如下:

Python code

```

1 from fenics import *
2 import numpy as np
3
4 T = 2.0          # final time
5 num_steps = 10    # number of time steps
6 dt = T / num_steps # time step size
7 alpha = 3         # parameter alpha
8 beta = 1.2        # parameter beta
9
10 # Create mesh and define function space
11 nx = ny = 8
12 mesh = UnitSquareMesh(nx, ny)
13 V = FunctionSpace(mesh, 'P', 1)
14
15 # Define boundary condition
16 u_D = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t',
17                  degree=2, alpha=alpha, beta=beta, t=0)
18
19 def boundary(x, on_boundary):
20     return on_boundary
21
22 bc = DirichletBC(V, u_D, boundary)
23
24 # Define initial value
25 u_n = interpolate(u_D, V)
26 #u_n = project(u_D, V)
27
28 # Define variational problem
29 u = TrialFunction(V)
30 v = TestFunction(V)
31 f = Constant(beta - 2 - 2*alpha)

```

```

32 F = u*v*dx + dt*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx
33 a, L = lhs(F), rhs(F)
34
35
36 # Time-stepping
37 u = Function(V)
38 t = 0
39 for n in range(num_steps):
40
41     # Update current time
42     t += dt
43     u_D.t = t
44
45     # Compute solution
46     solve(a == L, u, bc)
47
48     # Plot solution
49     plot(u)
50
51     # Compute error at vertices
52     u_e = interpolate(u_D, V)
53     error = np.abs(u_e.vector().array() - u.vector().array()).max()
54     print('t = %.2f: error = %.3g' % (t, error))
55
56     # Update previous solution
57     u_n.assign(u)
58
59 # Hold plot
60 interactive()

```

该示例程序可以在文件中找到

https://fenicsproject.org/pub/tutorial/python/vol1/ft03_heat.py

测试问题 2: Gaussian 函数的扩散。 让我们现在解决一个更有趣的测试问题, 即扩散一个 Gaussian 山。我们以初始值为准

$$u_0(x, y) = e^{-ax^2 - ay^2}$$

在 $a = 5$ 的域 $[-2, 2] \times [2, 2]$ 。为了这问题我们将使用均匀的 Dirichlet 边界条件 ($u_D = 0$)。

FEniCS 实现。以前的程序需要修改哪些? 一个专业更改是域不再是单位平方。新域名可以使用 RectangleMesh 在 FEniCS 中轻松创建:

Python code

```

1 nx = ny = 30
2 mesh = RectangleMesh(Point(-2, -2), Point(2, 2), nx, ny)

```

请注意, 我们比以前使用了比以前更高的分辨率解决解决方案的功能。我们还需要重新定义初始条件和边界条件。两者都很容易改变定义一个新的 Expression, 并在边界上设置 $u = 0$ 。

能够在外部程序中可视化解决方案, 如 ParaView, 我们将每次将解决方案保存为 VTK 格式的文件步。我们首先用后缀.pvd 创建一个 File:

Python code

```

1 vtkfile = File('heat_gaussian/solution.pvd')

```

在时间循环中，我们可能会将解值附加到这个文件：

Python code

```
1  vtkfile << (u, t)
```

在每个时间步骤中调用此行，从而创建一个包含后缀.vtu 的新文件，其中包含时间步长的所有数据（网格和顶点值）。文件 heat_gaussian/solution.pvd 将包含时间值和引用.vtu 文件，这意味着.pvd 文件将是单个小文件指向大量.vtu 文件包含实际数据。请注意，我们选择存储解决方案到一个名为heat_gaussian的子目录。这是为了避免混乱我们的源目录与所有生成的数据文件。在运行之前不需要创建目录程序将由 FEniCS 自动创建。

完整的程序如下所示。

Python code

```
1 from fenics import *
2 import time
3
4 T = 2.0           # final time
5 num_steps = 50    # number of time steps
6 dt = T / num_steps # time step size
7
8 # Create mesh and define function space
9 nx = ny = 30
10 mesh = RectangleMesh(Point(-2, -2), Point(2, 2), nx, ny)
11 V = FunctionSpace(mesh, 'P', 1)
12
13 # Define boundary condition
14 def boundary(x, on_boundary):
15     return on_boundary
16
17 bc = DirichletBC(V, Constant(0), boundary)
18
19 # Define initial value
20 u_0 = Expression('exp(-a*pow(x[0], 2) - a*pow(x[1], 2))',
21                  degree=2, a=5)
22 u_n = interpolate(u_0, V)
23
24 # Define variational problem
25 u = TrialFunction(V)
26 v = TestFunction(V)
27 f = Constant(0)
28
29 F = u*v*dx + dt*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx
30 a, L = lhs(F), rhs(F)
31
32 # Create VTK file for saving solution
33 vtkfile = File('heat_gaussian/solution.pvd')
34
35 # Time-stepping
36 u = Function(V)
37 t = 0
38 for n in range(num_steps):
39
40     # Update current time
41     t += dt
42
43     # Compute solution
44     solve(a == L, u, bc)
```

```

46 # Save to file and plot solution
47 vtkfile << (u, t)
48 plot(u)
49
50 # Update previous solution
51 u_n.assign(u)
52
53 # Hold plot
54 interactive()

```

该示例程序可以在文件中找到

https://fenicsproject.org/pub/tutorial/python/vol1/ft04_heat_gaussian.py

ParaView 中的可视化。 为了可视化高斯山的扩散，启动 ParaView，选择 File–Open...，打开 heat_gaussian/solution.pvd，然后点击 Apply 在 Properties 窗格中。点击播放按钮进行显示解决方案的动画。要将动画保存到文件中，请单击 File–Save Animation... 并将文件保存为所需的文件格式，例如 AVI 或 Ogg/Theora。动画一旦保存到文件中，就可以播放动画离线使用播放器，如 mplayer 或 VLC，或上传您的动画到 YouTube。图 3.1 显示了一个序列的解决方案的快照。

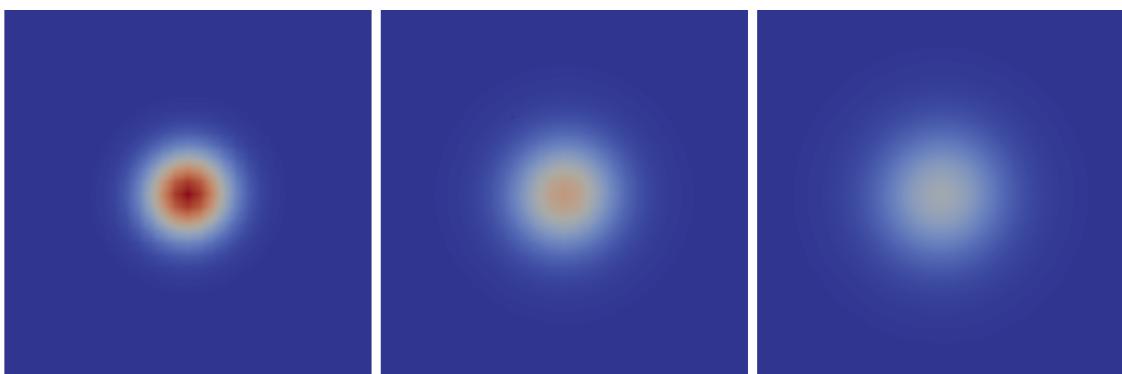


Figure 3.1: 使用 ParaView 创建的 Gaussian 山丘问题的解决方案的一系列快照。

3.2 非线性 Poisson 方程

现在我们将介绍如何解决非线性 PDE 问题。我们会看到非线性问题可以像线性问题那样容易地解决 FEniCS，通过简单地定义非线性变分问题和调用 solve 功能。当这样做时，我们会遇到一个微妙的变分问题如何定义的差异。

3.2.1 PDE 问题

作为解决非线性 PDE 的模型问题，我们取以下非线性 Poisson 方程：

$$-\nabla \cdot (q(u) \nabla u) = f, \quad (3.16)$$

在 Ω 中， $u = u_D$ 在边界 $\partial\Omega$ 上。系数 $q = q(u)$ 使得方程非线性（除非 $q(u)$ 在 u 中是不变的）。

3.2.2 变化公式

像往常一样，我们的 PDE 乘以一个测试函数 $v \in \hat{V}$ ，整合域，并整合二阶导数按部件。由零件整合产生的边界积分无论我们使用 Dirichlet 条件，都会消失。所结果的我们的模型问题的变分公式变成：找到 $u \in V$ 就这样

$$F(u; v) = 0 \quad \forall v \in \hat{V}, \quad (3.17)$$

哪里

$$F(u; v) = \int_{\Omega} (q(u) \nabla u \cdot \nabla v - f v) dx, \quad (3.18)$$

和

$$\begin{aligned} V &= \{v \in H^1(\Omega) : v = u_D \text{ on } \partial\Omega\}, \\ \hat{V} &= \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}. \end{aligned}$$

离散的问题像往常一样通过限制 V 和 \hat{V} 出现到一对离散空间。像以前一样，我们省略了任何下标离散空间和离散解。然后将离散的非线性问题写为：找到 $u \in V$

$$F(u; v) = 0 \quad \forall v \in \hat{V}, \quad (3.19)$$

与 $u = \sum_{j=1}^N U_j \phi_j$ 。由于 F 是非线性的 u ，变量语句产生一个系统未知数 U_1, \dots, U_N 中的非线性代数方程。

3.2.3 FEniCS 实现

测试问题。 为了解决测试问题，我们需要选择右边的 f ，系数 $q(u)$ 和边界值 u_D 。以前，我们与制造的解决方案一起工作，可以无需复制近似误差这在非线性问题上更为困难，代数更加繁琐。但是，我们可以利用 SymPy 符号计算，并将这些计算集成到 FEniCS 中求解。这允许我们轻松地尝试不同的制造解决方案。SymPy 的即将到来的代码需要一些基本熟悉这个包。特别是我们会用 SymPy 函数 diff 用于符号区分和 ccode for C/C++ 代码生成。

我们取 $q(u) = 1 + u^2$ 并定义二维制造 x 和 y 中的线性解决方案：

Python code

```

1 # Warning: from fenics import * will import both 'sym' and
2 # 'q' from FEniCS. We therefore import FEniCS first and then
3 # overwrite these objects.
4 from fenics import *
5
6 def q(u):
7     "Return nonlinear coefficient"
8     return 1 + u**2
9
10 # Use SymPy to compute f from the manufactured solution u
11 import sympy as sym
12 x, y = sym.symbols('x[0], x[1]')
13 u = 1 + x + 2*y
14 f = - sym.diff(q(u)*sym.diff(u, x), x) - sym.diff(q(u)*sym.diff(u, y), y)
15 f = sym.simplify(f)
16 u_code = sym.printing.ccode(u)
17 f_code = sym.printing.ccode(f)

```

```
18 | print('u =', u_code)
19 | print('f =', f_code)
```

注意

(在 Expression 对象中根据需要定义符号坐标) 请注意，我们通常会写 $x,y = \text{sym.symbols('x,y')}$ ，但是如果我们希望生成的表达式具有有效的语法 FEniCS Expression 对象，我们必须使用 $x[0]$ 和 $x[1]$ 。通过定义 x 的名称，sympy 可以轻松实现这一点 $y \text{ as } x[0] \text{ 和 } x[1]:x,y = \text{sym.symbols('x[0], x[1]')}$ 。

将 u 和 f 的表达式转换为 C 或 C++ 语法 FEniCS Expression 对象需要两个步骤。首先，我们要求 C 表达式的代码：

Python code

```
1 u_code = sym.printing.ccode(u)
2 f_code = sym.printing.ccode(f)
```

在某些情况下，需要编辑结果以匹配所需的结果 Expression 对象的语法，但不是这种情况。(首要的例如 M_{PI} 对于 π 在 C/C++ 中必须由 pi 替代 Expression 对象。) 在本例中， u_code 的输出和 f_code 是

$x[0] + 2*x[1] + 1 - 10*x[0] - 20*x[1] - 10$ 定义网格，功能空间和边界后，我们定义边界值 u_D 如

Python code

```
1 u_D = Expression(u_code, degree=1)
```

类似地，我们定义右边的函数

Python code

```
1 f = Expression(f_code, degree=1)
```

注意

(命名 FEniCS 与程序变量之间的冲突) 在上面的程序中，可能会出现奇怪的错误名字冲突。如果在做之前定义 sym 和 q 从 $\text{fenics import } *$ ，后一个语句也会导入变量名为 sym 和 q ，覆盖你以前定义的对象！这可能会导致奇怪错误。最安全的解决方案是做 import fenics 而不是 $\text{from fenics import } *$ ，然后前缀所有 FEniCS fenics 的对象名称。下一个最好的解决办法就是做 $\text{from fenics import } *$ 首先定义你自己的变量覆盖从 fenics 导入的那些。这是可以接受的如果我们不需要 fenics 中的 sym 和 q 。

FEniCS 实现。非线性 Poisson 方程的求解器很容易实现线性 Poisson 方程的求解器。我们所需要做的就是说明 F 和调用的公式 $\text{solve}(F == 0, u, bc)$ ，而不是 $\text{solve}(a == L, u, bc)$ 在线性情况下。这是一个简约代码：

Python code

```
1 from fenics import *
2
3 def q(u):
4     return 1 + u**2
5
6 mesh = UnitSquareMesh(8, 8)
```

```

7 |V = FunctionSpace(mesh, 'P', 1)
8 u_D = Expression(u_code, degree=1)
9
10 def boundary(x, on_boundary):
11     return on_boundary
12
13 bc = DirichletBC(V, u_D, boundary)
14
15 u = Function(V)
16 v = TestFunction(V)
17 f = Expression(f_code, degree=1)
18 F = q(u)*dot(grad(u), grad(v))*dx - f*v*dx
19
20 solve(F == 0, u, bc)

```

该示例程序的完整版本可以在该文件中找到

https://fenicsproject.org/pub/tutorial/python/vol1/ft05_poisson_nonlinear.py

与线性问题的主要区别在于未知功能 u 在非线性情况下的变分形式必须定义为 `Function`, 而不是 `TrialFunction`。在某种意义上这是从线性情况的简化, 我们必须定义 u 首先作为 `TrialFunction` 然后作为 `Function`。

`solve` 函数采用非线性方程, 从符号出发 Jacobian 矩阵, 并运行一个 Newton 方法来计算解。

当我们运行代码时, FEniCS 会报告 Newton 的进度迭代。以 $2 \cdot (8 \times 8)$ 单元格, 我们达到 8 的收敛具有容忍度 10^{-9} 的迭代, 以及错误数值解约为 10^{-16} 。这些结果带来了证据为正确的实施。思考有限差异在均匀网格上, P_1 元素模拟标准二阶差分, 其计算线性的导数或二次函数。在这里, ∇u 是一个常量向量, 但是然后乘以 $(1 + u^2)$, 它是二阶多项式 x 和 y , 分歧“差异运算符”应该是准确计算。因此, 我们可以使用 P_1 元素, 期望制造的 u 被转载数值方法。像 $1 + u^4$ 这样的非线性, 这不会是情况, 我们需要验证收敛率。

目前的例子显示了解决非线性问题的难度程度在 FEniCS。但是, 专家对非线性的数值解 PDEs 非常了解, 非线性的自动化程序可能会失败问题, 通常需要有更好的手册解决方案的流程控制比我们目前的流程要好案件。我们在 [ftut2] 中返回此问题, 并显示如何实现非线性方程式的定制解算法我们如何引导使用的自动化 Newton 方法中的参数以上。

3.3 线性弹性方程

结构分析是现代主要活动之一工程, 这可能使 PDE 建模变形弹性体是世界上最流行的 PDE。只需一个用于解决 FEniCS 中 2D 或 3D 弹性方程的代码页, 具体细节如下。

3.3.1 PDE 问题

控制身体的小弹性变形的方程式 Ω 的方程式可以写成

$$-\nabla \cdot \sigma = f \text{ in } \Omega, \quad (3.20)$$

$$\sigma = \lambda \operatorname{tr}(\varepsilon) I + 2\mu\varepsilon, \quad (3.21)$$

$$\varepsilon = \frac{1}{2} (\nabla u + (\nabla u)^\top), \quad (3.22)$$

其中 σ 是压力张量, f 是每单位的身体力量卷, λ 和 μ 是 Lamé's 弹性参数为物品在 Ω , I 是身份张量, tr 是跟踪运算符在张量上, ε 是对称应变率张量 (对称梯度), u 是位移矢量场。我们这里假设各向同性弹性条件。

我们结合在一起 (3.21) 和 (3.22) 获得

$$\sigma = \lambda(\nabla \cdot u)I + \mu(\nabla u + (\nabla u)^\top). \quad (3.23)$$

注意 (3.20)–(3.22) 可以轻松地转换为 u 的单个向量 PDE，这是管理未知 u (Navier 方程) 的 PDE。在里面然而，变分式的推导是方便的保持方程式如上所述。

3.3.2 变化公式

变分公式 (3.20)–(3.22) 包括形成内部产品 (3.20) 和一个 vector 测试函数 $v \in \hat{V}$ ，其中 \hat{V} 是向量值的测试函数空间，整合域 Ω :

$$-\int_{\Omega} (\nabla \cdot \sigma) \cdot v \, dx = \int_{\Omega} f \cdot v \, dx.$$

由于 $\nabla \cdot \sigma$ 包含主要的二阶导数未知 u ，我们将这个术语整合在一起:

$$-\int_{\Omega} (\nabla \cdot \sigma) \cdot v \, dx = \int_{\Omega} \sigma : \nabla v \, dx - \int_{\partial\Omega} (\sigma \cdot n) \cdot v \, ds,$$

冒号运算符是张量之间的内积 (相加所有元素的成对产品)， n 是向外单位正常在边界。数量 $\sigma \cdot n$ 被称为牵引力或应力矢量在边界，并经常规定作为边界条件。我们在这里假定它是在部分规定的 $\partial\Omega_T$ 的边界为 $\sigma \cdot n = T$ 。在上剩下的一部分边界，我们假设的价值位移作为 Dirichlet 条件给出。我们得到了

$$\int_{\Omega} \sigma : \nabla v \, dx = \int_{\Omega} f \cdot v \, dx + \int_{\partial\Omega_T} T \cdot v \, ds.$$

插入表达式 (3.23) σ 给出 u 的变体形式为未知。请注意剩余部分的边界积分由于 Dirichlet， $\partial\Omega \setminus \partial\Omega_T$ 消失条件。

我们现在可以将变分公式总结为: 找到 $u \in V$

$$a(u, v) = L(v) \quad \forall v \in \hat{V}, \quad (3.24)$$

哪里

$$a(u, v) = \int_{\Omega} \sigma(u) : \nabla v \, dx, \quad (3.25)$$

$$\sigma(u) = \lambda(\nabla \cdot u)I + \mu(\nabla u + (\nabla u)^\top), \quad (3.26)$$

$$L(v) = \int_{\Omega} f \cdot v \, dx + \int_{\partial\Omega_T} T \cdot v \, ds. \quad (3.27)$$

可以显示对称张量 A 和 a 的内积反对称张量 B 消失。如果我们表示 ∇v 作为总和的对称和反对称部分，只有对称部分在产品 $\sigma : \nabla v$ 中生存 σ 是 a 对称张量。因此用对称渐变替换 ∇u $\epsilon(u)$ 产生略微不同的变化形式

$$a(u, v) = \int_{\Omega} \sigma(u) : \epsilon(v) \, dx, \quad (3.28)$$

其中 $\epsilon(v)$ 是 ∇v 的对称部分:

$$\epsilon(v) = \frac{1}{2} (\nabla v + (\nabla v)^\top).$$

配方 (3.28) 是自然而然的是由弹性势能的最小化引起的流行的配方比 (3.25)。

3.3.3 FEniCS 实现

测试问题。 作为一个测试例子，我们将模拟在其下变形的夹紧梁自重在 3D。这可以通过设置右侧来建模每单位体积的身体力量 $f = (0, 0, -\rho g)$ 同 ρ 该梁的密度和 g 重力加速度。梁是长度为 L 的盒形，宽度为 W 的正方形横截面。我们在夹紧的末端设置 $u = u_D = (0, 0, 0)$, $x = 0$ 。其余的边界是无牵引力也就是说，我们设置 $T = 0$ 。

FEniCS 实现。 我们首先列出代码，然后对新的结构进行评论相比之前我们看到的例子。

Python code

```

1 from fenics import *
2
3 # Scaled variables
4 L = 1; W = 0.2
5 mu = 1
6 rho = 1
7 delta = W/L
8 gamma = 0.4*delta**2
9 beta = 1.25
10 lambda_ = beta
11 g = gamma
12
13 # Create mesh and define function space
14 mesh = BoxMesh(Point(0, 0, 0), Point(L, W, W), 10, 3, 3)
15 V = VectorFunctionSpace(mesh, 'P', 1)
16
17 # Define boundary condition
18 tol = 1E-14
19
20 def clamped_boundary(x, on_boundary):
21     return on_boundary and x[0] < tol
22
23 bc = DirichletBC(V, Constant((0, 0, 0)), clamped_boundary)
24
25 # Define strain and stress
26
27 def epsilon(u):
28     return 0.5*(nabla_grad(u) + nabla_grad(u).T)
29 #return sym(nabla_grad(u))
30
31 def sigma(u):
32     return lambda_*nabla_div(u)*Identity(d) + 2*mu*epsilon(u)
33
34 # Define variational problem
35 u = TrialFunction(V)
36 d = u.geometric_dimension() # space dimension
37 v = TestFunction(V)
38 f = Constant((0, 0, -rho*g))
39 T = Constant((0, 0, 0))
40 a = inner(sigma(u), epsilon(v))*dx
41 L = dot(f, v)*dx + dot(T, v)*ds
42
43 # Compute solution
44 u = Function(V)
45 solve(a == L, u, bc)
46
47 # Plot solution
48 plot(u, title='Displacement', mode='displacement')
```

```

49
50 # Plot stress
51 s = sigma(u) - (1./3)*tr(sigma(u))*Identity(d) # deviatoric stress
52 von_Mises = sqrt(3./2*inner(s, s))
53 V = FunctionSpace(mesh, 'P', 1)
54 von_Mises = project(von_Mises, V)
55 plot(von_Mises, title='Stress intensity')
56
57 # Compute magnitude of displacement
58 u_magnitude = sqrt(dot(u, u))
59 u_magnitude = project(u_magnitude, V)
60 plot(u_magnitude, 'Displacement magnitude')
61 print('min/max u: ',
62     u_magnitude.vector().array().min(),
63     u_magnitude.vector().array().max())

```

该示例程序可以在文件中找到

https://fenicsproject.org/pub/tutorial/python/vol1/ft06_elasticity.py

向量函数空间。 主要的未知数现在是矢量字段 u 而不是标量字段，所以我们需要使用向量函数空间：

Python code

```

1 V = VectorFunctionSpace(mesh, 'P', 1)

```

使用 $u = \text{Function}(V)$ ，我们得到 u 作为向量值有限元功能与这个 3D 问题的三个组件。

恒定向量。 对于边界条件 $u = (0, 0, 0)$ ，我们必须设置一个向量值为零，不仅仅是一个标量。这样的向量常数被指定为 $\text{Constant}((0, 0, 0))$ 在 FEniCS 中。相应的 2D 代码将使用 $\text{Constant}((0, 0))$ 。在代码中，我们还需要 f 作为向量并指定为 $\text{Constant}((0, 0, \rho * g))$ 。

nabla_grad 漾变和差异运算符现在有一个前缀 **nabla_** 这在目前的问题上是绝对不必要的，但是一般推荐由连续力学引起的载体 PDE，如果您将 ∇ 解释为 PDE 表示法中的向量；看到关于 **nabla_grad** 的框在 3.4.2 中。

压力计算。 一旦计算了位移 u ，我们可以计算各种压力措施。我们将计算 von Mises 应力定义为 $\sigma_M = \sqrt{\frac{3}{2}s : s}$ 其中 s 是偏差应力张量

$$s = \sigma - \frac{1}{3}\text{tr}(\sigma)I.$$

这些公式和 FEniCS 代码之间存在一对一的映射：

Python code

```

1 s = sigma(u) - (1./3)*tr(sigma(u))*Identity(d)
2 von_Mises = sqrt(3./2*inner(s, s))

```

von_Mises 变量现在是必须预计的表达式有限元空间，我们可以看到它：

Python code

```

1 V = FunctionSpace(mesh, 'P', 1)
2 von_Mises = project(von_Mises, V)
3 plot(von_Mises, title='Stress intensity')

```

缩放。 经常有利于缩小问题，因为它减少了设置的需要物理参数，一个获得无量纲的数字反映参数和物理效应的竞争。我们开发具有尺寸的原始模型的代码，并运行缩放通过适当调整参数的问题。缩放减少本应用程序的活动参数数目为 6 到 2。

在 Navier 的 u 方程中，由插入产生 (3.21) 和 (3.22) 成 (3.20)，

$$-(\lambda + \mu)\nabla(\nabla \cdot u) - \mu\nabla^2 u = f,$$

我们插入由 L 制成的维度坐标， $\bar{u} = u/U$ ，这导致无量纲控制方程

$$-\beta\bar{\nabla}(\bar{\nabla} \cdot \bar{u}) - \bar{\nabla}^2 \bar{u} = \bar{f}, \quad \bar{f} = (0, 0, \gamma),$$

其中 $\beta = 1 + \lambda/\mu$ 是无量纲弹性参数哪里

$$\gamma = \frac{\rho g L^2}{\mu U}$$

是反映负载比率的无量纲变量 ρg 和剪切应力在 PDE 中的期限 $\mu\nabla^2 u \sim \mu U/L^2$ 。

缩放的一个选项是选择 U ，使 γ 为单位大小 ($U = \rho g L^2 / \mu$)。然而，在弹性方面，这导致到几何尺寸的位移，这使得地块看起来很奇怪因此，我们想要特征位移是几何特征长度的一小部分。这可以通过选择 U 等于最大偏转来实现实际上存在一个公式: $U = \frac{3}{2}\rho g L^2 \delta^2/E$ ，其中 $\delta = L/W$ 是参数反映梁的细长度， E 是模数的弹性。因此，无量纲参数 δ 非常在问题上重要(如预期的那样，因为 $\delta \gg 1$ 是什么给出光束理论!)。以 E 为 μ 相同的顺序，这是许多材料的情况，我们意识到 $\gamma \sim \delta^{-2}$ 是一个合适的选择。试验代码在变形的地块中找到“看起来正确”的位移几何，指向 $\gamma = 0.4\delta^{-2}$ 作为我们的最终选择 γ 。

模拟代码实现了维度的问题物理参数 λ , μ , ρ , g , L 和 W 。但是，我们可以轻松地重用这个代码来解决一个缩放的问题：只需设置 $\mu = \rho = L = 1$, W 如 $W/L (\delta^{-1})$, $g = \gamma$ 和 $\lambda = \beta$ 。

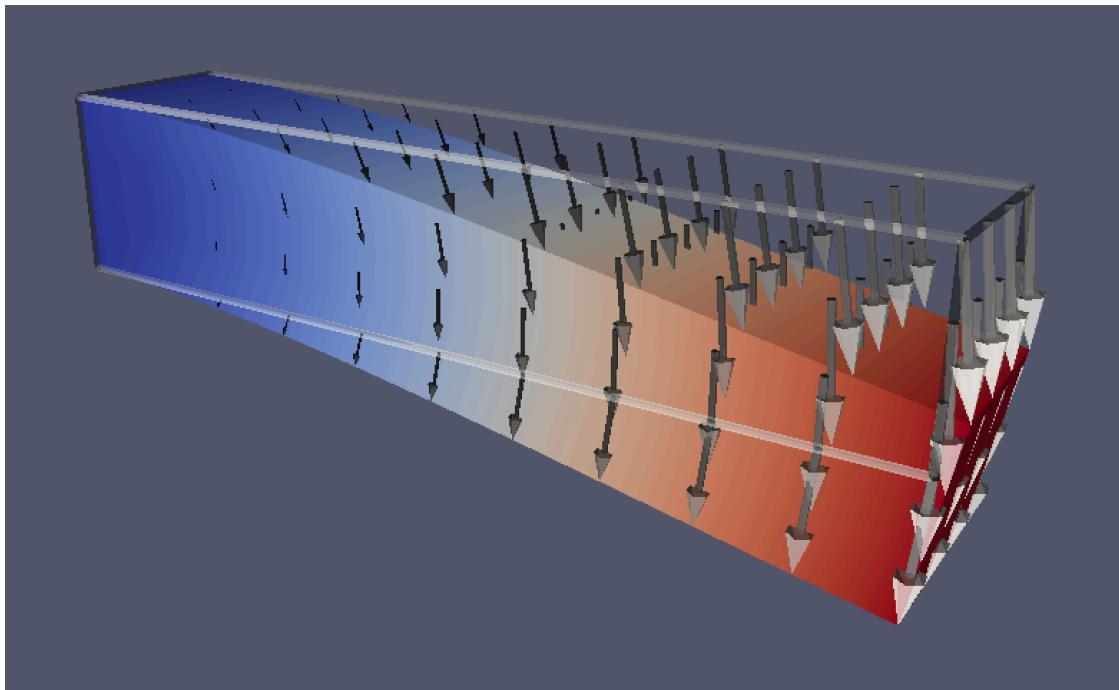


Figure 3.2: 用于弹性问题的夹紧梁中的重力引起的偏转曲线。

3.4 Navier–Stokes 方程

对于下一个例子，我们将解决不可压缩的 Navier–Stokes 方程。这个问题结合了我们的许多挑战以前研究过的问题：时间依赖，非线性和向量值变量。我们将介绍一些 FEniCS 主题，其中许多相当先进。但你会看到，即使是相对的复杂算法如二阶分割方法不可压缩的 Navier–Stokes 方程，可以实现 FEniCS 相对容易。

3.4.1 PDE 问题

不可压缩的 Navier–Stokes 方程组成一个方程组对于不可压缩流体的速度 u 和压力 p ：

$$\rho \left(\frac{\partial u}{\partial t} + u \cdot \nabla u \right) = \nabla \cdot \sigma(u, p) + f, \quad (3.29)$$

$$\nabla \cdot u = 0. \quad (3.30)$$

右边的 f 是每单位体积的给定力量就线性弹性方程而言， $\sigma(u, p)$ 表示应力张量，对于 Newtonian 流体是（谁）给的

$$\sigma(u, p) = 2\mu\epsilon(u) - pI, \quad (3.31)$$

其中 $\epsilon(u)$ 是应变率张量

$$\epsilon(u) = \frac{1}{2} (\nabla u + (\nabla u)^T).$$

参数 μ 是动态粘度。注意势头方程 (3.29) 非常类似于弹性方程 (3.20)。区别在于两个附加条款 $\rho(\partial u / \partial t + u \cdot \nabla u)$ 和不同的表达式为应力张量。两个额外的术语表达加速平衡的力量 $F = \nabla \cdot \sigma + f$ 每单位体积在 Newton 第二运动定律。

3.4.2 变化公式

Navier–Stokes 方程不同我们需要解决一个系统的时间依赖热方程方程式和这个系统是一种特殊的类型。如果我们应用相同的技术为热方程；那就是更换时间导数具有简单的差商，我们得到一个非线性方程组。这在本身对于 FEniCS 来说不是一个问题，我们在第 3.2 部分中看到，但系统有一个所谓的 saddle point structure，需要特殊技术（特殊预处理器和迭代方法）有效解决。

相反，我们将应用一种更为简单和经常非常有效的方法，称为 splitting method。这个想法是考虑两个方程 (3.29) 和 (3.30)。存在许多分裂不可压缩 Navier–Stokes 方程的策略。其中一个最古老的是由 Chorin [Chorin1968] 提出的方法 Temam [Temam1969]，通常称为 Chorin 的方法。我们会使用修改版本的 Chorin 的方法，即所谓的增量式压力校正计划 (IPCS) 由于 [Goda1979] 给出与原来的方案相比，提高了准确度成本。

IPCS 方案涉及三个步骤。首先，我们计算一个暂定的速度 u^* 通过推进动量方程 (3.29) 通过中点有限差分方案时间，但是使用压力 p^n 以前的时间间隔。我们还将线性化非线性对流通过使用已知速度 u^n 从上一个时间步长： $u^n \cdot \nabla u^n$ 。第一步的变分问题是

$$\begin{aligned} & \langle \rho(u^* - u^n)/\Delta t, v \rangle + \langle \rho u^n \cdot \nabla u^n, v \rangle + \langle \sigma(u^{n+\frac{1}{2}}, p^n), \epsilon(v) \rangle \\ & + \langle p^n n, v \rangle_{\partial\Omega} - \langle \mu \nabla u^{n+\frac{1}{2}} \cdot n, v \rangle_{\partial\Omega} = \langle f^{n+1}, v \rangle. \end{aligned} \quad (3.32)$$

这种符号，适用于变化中许多术语的问题配方，需要一些解释。首先我们用短手符号

$$\langle v, w \rangle = \int_{\Omega} vw \, dx, \quad \langle v, w \rangle_{\partial\Omega} = \int_{\partial\Omega} vw \, ds.$$

这使我们能够更加紧凑地表达变分问题办法。其次，我们使用符号 $u^{n+\frac{1}{2}}$ 。这个符号通常指的是间隔中点的 u 的值近似于算术平均值：

$$u^{n+\frac{1}{2}} \approx (u^n + u^{n+1})/2.$$

第三，我们注意到变分问题 (3.32) 源于该术语的部分整合 $\langle -\nabla \cdot \sigma, v \rangle$ 。就像弹性问题一样段 3.3，我们得到

$$\langle -\nabla \cdot \sigma, v \rangle = \langle \sigma, \epsilon(v) \rangle - \langle T, v \rangle_{\partial\Omega},$$

其中 $T = \sigma \cdot n$ 是边界牵引力。如果我们解决了有一个自由边界的问题，我们可以取 $T = 0$ 边界。但是，如果我们计算通过通道或管道的流量并想要将持续进入“虚拟通道”的流模型化外流，我们需要小心对待这个词。假设那么我们就是制造出速度的方向导数的通道在流出时为零，对应于流量“完全发达”或者不显着下降外流。这样做，流出的剩余边界就变成了 $p_n - \mu \nabla u \cdot n$ ，这个术语出现在变数问题 (3.32)。注意这个论点和实现取决于 ∇u 的确切定义组件 $\partial u_i / \partial x_j$ 或 $\partial u_j / \partial x_i$ 。我们这里选择后者， $\partial u_j / \partial x_i$ ，这意味着我们必须使用 FEniCS 运算符 `nabla_grad` 为执行。如果我们使用 `grad` 运算符和定义 $\partial u_i / \partial x_j$ ，我们必须保留条款 $p_n - \mu (\nabla u)^T \cdot n$ ！

注意

(`grad(u)` vs. `nabla_grad(u)`) 对于标量函数， ∇u 具有清晰的含义作为向量

$$\nabla u = \left(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial u}{\partial z} \right).$$

然而，如果 u 是向量值，则意义不太清楚。一些源定义 ∇u 作为具有元素的矩阵 $\partial u_j / \partial x_i$ ，而其他来源则喜欢 $\partial u_i / \partial x_j$ 。在 FEniCS 中，`grad(u)` 被定义为矩阵与元素 $\partial u_i / \partial x_j$ ，这是 ∇u 的自然定义，如果我们将其视为渐变或 u 的衍生。这样，可以应用矩阵 ∇u 一个差额 dx 给出增量 $du = \nabla u \, dx$ 。因为 ∇u 作为矩阵的替代解释与元素 $\partial u_j / \partial x_i$ 是非常常见的，在特别在连续力学，FEniCS 提供操作员 `nabla_grad` 以此目的。对于 Navier-Stokes 方程，重要的是考虑术语 $u \cdot \nabla u$ 应该被解释为向量 w 与元素 $w_i = \sum_j \left(u_j \frac{\partial}{\partial x_j} \right) u_i = \sum_j u_j \frac{\partial u_i}{\partial x_j}$ 。这个术语可以在 FEniCS 中实现 `grad(u)*u`，因为这是表达式 $\sum_j \partial u_i / \partial x_j u_j$ ，或 as `dot(u,nabla_grad(u))` 因为这个表达式变成了 $\sum_i u_i \partial u_j / \partial x_i$ 。我们将使用符号 `dot(u,nabla_grad(u))` 以下，因为它更紧密地对应到标准符号 $u \cdot \nabla u$ 。更准确地说，PDE 有三种不同的符号涉及梯度，发散和卷曲算子。一个雇用 `grad u`，`div u` 和 `curl u` 运营商。另一个雇用 ∇u 作为同义词 `grad u`， $\nabla \cdot u$ 表示 `div u` 和 $\nabla \times u$ 是 `curl u` 的名称。第三使用 ∇u ， $\nabla \cdot u$ 和 $\nabla \times u$ 其中 ∇ 是 vector，例如 ∇u 是一个二进制表达式： $(\nabla u)_{i,j} = \partial u_j / \partial x_i = (\text{grad } u)^T$ 。后一个符号，以 ∇ 为一向量运算符，在连续统计中得出方程时通常很方便力学，如果这个解释 ∇ 是基础你的 PDE，你必须使用 `nabla_grad`，`nabla_div`，和 `nabla_curl` 在 FEniCS 代码作为这些运算符兼容二进制计算。从 Navier-Stokes 方程，我们可以很容易地看到什么 ∇ 表示：如果对流术语的形式为 $u \cdot \nabla u$ ，实际上意思是 $(u \cdot \nabla) u$ ，那么 ∇ 是矢量和实现变成 `dot(u,nabla_grad(u))` 在 FEniCS，但是如果我们将看到 $\nabla u \cdot u$ 或 $(\text{grad } u) \cdot u$ ，相应的 FEniCS 表达式为 `dot(grad(u),u)`。

类似地，张量场的发散如应力张量 σ 也可以以两种不同的方式表达，或者 `div(sigma)` 或 `nabla_div(sigma)` 第一种情况对应于组件 $\partial \sigma_{ij} / \partial x_j$ 和第二个 $\partial \sigma_{ij} / \partial x_i$ 。一般来说，这些表达式将会有所不同，但当压力测量是对称的时候表达式具有相同的值。

我们现在进入我们分裂计划的第二步不可压缩的 Navier–Stokes 方程。在第一步，我们计算基于压力的初步速度 u^* 以前的时间步。我们现在可以使用计算的暂定速度计算新压力 p^n :

$$\langle \nabla p^{n+1}, \nabla q \rangle = \langle \nabla p^n, \nabla q \rangle - \Delta t^{-1} \langle \nabla \cdot u^*, q \rangle. \quad (3.33)$$

请注意， q 是压力的标量值测试函数空间，而测试函数 v 在 (3.32) 是一个来自速度空间的向量值测试函数。

考虑这一步的一个方法是减去 Navier–Stokes 动量方程 (3.29) 表示为暂定速度 u^* 和压力 p^n 来自以速度 u^{n+1} 表示的动量方程压力 p^{n+1} 。这导致等式

$$(u^{n+1} - u^*)/\Delta t + \nabla p^{n+1} - \nabla p^n = 0. \quad (3.34)$$

采取分歧，并要求 $\nabla \cdot u^{n+1} = 0$ 由 Navier–Stokes 连续性方程 (3.30)，我们获得方程 $-\nabla \cdot u^*/\Delta t + \nabla^2 p^{n+1} - \nabla^2 p^n = 0$ ，这是一个 Poisson 问题的压力 p^{n+1} 导致变量问题 (3.33)。

最后，我们从公式计算出校正的速度 u^{n+1} (3.34)。将该方程乘以测试函数 v ，我们获得

$$\langle u^{n+1}, v \rangle = \langle u^*, v \rangle - \Delta t \langle \nabla(p^{n+1} - p^n), v \rangle. \quad (3.35)$$

总而言之，我们可以解决不可压缩的 Navier–Stokes 通过求解三个线性变分的序列有效地计算方程每个时间步的问题。

3.4.3 FEniCS 实现

测试问题 1: 通道流。 作为第一个测试问题，我们计算两个无限大的流量板，所谓渠道或 Poiseuille 流。我们将看到，这个问题有一个已知的分析解决方案。让 H 成为距离盘间和 L 之间的通道长度。没有身体力量

我们可能首先扩大问题，摆脱看似独立的问题物理参数。这个问题的物理学是由粘性效应仅在垂直于流动的方向，所以 a 时间尺度应基于通道上的扩散: $t_c = H^2/\nu$ 。我们让 U ，一些特征流入速度是速度尺度和 H 空间尺度。采取压力秤作为特征剪切应力， $\mu U/H$ ，因为这是主要的剪切流动的例子。插入 $\bar{x} = x/H$, $\bar{y} = y/H$, $\bar{z} = z/H$, $\bar{u} = u/U$, $\bar{p} = Hp/(\mu U)$ ，方程式中的 $\bar{t} = H^2/\nu$ 导致缩放的 Navier-Stokes 方程式 (缩放后的条形):

$$\begin{aligned} \frac{\partial u}{\partial t} + \text{Re } u \cdot \nabla u &= -\nabla p + \nabla^2 u, \\ \nabla \cdot u &= 0. \end{aligned}$$

这里， $\text{Re} = \rho U H / \mu$ 是 Reynolds 数。因为的时间和压力尺度，这是不同的对流主导的流体流量，Reynolds 数量相关联具有对流项，而不是粘度项。

通过假设 $u = (u_x(x, y, z), 0, 0)$ ，得出确切的解 x 轴指向通道。由于 $\nabla \cdot u = 0$ ， u 不能依赖 x 。渠道流动的物理学也是二维，所以我们可以说省略 z 坐标 (更准确地说: $\partial/\partial z = 0$)。在 (缩放) 中插入 $u = (u_x, 0, 0)$ 控制方程式给出 $u_x(y) = \partial p / \partial x$ 。相对于 x 区分这个方程式表示 $\partial^2 p / \partial^2 x = 0$ 所以 $\partial p / \partial x$ 是一个常量，这里叫 $-\beta$ 。这是开车流量的力和可以指定为已知参数问题。在通道的宽度上集成 $u_x''(y) = -\beta$ [0, 1]，并要求 $u = (0, 0, 0)$ 在渠道墙，导致 $u_x = \frac{1}{2}\beta y(1-y)$ 。特征入口速度 U 可以作为 $y = 1/2$ 的最大流入，意味着 $\beta = 8$ 。频道的长度， L/H 在缩放模型，对结果没有影响，所以为了简单，我们只是计算在单位广场上。在数学上，必须规定压力在某一点上，由于 p 不依赖于 y ，我们可以将 p 设置为 a 沿着出口边界 $x = 1$ 的已知值，例如 0。结果是 $p(x) = 8(1-x)$ 和 $u_x = 4y(1-y)$ 。

边界条件可以设为 $p = 8$ 在 $x = 0$, $p = 0$ 在 $x = 1$ 和 $u = (0, 0, 0)$ 在墙上 $y = 0, 1$ 。这定义了压降并应导致入口和出口处的单位最大速度抛物线速度曲线，无进一步规格。注意解决 2D

中的 Navier–Stokes 方程式是有意义的 3D 几何，虽然潜在的数学问题崩溃了到两个 1D 问题，一个用于 $u_x(y)$ ，一个用于 $p(x)$ 。

缩放模型不容易使用标准模拟 Navier–Stokes 求解器。但是，人们可以争辩说对流项为零，所以这个术语前面的 Re 系数在缩放的 PDE 中并不重要，可以设置为一致。在那里在原始 Navier–Stokes 方程中设置 $\rho = \mu = 1$ 类似于缩放模型。

对于具体的工程问题，想要模拟一个特定的流体并设置相应的参数。因此，一般的解算器最自然地实现与尺寸和原始的物理参数。然而，缩放可以大大简化数值模拟。首先，它表明所有的流体都在同样的方式：我们是否有石油，天然气或水两块板之间流动，流动速度无关紧要是（达到某些批评价值的 Reynolds 号码的流量变得不稳定并转变成复杂的湍流完全不同的性质）。这意味着一个模拟就足够了涵盖所有类型的渠道流程！在其他应用程序中，缩放表明可能需要设置一些分数参数（无量纲数）而不是参数他们自己。这简化了探索输入参数空间往往是模拟的目的。通常，缩放的问题是通过将一些输入参数的尺寸设置为固定来运行价值观（通常统一）。

FEniCS 实现。 我们以前的例子都是从创建网格开始的然后在网格上定义一个 FunctionSpace。为了 Navier–Stokes 分裂方案我们将需要定义两个函数空间，一个为速度和一个为压力：

Python code

```
1 V = VectorFunctionSpace(mesh, 'P', 2)
2 Q = FunctionSpace(mesh, 'P', 1)
```

第一个空格 V 是速度的向量值函数空间而第二个空格 Q 是一个标量值的函数空间压力。我们使用分段二次元素作为速度分段线性元件用于压力。创建时 VectorFunctionSpace 在 FEniCS 中，值维（的长度向量）将被设置为等于该几何尺寸有限元网格。可以轻松创建向量值函数通过添加关键字参数，FEniCS 中的其他维度的空格 dim ：

Python code

```
1 V = VectorFunctionSpace(mesh, 'P', 2, dim=10)
```

注意

(**Navier–Stokes 方程的稳定有限元空间**) 众所周知，某些有限元空间不稳定对于 Navier–Stokes 方程，或者甚至用于更简单的 Stokes 方程。有限元不稳定对的典型例子空间是为了使用第一度连续分段多项式速度和压力。使用一个不稳定的空间对通常会产生一个解决方案虚假（想要的，非物理的）振荡的压力解。简单的补救办法是连续使用速度和连续分段线性的二次元素元素为压力。这些元素一起形成所谓的 Taylor-Hood 元素。杂散振荡也可能发生如果使用不稳定的元素对，则可以使用拆分方法。

由于我们有两个不同的函数空间，我们需要创建两个集合的试用和测试功能：

Python code

```
1 u = TrialFunction(V)
2 v = TestFunction(V)
3 p = TrialFunction(Q)
4 q = TestFunction(Q)
```

正如我们在前面的例子中所看到的，边界可能被定义在 FEniCS 通过定义返回 `True` 或 `False` 的 Python 函数取决于一个点是否应该被视为一部分边界

Python code

```
1 def boundary(x, on_boundary):
2     return near(x[0], 0)
```

该函数定义了所有点的边界 x - 坐标等于接近零。near 函数来自 FEniCS 并执行公差测试:abs($x[0] - 0$) < 3E-16 所以我们不要遇到烦恼。或者,我们可以给边界定义作为一个字符串的 C++ 代码,很像我们有先前定义的表达式,如 $u_D = \text{Expression}'1 + x[0]*x[0] + 2*x[1]*x[1]', \text{degree}=2$ 。上述边界的定义 Python 函数的术语可能由简单的 C ++ 代替串:

Python code

```
1 boundary = 'near(x[0], 0)'
```

这具有移动哪些节点的计算的优点属于从 Python 到 C++ 的边界,这提高了效率的程序。

对于当前的例子,我们将设置三个不同的边界条件。首先,我们将在通道的墙壁设置 $u = 0$;也就是说,在 $y = 0$ 和 $y = 1$ 。其次,我们将设置 $p = 8$ 流入 ($x = 0$), 最后流出 $p = 0$ ($x = 1$)。这个将导致压力梯度,从而加速流动零速度的初始状态。这些边界条件可能是定义如下:

Python code

```
1 # Define boundaries
2 inflow = 'near(x[0], 0)'
3 outflow = 'near(x[0], 1)'
4 walls = 'near(x[1], 0) || near(x[1], 1)'
5
6 # Define boundary conditions
7 bcu_noslip = DirichletBC(V, Constant((0, 0)), walls)
8 bcp_inflow = DirichletBC(Q, Constant(8), inflow)
9 bcp_outflow = DirichletBC(Q, Constant(0), outflow)
10 bcu = [bcu_noslip]
11 bcp = [bcp_inflow, bcp_outflow]
```

最后,我们收集速度的边界条件 Python 列表中的压力,所以我们可以轻松访问它们以下计算。

我们现在转向变分形式的定义。有要定义三个变量问题,一个为每个步骤 IPCS 方案。我们来看一下第一个变分的定义问题。我们从一些常数开始:

Python code

```
1 U = 0.5*(u_n + u)
2 n = FacetNormal(mesh)
3 f = Constant((0, 0))
4 k = Constant(dt)
5 mu = Constant(mu)
6 rho = Constant(rho)
```

下一步是设置第一步的变分形式 (3.32)。由于变异问题包含已知和未知数量的混合,我们将使用以下命名约定: u 是未知数 (数学上 u^{n+1}) 作为变体形式的试用函数, u_- 是个最近计算的近似值 (u^{n+1} 可用作为 Function 对象), u_n 是 u^n , 同样的约定 p , p_- (p^{n+1}) 和 $p_n(P^N)$ 。

Python code

```
1 # Define strain-rate tensor
2 def epsilon(u):
3     return sym(nabla_grad(u))
4
5 # Define stress tensor
6 def sigma(u, p):
7     return 2*mu*epsilon(u) - p*Identity(len(u))
8
9 # Define variational problem for step 1
10 F1 = rho*dot((u - u_n) / k, v)*dx +
11     rho*dot(dot(u_n, nabla_grad(u_n)), v)*dx +
12     + inner(sigma(U, p_n), epsilon(v))*dx \
```

```

13 + dot(p_n*n, v)*ds - dot(mu*nabla_grad(U)*n, v)*ds \
14 - dot(f, v)*dx
15 a1 = lhs(F1)
16 L1 = rhs(F1)

```

请注意，我们利用 Python 编程语言定义我们自己的运算符 sigma 和 epsilon。以这种方式使用 Python 可以轻松扩展 FEniCS 的数学语言特殊经营者和组织法。

还要注意，FEniCS 可以整理双线性形式 $a(u, v)$ 和线性形式 $L(v)$ 形式由 lhs 和 rhs 函数。这在更长的时间里是特别方便的更复杂的变化形式。

分裂方案需要解决三个序列变化问题在每个时间步。我们以前用过了内置的 FEniCS 函数 solve 来解决变分问题。下引擎盖，当用户调用 solve(a == L, u, bc) 时，FEniCS 会执行以下步骤：

Python code

```

1 A = assemble(A)
2 b = assemble(L)
3 bc.apply(A, b)
4 solve(A, u.vector(), b)

```

在最后一步中，FEniCS 使用重载的 solve 函数来解决线性系统 $AU = b$ 其中 U 是度的向量函数的自由 $u(x) = \sum_{j=1} U_j \phi_j(x)$ 。

在执行分裂计划时，我们将利用这些低级命令首先组装然后调用解决。这个具有我们可以控制何时组装和我们的优势解决线性系统。特别是，由于矩阵为三个变数问题都是时间无关的，这是有道理的在时间循环之外组合它们一次：

Python code

```

1 A1 = assemble(a1)
2 A2 = assemble(a2)
3 A3 = assemble(a3)

```

在时间步进循环中，我们然后可以组装右侧向量，应用边界条件，并调用求解功能如下这三个步骤中的第一步：

Python code

```

1 # Time-stepping
2 t = 0
3 for n in range(num_steps):
4
5     # Update current time
6     t += dt
7
8     # Step 1: Tentative velocity step
9     b1 = assemble(L1)
10    [bc.apply(b1) for bc in bcu]
11    solve(A1, u_.vector(), b1)

```

注意在 bcu 中的 bc 的 Python list comprehension [bc.apply(b1)] 它遍历列表 bcu 中的所有 bc。这是一个方便和紧凑的方式来构建一个适用的循环所有边界条件在一条线上。此外，代码工作如果我们在未来添加更多的 Dirichlet 边界条件。注意边界条件只需要应用于右边向量，因为它们已经被应用于矩阵（未示出）。

最后，我们来看一下我们如何使用参数的重要细节例如我们变分的定义中的时间步长 dt 问题。因为我们以后可能想改变这些，例如，如果我们想要尝试更小或更大的时间步骤，我们包装这些使用 FEniCS Constant：

Python code

```
1 |k = Constant(dt)
```

FEniCS 中的矩阵和向量的汇编基于代码代。这意味着每当我们改变一个变数问题时，FEniCS 将必须生成新的代码，这可能需要一点时间时间。新的代码也将生成和编译时的浮点值因为时间的改变。通过使用此参数包装 Constant，FEniCS 将把该参数视为通用常量不是一个特定的数值，它可以防止重复的代码代。在时间步长的情况下，我们选择一个新的名字 k 而不是 dt 为 Constant，因为我们也想使用变量 dt 作为 Python float 作为时间步长的一部分。

用于使用 FEniCS 模拟 2D 通道流的完整代码可以在文件中找到

https://fenicsproject.org/pub/tutorial/python/vol1/ft07_navier_stokes_channel.py

验证。 我们计算节点上的错误，就像我们之前做过的那样来验证我们的实施是正确的。我们的 Navier-Stokes 求解器计算解决时间依赖的不可压缩 Navier-Stokes 方程式，从初始条件 $u = (0, 0)$ 开始。我们有没有在我们的求解器中明确指定初始条件意味着 FEniCS 将初始化所有变量，特别是以前的和当前的速度 u_n 和 u ，为零。自从精确解是二次的，我们期望解决方案是准确的在无限时间的节点处的机器精度。对于我们实现时，错误快速接近零，大约为零 10^{-6} 在时间 $T = 10$ 。

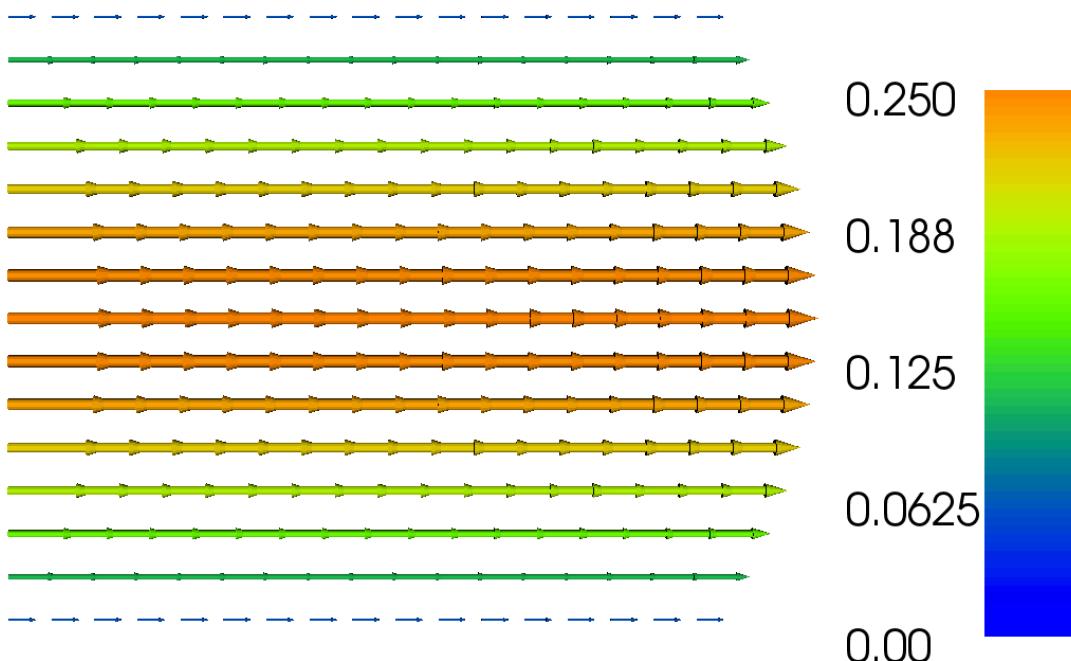


Figure 3.3: Navier - Stokes 通道流程示例的最终时间速度曲线图。

测试问题 2：流过气瓶。 现在我们把注意力转向一个更具挑战性的问题：流程经过一个圆柱体。几何和参数取自问题 DFG 2D-2 在

http://www.featflow.de/en/benchmarks/cfdbenchmarking/flow/dfg_benchmark2_re100.html
FEATFLOW/1995-DFG benchmark suite¹

¹http://www.featflow.de/en/benchmarks/cfdbenchmarking/flow/dfg_benchmark2_re100.html

并且如图 3.4 所示。运动粘度由 $\nu = 0.001 = \mu/\rho$ 给出，流入速度分布为指定为

$$u(x, y, t) = \left(1.5 \cdot \frac{4y(0.41 - y)}{0.41^2}, 0 \right),$$

其最大幅度为 1.5， $y = 0.41/2$ 。我们不使用任何缩放来解决这个问题，因为所有的精确参数都是已知的。

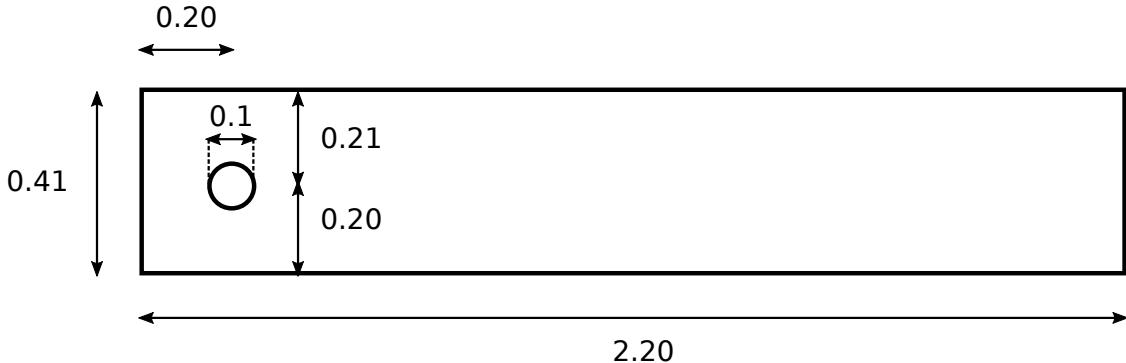


Figure 3.4: 流过气缸测试问题的几何。注意微扰和不对称几何。

FEniCS 实现。 到目前为止，我们所有的领域都是简单的形状，如单位面积或一个矩形盒子。可以创建许多这样简单的网格在 FEniCS 中使用内置的网格类 (UnitIntervalMesh, UnitSquareMesh, UnitCubeMesh, IntervalMesh, RectangleMesh, BoxMesh)。FEniCS 支持通过技术创建更复杂的网格称为 constructive solid geometry(CSG)，这使我们可以定义几何形状简单的形状（基元）和设置操作：联合，交集和设置差异。设置操作是使用运算符 +(联合), *(交集) 在 FEniCS 中编码，和-(设置差异)。要访问 FEniCS 中的 CSG 功能，必须导入提供的 FEniCS 模块 mshr FEniCS 的扩展网格划分功能。

气缸流量测试问题的几何可以很容易地定义通过首先定义矩形通道，然后减去圈：

Python code

```
1 channel = Rectangle(Point(0, 0), Point(2.2, 0.41))
2 cylinder = Circle(Point(0.2, 0.2), 0.05)
3 domain = channel - cylinder
```

然后我们可以通过调用函数 generate_mesh 创建网格：

Python code

```
1 mesh = generate_mesh(domain, 64)
```

这里的参数 64 表示我们要解决几何其直径为 64 个（通道长度）。

为了解决缸体测试问题，我们只需要做一些次要的更改我们为通道流测试编写的代码案件。除了定义新的网格，我们需要做出的唯一的改变是修改边界条件和时间步长。该边界规定如下：

Python code

```
1 inflow = 'near(x[0], 0)'
2 outflow = 'near(x[0], 2.2)'
3 walls = 'near(x[1], 0) || near(x[1], 0.41)'
4 cylinder = 'on_boundary && x[0]>0.1 && x[0]<0.3 && x[1]>0.1 && x[1]<0.3'
```

最后一行可能看起来似乎是隐秘的，然后才能抓住这个想法：我们想选择出现也位于 x2D 内的所有边界点 (on_boundary) 域 $[0.1, 0.3] \times [0.1, 0.3]$ ，参见图 3.4。唯一可能的点就是这些点循环边界！

除了这些重要的变化，我们还会做一些小的改变我们的解决方案。首先，因为我们需要选择一个相对较小的时间步长来计算解决方案（一个时间步长太大会使解决方案爆炸）我们添加了一个进度条我们可以跟随我们的计算进度。这可以做到如下：

```
Python code
1 progress = Progress('Time-stepping')
2 set_log_level(PROGRESS)
3
4 # Time-stepping
5 t = 0.0
6 for n in range(num_steps):
7
8     # Update current time
9     t += dt
10
11    # Place computation here
12
13    # Update progress bar
14    progress.update(t / T)
```

注意

(日志级别和 FEniCS 中的打印) 注意调用 `set_log_level(PROGRESS)` 这是至关重要的使 FEniCS 实际显示进度条。FEniCS 其实是相当丰富的信息，在计算过程中发生了什么打印到屏幕的信息量取决于当前日志水平。只有优先级高于或等于的消息将显示当前日志级别。预定义的日志级别 FEniCS 是 DBG, TRACE, PROGRESS, INFO, WARNING, ERROR, 和 CRITICAL。默认情况下，日志级别设置为 INFO 级别为 DBG, TRACE 和 PROGRESS 的邮件不会打印。用户可以使用 FEniCS 功能 `info` 打印消息，`warning` 和 `error`，它将在明显的日志中打印消息级别（在 `error` 的情况下也会抛出异常出口）。也可以使用呼叫 `log(level,message)` 来打印消息在特定日志级别。

由于线性方程组的系统明显大于对于简单的通道流量测试问题，我们选择使用迭代方法而不是使用的默认直接（稀疏）求解器调用 `solve` 时的 FEniCS。线性系统的有效解决方案由 PDEs 的离散化产生的，需要选择一个良好的迭代（Krylov 子空间）方法和良好的预条件。对于这个问题，我们将简单地使用双共轭梯度稳定法（BiCGSTAB）和共轭梯度法。这可以通过添加来完成在 `solve` 的调用中，关键字 `bicgstab` 或 `cg`。我们也指定适当的预处理器来加速计算：

```
Python code
1 solve(A1, u1.vector(), b1, 'bicgstab', 'hypre_amg')
2 solve(A2, p1.vector(), b2, 'bicgstab', 'hypre_amg')
3 solve(A3, u1.vector(), b3, 'cg', 'sor')
```

最后，为了能够在 ParaView 中对计算出的解决方案进行后处理，我们将解决方案存储在每个时间步长中的文件中。我们以前为此，使用后缀.pvd 创建文件。在例子中程序

https://fenicsproject.org/pub/tutorial/python/vol1/ft04_heat_gaussian.py,

我们首先创建一个名为 `heat_gaussian/solution.pvd` 的文件接着在每个时间步骤中保存解决方案

Python code

```
1 vtkfile << (u, t)
```

对于本示例，我们将选择保存解决方案到 XDMF 格式。此文件格式与.pvd 文件类似我们以前看过但有几个优点。首先，存储是在速度和文件大小方面都更有效率。第二，.xdmf 文件可以并行工作，无论是写作还是阅读（后期处理）。很像.pvd 文件，实际数据将不会存储在.xdmf 文件本身，而是存储在一个（单个）单独的数据文件与后缀.hdf5 是一个专为高性能计算设计的高级文件格式。我们创建 XDMF 文件如下：

Python code

```
1 xdmffile_u = XDMFFFile('navier_stokes_cylinder/velocity.xdmf')
2 xdmffile_p = XDMFFFile('navier_stokes_cylinder/pressure.xdmf')
```

在每个时间步骤中，我们可以存储速度和压力

Python code

```
1 xdmffile_u.write(u, t)
2 xdmffile_p.write(p, t)
```

我们还使用 FEniCS TimeSeries 存储解决方案。这让我们存储解决方案不是为了可视化，而是为了稍后重用计算，我们将在下一节中看到。使用 TimeSeries 从某些方面阅读解决方案是很容易和有效的模拟期间的时间。TimeSeries 类也使用 HDF5 用于高效存储和访问数据的文件格式。

图 3.5 和 3.6 显示速度和最终时间的压力在 ParaView 中可视化。对于可视化的速度，我们使用 Glyph 过滤器来可视化矢量速度场。为了可视化的压力，我们有使用 Warp By Scalar 过滤器。

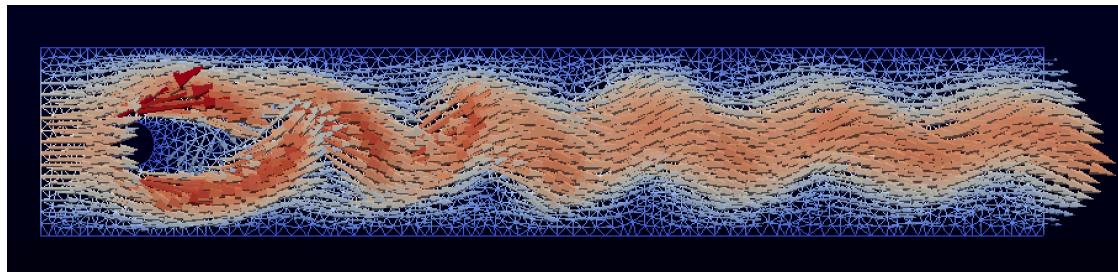


Figure 3.5: 最后时刻气瓶测试问题的速度曲线。

气缸测试问题的完整代码看起来像如下：

Python code

```
1 from fenics import *
2 from mshr import *
3 import numpy as np
4
5 T = 5.0          # final time
6 num_steps = 5000 # number of time steps
7 dt = T / num_steps # time step size
8 mu = 0.001       # dynamic viscosity
9 rho = 1          # density
10
11 # Create mesh
12 channel = Rectangle(Point(0, 0), Point(2.2, 0.41))
13 cylinder = Circle(Point(0.2, 0.2), 0.05)
```

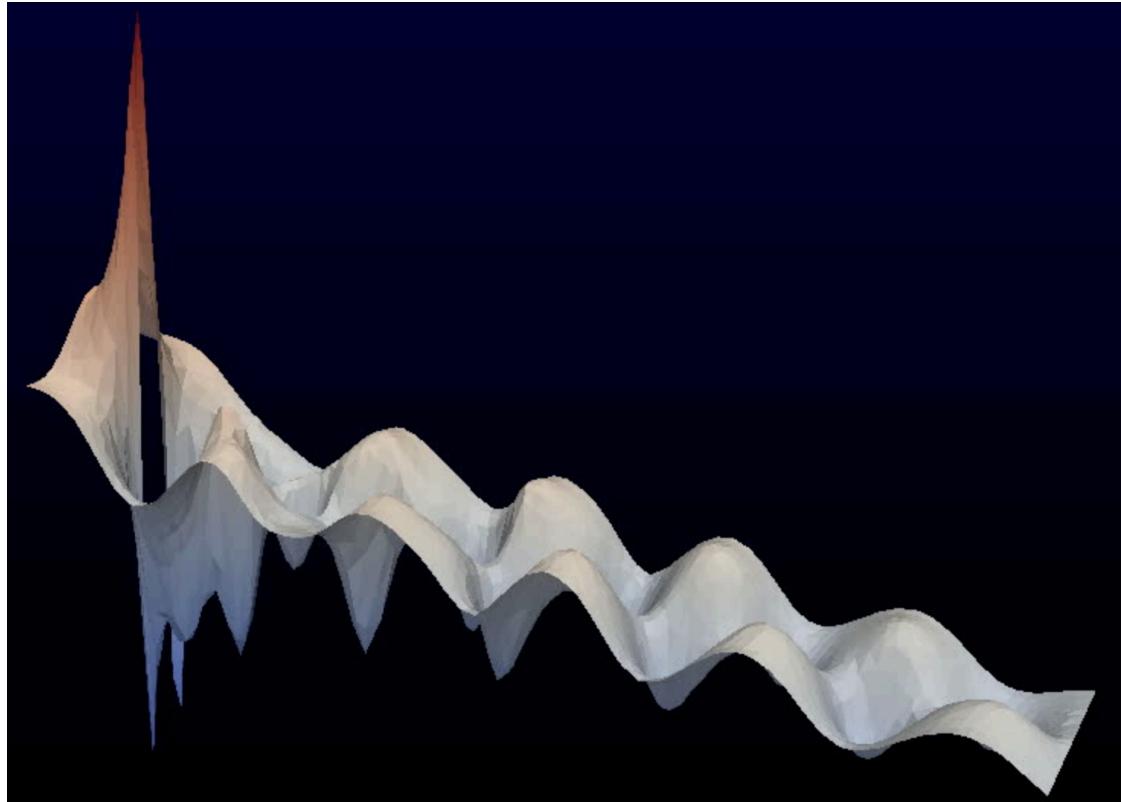


Figure 3.6: 最一刻气瓶测试问题的压力曲线。

```

14 domain = channel - cylinder
15 mesh = generate_mesh(domain, 64)
16
17 # Define function spaces
18 V = VectorFunctionSpace(mesh, 'P', 2)
19 Q = FunctionSpace(mesh, 'P', 1)
20
21 # Define boundaries
22 inflow = 'near(x[0], 0)'
23 outflow = 'near(x[0], 2.2)'
24 walls = 'near(x[1], 0) || near(x[1], 0.41)'
25 cylinder = 'on_boundary && x[0]>0.1 && x[0]<0.3 && x[1]>0.1 && x[1]<0.3'
26
27 # Define inflow profile
28 inflow_profile = ('4.0*1.5*x[1]*(0.41 - x[1]) / pow(0.41, 2)', '0')
29
30 # Define boundary conditions
31 bcu_inflow = DirichletBC(V, Expression(inflow_profile, degree=2), inflow)
32 bcu_walls = DirichletBC(V, Constant((0, 0)), walls)
33 bcu_cylinder = DirichletBC(V, Constant((0, 0)), cylinder)
34 bcp_outflow = DirichletBC(Q, Constant(0), outflow)
35 bcu = [bcu_inflow, bcu_walls, bcu_cylinder]
36 bcp = [bcp_outflow]
37
38 # Define trial and test functions
39 u = TrialFunction(V)
40 v = TestFunction(V)

```

```

41 p = TrialFunction(Q)
42 q = TestFunction(Q)
43
44 # Define functions for solutions at previous and current time steps
45 u_n = Function(V)
46 u_ = Function(V)
47 p_n = Function(Q)
48 p_ = Function(Q)
49
50 # Define expressions used in variational forms
51 U = 0.5*(u_n + u)
52 n = FacetNormal(mesh)
53 f = Constant((0, 0))
54 k = Constant(dt)
55 mu = Constant(mu)
56 rho = Constant(rho)
57
58 # Define symmetric gradient
59 def epsilon(u):
60     return sym(nabla_grad(u))
61
62 # Define stress tensor
63 def sigma(u, p):
64     return 2*mu*epsilon(u) - p*Identity(len(u))
65
66 # Define variational problem for step 1
67 F1 = rho*dot((u - u_n) / k, v)*dx \
68     + rho*dot(dot(u_n, nabla_grad(u_n)), v)*dx \
69     + inner(sigma(U, p_n), epsilon(v))*dx \
70     + dot(p_n*n, v)*ds - dot(mu*nabla_grad(U)*n, v)*ds \
71     - dot(f, v)*dx
72 a1 = lhs(F1)
73 L1 = rhs(F1)
74
75 # Define variational problem for step 2
76 a2 = dot(nabla_grad(p), nabla_grad(q))*dx
77 L2 = dot(nabla_grad(p_n), nabla_grad(q))*dx - (1/k)*div(u_)*q*dx
78
79 # Define variational problem for step 3
80 a3 = dot(u, v)*dx
81 L3 = dot(u_, v)*dx - k*dot(nabla_grad(p_ - p_n), v)*dx
82
83 # Assemble matrices
84 A1 = assemble(a1)
85 A2 = assemble(a2)
86 A3 = assemble(a3)
87
88 # Apply boundary conditions to matrices
89 [bc.apply(A1) for bc in bcu]
90 [bc.apply(A2) for bc in bcp]
91
92 # Create XDMF files for visualization output
93 xdmffile_u = XDMFFile('navier_stokes_cylinder/velocity.xdmf')
94 xdmffile_p = XDMFFile('navier_stokes_cylinder/pressure.xdmf')
95
96 # Create time series (for use in reaction_system.py)
97 timeseries_u = TimeSeries('navier_stokes_cylinder/velocity_series')
98 timeseries_p = TimeSeries('navier_stokes_cylinder/pressure_series')
99

```

```

100 # Save mesh to file (for use in reaction_system.py)
101 File('navier_stokes_cylinder/cylinder.xml.gz') << mesh
102
103 # Create progress bar
104 progress = Progress('Time-stepping')
105 set_log_level(PROGRESS)
106
107 # Time-stepping
108 t = 0
109 for n in range(num_steps):
110
111     # Update current time
112     t += dt
113
114     # Step 1: Tentative velocity step
115     b1 = assemble(L1)
116     [bc.apply(b1) for bc in bcu]
117     solve(A1, u_.vector(), b1, 'bicgstab', 'hypre_amg')
118
119     # Step 2: Pressure correction step
120     b2 = assemble(L2)
121     [bc.apply(b2) for bc in bcp]
122     solve(A2, p_.vector(), b2, 'bicgstab', 'hypre_amg')
123
124     # Step 3: Velocity correction step
125     b3 = assemble(L3)
126     solve(A3, u_.vector(), b3, 'cg', 'sor')
127
128     # Plot solution
129     plot(u_, title='Velocity')
130     plot(p_, title='Pressure')
131
132     # Save solution to file (XDMF/HDF5)
133     xdmffile_u.write(u_, t)
134     xdmffile_p.write(p_, t)
135
136     # Save nodal values to file
137     timeseries_u.store(u_.vector(), t)
138     timeseries_p.store(p_.vector(), t)
139
140     # Update previous solution
141     u_n.assign(u_)
142     p_n.assign(p_)
143
144     # Update progress bar
145     progress.update(t / T)
146     print('u max:', u_.vector().array().max())
147
148     # Hold plot
149     interactive()

```

该程序可以在文件

https://fenicsproject.org/pub/tutorial/python/vol1/ft08_navier_stokes_cylinder.py

中找到。应该告诉读者，这个例子程序很大比我们以前的例子在 CPU 时间和要求更高记忆，但应该可以合理地运行程序现代笔记本电脑。

3.5 平流-扩散-反应方程组

到目前为止我们遇到的问题 - 有显着的例外的 Navier-Stokes 方程 - 都有一个共同的特征: 它们都是涉及由单个标量或向量 PDE 表达的模型。在很多情况下, 模型被表示为 PDE 系统, 描述可能由 (非常) 不同的管理的不同数量物理。正如我们所看到的 Navier-Stokes 方程式, 一种解决方法 FEniCS 中的 PDE 系统是使用我们解决的分割方法一个方程, 并将解从一个方程提供给下一个。然而, FEniCS 的优势之一是轻松哪—一个可以代替几个相互联系的变分问题 PDE 成为一个复合系统。在本节中, 我们将介绍如何使用 FEniCS 为这种耦合 PDE 系统写出求解器。目标是展示实现完全隐含的容易程度, 在 FEniCS 中也称为单片, 求解器。

3.5.1 PDE 问题

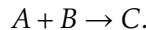
我们的模型问题是以下系统平流-扩散-反应方程:

$$\frac{\partial u_1}{\partial t} + w \cdot \nabla u_1 - \nabla \cdot (\epsilon \nabla u_1) = f_1 - Ku_1 u_2, \quad (3.36)$$

$$\frac{\partial u_2}{\partial t} + w \cdot \nabla u_2 - \nabla \cdot (\epsilon \nabla u_2) = f_2 - Ku_1 u_2, \quad (3.37)$$

$$\frac{\partial u_3}{\partial t} + w \cdot \nabla u_3 - \nabla \cdot (\epsilon \nabla u_3) = f_3 + Ku_1 u_2 - Ku_3. \quad (3.38)$$

该系统模拟两种物质之间的化学反应 A 和 B 在某些域 Ω :



我们假设反应是 first-order, 意思是反应速率与浓度 $[A]$ 和 $[B]$ 成正比两种物品 A 和 B :

$$\frac{d}{dt}[C] = K[A][B].$$

我们还假设形成的物种 C 自发衰减速率与浓度成比例 $[C]$ 。在 PDE 系统中 (3.36)–(3.38), 我们使用变量 u_1 , u_2 和 u_3 来表示三种物种的浓度:

$$u_1 = [A], \quad u_2 = [B], \quad u_3 = [C].$$

我们看到化学反应是在 PDE 系统的右侧 (3.36)–(3.38)。

化学反应参与域中的每个点 Ω 。此外, 我们假设物种 A , B 和 C 在整个域扩散扩散性 $\epsilon(\text{术语 } -\nabla \cdot (\epsilon \nabla u_i))$, 并以速度平缓 w (术语 $w \cdot \nabla u_i$)。

为了使事物的视觉和身体感兴趣, 我们将让它化学反应发生在从中计算出的速度场解决不可压缩的 Navier-Stokes 方程圆柱体从上一节。总而言之, 我们将会是求解以下非线性 PDE 耦合系统:

$$\rho \left(\frac{\partial w}{\partial t} + w \cdot \nabla w \right) = \nabla \cdot \sigma(w, p) + f, \quad (3.39)$$

$$\nabla \cdot w = 0, \quad (3.40)$$

$$\frac{\partial u_1}{\partial t} + w \cdot \nabla u_1 - \nabla \cdot (\epsilon \nabla u_1) = f_1 - Ku_1 u_2, \quad (3.41)$$

$$\frac{\partial u_2}{\partial t} + w \cdot \nabla u_2 - \nabla \cdot (\epsilon \nabla u_2) = f_2 - Ku_1 u_2, \quad (3.42)$$

$$\frac{\partial u_3}{\partial t} + w \cdot \nabla u_3 - \nabla \cdot (\epsilon \nabla u_3) = f_3 + Ku_1 u_2 - Ku_3. \quad (3.43)$$

我们假设 $u_1 = u_2 = u_3 = 0$ at $t = 0$ 并注入物种 A 和 B 通过指定非零源术语 f_1 到系统中和 f_2 接近流入的角落，并取 $f_3 = 0$ 。该结果将是 A 和 B 对流对流在整个渠道扩散，当他们混合物种 C 时将形成。

由于系统是从 Navier–Stokes 子系统单向耦合的对于平流–扩散–反应子系统，我们不需要重新计算 Navier–Stokes 方程的解，但只能读回以前计算的速度场 w 并将其输入我们的方程式但是我们需要学习如何读写时间依赖 PDE 问题的解决方案。

3.5.2 变化公式

我们通过乘法获得我们系统的变分公式每个方程通过一个测试函数，整合二阶项 $-\nabla \cdot (\epsilon \nabla u_i)$ 部分，并总结方程。当与 FEniCS 合作时，可以方便的考虑 PDE 系统作为方程的向量。测试功能收集在矢量也是，变分公式是内在的产物矢量 PDE 和矢量测试功能。

我们还需要及时介绍一些离散化。我们会用反向 Euler 方法与以前一样，当我们解决热方程和通过 $(u_i^{n+1} - u_i^n)/\Delta t$ 近似时间导数。让 v_1 , v_2 和 v_3 是测试功能或组件测试向量函数。内部产品产生

$$\begin{aligned} & \int_{\Omega} (\Delta t^{-1}(u_1^{n+1} - u_1^n)v_1 + w \cdot \nabla u_1^{n+1} v_1 + \epsilon \nabla u_1^{n+1} \cdot \nabla v_1) dx \\ & + \int_{\Omega} (\Delta t^{-1}(u_2^{n+1} - u_2^n)v_2 + w \cdot \nabla u_2^{n+1} v_2 + \epsilon \nabla u_2^{n+1} \cdot \nabla v_2) dx \\ & + \int_{\Omega} (\Delta t^{-1}(u_3^{n+1} - u_3^n)v_3 + w \cdot \nabla u_3^{n+1} v_3 + \epsilon \nabla u_3^{n+1} \cdot \nabla v_3) dx \\ & - \int_{\Omega} (f_1 v_1 + f_2 v_2 + f_3 v_3) dx \\ & - \int_{\Omega} (-K u_1^{n+1} u_2^{n+1} v_1 - K u_1^{n+1} u_2^{n+1} v_2 + K u_1^{n+1} u_2^{n+1} v_3 - K u_3^{n+1} v_3) dx = 0. \end{aligned} \quad (3.44)$$

对于这个问题，很自然地假设均匀的 xxNeumann 边界 u_1 , u_2 和 u_3 的整个边界条件；那是， $\partial u_i / \partial n = 0$ 对于 $i = 1, 2, 3$ 。这意味着当我们通过零件整合时，边界条件消失。

3.5.3 FEniCS 实现

第一步是从文件中读取网格。幸运的是，我们确信将网格保存到 Navier–Stokes 示例中，现在可以轻松实现从文件读取：

```
Python code
1 mesh = Mesh('navier_stokes_cylinder/cylinder.xml.gz')
```

网格以本地 FEniCS XML 格式存储（附加 gzipping 减少文件大小）。

接下来，我们需要定义有限元函数空间。为了这问题，我们需要定义几个空格。我们创建的第一个空间是 Navier–Stokes 的速度场 w 的空间模拟。我们称这个空格为 W 并定义空格

```
Python code
1 W = VectorFunctionSpace(mesh, 'P', 2)
```

重要的是这个空间与我们的空间完全一样用于 Navier–Stokes 求解器中的速度场。阅读值为速度字段，我们使用 TimeSeries：

```
Python code
1 timeseries_w = TimeSeries('navier_stokes_cylinder/velocity_series')
```

这将初始化对象timeseries_w 我们会打电话给我们后来在时间循环中从中检索值文件velocity_series.h5 (二进制 HDF5 格式)。

对于三个浓度 u_1 , u_2 和 u_3 , 我们想要使用表示完整系统的函数创建混合空间 (u_1, u_2, u_3) 作为单个实体。为此, 我们需要定义一个 MixedElement 作为三个简单有限元素的乘积空间然后使用混合元素定义函数空间:

```
Python code
1 P1 = FiniteElement('P', triangle, 1)
2 element = MixedElement([P1, P1, P1])
3 V = FunctionSpace(mesh, element)
```

注意

(混合元素作为元素的产物) FEniCS 还允许将有限元素定义为简单的产品元素 (或混合元素)。例如, 众所周知的 Taylor-Hood 元素, 具有二次速度分量和线性压力函数, 可以定义如下:

```
Python code
1 P2 = VectorElement('P', triangle, 2)
2 P1 = FiniteElement('P', triangle, 1)
3 TH = P2 * P1
```

这个语法对于两个元素是非常有用的, 但是对于三个或更多个这样的语法是非常有用的我们遇到 Python 解释器处理的一个微妙问题 * 运算符。对于反应体系, 我们创建混合元素通过 element = MixedElement([P1, P1, P1]), 一个将被诱惑写

```
Python code
1 element = P1 * P1 * P1
```

但是, 这相当于写入 element = (P1 * P1) * P1 结果将是由两个子系统组成的混合元素其中首先由两个标量子系统组成。

最后, 我们说一个混合系统的简单情况由反应体系的三个标量元素组成定义实际上等同于使用标准向量值元件:

```
Python code
1 element = VectorElement('P', triangle, 1, dim=3)
2 V = FunctionSpace(mesh, element)
```

一旦创建空间, 我们需要定义我们的测试功能和有限元函数。混合功能的测试功能可以通过将 TestFunction 替换为 TestFunctions 来创建空格:

```
Python code
1 v_1, v_2, v_3 = TestFunctions(V)
```

由于这个问题是非线性的, 所以我们需要使用功能比未知数的试验功能。这可以通过使用 FEniCS 中对应的 Functions 构造。但是, 我们将会首先需要访问整个系统本身的 Function 需要创建该函数, 然后访问其组件:

```
Python code
1 u = Function(V)
2 u_1, u_2, u_3 = split(u)
```

这些函数将用于表示未知数 u_1 , u_2 和 u_3 在新的时间级别 $n+1$ 。前一个相应的值时间级别 n 由 u_{n1} , u_{n2} , 和 u_{n3} 表示在我们的程序中。

现在所有功能和测试功能都已经定义了，我们可以表达非线性变分问题 (3.44):

```

    Python code
1 F = ((u_1 - u_n1) / k)*v_1*dx + dot(w, grad(u_1))*v_1*dx \
2 + eps*dot(grad(u_1), grad(v_1))*dx + K*u_1*u_2*v_1*dx \
3 + ((u_2 - u_n2) / k)*v_2*dx + dot(w, grad(u_2))*v_2*dx \
4 + eps*dot(grad(u_2), grad(v_2))*dx + K*u_1*u_2*v_2*dx \
5 + ((u_3 - u_n3) / k)*v_3*dx + dot(w, grad(u_3))*v_3*dx \
6 + eps*dot(grad(u_3), grad(v_3))*dx - K*u_1*u_2*v_3*dx + K*u_3*v_3*dx \
7 - f_1*v_1*dx - f_2*v_2*dx - f_3*v_3*dx

```

时间步长只是解决这个变分问题在每个时间段中通过调用 solve 函数:

```

    Python code
1 t = 0
2 for n in range(num_steps):
3     t += dt
4     timeseries_w.retrieve(w.vector(), t)
5     solve(F == 0, u)
6     u_n.assign(u)

```

在每个时间步长中，我们首先读取速度的当前值字段从我们以前存储的时间序列。然后我们解决非线性系统，并将计算的值分配给左侧下一个时间间隔的边值。从 a 检索值时 TimeSeries，默认情况下将内插（线性）到给定的时间 t 如果时间不完全匹配的样本系列。

最后一次的解决方案如图 3.7 所示。我们清楚地看到物种 A 和 B 的平流和形成的 C 沿着 A 和 B 相遇的频道中心。

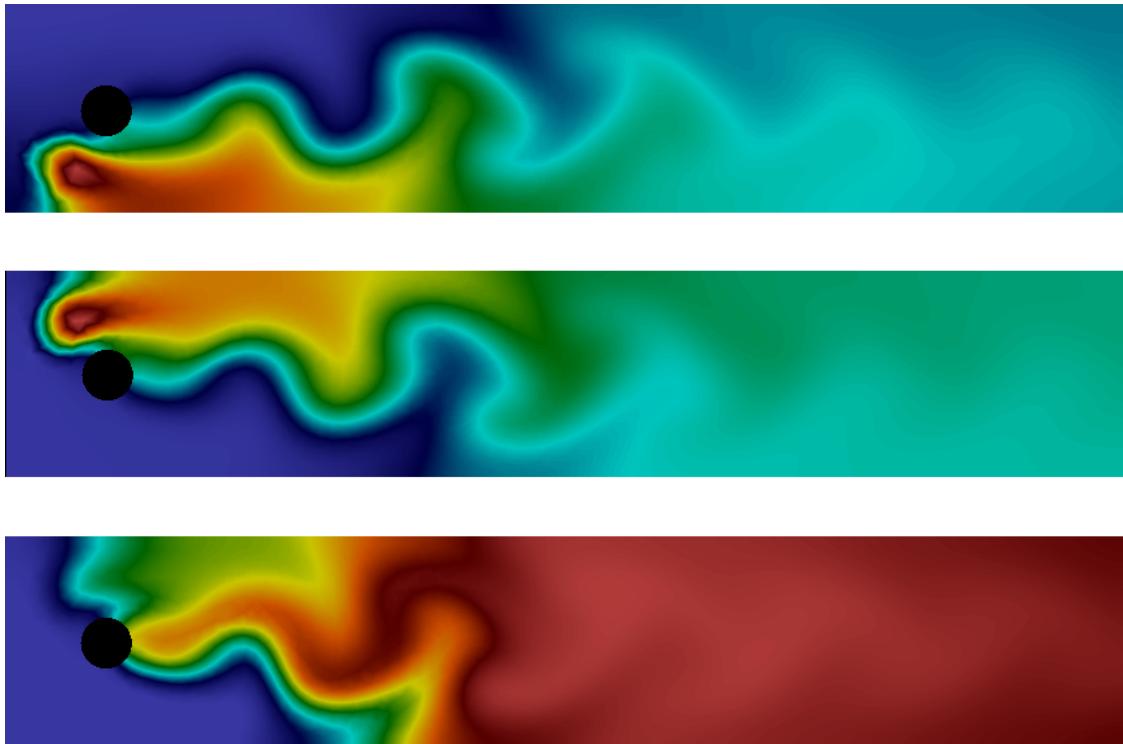


Figure 3.7: 在最终时间，从三种物种的 A, B 和 C(从上到下) 的浓度。

完整的代码如下。

Python code

```

1 from fenics import *
2
3 T = 5.0           # final time
4 num_steps = 500   # number of time steps
5 dt = T / num_steps # time step size
6 eps = 0.01        # diffusion coefficient
7 K = 10.0          # reaction rate
8
9 # Read mesh from file
10 mesh = Mesh('navier_stokes_cylinder/cylinder.xml.gz')
11
12 # Define function space for velocity
13 W = VectorFunctionSpace(mesh, 'P', 2)
14
15 # Define function space for system of concentrations
16 P1 = FiniteElement('P', triangle, 1)
17 element = MixedElement([P1, P1, P1])
18 V = FunctionSpace(mesh, element)
19
20 # Define test functions
21 v_1, v_2, v_3 = TestFunctions(V)
22
23 # Define functions for velocity and concentrations
24 w = Function(W)
25 u = Function(V)
26 u_n = Function(V)
27
28 # Split system functions to access components
29 u_1, u_2, u_3 = split(u)
30 u_n1, u_n2, u_n3 = split(u_n)
31
32 # Define source terms
33 f_1 = Expression('pow(x[0]-0.1,2)+pow(x[1]-0.1,2)<0.05*0.05 ? 0.1 : 0',
34                      degree=1)
35 f_2 = Expression('pow(x[0]-0.1,2)+pow(x[1]-0.3,2)<0.05*0.05 ? 0.1 : 0',
36                      degree=1)
37 f_3 = Constant(0)
38
39 # Define expressions used in variational forms
40 k = Constant(dt)
41 K = Constant(K)
42 eps = Constant(eps)
43
44 # Define variational problem
45 F = ((u_1 - u_n1) / k)*v_1*dx + dot(w, grad(u_1))*v_1*dx \
46     + eps*dot(grad(u_1), grad(v_1))*dx + K*u_1*u_2*v_1*dx \
47     + ((u_2 - u_n2) / k)*v_2*dx + dot(w, grad(u_2))*v_2*dx \
48     + eps*dot(grad(u_2), grad(v_2))*dx + K*u_1*u_2*v_2*dx \
49     + ((u_3 - u_n3) / k)*v_3*dx + dot(w, grad(u_3))*v_3*dx \
50     + eps*dot(grad(u_3), grad(v_3))*dx - K*u_1*u_2*v_3*dx + K*u_3*v_3*dx \
51     - f_1*v_1*dx - f_2*v_2*dx - f_3*v_3*dx
52
53 # Create time series for reading velocity data
54 timeseries_w = TimeSeries('navier_stokes_cylinder/velocity_series')
55
56 # Create VTK files for visualization output
57 vtkfile_u_1 = File('reaction_system/u_1.pvd')
58 vtkfile_u_2 = File('reaction_system/u_2.pvd')
```

```

59 |vtkfile_u_3 = File('reaction_system/u_3.pvd')
60 |
61 # Create progress bar
62 progress = Progress('Time-stepping')
63 set_log_level(PROGRESS)
64 |
65 # Time-stepping
66 t = 0
67 for n in range(num_steps):
68     # Update current time
69     t += dt
70 |
71     # Read velocity from file
72     timeseries_w.retrieve(w.vector(), t)
73 |
74     # Solve variational problem for time step
75     solve(F == 0, u)
76 |
77     # Save solution to file (VTK)
78     _u_1, _u_2, _u_3 = u.split()
79     vtkfile_u_1 << (_u_1, t)
80     vtkfile_u_2 << (_u_2, t)
81     vtkfile_u_3 << (_u_3, t)
82 |
83     # Update previous solution
84     u_n.assign(u)
85 |
86     # Update progress bar
87     progress.update(t / T)
88 |
89 # Hold plot
90 interactive()

```

该示例程序可以在文件

https://fenicsproject.org/pub/tutorial/python/vol1/ft09_reaction_system.py

中找到。

最后，我们评论三种非常有用的重要技术当使用 PDE 系统时：设置初始条件，设置边界条件，并提取系统的组件绘图或后处理。

设置混合系统的初始条件。 在我们的示例中，我们不需要担心设置初始化条件，因为我们从 $u_1 = u_2 = u_3 = 0$ 开始。这发生了自动在代码中设置 $u_n = \text{Function}(V)$ 这个为整个系统和所有自由度创建一个 Function 设置为零。

如果我们要为系统的组件设置初始条件单独地，最简单的解决方案是定义初始条件作为矢量值 Expression，然后进行项目（或内插）代表整个系统的 Function。例如，

Python code

```

1 u_0 = Expression(( 'sin(x[0])' , 'cos(x[0]*x[1])' , 'exp(x[1])' ), degree=1)
2 u_n = project(u_0, V)

```

这将定义 u_1 , u_2 和 u_3 作为预测 $\sin x$, $\cos(xy)$ 和 $\exp(y)$ 。

设定混合系统的边界条件。 在我们的例子中，我们也不用担心设置边界条件，因为我们使用了一个天然的 Neumann 条件。如果我们想设置 Dirichlet 条件对于系统的各个组件，这可

以像 `x` DirichletBC 一样照常执行，但是我们必须指定哪个子系统设定边界条件。例如，指定 u_2 应该等于 xy 在边界定义 boundary，我们做

```
Python code
1 u_D = Expression('x[0]*x[1]', degree=1)
2 bc = DirichletBC(V.sub(1), u_D, boundary)
```

对象 `bc` 或包含不同的这些对象的列表边界条件，可以像往常一样传递给 `solve` 函数。请注意，在 FEniCS 中，编号从 0 开始，因此子空间对应于 u_2 是 `V.sub(1)`。

访问混合系统的组件。 如果 `u` 是在 FEniCS 的混合函数空间中定义的 Function `u` 可以是 split 到组件的几种方式。以上我们已经看到了第一个例子：

```
Python code
1 u_1, u_2, u_3 = split(u)
```

这将 `u` 的组件提取为可以在 `a` 中使用的 symbols 变分问题。上述声明其实相当于

```
Python code
1 u_1 = u[0]
2 u_2 = u[1]
3 u_3 = u[2]
```

注意 `u[0]` 不是真正的 Function 对象，而只是一个符号表达式，就像 `grad(u)` 在 FEniCS 中是一个象征意义表达式而不是 Function 表示渐变。意即那个 `u_1`, `u_2`, `u_3` 可以用于变分问题，但是不能用于绘图或后处理。

访问 `u` 的组件以绘制和保存解决方案要文件，我们需要使用 `split` 函数的不同变体：

```
Python code
1 u_1_, u_2_, u_3_ = u.split()
```

这将返回三个子函数作为具有访问权限的实际对象共同的底层数据存储在 `u` 中，这样可以绘制和保存提交可能。或者，我们可以做

```
Python code
1 u_1_, u_2_, u_3_ = u.split(deepcopy=True)
```

这将创建 `u_1_`, `u_2_`, 和 `u_3_` 作为独立的 Function 对象，每个持有从中提取的子功能数据的副本 `u`。这在许多情况下是有用的，但不是必需的绘制和保存文件解决方案。

4. 子域和边界条件

4.1	结合 Dirichlet 和 Neumann 条件	65
4.2	设置多个 Dirichlet 条件	67
4.3	定义不同材质的子域	68
4.4	设置多个 Dirichlet, Neumann 和 Robin 条件	73
4.5	用子域生成网格	78

到目前为止，我们只是简单地看一下如何指定边界条件。在本章中，我们将更仔细地研究如何指定边界的特定部分（子域）的边界条件如何组合多个边界条件。我们也会看看怎么样生成与子域的网格以及如何定义系数在不同的子域中具有不同的值。

4.1 结合 Dirichlet 和 Neumann 条件

让我们从 Chapter 2 返回 xPoisson 问题并看看如何扩展数学和实现来处理 Dirichlet 条件与 Neumann 条件相结合。该域仍然是单位广场，但现在我们设置了 Dirichlet 条件 $u = u_D$ 在左侧和右侧， $x = 0$ 和 $x = 1$ ，而 Neumann 条件

$$-\frac{\partial u}{\partial n} = g$$

适用于剩余的边 $y = 0$ 和 $y = 1$ 。

4.1.1 PDE 问题

让 Γ_D 和 Γ_N 表示边界 $\partial\Omega$ 的部分分别适用 Dirichlet 和 Neumann 条件。该完整的边界值问题可以写成

$$-\nabla^2 u = f \quad \text{in } \Omega, \tag{4.1}$$

$$u = u_D \quad \text{on } \Gamma_D, \tag{4.2}$$

$$-\frac{\partial u}{\partial n} = g \quad \text{on } \Gamma_N. \tag{4.3}$$

再次，我们选择 $u = 1 + x^2 + 2y^2$ 作为确切的解，并调整 f , g 和 u_D 相应：

$$\begin{aligned} f(x, y) &= -6, \\ g(x, y) &= \begin{cases} 0, & y = 0, \\ -4, & y = 1, \end{cases} \\ u_D(x, y) &= 1 + x^2 + 2y^2. \end{aligned}$$

为了便于编程，我们将 g 定义为整体的一个函数域 Ω ，使得 g 在 $y = 0$ 和上具有正确的值 $Y = 1$ 。一个可能的扩展是

$$g(x, y) = -4y.$$

4.1.2 变化公式

第一个任务是推导变分公式。这一次我们不能因为省略由零件整合产生的边界术语 v 在 Γ_D 上为零。我们有

$$-\int_{\Omega} (\nabla^2 u)v \, dx = \int_{\Omega} \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \, ds,$$

因为 $v = 0$ 上 Γ_D ，

$$-\int_{\partial\Omega} \frac{\partial u}{\partial n} v \, ds = -\int_{\Gamma_N} \frac{\partial u}{\partial n} v \, ds = \int_{\Gamma_N} gv \, ds,$$

通过在 Γ_N 上应用边界条件。得到的弱表单读

$$\int_{\Omega} \nabla u \cdot \nabla v \, dx = \int_{\Omega} fv \, dx - \int_{\Gamma_N} gv \, ds. \quad (4.4)$$

表达这个方程在标准符号中， $a(u, v) = L(v)$ 是直接的

$$a(u, v) = \int_{\Omega} \nabla u \cdot \nabla v \, dx, \quad (4.5)$$

$$L(v) = \int_{\Omega} fv \, dx - \int_{\Gamma_N} gv \, ds. \quad (4.6)$$

4.1.3 FEniCS 实现

Neumann 条件如何影响实现？让我们重新审视我们以前的实施

https://fenicsproject.org/pub/tutorial/python/vol1/ft01_poisson.py

从部分 2.2，并检查哪些更改我们需要使 xNeumann 条件合并。事实证明只需要两个更改：

- 定义 Dirichlet 边界的函数 boundary 必须修改。
- 新的边界项必须添加到 L 的表达式中。

第一次调整可以编码为

Python code

```

1 tol = 1E-14
2
3 def boundary_D(x, on_boundary):
4     if on_boundary:
5         if near(x[0], 0, tol) or near(x[0], 1, tol):
6             return True
7         else:
8             return False
9     else:
10        return False

```

更紧凑的实现读取

Python code

```

1 def boundary_D(x, on_boundary):
2     return on_boundary and (near(x[0], 0, tol) or near(x[0], 1, tol))

```

我们的程序的第二个调整涉及 L 的定义，其中包括 Neumann 条件：

Python code

```

1 g = Expression(' -4*x[1]', degree=1)
2 L = f*v*dx - g*v*ds

```

ds 变量意味着边界积分，而 dx 意味着域 Ω 的整数。不需要其他修改。

请注意，整合 $*ds$ 是整个执行的边界，包括 Dirichlet 边界。但是，自从测试函数 v 在 Dirichlet 边界上消失（因此指定一个 DirichletBC），积分将只包括来自 Neumann 边界的贡献。

4.2 设置多个 Dirichlet 条件

在上一节中，我们使用单个函数 $u_D(x, y)$ 在边界的两个部分设置 Dirichlet 条件。通常是更实用的是使用多个功能，每个子域一个边界。让我们再回到 4.1” 部分的情况。并根据两个 Dirichlet 条件重新定义问题：

$$\begin{aligned} -\nabla^2 u &= f \quad \text{in } \Omega, \\ u &= u_L \quad \text{on } \Gamma_D^L, \\ u &= u_R \quad \text{on } \Gamma_D^R, \\ -\frac{\partial u}{\partial n} &= g \quad \text{on } \Gamma_N. \end{aligned}$$

这里， Γ_D^L 是左边界 $x = 0$ ，而 Γ_D^R 是右边界 $x = 1$ 。我们注意到 $u_L(x, y) = 1 + 2y^2$ ， $u_R(x, y) = 2 + 2y^2$ ，和 $g(x, y) = 4y$ 。

对于 Γ_D^L 的边界条件，我们定义了通常三重表达式的边界值，一个函数定义边界的位置，以及一个 DirichletBC 对象：

Python code

```

1 u_L = Expression('1 + 2*x[1]*x[1]', degree=2)
2
3 def boundary_L(x, on_boundary):
4     tol = 1E-14

```

```

5     return on_boundary and near(x[0], 0, tol)
6
7 bc_L = DirichletBC(V, u_L, boundary_L)

```

对于 Γ_D^R 的边界条件，我们写一个类似的代码段：

Python code

```

1 u_R = Expression('2 + 2*x[1]*x[1]', degree=2)
2
3 def boundary_R(x, on_boundary):
4     tol = 1E-14
5     return on_boundary and near(x[0], 1, tol)
6
7 bc_R = DirichletBC(V, u_R, boundary_R)

```

我们收集列表中的两个边界条件我们可以传递给 solve 函数来计算解决方案：

Python code

```

1 bcs = [bc_L, bc_R]
2 ...
3 solve(a == L, u, bcs)

```

请注意，对于不依赖于 x 或 y 的边界值，我们可以用 Constant 对象替换 Expression 对象。

4.3 定义不同材质的子域

在不同材料构成的领域解决 PDE 是经常发生的遇到任务在 FEniCS 中，这些问题由处理定义域内的子域。一个简单的例子与两个 2D 中的材料（子域）将展示这一想法。我们认为 Poisson 方程的以下可变系数扩展从章 2：

$$-\nabla \cdot [\kappa(x, y) \nabla u(x, y)] = f(x, y), \quad (4.7)$$

在某些域 Ω 。在物理上，这个问题可以被看作是热传导的模型，具有可变热导率 $\kappa(x, y) \geq \underline{\kappa} > 0$ 。

为了说明的目的，我们考虑域 $\Omega = [0, 1] \times [0, 1]$ 并将其分成两个相等的子域，如如图 4.1 所示：

$$\Omega_0 = [0, 1] \times [0, 1/2], \quad \Omega_1 = [0, 1] \times (1/2, 1].$$

我们定义 $\kappa(x, y) = \kappa_0$ 在 Ω_0 和 $\kappa(x, y) = \kappa_1$ 在 Ω_1 ，哪里 $\kappa_0, \kappa_1 > 0$ 被赋予常量。

变分公式可以很容易地表达在 FEniCS 中如下：

Python code

```

1 a = kappa*dot(grad(u), grad(v))*dx
2 L = f*v*dx

```

在本节的其余部分，我们将讨论不同的策略用于将系数 kappa 定义为需要的 Expression 两个子域中的值不同。

4.3.1 使用表达式来定义子域

实现可变系数的最简单的方法 $\kappa = \kappa(x, y)$ 是定义一个 Expression，它依赖于坐标 x 和 y 。我们以前用过 Expression 类基于简单公式定义表达式。或者，一个 Expression 可以被定义为一个允许更多的 Python 类复杂的逻辑。以下代码片段说明了这一点施工：

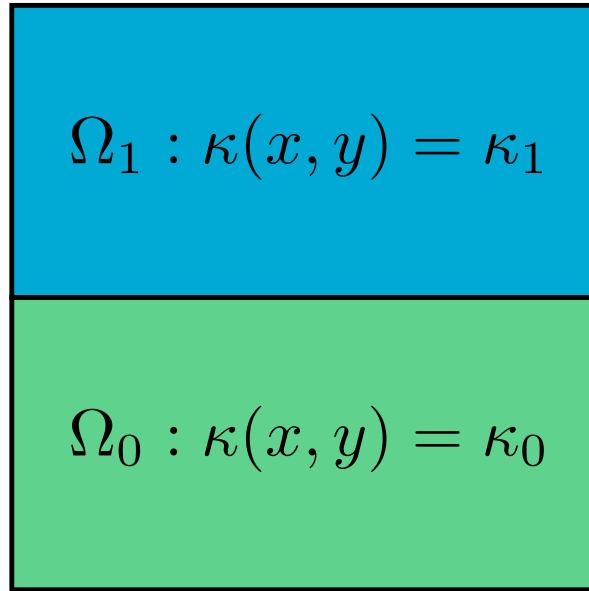


Figure 4.1: 两个具有不同材料参数的子域。

Python code

```

1 class K(Expression):
2     def set_k_values(self, k_0, k_1):
3         self.k_0, self.k_1 = k_0, k_1
4
5     def eval(self, value, x):
6         "Set value[0] to value at point x"
7         tol = 1E-14
8         if x[1] <= 0.5 + tol:
9             value[0] = self.k_0
10        else:
11            value[0] = self.k_1
12
13 # Initialize kappa
14 kappa = K(degree=0)
15 kappa.set_k_values(1, 0.01)

```

`eval` 方法在定义函数方面提供了很大的灵活性，但一个缺点是 FEniCS 将为 Python 中的每个节点 `x` 调用 `eval` 这是一个缓慢的过程。

另一种方法是使用 C++ 字符串表达式以前看过，在 FEniCS 中效率更高。这可以做到使用内联如果测试：

Python code

```

1 tol = 1E-14
2 k_0 = 1.0
3 k_1 = 0.01
4 kappa = Expression('x[1] <= 0.5 + tol ? k_0 : k_1', degree=0,
5                      tol=tol, k_0=k_0, k_1=k_1)

```

如果子域名定义变量系数的这种方法起作用是可以用几何表达的简单形状不平等。但是，对于更复杂的子域，我们将需要使用更一般的技术，我们将在下面看到。

4.3.2 使用网格函数来定义子域

现在我们将介绍如何指定子域 Ω_0 和 Ω_1 使用更一般的技术。这种技术涉及使用两种在使用子域时，FEniCS 中必不可少的课程：SubDomain 和 MeshFunction。考虑下面的定义边界 $x = 0$ ：

Python code

```

1 def boundary(x, on_boundary):
2     tol = 1E-14
3     return on_boundary and near(x[0], 0, tol)

```

这个边界定义实际上是更通用的捷径 FEniCS 概念 SubDomain。一个 SubDomain 是一个定义 a 的类区域在空间 (子域) 的成员函数 inside 它返回 True 属于子域的点 False 对于不属于子域的点。这是怎么回事指定边界 $x = 0$ 作为 SubDomain：

Python code

```

1 class Boundary(SubDomain):
2     def inside(self, x, on_boundary):
3         tol = 1E-14
4         return on_boundary and near(x[0], 0, tol)
5
6 boundary = Boundary()
7 bc = DirichletBC(V, Constant(0), boundary)

```

我们注意到 Boundary 类的 inside 函数是 (几乎) 与以前的边界定义相同 boundary 功能。技术上，我们的类 Boundary 是一个 subclass 的 FEniCS 类 SubDomain。

我们将使用两个 SubDomain 子类来定义两个子域 Ω_0 和 Ω_1 ：

Python code

```

1 tol = 1E-14
2
3 class Omega_0(SubDomain):
4     def inside(self, x, on_boundary):
5         return x[1] <= 0.5 + tol
6
7 class Omega_1(SubDomain):
8     def inside(self, x, on_boundary):
9         return x[1] >= 0.5 - tol

```

请注意在两个测试中使用 \leq 和 \geq 。FEniCS 会打电话给 inside 函数为单元格中的每个顶点确定是否不是该单元属于特定的子域。因此，它是重要的，测试对于与对齐的单元格中的所有顶点保持一致边界。另外，我们使用宽容来确保内部边界上的顶点 $y = 0.5$ 将属于两者子域。这是一个有点反直觉，但是是必要的使内部边界以上和下方的细胞属于 Ω_0 或 Ω_1 。

要定义变量系数 κ ，我们将使用一个强大的工具 FEniCS 称为 MeshFunction。一个 MeshFunction 是一个离散的函数可以在一组所谓的网格中进行评估实体。FEniCS 中的网格实体是顶点、边缘、面或细胞（三角形或四面体）。一个 MeshFunction 在单元格上适合代表子域（材料），而 a MeshFunction 超过面（边或面）用于表示外部或内部边界。一个 MeshFunction 在单元格上也可用于表示网格细化的边界标记。一个 FEniCS MeshFunction 对其数据类型（如整数或布尔值）及其维数（0 = 顶点，1 = 边等等。）特殊子类 VertexFunction, EdgeFunction 等等提供了一个特定的 MeshFunction 的容易定义尺寸。

由于在本示例中我们需要定义 Ω 的子域，我们使用 CellFunction。构造函数给出两个参数：(1) 值的类型：'int' 用于整数，'size_t' 用于非负（无符号）整数，'double' 为真数字和 'bool' 用于逻辑值；(2) 一个 Mesh 对象。或者，构造函数只能使用一个文件名并进行初始化 CellFunction 从文件中的数据。

我们首先创建一个非负的 CellFunction 整数值 ('size_t'):

```
Python code
1 materials = CellFunction('size_t', mesh)
```

接下来，我们使用两个子域来 mark 属于每个的单元格子域名:

```
Python code
1 subdomain_0 = Omega_0()
2 subdomain_1 = Omega_1()
3 subdomain_0.mark(materials, 0)
4 subdomain_1.mark(materials, 1)
```

这将将网格函数 materials 的值设置为 0 属于所有单元格的每个单元格属于 Ω_0 和 1 Ω_1 。或者，我们可以使用以下等效代码标记单元格:

```
Python code
1 materials.set_all(0)
2 subdomain_1.mark(materials, 1)
```

检查网格函数的值，看到我们确实正确定义了我们的子域，我们可以简单地绘制网格功能:

```
Python code
1 plot(materials, interactive=True)
```

我们也可能希望以后存储网格功能的值使用:

```
Python code
1 File('materials.xml.gz') << materials
```

这可以稍后从文件中读取如下:

```
Python code
1 File('materials.xml.gz') >> materials
```

现在，使用网格函数 materials 的值来定义可变系数 κ ，我们创建一个 FEniCS Expression:

```
Python code
1 class K(Expression):
2     def __init__(self, materials, k_0, k_1, **kwargs):
3         self.materials = materials
4         self.k_0 = k_0
5         self.k_1 = k_1
6
7     def eval_cell(self, values, x, cell):
8         if self.materials[cell.index] == 0:
9             values[0] = self.k_0
10        else:
11            values[0] = self.k_1
12
13 kappa = K(materials, k_0, k_1, degree=0)
```

这与我们上面定义的 Expression 子类似，但是我们使用成员函数 eval_cell 代替常规 eval 函数。这个版本的评估功能有一个附加 cell 参数，我们可以用来检查我们在哪个单元格目前正在评估功能。我们也定义了特殊的功能 __init__ (构造函数)，以便我们可以将所有数据传递给 Expression 创建时。

由于我们使用几何测试来定义两个 SubDomains 对于 Ω_0 和 Ω_1 ，MeshFunction 方法可能看起来像使用简单方法的不必要的复杂性 Expression 与 if-test。但是，一般的定义是子域

可能作为 MeshFunction(从数据文件) 可用, 可能生成的网格生成过程的一部分, 而不是作为一个简单的几何测试。在这种情况下, 这里所示的方法是推荐使用子域名的方式。

4.3.3 使用 C ++ 代码片段来定义子域

SubDomain 和 Expression Python 类非常方便, 但是它们的使用导致每个节点从 C++ 到 Python 的函数调用在网格中。由于这涉及到巨大的成本, 我们需要做出如果性能是一个问题, 使用 C ++ 代码。

而不是在 Python 中编写 SubDomain 子类, 我们可以改为使用在 FEniCS 中使用 CompiledSubDomain 工具来指定 C++ 中的子域代码, 从而加快我们的代码。考虑类的定义Omega_0 和Omega_1 以上在 Python。该定义这些子域的关键字符串可以用 C++ 语法表示并作为参数给出 CompiledSubDomain, 如下所示:

Python code

```
1 tol = 1E-14
2 subdomain_0 = CompiledSubDomain('x[1] <= 0.5 + tol', tol=tol)
3 subdomain_1 = CompiledSubDomain('x[1] >= 0.5 - tol', tol=tol)
```

如所见, 可以使用关键字参数指定参数。产生的对象, subdomain_0 和subdomain_1, 可以使用作为普通的 SubDomain 对象。

可以应用编译的子域字符串来指定边界好:

Python code

```
1 boundary_R = CompiledSubDomain('on_boundary && near(x[0], 1, tol)',
2                                tol=1E-14)
```

也可以提供 C++ 字符串 (不带参数) 直接作为 DirichletBC 的第三个参数构造一个 CompiledSubDomain 对象:

Python code

```
1 bc1 = DirichletBC(V, value, 'on_boundary && near(x[0], 1, tol)')
```

Python Expression 类也可以使用 C ++ 重新定义更多高效的代码。再次考虑上面的 class K 的定义对于变量系数 $\kappa = \kappa(x)$ 。这可以使用 a 重新定义 C++ 代码片段和关键字 cppcode 到常规 FEniCS Expression 类:

Python code

```
1 cppcode = """
2 class K : public Expression
3 {
4 public:
5
6     void eval(Array<double>& values,
7               const Array<double>& x,
8               const ufc::cell& cell) const
9     {
10        if ((*materials)[cell.index] == 0)
11            values[0] = k_0;
12        else
13            values[0] = k_1;
14    }
15
16    std::shared_ptr<MeshFunction<std::size_t>> materials;
17    double k_0;
18    double k_1;
```

```

19 }
20 };
21 """
22
23 kappa = Expression(cppcode=cppcode, degree=0)
24 kappa.materials = materials
25 kappa.k_0 = k_0
26 kappa.k_1 = k_1

```

4.4 设置多个 Dirichlet, Neumann 和 Robin 条件

再次考虑可变系数 Poisson 问题从部分 4.3。我们现在讨论一下如何实现边界条件的一般组合 Dirichlet, Neumann 和 Robin 类型的这个模型问题。

4.4.1 三种边界条件

我们将我们的边界条件扩展到三种类型: Dirichlet, Neumann 和 Robin。Dirichlet 条件适用于某些部分 $\Gamma_D^0, \Gamma_D^1, \dots$, 的边界:

$$u = u_D^0 \text{ on } \Gamma_D^0, \quad u = u_D^1 \text{ on } \Gamma_D^1, \quad \dots,$$

其中 u_D^i 是规定的函数, $i = 0, 1, \dots$ 。在其他部分, $\Gamma_N^0, \Gamma_N^1, \dots$, 我们有 Neumann 条件:

$$-\kappa \frac{\partial u}{\partial n} = g_0 \text{ on } \Gamma_N^0, \quad -\kappa \frac{\partial u}{\partial n} = g_1 \text{ on } \Gamma_N^1, \quad \dots,$$

最后, 我们有 Robin 条件:

$$-\kappa \frac{\partial u}{\partial n} = r(u - s),$$

其中 r 和 s 是指定的函数。Robin 条件是最常用于将热量传递给周围环境并产生自然是从 Newton 的冷静法。在这种情况下, r 是一个热量传递系数, s 是温度环境。两者都可以是空间和时间依赖的。Robin 条件适用在某些部分 $\Gamma_R^0, \Gamma_R^1, \dots$, 的边界:

$$-\kappa \frac{\partial u}{\partial n} = r_0(u - s_0) \text{ on } \Gamma_R^0, \quad -\kappa \frac{\partial u}{\partial n} = r_1(u - s_1) \text{ on } \Gamma_R^1, \quad \dots$$

4.4.2 PDE 问题

用上面的符号, 模型问题要用多个解决 Dirichlet, Neumann 和 Robin 条件可以表达如下:

$$-\nabla \cdot (\kappa \nabla u) = f \quad \text{in } \Omega, \tag{4.8}$$

$$u = u_D^i \quad \text{on } \Gamma_D^i, \quad i = 0, 1, \dots \tag{4.9}$$

$$-\kappa \frac{\partial u}{\partial n} = g_i \quad \text{on } \Gamma_N^i, \quad i = 0, 1, \dots \tag{4.10}$$

$$-\kappa \frac{\partial u}{\partial n} = r_i(u - s_i) \quad \text{on } \Gamma_R^i, \quad i = 0, 1, \dots \tag{4.11}$$

4.4.3 变化公式

像往常一样，我们乘以一个测试函数 v 并按部分进行集成:

$$-\int_{\Omega} \nabla \cdot (\kappa \nabla u) v \, dx = \int_{\Omega} \kappa \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} \kappa \frac{\partial u}{\partial n} v \, ds.$$

在边界的 Dirichlet 部分 (Γ_D^i)，边界积分 $v = 0$ 以后消失。在边界的剩余部分，我们将边界积分分解为 Neumann 部分的贡献 (Γ_N^i) 和 Robin 部分 (Γ_R^i)。插入边界条件，我们获得

$$\begin{aligned} -\int_{\partial\Omega} \kappa \frac{\partial u}{\partial n} v \, ds &= -\sum_i \int_{\Gamma_N^i} \kappa \frac{\partial u}{\partial n} \, ds - \sum_i \int_{\Gamma_R^i} \kappa \frac{\partial u}{\partial n} \, ds \\ &= \sum_i \int_{\Gamma_N^i} g_i \, ds + \sum_i \int_{\Gamma_R^i} r_i(u - s_i) \, ds. \end{aligned}$$

因此，我们得到以下变分问题:

$$F = \int_{\Omega} \kappa \nabla u \cdot \nabla v \, dx + \sum_i \int_{\Gamma_N^i} g_i v \, ds + \sum_i \int_{\Gamma_R^i} r_i(u - s_i) v \, ds - \int_{\Omega} f v \, dx = 0. \quad (4.12)$$

我们已经习惯了在这个写作中写出这种变异形式标准符号 $a(u, v) = L(v)$ ，这要求我们识别所有积分取决于试验功能 u ，并收集这些 $a(u, v)$ ，而剩余的积分则为 $L(v)$ 。积分由于 Robin 条件必须分为两部分:

$$\int_{\Gamma_R^i} r_i(u - s_i) v \, ds = \int_{\Gamma_R^i} r_i u v \, ds - \int_{\Gamma_R^i} r_i s_i v \, ds.$$

我们有

$$a(u, v) = \int_{\Omega} \kappa \nabla u \cdot \nabla v \, dx + \sum_i \int_{\Gamma_R^i} r_i u v \, ds, \quad (4.13)$$

$$L(v) = \int_{\Omega} f v \, dx - \sum_i \int_{\Gamma_N^i} g_i v \, ds + \sum_i \int_{\Gamma_R^i} r_i s_i v \, ds. \quad (4.14)$$

或者，我们可以保留配方 (4.12)，并解决变分作为 FEniCS 中的非线性问题 ($F == 0$) 或使用 FEniCS 函数 `lhs` 和 `rhs` 来提取双线性和线性部分 F :

Python code

```
1 a = lhs(F)
2 L = rhs(F)
```

请注意，如果我们选择将该线性问题解决为非线性问题，Newton 迭代将在单次迭代中收敛。

4.4.4 FEniCS 实现

让我们来看看如何扩展我们的 Poisson 求解器来处理一般 Dirichlet, Neumann 和 Robin 边界条件的组合。与以前的代码相比，我们必须考虑以下几点扩展:

- 定义边界不同部分的标记。
- 使用标记将边界积分分成部分。

第一个任务的一般方法是标记所需的每个标记 0,1,2 等的边界部分。在这里我们瞄准了单位正方形的四边，标有 $0(x = 0)$, $1(x = 1)$, $2(y = 0)$ 和 $3(y = 1)$ 。标记将使用 `a` 定义 `MeshFunction`, 但与第 4.3 相反, 这不是单元格的一个功能, 而是一个超过网格面的功能。我们使用 `FacetFunction` 这个目的:

Python code

```
1 boundary_markers = FacetFunction('size_t', mesh)
```

如在 4.3 中, 我们使用一个子类 `SubDomain` 来标识网格的各个部分功能。可能设置更复杂几何域的问题用于将边界标记为网格的一部分的网格功能代。在我们的例子中, 可以标记边界 $x = 0$ 如下:

Python code

```
1 class BoundaryX0(SubDomain):
2     tol = 1E-14
3     def inside(self, x, on_boundary):
4         return on_boundary and near(x[0], 0, tol)
5
6 bxo = BoundaryX0()
7 bxo.mark(boundary_markers, 0)
```

类似地, 我们创建类 `BoundaryY0`($y = 0$), `BoundaryY1`($y = 1$) 边界, 并将其标记为子域 1,2 和 3。

为了实现的一般性, 我们让用户指定什么样的边界条件适用于四个中的每一个边界。为此, 我们设置了一个 Python 字典键作为子域号, 值作为字典指定关键的条件和作为其价值的功能。例如,

Python code

```
1 boundary_conditions = {0: {'Dirichlet': u_D},
2                         1: {'Robin': (r, s)},
3                         2: {'Neumann': g},
4                         3: {'Neumann': 0}}
```

指定

- 一个 Dirichlet 条件 $u = u_D$ 对于 $x = 0$;
- 一个 Robin 条件 $-\kappa \partial_n u = r(u - s)$ 对于 $x = 1$;
- 一个 Neumann 条件 $-\kappa \partial_n u = g$ 对于 $y = 0$;
- 一个 Neumann 条件 $-\kappa \partial_n u = 0$ 对于 $y = 1$.

如在 4.2 中所述的多个 Dirichlet 条件必须收集在 `DirichletBC` 的列表中对象。基于 `boundary_conditions` 数据结构如上, 我们可以通过以下代码片段构建此列表:

Python code

```
1 bcs = []
2 for i in boundary_conditions:
3     if 'Dirichlet' in boundary_conditions[i]:
4         bc = DirichletBC(V, boundary_conditions[i]['Dirichlet'],
5                           boundary_markers, i)
6         bcs.append(bc)
```

变分问题的一个新方面是两个不同的边界积分超过 Γ_N^i 和 Γ_R^i 。在外部细胞方面有一个网格功能 (我们的 boundary_markers 对象), 子域(边界部分) 编号为 0,1,2,..., 特殊符号 ds(0) 意味着在子域(部分)0, ds(1) 表示的集成整合子域(部分)1, 依此类推。多个 ds 类型对象的想法概括为卷积分:dx(0), dx(1) 等, 用于在 Ω 内的子域 0, 1 等集成。

为了使用 ds(i) 在边界部分表达积分, 我们必须首先根据边界标记重新定义度量 ds:

Python code

```
1 ds = Measure('ds', domain=mesh, subdomain_data=boundary_markers)
```

同样, 如果我们想要在域的不同部分进行集成, 我们将 dx 重新定义为

Python code

```
1 dx = Measure('dx', domain=mesh, subdomain_data=domain_markers)
```

domain_markers 是 CellFunction 定义 Ω 中的子域名。

假设我们在子域上具有值为 r 和 s 的 Robin 条件 R, 在子域 N 上的值为 g 的 Neumann 条件。该然后可以写出变化形式

Python code

```
1 a = kappa*dot(grad(u), grad(v))*dx + r*u*v*ds(R)
2 L = f*v*dx - g*v*ds(N) + r*s*v*ds(R)
```

在我们的情况下, 事情变得复杂了一些关于 Neumann 和 Robin 条件的积分的信息在 boundary_conditions 数据结构。我们可以收集所有 Neumann 条件由以下代码片段:

Python code

```
1 integrals_N = []
2 for i in boundary_conditions:
3     if 'Neumann' in boundary_conditions[i]:
4         if boundary_conditions[i]['Neumann'] != 0:
5             g = boundary_conditions[i]['Neumann']
6             integrals_N.append(g*v*ds(i))
```

应用sum(integrals_N) 将 + 运算符应用到变体形式在integrals_N 列表和结果在积分中, 我们需要右边的 L 变化形式。

Robin 条件的积分可以同样收集列表中:

Python code

```
1 integrals_R_a = []
2 integrals_R_L = []
3 for i in boundary_conditions:
4     if 'Robin' in boundary_conditions[i]:
5         r, s = boundary_conditions[i]['Robin']
6         integrals_R_a.append(r*u*v*ds(i))
7         integrals_R_L.append(r*s*v*ds(i))
```

我们现在可以定义 a 和 L 表达式在变分公式中:

Python code

```
1 a = kappa*dot(grad(u), grad(v))*dx + sum(integrals_R_a)
2 L = f*v*dx - sum(integrals_N) + sum(integrals_R_L)
```

或者, 我们可以使用 FEniCS 函数 lhs 和 rhs 作为以简化 Robin 的术语提取积分:

Python code

```
1 integrals_R = []
```

```

2 | for i in boundary_conditions:
3 |     if 'Robin' in boundary_conditions[i]:
4 |         r, s = boundary_conditions[i]['Robin']
5 |         integrals_R.append(r*(u - s)*v*ds(i))
6 |
7 F = kappa*dot(grad(u), grad(v))*dx + \
8     sum(integrals_R) - f*v*dx + sum(integrals_N)
9 a, L = lhs(F), rhs(F)

```

这一次，我们可以更自然地定义积分 Robin 条件为 $r*(u - s)*v*ds(i)$ 。

完整的代码可以在函数中找到 solver_bcs 在程序中

https://fenicsproject.org/pub/tutorial/python/vol1/ft10_poisson_extended.py。

4.4.5 测试问题

我们将使用相同的确切解决方案 $u_e = 1 + x^2 + 2y^2$, 如“2”所示, 因此需要 $\kappa = 1$ 和 $f = -6$ 。我们的域名是单位平方, 我们将 Dirichlet 条件分配给 $x = 0$ 和 $x = 1$, $y = 0$ 的 Robin 条件, 以及 Neumann 条件 $y = 1$ 。给定的确切解决方案 u_e , 我们意识到 $y = 1$ 的 Neumann 条件是 $-\partial u / \partial n = -\partial u / \partial y = 4y = 4$, 而可以选择 $y = 0$ 的 Robin 条件很多方法。由于 $-\partial u / \partial n = \partial u / \partial y = 0$ at $y = 0$, 我们可以选择 $s = u_e$ 并指定 $r \neq 0$ Robin 条件。我们将设置 $r = 1000$ 和 $s = u_e$ 。

因此边界部分 Γ_D^0 : $x = 0$, Γ_D^1 : $x = 1$, Γ_R^0 : $y = 0$, 和 Γ_N^0 : $y = 1$ 。

在实施这个测试问题时, 尤其是其他测试使用更复杂的表达式的问题, 有利于使用符号计算。下面我们将确切的解决方案定义为 sympy 表达并从他们的数学派生其他功能定义。然后我们将这些表达式转换成 C/C++ 代码, 然后可以用来定义 Expression 对象。

Python code

```

1 # Define manufactured solution in sympy and derive f, g, etc.
2 import sympy as sym
3 x, y = sym.symbols('x[0], x[1]')
4 u = 1 + x**2 + 2*y**2
5 u_e = u
6 u_00 = u.subs(x, 0)
7 u_01 = u.subs(x, 1)
8 f = -sym.diff(u, x, 2) - sym.diff(u, y, 2)
9 f = sym.simplify(f)
10 g = -sym.diff(u, y).subs(y, 1)
11 r = 1000
12 s = u
13
14 # Collect variables
15 variables = [u_e, u_00, u_01, f, g, r, s]
16
17 # Turn into C/C++ code strings
18 variables = [sym.printing.ccode(var) for var in variables]
19
20 # Turn into FEniCS Expressions
21 variables = [Expression(var, degree=2) for var in variables]
22
23 # Extract variables
24 u_e, u_00, u_01, f, g, r, s = variables
25
26 # Define boundary conditions
27 boundary_conditions = {0: {'Dirichlet': u_00}, # x = 0

```

```

28     1: {'Dirichlet': u_01}, # x = 1
29     2: {'Robin': (r, s)}, # y = 0
30     3: {'Neumann': g}} # y = 1

```

完整的代码可以在函数中找到 demo_bcs 在程序中

https://fenicsproject.org/pub/tutorial/python/vol1/ft10_poisson_extended.py。

4.4.6 调试边界条件

在执行许多问题时很容易犯错误不同类型的边界条件，如本案。一调试边界条件的方法是遍历所有顶点协调并检查 SubDomain.inside 方法是否标记顶点在边界上。另一个有用的方法是列出哪个受 Dirichlet 条件影响的自由度一级 Lagrange(P_1) 元素，打印对应的顶点坐标如下图所示代码段：

Python code

```

1 if debug1:
2
3     # Print all vertices that belong to the boundary parts
4     for x in mesh.coordinates():
5         if bx0.inside(x, True): print('%s is on x = 0' % x)
6         if bx1.inside(x, True): print('%s is on x = 1' % x)
7         if by0.inside(x, True): print('%s is on y = 0' % x)
8         if by1.inside(x, True): print('%s is on y = 1' % x)
9
10    # Print the Dirichlet conditions
11    print('Number of Dirichlet conditions:', len(bcs))
12    if V.ufl_element().degree() == 1: # P1 elements
13        d2v = dof_to_vertex_map(V)
14        coor = mesh.coordinates()
15        for i, bc in enumerate(bcs):
16            print('Dirichlet condition %d' % i)
17            boundary_values = bc.get_boundary_values()
18            for dof in boundary_values:
19                print('    dof %2d: u = %g, % (dof, boundary_values[dof]))')
20                if V.ufl_element().degree() == 1:
21                    print('        at point %s, %
22                          (str(tuple(coor[d2v[dof]].tolist())))))

```

注意

(**调用 inside 方法**) 在上面的代码片段中，我们称之为每个的 inside 方法网格的坐标。我们也可以打印输出 inside 方法。那么这个方法就是令人惊讶的不仅要求与自由度相关的要点。对于 P_1 元素，也为每个元素调用该方法细胞每一面的中点。这是因为 Dirichlet 默认情况下，只有整个方面可以说是设置的在限定边界的条件下。

4.5 用子域生成网格

到目前为止，我们的工作主要是简单的网格（单位面积）和根据简单的几何测试定义边界和子域像 $x = 0$ 或 $y \leq 0.5$ 。对于更复杂的几何，它不是以这种方式指定边界和子域的现实。代替，边界和子域必须定义为网格的一部分生成过程。现在我们来看一下如何使用 FEniCS 网格生成工具 mshr 以生成网格并定义子域。

4.5.1 PDE 问题

我们将再次解决 Poisson 方程，但这次为一个不同的应用。考虑一个带铜线的铁芯如图 4.2 所示。铁芯由一个外圆和一个内圆组成，中间有 8 个槽。通过铜线的静态电流 $J = 1\text{ A}$ 正在流动，我们要计算铁芯中的磁场 B ，铜线，和周围的真空。

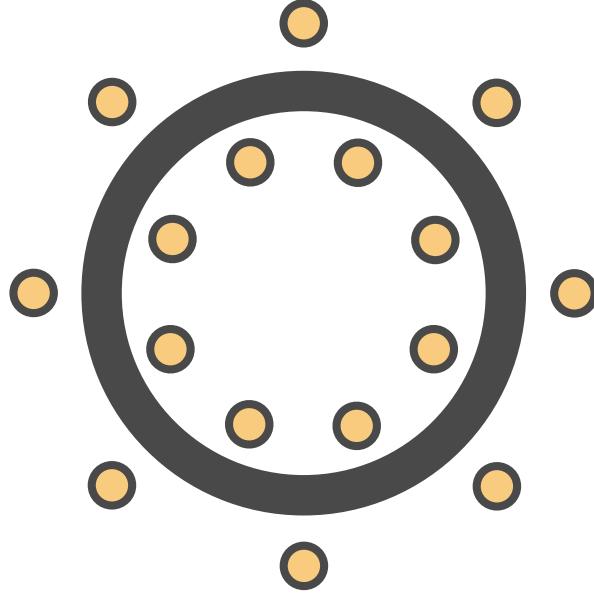


Figure 4.2: 在圆柱体上缠绕铜线的铁芯横截面，这里以 $n = 8$ 的绕组。内圆是铜线的横截面（“北”），外圆是铜线向下进入平面（“南”）的横截面。

首先，我们将问题简化为 2D 问题。我们可以这样做假设圆柱体沿着 z -axis 和 a 延伸很远，后果是该领域几乎独立于 Z -协调。接下来，我们考虑 Maxwell 的方程式得出 a 磁场的 Poisson 方程（或其电位）：

$$\nabla \cdot D = \rho, \quad (4.15)$$

$$\nabla \cdot B = 0, \quad (4.16)$$

$$\nabla \times E = -\frac{\partial B}{\partial t}, \quad (4.17)$$

$$\nabla \times H = \frac{\partial D}{\partial t} + J. \quad (4.18)$$

这里， D 是位移场， B 是磁性的场， E 是电场， H 是磁场。在除了 Maxwell 方程之外，我们还需要一个本构关系在 B 和 H 之间，

$$B = \mu H, \quad (4.19)$$

其适用于各向同性线性磁介质。这里， μ 是材料的磁导率。现在，因为 B 是螺线管的（不分散）根据 Maxwell 方程，我们知道 B 必须是一些矢量字段 A 的卷曲。这个字段叫做磁矢量电位。由于问题是静态的，因此 $\partial D / \partial t = 0$ ，就是这样

$$J = \nabla \times H = \nabla \times (\mu^{-1} B) = \nabla \times (\mu^{-1} \nabla \times A) = -\nabla \cdot (\mu^{-1} \nabla A). \quad (4.20)$$

在最后一步中，我们扩大了二次衍生品并使用了量纲自由 A 简化方程式简单磁矢量势的向量值 Poisson 问题；如果 $B = \nabla \times A$ ，然后 $B = \nabla \times (A + \nabla \psi)$ 标量场 ψ （量表功能）。对于目前的问题，我们因此需要解决以下 2D Poisson 问题 z -组件 A_z 的磁矢量电位：

$$-\nabla \cdot (\mu^{-1} \nabla A_z) = J_z \quad \text{in } \mathbb{R}^2, \quad (4.21)$$

$$\lim_{|(x,y)| \rightarrow \infty} A_z = 0. \quad (4.22)$$

由于我们无法在无限的域上解决这个问题，所以我们会使用大型磁盘截断域，并设置 $A_z = 0$ 边界。当前的 J_z 设置为 $+1\text{ A}$ 在内部一组圈（铜线横截面）和 -1 A 图中的外部圆圈 4.2。

一旦计算了磁矢量电位，我们就可以计算磁场 $B = B(x, y)$

$$B(x, y) = \left(\frac{\partial A_z}{\partial y}, -\frac{\partial A_z}{\partial x} \right). \quad (4.23)$$

4.5.2 变化公式

变分问题是通过乘以 PDE 得到的具有 v 的测试函数，并以部件集成。自边界由于 Dirichlet 条件，积分消失，我们得到

$$\int_{\Omega} \mu^{-1} \nabla A_z \cdot \nabla v \, dx = \int_{\Omega} J_z v \, dx, \quad (4.24)$$

或者换句话说， $a(A_z, v) = L(v)$ 同

$$a(A_z, v) = \int_{\Omega} \mu^{-1} \nabla A_z \cdot \nabla v \, dx, \quad (4.25)$$

$$L(v) = \int_{\Omega} J_z v \, dx. \quad (4.26)$$

4.5.3 FEniCS 实现

第一步是为几何描述生成一个网格图 4.2。我们让 a 和 b 成为铁缸的内外半径，让 c_1 和 c_2 是铜线的两个同心分布的半径交叉区域。此外，我们让 r 是铜的半径线， R 是我们域的半径， n 是数绕组（总共 $2n$ 铜线截面）。这个几何可以很容易的用 mshr 和一点点描述 Python 编程：

Python code

```

1 # Define geometry for background
2 domain = Circle(Point(0, 0), R)
3
4 # Define geometry for iron cylinder
5 cylinder = Circle(Point(0, 0), b) - Circle(Point(0, 0), a)
6
7 # Define geometry for wires (N = North (up), S = South (down))
8 angles_N = [i*2*pi/n for i in range(n)]
9 angles_S = [(i + 0.5)*2*pi/n for i in range(n)]
10 wires_N = [Circle(Point(c_1*cos(v), c_1*sin(v)), r) for v in angles_N]
11 wires_S = [Circle(Point(c_2*cos(v), c_2*sin(v)), r) for v in angles_S]
```

我们生成的网格将是整个磁盘的网格半径 R ，但我们需要网格生成来尊重内部由铁缸和铜线限定的边界。我们也想要 mshr 标记子域，以便我们可以轻松指定材料参数 (μ) 和电流。为此，我们使用 mshr 功能 set_subdomain 如下：

Python code

```

1 # Set subdomain for iron cylinder
2 domain.set_subdomain(1, cylinder)
3
4 # Set subdomains for wires
5 for (i, wire) in enumerate(wires_N):
6     domain.set_subdomain(2 + i, wire)
7 for (i, wire) in enumerate(wires_S):
8     domain.set_subdomain(2 + n + i, wire)

```

一旦子域被创建，我们可以生成网格：

Python code

```
1 mesh = generate_mesh(domain, 128)
```

网格的细节如图 4.3 所示。

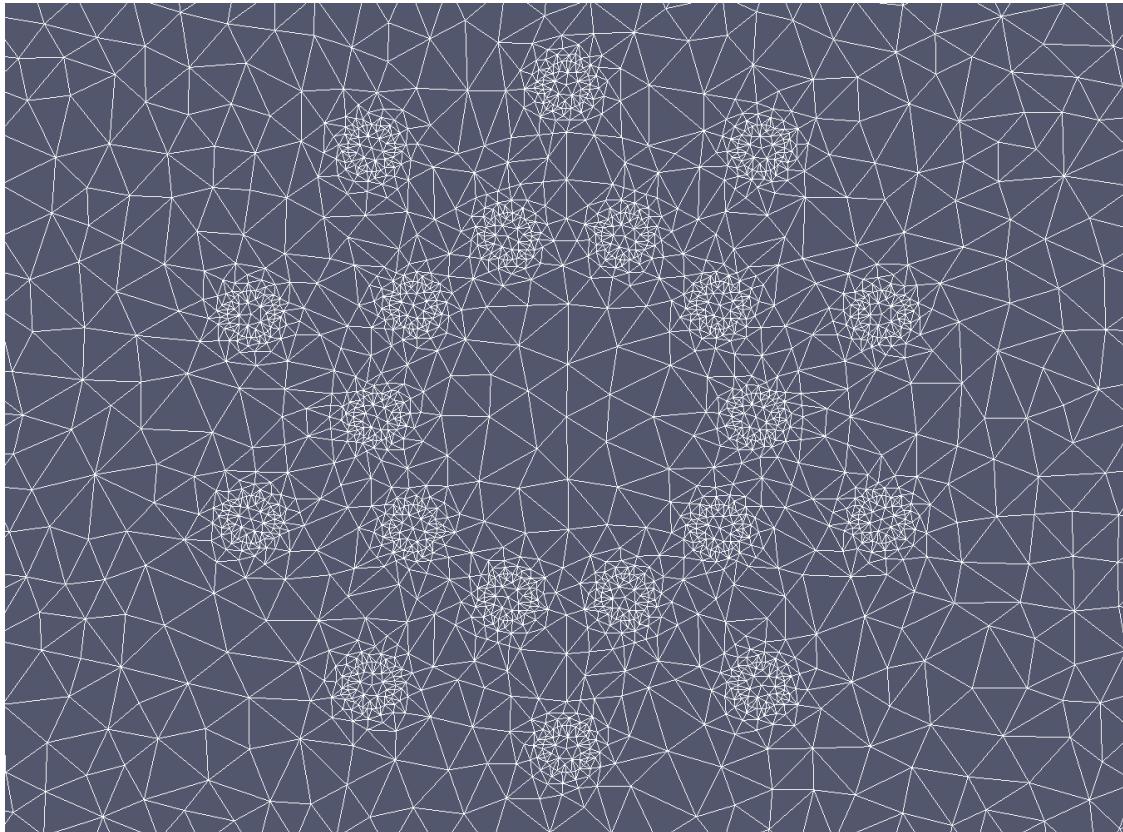


Figure 4.3: 绘制（部分）为静磁测试问题产生的网格。铁桶和铜线的子域清晰可见。

使用 mshr 生成的网格将包含有关的信息我们定义的子域。在定义中使用这些信息我们的变分问题和子域依赖参数，我们将需要创建一个标记子域的 MeshFunction。这可以很容易通过调用成员函数 mesh.domains 创建的 mshr 生成的子域数据：

Python code

```
1 markers = MeshFunction('size_t', mesh, 2, mesh.domains())
```

该行用无符号整数值创建一个 MeshFunction 子域号，其尺寸为 2，其为单元格维度这个 2D 问题。

我们现在可以像以前那样使用标记来重新定义整合度量 dx :

```
Python code
1 dx = Measure('dx', domain=mesh, subdomain_data=markers)
```

然后可以通过 $dx(0)$, $dx(1)$ 来表示子域上的积分, 等等。我们用它来定义当前的 $J_z = \pm 1 \text{ A}$ 在铜线上:

```
Python code
1 J_N = Constant(1.0)
2 J_S = Constant(-1.0)
3 A_z = TrialFunction(V)
4 v = TestFunction(V)
5 a = (1 / mu)*dot(grad(A_z), grad(v))*dx
6 L_N = sum(J_N*v*dx(i) for i in range(2, 2 + n))
7 L_S = sum(J_S*v*dx(i) for i in range(2 + n, 2 + 2*n))
8 L = L_N + L_S
```

Permeability 被定义为取决于该文件的 Expression 子域号:

```
Python code
1 class Permeability(Expression):
2     def __init__(self, markers, **kwargs):
3         self.markers = markers
4     def eval_cell(self, values, x, cell):
5         if self.markers[cell.index] == 0:
6             values[0] = 4*pi*1e-7 # vacuum
7         elif self.markers[cell.index] == 1:
8             values[0] = 1e-5      # iron (should really be 6.3e-3)
9         else:
10            values[0] = 1.26e-6 # copper
11
12 mu = Permeability(markers, degree=1)
```

从这段代码片段可以看出, 我们使用的是一个稍微不那么的极端铁的磁导率值。这是做的解决方案有点有趣。否则将是完全的以铁缸为主。

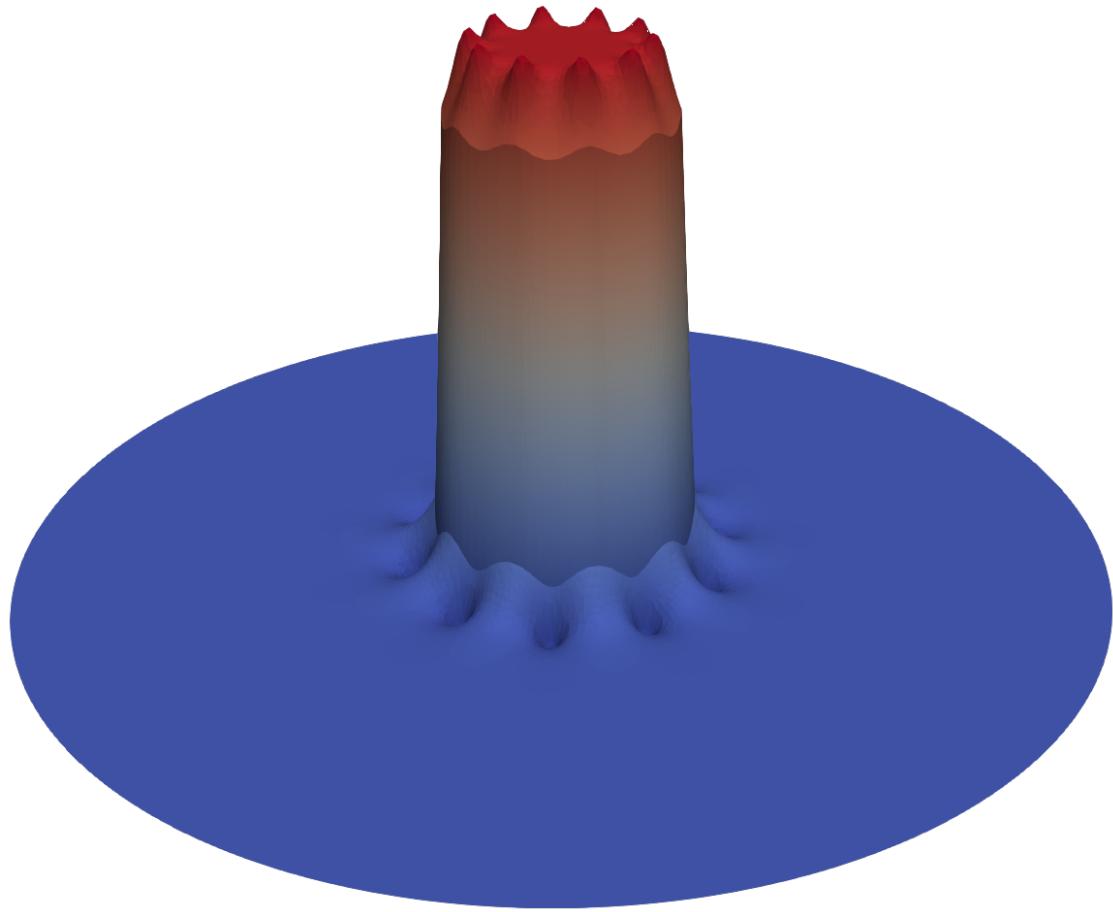
最后, 当计算 A_z 时, 我们可以计算磁性领域:

```
Python code
1 W = VectorFunctionSpace(mesh, 'P', 1)
2 B = project(as_vector((A_z.dx(1), -A_z.dx(0))), W)
```

我们使用 `as_vector` 解释 $(A_z.dx(1), -A_z.dx(0))$ 作为 UFL 意义上的向量形式语言, 而不是 Python 元组。所得的地块磁矢量电位和磁场如图 4.4 和 4.5 所示。

计算磁场的完整代码如下。

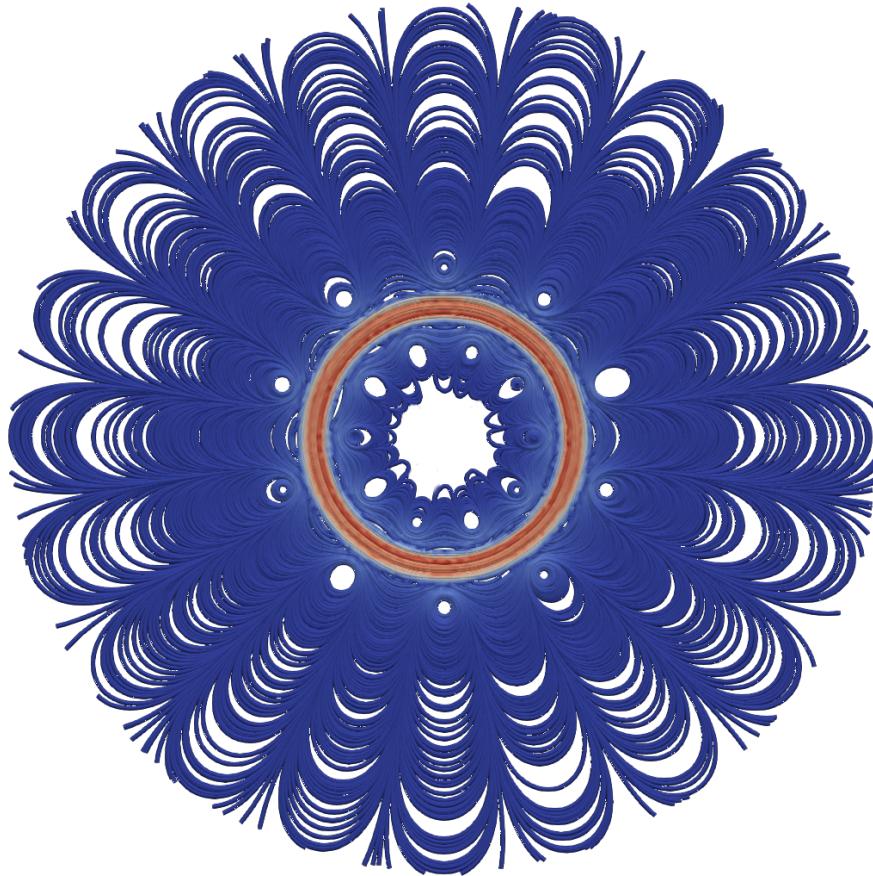
```
Python code
1 from fenics import *
2 from mshr import *
3 from math import sin, cos, pi
4
5 a = 1.0 # inner radius of iron cylinder
6 b = 1.2 # outer radius of iron cylinder
7 c_1 = 0.8 # radius for inner circle of copper wires
8 c_2 = 1.4 # radius for outer circle of copper wires
9 r = 0.1 # radius of copper wires
10 R = 5.0 # radius of domain
```

Figure 4.4: 剧情 z 元素 A_z 的磁矢量势。

```

11 n = 10      # number of windings
12
13 # Define geometry for background
14 domain = Circle(Point(0, 0), R)
15
16 # Define geometry for iron cylinder
17 cylinder = Circle(Point(0, 0), b) - Circle(Point(0, 0), a)
18
19 # Define geometry for wires (N = North (up), S = South (down))
20 angles_N = [i*2*pi/n for i in range(n)]
21 angles_S = [(i + 0.5)*2*pi/n for i in range(n)]
22 wires_N = [Circle(Point(c_1*cos(v), c_1*sin(v)), r) for v in angles_N]
23 wires_S = [Circle(Point(c_2*cos(v), c_2*sin(v)), r) for v in angles_S]
24
25 # Set subdomain for iron cylinder
26 domain.set_subdomain(1, cylinder)
27
28 # Set subdomains for wires
29 for (i, wire) in enumerate(wires_N):
30     domain.set_subdomain(2 + i, wire)
31 for (i, wire) in enumerate(wires_S):
32     domain.set_subdomain(2 + n + i, wire)
33

```

Figure 4.5: 在 xy 平面上绘制磁场 B 。

```

34 # Create mesh
35 mesh = generate_mesh(domain, 32)
36
37 # Define function space
38 V = FunctionSpace(mesh, 'P', 1)
39
40 # Define boundary condition
41 bc = DirichletBC(V, Constant(0), 'on_boundary')
42
43 # Define subdomain markers and integration measure
44 markers = MeshFunction('size_t', mesh, 2, mesh.domains())
45 dx = Measure('dx', domain=mesh, subdomain_data=markers)
46
47 # Define current densities
48 J_N = Constant(1.0)
49 J_S = Constant(-1.0)
50
51 # Define magnetic permeability
52 class Permeability(Expression):
53     def __init__(self, markers, **kwargs):
54         self.markers = markers
55     def eval_cell(self, values, x, cell):
56         if self.markers[cell.index] == 0:
57             values[0] = 4*pi*1e-7 # vacuum

```

```
58     elif self.markers[cell.index] == 1:
59         values[0] = 1e-5      # iron (should really be 6.3e-3)
60     else:
61         values[0] = 1.26e-6  # copper
62
63 mu = Permeability(markers, degree=1)
64
65 # Define variational problem
66 A_z = TrialFunction(V)
67 v = TestFunction(V)
68 a = (1 / mu)*dot(grad(A_z), grad(v))*dx
69 L_N = sum(J_N*v*dx(i) for i in range(2, 2 + n))
70 L_S = sum(J_S*v*dx(i) for i in range(2 + n, 2 + 2*n))
71 L = L_N + L_S
72
73 # Solve variational problem
74 A_z = Function(V)
75 solve(a == L, A_z, bc)
76
77 # Compute magnetic field (B = curl A)
78 W = VectorFunctionSpace(mesh, 'P', 1)
79 B = project(as_vector((A_z.dx(1), -A_z.dx(0))), W)
80
81 # Plot solution
82 plot(A_z)
83 plot(B)
84
85 # Save solution to file
86 vtkfile_A_z = File('magnetostatics/potential.pvd')
87 vtkfile_B = File('magnetostatics/field.pvd')
88 vtkfile_A_z << A_z
89 vtkfile_B << B
90
91 # Hold plot
92 interactive()
```

该示例程序可以在文件中找到

https://fenicsproject.org/pub/tutorial/python/vol1/ft11_magnetostatics.py。

5. 扩展：改进 Poisson 求解器

5.1	重构 Poisson 求解器	86
5.2	使用线性求解器	90
5.3	高级和低级解算器接口	93
5.4	自由度和功能评估	96
5.5	后处理计算	100
5.6	下一步	111

我们迄今为止撰写的 FENICS 计划已经设计为平面设计 Python 脚本。这对于解决简单的演示非常有用问题。但是，当您构建高级的求解器时应用程序，你会很快找到需要更多的结构化节目。特别是，您可能希望重用您的求解器来解决大量的问题，你们改变边界条件，域，以及材料参数等系数。在本章中，我们将看到如何编写一般的求解器函数来改进 FEnICS 计划的可用性。我们也会讨论如何利用具有预处理器的迭代求解器来求解线性系统，如何计算派生数量，例如通量在边界的一部分，以及如何计算错误和收敛率。

5.1 重构 Poisson 求解器

本书中讨论的大多数程序是“平”就是他们是在 Python 方面没有组织成逻辑，可重复使用的单位功能。这样的平面程序对于快速测试想法是有用的素描解算法，但不适合严重解决问题因此，我们将看看如何 refactor Poisson 求解器从 Chapter 2。一开始就这样意味着将代码分解为函数。但重构不仅仅是一个重新排列现有声明。在重构期间，我们也尝试使我们创建的功能尽可能在其他方面可重用上下文。我们也会封装一些具体的陈述问题变成(不可重用)的功能。能够区分当重构时，专用代码的可重用代码是一个关键问题代码，这个能力取决于一个很好的数学理解手头的问题(什么是一般的，什么是特殊的)。在一个单位程序，一般和专门的代码(和数学)经常混合在一起，这往往给人一种模糊的理解手头的问题。

5.1.1 更一般的求解器函数

我们考虑平面计划

https://fenicsproject.org/pub/tutorial/python/vol1/ft01_poisson.py

解决了 Poisson 问题的开发在章节 2。这个程序中的一些代码需要解决任何 Poisson 问题 $-\nabla^2 u = f$ on $[0, 1] \times [0, 1]$ 与 $u = u_b$ 在边界上，而其他语句来自我们简单的测试问题。让我们收集一般的，可重复使用的代码一个名为 solver 的函数。我们的特殊测试问题就是这样我们的 solver 的一个应用程序附加了一些其他语句。我们限制 solver 函数来计算数值解。绘制和比较解决方案与确切的解决方案被认为是要执行的特定于问题的活动别处。

我们将 solver 参数化为 f , u_D 和分辨率目。因为使用高阶有限元是如此微不足道函数通过将第三个参数改为 FunctionSpace, 我们还加入有限元函数空间的多项式度作为 solver 的参数。

Python code

```

1 from fenics import *
2 import numpy as np
3
4 def solver(f, u_D, Nx, Ny, degree=1):
5     """
6     Solve -Laplace(u) = f on [0,1] x [0,1] with 2*Nx*Ny Lagrange
7     elements of specified degree and u = u_D (Expression) on
8     the boundary.
9     """
10
11    # Create mesh and define function space
12    mesh = UnitSquareMesh(Nx, Ny)
13    V = FunctionSpace(mesh, 'P', degree)
14
15    # Define boundary condition
16    def boundary(x, on_boundary):
17        return on_boundary
18
19    bc = DirichletBC(V, u_D, boundary)
20
21    # Define variational problem
22    u = TrialFunction(V)
23    v = TestFunction(V)
24    a = dot(grad(u), grad(v))*dx
25    L = f*v*dx
26
27    # Compute solution
28    u = Function(V)
29    solve(a == L, u, bc)
30
31    return u

```

我们的初始程序的其余任务, 如调用 solver 功能与问题特定的参数和绘图, 可以放在一个单独的功能。这里我们选择放这个代码在一个名为run_solver的函数中:

Python code

```

1 def run_solver():
2     "Run solver to compute and post-process solution"
3
4     # Set up problem parameters and call solver
5     u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', degree=2)
6     f = Constant(-6.0)
7     u = solver(f, u_D, 8, 8, 1)
8
9     # Plot solution and mesh
10    plot(u)
11    plot(u.function_space().mesh())
12
13    # Save solution to file in VTK format
14    vtkfile = File('poisson_solver/solution.pvd')
15    vtkfile << u

```

该解决方案现在可以被计算, 绘制和保存到文件只需调用run_solver 功能。

5.1.2 将求解器写为 Python 模块

重构的代码放在一个文件中

https://fenicsproject.org/pub/tutorial/python/vol1/ft12_poisson_solver.py。

我们应该确保这样的文件可以导入（因此重用）在其他程序。这意味着所有的语句在主不在功能内的程序应该出现在测试中 if `__name__ == '__main__'`: 如果文件被执行，则该测试是真实的一个程序，但是假的如果导入文件。如果我们想运行这个文件的方式与我们可以运行 `ft01_poisson.py`, 相同主程序只是一个调用 `run_solver` 其次是电话 `interactive` 保存情节:

Python code

```

1 if __name__ == '__main__':
2     run_solver()
3     interactive()
```

这个完整的程序可以在文件中找到

https://fenicsproject.org/pub/tutorial/python/vol1/ft12_poisson_solver.py。

5.1.3 验证和单元测试

我们的第一个程序的剩余部分是比较数字和确切的解决方案。每次我们编辑代码，我们必须重新运行测试并检查 `error_max` 足够小，所以我们知道该代码仍然有效。为此，我们将采用单元测试，意味着我们创建了一个数学测试和相应的软件可以自动运行所有测试，并检查所有测试通过。Python 有几个单元测试工具。两个很受欢迎一个是 `pytest` 和 `nose`。这几乎是同样的，非常容易使用。通过测试类提供更经典的单元测试内置模块 `unittest`，但是这里我们要使用 `pytest`（或 `nose`），因为这将导致更短和更清晰的代码。

在数学上，我们的单元测试是有限元解我们的问题当 $f = -6$ 等于确切的解决方案 $u = u_D = 1 + x^2 + 2y^2$ 在网格的顶点。我们已经创建了一个在顶点找到错误的代码我们的数值解。由于四舍五入的错误，我们不能要求这个错误为零，但是我们必须使用一个容差，哪个取决于元素的数量和多项式的程度在有限元的基础上。如果我们要测试那个 `solver` 函数适用于高达 $2 \times (20 \times 20)$ 的网格元素和立方体 Lagrange 元素， 10^{-10} 是适当的容忍测试最大误差消失。

为了使我们的测试用例与 `pytest` 和 `nose` 一起工作，我们必须对我们的程序进行几个小的调整。简单规则是每个测试都必须放在一个函数中

- 名字以 `test_` 开头
- 没有论据，
- 执行表示为 `assert success, msg` 的测试。

关于最后一点，`success` 是一个布尔表达式 `False` 如果测试失败，在这种情况下，字符串 `msg` 是写入屏幕。当测试失败时，`assert` 会引发一个 Python 中的 `AssertionError` 异常，否则运行默默。`msg` 字符串是可选的，所以 `assert success` 是最小测试。在我们的例子中，我们会写 `assert error_max < tol`，其中 `tol` 是上述容限。

在该单元测试中执行适当的测试功能 `pytest` 或 `nose` 测试框架具有以下形式。注意我们对不同的网格分辨率和度数执行测试有限元。

Python code

```

1 def test_solver():
2     "Test solver by reproducing  $u = 1 + x^2 + 2y^2$ "
3
4     # Set up parameters for testing
5     tol = 1E-10
6     u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', degree=2)
7     f = Constant(-6.0)
8
9     # Iterate over mesh sizes and degrees
10    for Nx, Ny in [(3, 3), (3, 5), (5, 3), (20, 20)]:
11        for degree in 1, 2, 3:
12            print('Solving on a 2 x (%d x %d) mesh with P%d elements.' %
13                  (Nx, Ny, degree))
14
15        # Compute solution
16        u = solver(f, u_D, Nx, Ny, degree)
17
18        # Extract the mesh
19        mesh = u.function_space().mesh()
20
21        # Compute maximum error at vertices
22        vertex_values_u_D = u_D.compute_vertex_values(mesh)
23        vertex_values_u = u.compute_vertex_values(mesh)
24        error_max = np.max(np.abs(vertex_values_u_D - \
25                            vertex_values_u))
26
27        # Check maximum error
28        msg = 'error_max = %g' % error_max
29        assert error_max < tol, msg

```

要运行测试，我们键入以下命令：

Bash code

```
1 $ py.test ft12_poisson_solver.py
```

这将运行所有名为`test_*`的函数（目前只有`test_solver`函数）在文件中找到并报告结果。对于更详细的输出，请添加`-s -v`。

我们将会习惯于将数字测试问题包含进来单元测试如上，我们强烈鼓励读者创建每当实施 FEniCS 求解器时，进行类似的单元测试。

注意

(提示: 在测试功能中打印消息) 当测试通过时，`assert` 语句静默运行，以便用户可以测试功能中的所有语句是否真的会变得不确定执行。心理帮助是在`assert` 之前打印出一些东西（正如我们在上面的例子中所做的那样），这样很清楚测试真的发生了。请注意，`py.test` 需要`-s` 选项来显示打印输出从测试功能。

注意

(提示: 使用 iPython 进行调试) 可以通过添加以下内容从 Python 脚本轻松输入 iPython 代码中的任何位置：

Python code

```
1 from IPython import embed; embed()
```

这一行开始一个交互式的 Python 会话，让你打印和绘图变量，这对调试非常有帮助。

5.1.4 参数化空间维数

FEniCS 可以很容易地编写一个可以统一的模拟代码操作在 1D, 2D 和 3D。作为开胃菜，回到以前的节目

https://fenicsproject.org/pub/tutorial/python/vol1/ft01_poisson.py 要么
https://fenicsproject.org/pub/tutorial/python/vol1/ft12_poisson_solver.py

并将网格构造从 UnitSquareMesh(8,8) 更改为 UnitCubeMesh(8,8,8)。现在域是单位立方体分为 $8 \times 8 \times 8$ 盒子和每个盒子分为六个四面体形有限元计算。运行程序，观察我们可以解决 3D 问题没有任何其他修改！（在 1D 中，表达式必须是修改为不依赖于 $x[1]$ ）。可视化允许你旋转立方体并观察功能值作为颜色边界。

如果我们要参数化单位间隔的创建，单位平方，或单位立方体尺寸，我们可以通过封装这部分来实现的函数中的代码。给定一个列表或元组指定除法进入空格坐标中的单元格，具有以下功能返回 d 维数多维数据集的网格：

Python code

```
1 def UnitHyperCube(divisions):
2     mesh_classes = [UnitIntervalMesh, UnitSquareMesh, UnitCubeMesh]
3     d = len(divisions)
4     mesh = mesh_classes[d - 1](*divisions)
5     return mesh
```

施工 `mesh_class[d - 1]` 会选出正确的名字用于定义域并生成网格的对象。而且，参数 `* divisions` 发送列表的所有组件 `divisions` 作为网格构造的构造函数的单独参数类由 `mesh_class[d - 1]` 挑选出来。例如，在 2D 问题中其中 `divisions` 有两个元素，即语句

Python code

```
1 mesh = mesh_classes[d - 1](*divisions)
```

相当于

Python code

```
1 mesh = UnitSquareMesh(divisions[0], divisions[1])
```

solver 函数

https://fenicsproject.org/pub/tutorial/python/vol1/ft12_poisson_solver.py

可以通过替换来修改 d 维度问题 N_x 和 N_y 参数由 `divisions` 调用，并调用该函数 `UnitHyperCube` 创建网格。注意 `UnitHyperCube` 是一个 `function` 而不是 `class`，但是我们使用所谓的命名它 `CamelCase` 符号使其看起来像一个类：

Python code

```
1 mesh = UnitHyperCube(divisions)
```

5.2 使用线性求解器

默认使用稀疏 LU 分解 (Gaussian 消除) 在 FEniCS 程序中求解线性方程组。这是非常鲁棒简单的方法。这是系统的推荐方法最多有几千个未知数，因此可能是其中的一种许多 2D 和

更小的 3D 问题的选择。但是，稀疏 LU 分解变慢，一个快速耗尽内存更大的问题对于大问题，我们需要使用 iterative 方法它们更快，需要更少的内存。我们现在看看如何利用最先进的迭代解决方案 FEniCS 中的方法。

5.2.1 选择线性求解器和预处理器

预处理 Krylov 求解器是一种流行的迭代方法在 FEniCS 程序中可以轻松访问。Poisson 方程产生一个对称的正定系统矩阵，为此，最优 Krylov 求解器是 Conjugate Gradient(CG) 方法。对于非对称问题，非对称系统的 Krylov 求解器，如 GMRES，是一个更好的选择。不完整 LU 分解 (ILU) 是一个流行和强大的全面预处理器，所以让我们试试 GMRES-ILU 对：

Python code

```

1 solve(a == L, u, bc,
2        solver_parameters={'linear_solver': 'gmres',
3                            'preconditioner': 'ilu'})
4 # Alternative syntax
5 solve(a == L, u, bc,
6       solver_parameters=dict(linear_solver='gmres',
7                               preconditioner='ilu'))

```

部分 5.2.5 列出了最受欢迎的选择 Krylov 解决方案和预处理器可用于 FEniCS。

5.2.2 选择线性代数后端

实际的 GMRES 和 ILU 实施被采取行动取决于线性代数包的选择。FEniCS 接口几个线性代数包，称为 linear algebra backends FEniCS 术语。如果 FEniCS 被编译，PETSc 是默认选择与 PETSc。如果 PETSc 不可用，则 FEniCS 将回退使用 Eigen 后端。FEniCS 中的线性代数后端可以设置使用以下命令：

Python code

```
1 parameters.linear_algebra_backend = backendname
```

backendname 是一个字符串。查看哪个线性代数后端可用，您可以调用 FEniCS 功能 list_linear_algebra_backends 同样，可以查看哪一个线性代数后端正在被以下使用命令：

Python code

```
1 print(parameters.linear_algebra_backend)
```

5.2.3 设置求解器参数

我们通常会在停止时控制容差标准和运行时的最大迭代次数迭代法这样的参数可以在全局控制和地方一级。我们将从如何设定全球化开始参数。对于更高级的程序，可能需要使用一个数字的不同线性求解器并设置不同的公差等参数。那么在 a 处控制参数变得很重要地方一级。我们将在 5.3.1 部分中回到此问题。

更改全局 FEniCS 参数数据库中的参数会影响所有线性求解器（在参数设置后创建）。全局 FEniCS 参数数据库简称为 parameters 和它表现为一个嵌套字典。写

Python code

```
1 info(parameters, verbose=True)
```

列出数据库中的所有参数及其默认值。参数集的嵌套通过缩进表示从 info 输出。根据该输出，相关参数集为命名为 'krylov_solver'，参数设置如下：

```
Python code
1 prm = parameters.krylov_solver # short form
2 prm.absolute_tolerance = 1E-10
3 prm.relative_tolerance = 1E-6
4 prm.maximum_iterations = 1000
```

针对 Krylov 求解器的停止标准通常涉及一些规范剩余值必须小于绝对公差参数或小于相对公差参数次数初始残留。

我们注意到，全局参数数据库的默认值可以是在 XML 文件中定义。从当前集合生成这样的文件的程序中的参数，运行

```
Python code
1 File('parameters.xml') << parameters
```

如果一个dolfin_parameters.xml 文件在目录中找到运行 FEniCS 程序，该文件被读取并用于初始化 parameters 对象。否则，该文件 .config/fenics/dolfin_parameters.xml 在用户的主目录是阅读，如果存在。另一个选择是加载 XML 文件 (与任何名称) 手动在程序中：

```
Python code
1 File('parameters.xml') >> parameters
```

XML 文件也可以是 gzip 的形式，扩展名为.xml.gz。

5.2.4 扩展求解器函数

我们可能会扩展以前的求解器函数

https://fenicsproject.org/pub/tutorial/python/vol1/ft12_poisson_solver.py

在部分 [5.1.1](#) 这样它也提供了 GMRES + ILU 预处理 Krylov 求解器：

这个新的 solver 函数在文件中找到

https://fenicsproject.org/pub/tutorial/python/vol1/ft10_poisson_extended.py, 取而代之
https://fenicsproject.org/pub/tutorial/python/vol1/ft12_poisson_solver.py。

它具有以前 solver 功能的所有功能，但是也可以用迭代法解决线性系统。

`subsection` 关于单元测试的评论

关于以单位来验证新的 solver 函数测试，事实证明单位测试的问题在哪里当我们使用时，近似误差消失会变得更加复杂迭代方法。问题是由于迭代而保持错误解决方案小于验证中使用的公差试验。首先，这意味着 Krylov 中使用的公差求解器必须小于 assert 测试中使用的公差，但是这并不能保证线性求解误差小。对于线性元素和小网格，容差为 10^{-11} 作品在 Krylov 求解器的情况下（使用容限 10^{-12} ）在那些解答者）。有兴趣的读者参考 `demo_solvers` 功能在

https://fenicsproject.org/pub/tutorial/python/vol1/ft10_poisson_extended.py

详情请见：该函数测试直接和迭代的数值解线性求解器，适用于不同网格，不同程度的有限元基函数中的多项式。

5.2.5 线性求解器方法和预处理器列表

哪些线性求解器和预处理器可用在 FEniCS 中取决于 FEniCS 的配置以及哪些线性代数后端当前处于活动状态。下表显示了可用的线性求解器的示例当 PETSc 后端处于活动状态时，通过 FEniCS：

名称	方法
'bicgstab'	Biconjugate gradient stabilized method
'cg'	Conjugate gradient method
'gmres'	Generalized minimal residual method
'minres'	Minimal residual method
'petsc'	PETSc built in LU solver
'richardson'	Richardson method
'superlu_dist'	Parallel SuperLU
'tfqmr'	Transpose-free quasi-minimal residual method
'umfpack'	UMFPACK

该组可用的预处理器还取决于配置和线性代数后端。下表显示了一个例子预处理器可能可用：

名称	方法
'icc'	Incomplete Cholesky factorization
'ilu'	Incomplete LU factorization
'petsc_amg'	PETSc algebraic multigrid
'sor'	Successive over-relaxation

可用的求解器和预处理器的最新列表为您的 FEniCS 安装可以生产

Python code

```
1 list_linear_solver_methods()
2 list_krylov_solver_preconditioners()
```

5.3 高级和低级解算器接口

FEniCS 接口允许不同的方式访问核心功能，从非常高级到低级访问。所以很远，我们大多使用高层次的电话 `solve(a == L, u, bc)` 来解决具有一定边界条件的变分问题 `a == L` `bc`。但是，有时您可能需要更细粒度的控制解决过程。特别是，将会创建对 `solve` 的调用在解决方案之后抛出的某些对象并且重用它们可能是实际的或有效的对象。

5.3.1 线性变分问题和求解器对象

在本节中，我们将看一个替代的解决方案 FEniCS 中的线性变分问题，可能是优选的很多情况与高级 `solve` 功能接口相比。此接口使用两个类 `LinearVariationalProblem` 和 `LinearVariationalSolver`。使用这个界面，相当于 `solve(a == L, u, bc)` 看起来如下：

Python code

```
1 u = Function(V)
2 problem = LinearVariationalProblem(a, L, u, bc)
3 solver = LinearVariationalSolver(problem)
4 solver.solve()
```

许多 FEniCS 对象具有属性 `parameters`，类似于全局 `parameters` 数据库，但本地对象。这里，`solver.parameters` 播放角色。用 ILU 预处理器设置 CG 方法作为解决方案方法和指定求解器特定的参数可以完成喜欢这个：

Python code

```
1 solver.parameters.linear_solver = 'gmres'
```

```

2 | solver.parameters.preconditioner = 'ilu'
3 | prm = solver.parameters.krylov_solver # short form
4 | prm.absolute_tolerance = 1E-7
5 | prm.relative_tolerance = 1E-4
6 | prm.maximum_iterations = 1000

```

全局 parameters 数据库中的设置为传播到各个对象中的参数集可能被覆盖如上。注意全局参数值只能在时间之前设置才能影响本地参数值创建本地对象。因此，改变价值全局参数数据库中的公差不会影响已经创建的求解器的参数。

5.3.2 明确组装和解决

正如我们在“[F3.4](#)”节中已经看到的，线性变分问题可以在 FEniCS 中明确地组合成矩阵使用 assemble 函数的向量。这允许更多解决方案的细粒度控制与使用相比高级 solve 函数或使用类 LinearVariationalProblem 和 LinearVariationalSolver。我们现在将更加关注如何使用 assemble 功能，以及如何将它与低级别组合要求解决组装的线性系统。

给定变量问题 $a(uv) = L(v)$ ，离散解 u 通过将 $u = \sum_{j=1}^N U_j \phi_j$ 插入 $a(uv)$ 和对 N 测试函数要求 $a(uv) = L(v) \hat{\phi}_1 \dots \hat{\phi}_N$ 。这意味着

$$\sum_{j=1}^N a(\phi_j, \hat{\phi}_i) U_j = L(\hat{\phi}_i), \quad i = 1, \dots, N,$$

which is nothing but a linear system,

$$AU = b,$$

where the entries of A and b are given by

$$\begin{aligned} A_{ij} &= a(\phi_j, \hat{\phi}_i), \\ b_i &= L(\hat{\phi}_i). \end{aligned}$$

迄今为止的例子已经指定了左侧和右侧变分制定，然后要求 FEniCS 组装线性系统解决。一个替代方法是明确地调用用于组合系数矩阵 A 和右手的功能侧向量 b ，然后解决线性系统 $AU = b$ 矢量 U 。我们现在写的不是 `solve(a == L, U, b)`

Python code

```

1 | A = assemble(a)
2 | b = assemble(L)
3 | bc.apply(A, b)
4 | u = Function(V)
5 | U = u.vector()
6 | solve(A, U, b)

```

变量 a 和 L 与以前相同；也就是 a 指涉及一个 TrialFunction 对象 u 的双线性形式和一个 TestFunction 对象 v 和 L 涉及相同的 TestFunction 目的 v 。从 a 和 L ，assemble 函数可以计算 A 和 b 。

在程序中明确创建线性系统可以有一些更先进的问题设置优点。例如， A 可以在整个时间依赖的模拟中是恒定的，所以我们可以避免在每个时间级别重新计算 A ，并节省大量资金的模拟时间。

矩阵 A 和向量 b 首先被组合而没有并入基本 (Dirichlet) 边界条件。此后，调用 `bc.apply(A, b)` 执行必要的修改线性系统，使 u 保证等于规定边界值。当我们有多个 Dirichlet 条件存储在一个列表 `bcs`，我们必须将 `bcs` 中的每个条件应用于系统：

Python code

```

1 for bc in bcs:
2     bc.apply(A, b)
3
4 # Alternative syntax using list comprehension
5 [bc.apply(A, b) for bc in bcs]

```

或者，我们可以使用函数 `assemble_system`，它需要组装线性时考虑边界条件系统。该方法保留了线性系统的对称性对称双线性形式。即使出现的矩阵 A 对 `assemble` 的调用是对称的（对于双线性对称形式） a ，对 `bc.apply` 的调用将会破坏对称性。保存变异问题的对称性在使用特定时是很重要的为对称系统设计的线性求解器，如共轭梯度法。

线性系统组装完成后，我们需要计算出解 $U = A^{-1}b$ 并将解 U 存储在向量中 $U = u.vector()$ 。以与线性变分问题相同的方式可以在 FEniCS—高级别编程使用不同的接口 `solve` 函数，

类 `LinearVariationalSolver` 和低级 `assemble` 功能—线性系统也可编程在 FEniCS 中使用不同的接口。高级接口在 FEniCS 中求解线性系统也被命名为 `solve`：

Python code

```
1 solve(A, U, b)
```

默认情况下，`solve(A, U, b)` 使用稀疏 LU 分解进行计算解决方案。迭代求解器和预处理器的规范可以通过两个可选参数：

Python code

```
1 solve(A, U, b, 'cg', 'ilu')
```

找到适当名称的求解器和预处理器第 5.2.5。

这个高级界面对许多应用程序很有用，但是有时需要更细粒度的控制。然后可以创建一个或更多 `KrylovSolver` 对象，然后用于求解线性系统。每个不同的求解器对象都可以有自己的集合参数和选择迭代法和预处理器。这里是一个例子：

Python code

```

1 solver = KrylovSolver('cg', 'ilu')
2 prm = solver.parameters
3 prm.absolute_tolerance = 1E-7
4 prm.relative_tolerance = 1E-4
5 prm.maximum_iterations = 1000
6 u = Function(V)
7 U = u.vector()
8 solver.solve(A, U, b)

```

函数 `solver_linalg` 在程序文件中

https://fenicsproject.org/pub/tutorial/python/vol1/ft10_poisson_extended.py

实现这样的解算器。

在线性求解器中迭代的起始向量的选择是往往很重要。默认情况下， u 的值和向量 $U = u.vector()$ 的值将被初始化为零。如果我们想要的话使用间隔 $[-100, 100]$ 中的随机数初始化 U 可以做到如下：

Python code

```

1 n = u.vector().array().size
2 U = u.vector()
3 U[:] = numpy.random.uniform(-100, 100, n)
4 solver.parameters.nonzero_initial_guess = True
5 solver.solve(A, U, b)

```

请注意，我们必须同时关闭设置开始的默认行为向量（“初始猜测”）为零，并设置值的值向量 U 为非零值。这是有用的，如果我们碰巧知道一个很好的初步猜测的解决方案。

使用非零初始猜测对于以下情况可能尤为重要的时间依赖性的问题或解决线性系统的一部分非线性迭代，从此之前的解向量 U 将会通常是下一个时间步骤中解决方案的一个很好的初步猜测或迭代。在这种情况下，矢量中的值 U 会自然会用以前的解决方案向量进行初始化（如果我们只是）使用它来解决线性系统），所以唯一额外的步骤是必要的设置参数 `nonzero_initial_guess` 到 `True`。

5.3.3 检查矩阵和向量值

当调用 `A = assemble(a)` 和 `b = assemble(L)` 时，对象 `A` 将是 `Matrix` 类型，而 `b` 和 `u.vector()` 是类型 `Vector`。为了检查这些值，我们可以转换矩阵和向量通过调用 `array` 方法将数据转换为 `numpy` 数组，如图所示之前。例如，如果你想知道边界条件是多么重要并入线性系统，您可以打印出 `A` 和 `b` 之前和之后的 `bc.apply(A, b)` 调用：

Python code

```

1 A = assemble(a)
2 b = assemble(L)
3 if mesh.num_cells() < 16: # print for small meshes only
4     print(A.array())
5     print(b.array())
6 bc.apply(A, b)
7 if mesh.num_cells() < 16:
8     print(A.array())
9     print(b.array())

```

通过 `numpy` 数组访问 `A` 中的元素，我们可以轻松实现对该矩阵执行计算，例如计算特征值（使用 `numpy.linalg` 中的 `eig` 函数）。我们可以选择转储 `A.array()` 和 `b.array()` 以 MATLAB 格式进行文件调用 MATLAB 或 Octave 分析线性系统。将数组转储为 MATLAB 格式完成

Python code

```

1 import scipy.io
2 scipy.io.savemat('Ab.mat', {'A': A.array(), 'b': b.array()})

```

然后在 MATLAB 或 Octave 中写 `load Ab.mat` 数组变量 `A` 和 `b` 可用于计算。

在 Python 或 MATLAB/Octave 中进行矩阵处理是可行的小的 PDE 问题，因为 `numpy` 数组或 MATLAB 文件中的矩阵格式是密集矩阵。FEniCS 也有一个接口固定软件包 SLEPc，它是首选的计算工具大型、稀疏矩阵的特征值遇到的类型 PDE 问题（见 `demo/recorder/eigenvalue/python/` 中的 FEniCS/DOLFIN 演示的源代码树）。

5.4 自由度和功能评估

5.4.1 检查自由度

我们以前看过如何从 `a` 获得自由度阵列有限元函数 `u`:

Python code

```
1 nodal_values = u.vector().array()
```

对于来自标准连续分段线性的有限元函数函数空间 (P_1 Lagrange 元素)，这些值将会与我们通过以下语句得到的值相同：

Python code

```
1 vertex_values = u.compute_vertex_values(mesh)
```

两个nodal_values 和vertex_values 将 numpy 数组和它们将具有相同的长度并包含相同的值(对于 P_1 元素)，但可能有不同的排序。该排列vertex_values 将具有与顶点相同的顺序网格，而nodal_values 将以一种方式(几乎)最小化系统矩阵的带宽，从而改善系统矩阵的带宽数值求解器的效率。

一个根本的问题是：什么是顶点的坐标值为nodal_values[i]？回答这个问题，我们需要了解如何得到我们的手协调，特别是编号的自由度以及网格中顶点的编号。

函数 mesh.coordinates 返回的坐标顶点为 numpy 数组，其形状为 (Md)， M 为数字网格中的顶点数为 d 为空间维数：

Python code

```
1 >> from fenics import *
2 >> mesh = UnitSquareMesh(2, 2)
3 >> coordinates = mesh.coordinates()
4 >> coordinates
5 array([[ 0. ,  0. ],
6        [ 0.5,  0. ],
7        [ 1. ,  0. ],
8        [ 0. ,  0.5],
9        [ 0.5,  0.5],
10       [ 1. ,  0.5],
11       [ 0. ,  1. ],
12       [ 0.5,  1. ],
13       [ 1. ,  1. ]])
```

从这个输出我们看到，对于这个特定的网格，顶点首先按 $y = 0$ 进行编号随着 x 坐标的增加，那么沿 $y = 0.5$ ，依此类推。

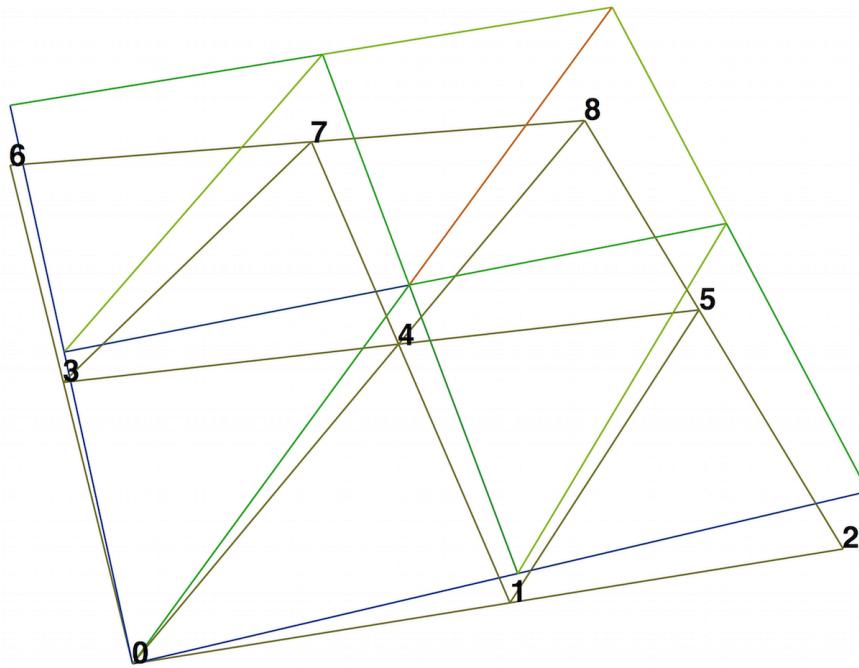
接下来我们在这个网格上计算一个函数 u 。我们来拿 $u = x + y$ ：

Python code

```
1 >> V = FunctionSpace(mesh, 'P', 1)
2 >> u = interpolate(Expression('x[0] + x[1]', degree=1), V)
3 >> plot(u, interactive=True)
4 >> nodal_values = u.vector().array()
5 >> nodal_values
6 array([ 1. ,  0.5,  1.5,  0. ,  1. ,  2. ,  0.5,  1.5,  1. ])
```

我们观察到nodal_values[0] 是 not $x+y$ 的值顶点数为 0，因为该顶点具有坐标 $x=y=0$ 。该编号的节点值(自由度) U_1, \dots, U_N 显然与顶点的编号不同。

顶点编号可以通过使用 FEniCS plot 命令。为此，绘制函数 u ，按 w 打开线框而不是完全有色的表面，m 显示网格，然后 v 显示顶点的编号。



我们通过调用 `u.compute_vertex_values`:

Python code

```

1 >> vertex_values = u.compute_vertex_values()
2 >> for i, x in enumerate(coordinates):
3 ...     print('vertex %d: vertex_values[%d] = %g\ntu(%s) = %g' %
4 ...           (i, i, vertex_values[i], x, u(x)))
5 vertex 0: vertex_values[0] = 0             tu([ 0.  0.]) = 8.46545e-16
6 vertex 1: vertex_values[1] = 0.5          tu([ 0.5  0.]) = 0.5
7 vertex 2: vertex_values[2] = 1             tu([ 1.  0.]) = 1
8 vertex 3: vertex_values[3] = 0.5          tu([ 0.  0.5]) = 0.5
9 vertex 4: vertex_values[4] = 1             tu([ 0.5  0.5]) = 1
10 vertex 5: vertex_values[5] = 1.5         tu([ 1.  0.5]) = 1.5
11 vertex 6: vertex_values[6] = 1             tu([ 0.  1.]) = 1
12 vertex 7: vertex_values[7] = 1.5         tu([ 0.5  1.]) = 1.5
13 vertex 8: vertex_values[8] = 2             tu([ 1.  1.]) = 2

```

我们可以要求 FEniCS 给我们从顶点到度数的映射某个功能空间的自由 V :

Python code

```
1 v2d = vertex_to_dof_map(V)
```

现在, `nodal_values[v2d[i]]` 会给我们带来的价值自由对应于顶点 $i(v2d[i])$ 。特别是 `nodal_values[v2d]` 是具有相同(顶点编号)顺序的所有元素的数组作为 `coordinates`。逆映射, 从自由度数到顶点数由 `dof_to_vertex_map(V)` 给出。这意味着我们可以打电话 `coordinates[dof_to_vertex_map(V)]` 得到所有的数组坐标与自由度相同。注意这些映射仅适用于 P_1 元素的 FEniCS。

对于 Lagrange 度数大于 1 的元素, 有度数自由度(节点)不对应于顶点。对于这些元素, 我们可以通过调用获得顶点值 `u.compute_vertex_values(mesh)`, 我们可以获得自由度通过调用 `u.vector().array()`。获取坐标相关联我们需要对所有的自由度进行迭代网格并要求 FEniCS 返回与之相关的坐标和自由度与每个元素(单元格)。此信息存储在 `FunctionSpace` 的 `FiniteElement` 和 `DofMap` 对象。该以下代码说明了如何迭代网格的所有元素并打印与之相关的坐标和自由度元件。

Python code

```

1 element = V.element()
2 dofmap = V.dofmap()
3 for cell in cells(mesh):
4     print(element.tabulate_dof_coordinates(cell))
5     print(dofmap.cell_dofs(cell.index()))

```

5.4.2 设定自由度

我们已经看到了如何在 numpy 数组中提取节点值。如果需要，我们也可以调整节点值。说我们想要规范化解决方案，使 $\max_j |U_j| = 1$ 。然后我们必须划分所有 U_j 值由 $\max_j |U_j|$ 。以下功能执行任务：

Python code

```

1 def normalize_solution(u):
2     """Normalize u: return u divided by max(u)"""
3     u_array = u.vector().array()
4     u_max = np.max(np.abs(u_array))
5     u_array /= u_max
6     u.vector()[:] = u_array
7     #u.vector().set_local(u_array) # alternative
8     return u

```

当使用 Lagrange 元素时，这个（大约）确保了函数 u 的最大值为 1。

$/=$ 运算符意味着在左边的对象的就地修改：全部数组元素 `nodal_values` 按值 `u_max` 划分。或者，我们可以做 `nodal_values = nodal_values / u_max`，其中意味着在右边创建一个新的数组并分配这个数组命名为 `nodal_values`。

注意

(操纵自由度时要小心) 一个像 `u.vector().array()` 的调用返回一个数据的副本 `u.vector()`。因此，必须永远不要执行任务 `u.vector.array()[:] = ...`，而是提取 numpy 数组（即复制），操作它，并用 `u.vector()[:] =` 插入，或使用 `u.set_local(...)`。

5.4.3 功能评估

FEniCS Function 对象在内部是唯一定义的有限元网格的每个单元格。对于连续 (Lagrange) 函数空间，函数值也是唯一定义的细胞界限可以简单地对 Function 对象 u 进行评估调用

Python code

```
1 u(x)
```

x 是 Point 或正确空间的 Python 元组尺寸。当 Function 被评估时，FEniCS 必须首先查找包含给定点（如果有）的网格的单元格，以及然后评估给定的基函数的线性组合指向有问题的单元格内。FEniCS 使用高效数据结构（边界框树）快速找到点，但是建造树是一个相对昂贵的操作，所以成本在单点评估 Function 是很昂贵的。重复评估将重用计算的数据结构，从而相对便宜。

注意

(廉价 vs 昂贵的功能评估) 一个 Function 对象 u 可以以各种方式进行评估：

1. $u(x)$ 为任意点 x

2. `u.vector().array()[i]` 自由度数 i
3. `u.compute_vertex_values()[i]` 在顶点编号 i

第一种方法虽然非常灵活，但通常是昂贵的而另外两个是非常有效的（但限于某些点）。

为了演示使用 Function 对象的点评估，我们打印计算有限元解的值 u Poisson 问题在域的中心点，并与之进行比较确切的解决方案：

Python code

```
1 center = (0.5, 0.5)
2 error = u_D(center) - u(center)
3 print('Error at %s: %g' % (center, error))
```

一个 $2 \times (3 \times 3)$ 网格，输出以前的片段变成了

Python code

```
1 Error at (0.5, 0.5): -0.0833333
```

差异是由于中心点不是节点在这个特定的网格中，而是细胞内部的一个点 u 在 u_D 的细胞上线性变化！是一个二次方功能。当中心点是一个节点时，如 $2 \times (2 \times 2)$ 或 $2 \times (4 \times 4)$ 网格，错误是顺序的 10^{-15} 。

5.5 后处理计算

作为本章的最后一个主题，我们将看看如何后处理计算；那就是如何计算各种派生的来自 PDE 的计算溶液的量。解决方案 u 本身可能是可视化的一般特征的兴趣解决方案，但有时一个人对计算解决方案感兴趣用于计算从解决方案中导出的特定数量的 PDE，例如，通量，点值或某些平均值解。

5.5.1 测试问题

作为测试问题，我们再次考虑可变系数 Poisson 单个 Dirichlet 边界条件的问题：

$$-\nabla \cdot (\kappa \nabla u) = f \quad \text{in } \Omega, \tag{5.1}$$

$$u = u_D \quad \text{on } \partial\Omega. \tag{5.2}$$

让我们继续使用我们最喜欢的解决方案 $u(x, y) = 1 + x^2 + 2y^2$ 和然后开出 $\kappa(x, y) = x + y$ 。它遵循 $u_D(x, y) = 1 + x^2 + 2y^2$ 和 $f(x, y) = -8x - 10y$ 。

像以前一样，这个模型问题的变分公式可以在 FEniCS 中指定为

Python code

```
1 a = kappa*dot(grad(u), grad(v))*dx
2 L = f*v*dx
```

系数为 κ ，右边为 f

Python code

```
1 kappa = Expression('x[0] + x[1]', degree=1)
2 f = Expression('-8*x[0] - 10*x[1]', degree=1)
```

5.5.2 通量计算

计算通量 $Q = -\kappa \nabla u$ 通常是有意义的。由于 $u = \sum_{j=1}^N U_j \phi_j$, 因此

$$Q = -\kappa \sum_{j=1}^N U_j \nabla \phi_j.$$

我们注意到分段连续有限元标量的梯度因为基函数是一个不连续的矢量场 $\{\phi_j\}$ 在边界处有不连续的派生词细胞。例如，使用 1 级的 Lagrange 元素， u 是线性的在每个细胞上，梯度变成分段恒定矢量场。相反，确切的梯度是连续。为了可视化和数据分析的目的，我们经常希望计算的梯度是一个连续的矢量场。通常情况下，我们想要 ∇u 的每个组件以相同的方式表示作为 u 本身。为此，我们可以投影组件 ∇u 与我们用于 u 的功能空间相同。这意味着我们解决 $w = \nabla u$ 大约通过有限元法，对于 w 的组件使用相同的元素，就像我们所使用的那样 u 。这个过程称为 projection。

Projection 是有限元分析中的常用操作，我们已经看过，FEniCS 具有轻松执行 projection 的功能：project(expression, W)，它返回一些 projection 表达进空格 W。

在我们的情况下，通量 $Q = -\kappa \nabla u$ 是向量值的，我们需要选择 W 作为向量值函数与 u 所在的空格 V 相同程度的空间：

Python code

```

1 V = u.function_space()
2 mesh = V.mesh()
3 degree = V.ufl_element().degree()
4 W = VectorFunctionSpace(mesh, 'P', degree)
5
6 grad_u = project(grad(u), W)
7 flux_u = project(-k*grad(u), W)
```

投影的应用很多，包括不连续的转动梯度场变为连续的，比较高阶和低阶函数近似，并转换高阶有限元解决了一个分段线性场，这是许多需要的可视化包。

绘制通量矢量场自然就像绘图一样简单还要别的吗：

Python code

```

1 plot(flux_u, title='flux field')
2
3 flux_x, flux_y = flux_u.split(deepcopy=True) # extract components
4 plot(flux_x, title='x-component of flux (-kappa*grad(u))')
5 plot(flux_y, title='y-component of flux (-kappa*grad(u))')
```

deepcopy=True 参数表示 deep copy，它是计算机科学的一般术语意味着数据的副本是回。（相反的，deepcopy=False，意思是 shallow copy，其中返回的对象只是指向原始数据的指针。）

对于通量场的节点值的数据分析，我们可以抓取底层的 numpy 数组（需要一个 deepcopy=True 在 flux 的分割中）：

Python code

```

1 flux_x_nodal_values = flux_x.vector().dofs()
2 flux_y_nodal_values = flux_y.vector().dofs()
```

自由度flux_u 矢量场也可以达到了

Python code

```
1 flux_u_nodal_values = flux_u.vector().array()
```

但是，这是一个包含度数的 numpy 数组自由的 x 和 y 组件的通量和 FEniCS 可以将组件的顺序混合起来提高计算效率。

功能demo_flux 在程序中

https://fenicsproject.org/pub/tutorial/python/vol1/ft10_poisson_extended.py

演示了上述计算。

注意

(手动投影。) 虽然您将始终使用 project 来投影有限元素功能，看看如何制定它可以是有启发性的在数学上投影并在 FEniCS 中手动执行其步骤。

假设我们有一个表达式 $g = g(u)$ ，我们想要项目进入一些空间 W 。 (L^2) 的数学公式投影 $w = P_W g$ 成 W 是变分问题

$$\int_{\Omega} wv \, dx = \int_{\Omega} gv \, dx \quad (5.3)$$

对于所有测试函数 $v \in W$ 。换句话说，我们有一个标准变量问题 $a(w, v) = L(v)$ 现在

$$a(w, v) = \int_{\Omega} wv \, dx, \quad (5.4)$$

$$L(v) = \int_{\Omega} gv \, dx. \quad (5.5)$$

注意，当 W 中的函数是向量值的时候，就像这样当我们投影渐变 $g(u) = \nabla u$ 时，我们必须替换以上产品由 $w \cdot v$ 和 $g \cdot v$ 。

变异问题在 FEniCS 中很容易定义。

Python code

```

1 w = TrialFunction(W)
2 v = TestFunction(W)
3
4 a = w*v*dx # or dot(w, v)*dx when w is vector-valued
5 L = g*v*dx # or dot(g, v)*dx when g is vector-valued
6 w = Function(W)
7 solve(a == L, w)

```

solve 的边界条件参数被丢弃这个问题没有必要的边界条件。

5.5.3 计算功能

在计算了 PDE 的解决方案 u 之后，我们偶尔想要计算 u 的功能，例如，

$$\frac{1}{2} \|\nabla u\|^2 = \frac{1}{2} \int_{\Omega} \nabla u \cdot \nabla u \, dx, \quad (5.6)$$

这通常反映了一些能量。另一个经常出现的功能是错误

$$\|u_e - u\| = \left(\int_{\Omega} (u_e - u)^2 \, dx \right)^{1/2}, \quad (5.7)$$

其中 u_e 是确切的解决方案。这个错误是特别的研究有限元收敛性时的兴趣方法。其他时候，我们可能会对计算感兴趣通过一部分流出 Γ 的边界 $\partial\Omega$ ，

$$F = - \int_{\Gamma} \kappa \nabla u \cdot n \, ds, \quad (5.8)$$

其中 n 是 Γ 上的向外指向单元。

所有这些功能都很容易用 FEniCS 进行计算，我们将看到在下面的例子中。

能量功能。 能量函数的被积函数 (5.6) 在 UFL 中描述语言与我们描述的形式相同：

Python code

```
1 energy = 0.5 * dot(grad(u), grad(u)) * dx
2 E = assemble(energy)
```

通过调用 `assemble` 来评估函数 `energy` 我们以前用来组装矩阵的函数向量。FEniCS 将会认识到该表单具有“0 级”（因为它）不包含试用和测试功能），并返回结果标量值。

错误功能。 功能 (5.7) 可以计算如下：

Python code

```
1 error = (u_e - u)**2 * dx
2 E = sqrt(abs(assemble(error)))
```

确切的解决方案 u_e 在这里由 Function 或 Expression 对象 `u_e`，而 `u` 是有限元近似（因此也是一个 Function）。有时，非常小错误值，`assemble(error)` 的结果可以是（非常小的）负数，所以我们使用 `E` 的表达式中使用了 `abs` 以确保 `sqrt` 函数的正值。

正如在 5.5.4 部分中所解释和演示的， $(u_e - u)^{**2} * dx$ 可能导致太乐观的收敛速度，除非有一个小心点差异 $u_e - u$ 被评估。一般建议为了可靠的错误计算是使用 `errornorm` 函数：

Python code

```
1 E = errornorm(u_e, u)
```

助焊剂功能。 计算通量积分，如 $F = - \int_{\Gamma} \kappa \nabla u \cdot n \, ds$ ，我们需要定义 n 向量，在 FEniCS 中称为 facet normal。如果是表面域 Γ 中的通量积分是完整的边界，我们可以执行通量计算

Python code

```
1 n = FacetNormal(mesh)
2 flux = -k * dot(grad(u), n) * ds
3 total_flux = assemble(flux)
```

虽然 `grad(u)` 和 `nabla_grad(u)` 在上述可互换当 `u` 是一个标量函数时，我们选择写入 `grad(u)`，因为如果我们概括一下，这是正确的表达式向量 PDE 的基本方程。用 `nabla_grad(u)` 我们在这种情况下必须写 `dot(n, nabla_grad(u))`

可以将整合限制到边界的一部分通过使用网格函数来标记相关部分，如下所述第 4.4。假设该部分对应到子域号 `i`，变体的相关语法通量的公式为 $-k * dot(grad(u), n) * ds(i)$ 。

注意

(关于整合准确性的说明) 如前所述，FEniCS Expressions 必须使用定义一个特殊的程度该学位告诉 FEniCS 到哪个地方有限元空间的表达式应该是内插的执行本地计算（集成）。作为说明，考虑积分的计算 $\int_0^1 \cos x \, dx = \sin 1$ 。这可以在 FEniCS 中计算

Python code

```

1 mesh = UnitIntervalMesh(1)
2 I = assemble(Expression('cos(x[0])', degree=degree)*dx(domain=mesh))

```

请注意，我们必须在这里指定参数 `domain = mesh` 测量 `dx`。在定义表单时通常不是必需的在 FEniCS 中，但是由于 `cos(x[0])` 未关联，所以必需与任何域（如我们集成 Function 来自某些 Mesh 上定义的 FunctionSpace）。

0 和 5 之间的变化值， $|\sin(1) - I|$ 的值 0.036, 0.071, 0.00030, 0.00013, 4.5E-07, and 2.5E-07.

FEniCS 还允许表达式直接表达为一部分表单。这需要创建一个 `SpatialCoordinate`。在这种情况下，精度由准确度决定集成，可以由 `degree` 参数控制整合度量 `dx`。`degree` 参数指定对于该程度的多项式，积分应该是精确的。

以下代码片段显示如何使用这种方法计算积分 $\int_0^1 \cos x dx$:

Python code

```

1 mesh = UnitIntervalMesh(1)
2 x = SpatialCoordinate(mesh)
3 I = assemble(cos(x[0])*dx(degree=degree))

```

0 和 5 之间的变化值， $|\sin(1) - I|$ 的值 0.036, 0.036, 0.00020, 0.00020, 4.3E-07, 4.3E-07.

请注意，正交程度仅适用于奇数，以便 0 的程度将使用相同的正交规则为 1，学位 2 将给出相同的正交规则为度 3 等等。

5.5.4 计算收敛速度

任何数值方法的核心问题是它的 convergence rate: 当分辨率增加时，误差逼近为零增加？对于有限元方法，这通常对应于从理论上或经验证明，错误 $e = u_e - u$ 被网格大小 h 限制到某些权力 r ；也就是 $\|e\| \leq Ch^r$ 为一些常数 C 。数字 r 被称为 convergence rate 的方法。注意不同的规范，像 L^2 -规范 $\|e\|$ 或 H_0^1 -标准 $\|\nabla e\|$ 通常有收敛速度不同。

为了说明如何在 FEniCS 中计算误差和收敛速度，我们已经包含了函数 `compute_convergence_rates` 在里面教程课程

https://fenicsproject.org/pub/tutorial/python/vol1/ft10_poisson_extended.py。

这是一种在验证有限元代码时非常方便的工具因此将在此详细解释。

计算错误规范。 正如我们已经看到的， L^2 -错误的规范 $u_e - u$ 能够在 FEniCS 中实施

Python code

```

1 error = (u_e - u)**2*dx
2 E = sqrt(abs(assemble(error)))

```

如上所述，我们在 `E` 的表达式中使用了 `abs` 来确保 `sqrt` 函数的正值。

了解 FEniCS 计算错误是非常重要的上面的代码，因为我们可能会在使用时遇到微妙的问题计算收敛速度的价值。第一个微妙的问题是如果 `u_e` 不是一个有限元素的函数（一个创建的对象）使用 `Function(V)`，如果 `u_e` 被定义为 `Expression`，FEniCS 必须插入 `u_e` 变成一些局部有限的网格的每个元素上的元素空间。用于的程度插值是由强制关键字参数决定的 `Expression` 类，例如：

Python code

```

1 u_e = Expression('sin(x[0])', degree=1)

```

这意味着计算出的误差将不等于实际值错误 $\|u_e - u\|$ 而是有限的差异元素解决方案 u 和分段线性插值 u_e 。这可能会产生太乐观（太小）的价值错误。通过内插精确度可以获得更好的价值解决方案进入高阶函数空间，这可以通过简单地增加学位：

```
Python code
1 u_e = Expression('sin(x[0])', degree=3)
```

第二个微妙的问题是当 FEniCS 评估表达式时 $(u_e - u)^{** 2}$, 这将被扩展为 $u_e^{**2} + u^{**2} - 2*u_e*u$ 如果错误很小（解决方案本身就是中等大小），这个计算将对应于减法两个正数 ($u_e^{**2} + u^{**2} \sim 1$ and $2*u_e*u \sim 1$) 产生少量。这样的计算很容易四舍五入的错误，这可能再次导致不可靠的价值错误。为了使这种情况更糟，FEniCS 可能会扩大这一点计算大量的术语，特别是较高的订单元素，使计算非常不稳定。

为了帮助解决这些问题，FEniCS 提供了内置的功能 `errornorm`，它以更智能的方式计算错误规范办法。首先，两个 u_e 和 u 被插入到高阶功能空间。然后， u_e 的自由度和 u 是减去在高阶函数中产生新的函数空间。最后，FEniCS 整合了平方的差异函数，然后取平方根来获取错误的值规范。使用 `errornorm` 功能很简单：

```
Python code
1 E = errornorm(u_e, u, normtype='L2')
```

这是一个简短的实现 `errornorm` 的说明：

```
Python code
1 def errornorm(u_e, u):
2     V = u.function_space()
3     mesh = V.mesh()
4     degree = V.ufl_element().degree()
5     W = FunctionSpace(mesh, 'P', degree + 3)
6     u_e_W = interpolate(u_e, W)
7     u_W = interpolate(u, W)
8     e_W = Function(W)
9     e_W.vector()[:] = u_e_W.vector().array() - u_W.vector().array()
10    error = e_W**2*dx
11    return sqrt(abs(assemble(error)))
```

有时候计算错误是有意义的渐变字段: $\|\nabla(u_e - u)\|$, 通常被称为 H_0^1 或 H^1 错误的 seminorm。这可以用上面的表达式来代替 `error` 通过 `error = dot(grad(e_W), grad(e_W))*dx`, 或通过调用在 FEniCS 中的 `errornorm`:

```
Python code
1 E = errornorm(u_e, u, norm_type='H10')
```

有关可用的更多信息，请在 Python 中键入 `help(errornorm)` 规范类型。

函数 `compute_errors` 在

https://fenicsproject.org/pub/tutorial/python/vol1/ft10_poisson_extended.py

说明了 FEniCS 中各种误差规范的计算。

计算收敛速度。 我们来研究如何在 FEniCS 中计算收敛速度。solver 函数

https://fenicsproject.org/pub/tutorial/python/vol1/ft10_poisson_extended.py

使我们能够轻松地为更精细和更细的网格计算解决方案使我们能够研究收敛速度。定义元素大小 $h = 1/n$, 其中 n 是 x 和 y 中的单元格分隔数方向 ($n = Nx = Ny$)。我们进行实验

$h_0 > h_1 > h_2 > \dots$ 并计算相应的错误 E_0, E_1, E_2 等等。假设 $E_i = Ch_i^r$ 为未知常数 C 和 r , 我们可以比较两个连续的实验, $E_{i-1} = Ch_{i-1}^r$ 和 $E_i = Ch_i^r$, 并解决 r :

$$r = \frac{\ln(E_i/E_{i-1})}{\ln(h_i/h_{i-1})}.$$

r 值应接近预期收敛速度 (通常为 L^2 的多项式度 +1) 错误) 为 i 增加。

以上过程可以很容易地转化为 Python 代码。这里我们通过元素度量列表 (P_1 , P_2 和 P_3), 对一系列细化的网格进行实验每个实验报告由 `compute_errors` 返回的六个错误类型。

测试问题。 为了说明收敛速度的计算, 我们选择一个精确的解决方案 u_e , 这个时间比一点更有趣对于 2 中的测试问题:

$$u_e(x, y) = \sin(\omega\pi x)\sin(\omega\pi y).$$

这个选择意味着 $f(x, y) = 2\omega^2\pi^2 u(x, y)$ 。将 ω 限制为整数, 因此边界值由 $u_D = 0$ 给出。

我们需要定义适当的边界条件, 确切的解决方案和代码中的 f 函数:

Python code

```

1 def boundary(x, on_boundary):
2     return on_boundary
3
4 bc = DirichletBC(V, Constant(0), boundary)
5
6 omega = 1.0
7 u_e = Expression('sin(omega*pi*x[0])*sin(omega*pi*x[1])',
8                  degree=6, omega=omega)
9
10 f = 2*pi**2*omega**2*u_e

```

实验。 可以实现收敛速度的计算在函数 `demo_convergence_rates` 中找到在演示程序中

https://fenicsproject.org/pub/tutorial/python/vol1/ft10_poisson_extended.py。

我们取得一些有趣的结果。使用无限度的自由度差异规范, 我们获得下表:

element	$n = 8$	$n = 16$	$n = 32$	$n = 64$
P_1	1.99	2.00	2.00	2.00
P_2	3.99	4.00	4.00	4.01
P_3	3.95	3.99	3.99	3.92

$n = 32$ 和 P_3 的条目如 3.99 表示我们通过比较两个网格, 分辨率来估计 3.99 的比率 $n = 32$ 和 $n = 16$, 使用 P_3 元素。注意节点上 P_2 的超会聚。最好的估计的价格出现在最右边的列, 因为这些费率是基于最好的决议, 因此最深入渐近的制度 (直到我们达到了四舍五入的错误和线性系统的不精确解决方案开始起作用)。

L^2 -使用 FEniCS 计算的范数误差 `errornorm` 功能显示 u 的预期 h^{d+1} 率:

element	$n = 8$	$n = 16$	$n = 32$	$n = 64$
P_1	1.97	1.99	2.00	2.00
P_2	3.00	3.00	3.00	3.00
P_3	4.04	4.02	4.01	4.00

但是，使用 $(u_e - u)^{**2}$ 对于误差计算，用相同的 u_e 插值的程度对于 u ，给出奇怪的结果：

element	$n = 8$	$n = 16$	$n = 32$	$n = 64$
P ₁	1.97	1.99	2.00	2.00
P ₂	3.00	3.00	3.00	3.01
P ₃	4.04	4.07	1.91	0.00

这是一个例子，重要的是插入 u_e 到一个高阶空间（3 阶多项式在这里就足够了）。这个通过使用 `errornorm` 函数自动处理。

检查收敛速度是验证 PDE 的极好方法码。

5.5.5 利用结构化网格数据

许多读者在可视化和数据方面有丰富的经验分析 1D, 2D 和 3D 标量和矢量场均匀，结构化网格，而 FEniCS 解决方案专门工作非结构化网格由于它可以很多次实践使用结构化数据，我们在本节中讨论如何提取用于计算的有限元解的结构化数据 FENICS。

必要的第一步是将我们的 Mesh 对象转换为对象表示具有同样形状的矩形（或 3D 框）矩形细胞。第二步是转变节点值的一维数组为二维数组将值保存在结构化的单元格的角落里。我们要通过 i 和 j 索引 i 来访问一个值计数 x 方向的单元格， j 计数 y 中的单元格方向。但这种转变原则上是直截了当的它经常导致模糊的索引错误，所以使用软件减轻工作的工具是有利的。

在本书附带的示例程序目录中，我们有包括 Python 模块

<https://fenicsproject.org/pub/tutorial/python/vol1/boxfield.py>

它提供了用于处理结构化网格数据的实用程序 FENICS。给定有限元函数 u ，以下函数在结构化网格上返回表示 u 的 BoxField 对象：

Python code

```
1 from boxfield import *
2 u_box = FEniCSBoxField(u, (nx, ny))
```

u_{box} 对象包含几个有用的数据结构：

- $u_{\text{box}}.grid$: 对象为结构化网格
- $u_{\text{box}}.grid.coor[X]$: $X=0$ 方向的网格坐标
- $u_{\text{box}}.grid.coor[Y]$: $Y=1$ 方向的网格坐标
- $u_{\text{box}}.grid.coor[Z]$: $Z=2$ 方向的网格坐标
- $u_{\text{box}}.grid.coorv[X]$: $u_{\text{box}}.grid.coor[X]$ 的矢量化版本
- $u_{\text{box}}.grid.coorv[Y]$: $u_{\text{box}}.grid.coor[Y]$ 的矢量化版本
- $u_{\text{box}}.grid.coorv[Z]$: $u_{\text{box}}.grid.coor[Z]$ 的矢量化版本
- $u_{\text{box}}.values$: numpy 数组，保存 u 值； $u_{\text{box}}.values[I, J]$ 在网格点保持 u ，坐标为 $(u_{\text{box}}.grid.coor[X][i], u_{\text{box}}.grid.coor[Y][j])$

迭代点数和值。 现在让我们使用 solver 函数

https://fenicsproject.org/pub/tutorial/python/vol1/ft10_poisson_extended.py

代码来计算 u , 将其映射到一个 BoxField 对象上的结构化网格表示, 并打印坐标和函数值在所有网格点:

Python code

```

1 u = solver(p, f, u_b, nx, ny, 1, linear_solver='direct')
2 u_box = structured_mesh(u, (nx, ny))
3 u_ = u_box.values
4
5 # Iterate over 2D mesh points (i, j)
6 for j in range(u_.shape[1]):
7     for i in range(u_.shape[0]):
8         print('u[%d, %d] = u(%g, %g) = %g, %'
9               (i, j,
10                u_box.grid.coor[X][i], u_box.grid.coor[Y][j],
11                u_[i, j]))
```

计算有限差分近似。 使用多维数组 $u_ = u_box.values$, 我们可以轻松地表示衍生物的有限差分近似:

Python code

```

1 x = u_box.grid.coor[X]
2 dx = x[1] - x[0]
3 u_xx = (u_[i - 1, j] - 2*u_[i, j] + u_[i + 1, j]) / dx**2
```

制作曲面图。 作为结构化数据访问有限元字段的能力在许多情况下很方便, 例如用于可视化和数据分析。使用 Matplotlib, 我们可以创建一个曲面图, 如图所示图 5.1(左上):

Python code

```

1 import matplotlib.pyplot as plt
2 from mpl_toolkits.mplot3d import Axes3D # necessary for 3D plotting
3 from matplotlib import cm
4 fig = plt.figure()
5 ax = fig.gca(projection='3d')
6 cv = u_box.grid.coorv # vectorized mesh coordinates
7 ax.plot_surface(cv[X], cv[Y], u_, cmap=cm.coolwarm,
8                 rstride=1, cstride=1)
9 plt.title('Surface plot of solution')
```

关键问题是知道曲面所需的坐标情节在 $u_box.grid.coorv$ 并且值在 $u_$ 中。

制作轮廓图。 一个轮廓图也可以由 Matplotlib 制作:

Python code

```

1 fig = plt.figure()
2 ax = fig.gca()
3 levels = [1.5, 2.0, 2.5, 3.5]
4 cs = ax.contour(cv[X], cv[Y], u_, levels=levels)
5 plt.clabel(cs) # add labels to contour lines
6 plt.axis('equal')
7 plt.title('Contour plot of solution')
```

结果如图 5.1(右上) 所示。

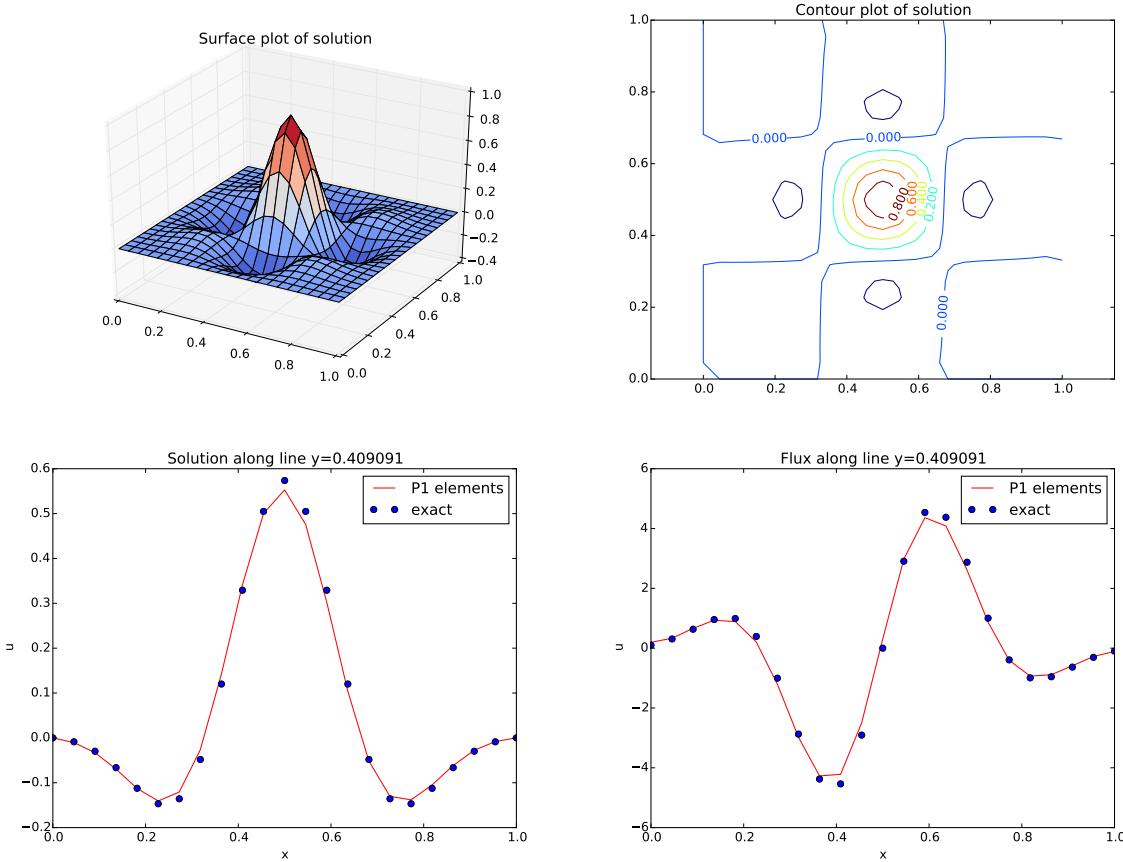


Figure 5.1: 在结构化网格上的各种解决方案。

通过域进行曲线图。 BoxField 对象的一个方便的功能是启动的能力指向域中的方向，然后提取字段和沿 mesh points 最近行的相应坐标。我们有已经看到如何在网格中沿线插入解决方案，但是使用 BoxField，您可以选择计算点(顶点)检查这些要点。数值方法经常表现出改善的行为在这样的点，所以这是有趣的。对于 3D 字段还可以在平面中提取数据。

说我们要绘制 u 行 $y = 0.4$ 。网格点， x 和 u 值沿着这一行， u_val ，可以被提取出来

Python code

```
1 start = (0, 0.4)
2 x, u_val, y_fixed, snapped = u_box.gridline(start, direction=X)
```

如果行被贴到最接近的位置，则变量 `snapped` 为 true 网格线，在这种情况下 `y_fixed` 持有攫取(改变) y 值。关键字参数 `snap` 默认情况下为 True 以避免插补和力捕捉。

沿线的数值和精确解的比较 $y \approx 0.41$ (从 $y = 0.4$ 中扣除) 由以下代码制成:

Python code

```
1 # Plot u along a line y = const and compare with exact solution
2 start = (0, 0.4)
3 x, u_val, y_fixed, snapped = u_box.gridline(start, direction=X)
4 u_e_val = [u_D((x_, y_fixed)) for x_ in x]
5 plt.figure()
6 plt.plot(x, u_val, 'r-')
7 plt.plot(x, u_e_val, 'bo')
8 plt.legend(['P1 elements', 'exact'], loc='best')
9 plt.title('Solution along line y=%g % y_fixed')
```

```
10 plt.xlabel('x'); plt.ylabel('u')
```

对于所得曲线图, 请参见图 5.1(左下)。

制作通量的曲线图。 我们还可以比较数值和精确通量 $-\kappa \partial u / \partial x$ 与上述相同:

Python code

```
1 # Plot the numerical and exact flux along the same line
2 flux_u = flux(u, kappa)
3 flux_u_x, flux_u_y = flux_u.split(deepcopy=True)
4 flux2_x = flux_u_x if flux_u_x.ufl_element().degree() == 1 \
5 else interpolate(flux_x,
6 FunctionSpace(u.function_space().mesh(), 'P', 1))
7 flux_u_x_box = FEniCSBoxField(flux_u_x, (nx, ny))
8 x, flux_u_val, y_fixed, snapped = \
9 flux_u_x_box.gridline(start, direction=X)
10 y = y_fixed
11 plt.figure()
12 plt.plot(x, flux_u_val, 'r-')
13 plt.plot(x, flux_u_x_exact(x, y_fixed), 'bo')
14 plt.legend(['P1 elements', 'exact'], loc='best')
15 plt.title('Flux along line y=%g' % y_fixed)
16 plt.xlabel('x'); plt.ylabel('u')
```

在代码片段开头调用的函数 `flux` 是在示例程序中定义

https://fenicsproject.org/pub/tutorial/python/vol1/ft10_poisson_extended.py

并将通量内插到功能空间中。

请注意, Matplotlib 是绘图包的选择。随着统一的界面在

<https://github.com/hplgit/scitools SciTools package>¹

中可以访问 Matplotlib, Gnuplot, MATLAB, OpenDX, VisIt 等绘图引擎通过相同的 API。

测试问题。 图中引用的图形 5.1 对应于规定解的一个测试问题 $u_e = H(x)H(y)$, 其中

$$H(x) = e^{-16(x-\frac{1}{2})^2} \sin(3\pi x).$$

相应的右侧 f 是通过插入确切的获得的解决方案进入 PDE 并像以前一样进行区分。虽然很容易进行用手分解 f 并对结果表达式进行硬编码在 Expression 对象中, 更可靠的习惯是使用 Python 符号计算引擎, SymPy, 执行数学和自动将公式转换为 Expression 对象的 C ++ 语法。简要介绍了第 3.2.3。

我们从 sympy 中定义确切的解决方案开始:

Python code

```
1 from sympy import exp, sin, pi # for use in math formulas
2 import sympy as sym
3
4 H = lambda x: exp(-16*(x-0.5)**2)*sin(3*pi*x)
5 x, y = sym.symbols('x[0], x[1]')
6 u = H(x)*H(y)
```

将 u 的表达式转换为 Expression 对象的 C 或 C ++ 语法需要两个步骤。首先我们要求表达式的 C 代码:

¹<https://github.com/hplgit/scitools>

Python code

```
1 u_code = sym.printing.ccode(u)
```

打印u_code 给出 (这里的输出手动分为两部分线):

Python code

```
1 -exp(-16*pow(x[0] - 0.5, 2) - 16*pow(x[1] - 0.5, 2))*  
2 sin(3*M_PI*x[0])*sin(3*M_PI*x[1])
```

必要的语法调整正在替换符号M_PI 对于 pi(或DOLFIN_PI) 的 C/C++ 中的 π :

Python code

```
1 u_code = u_code.replace('M_PI', 'pi')  
2 u_b = Expression(u_code, degree=1)
```

此后，我们可以随着计算进展 $f = -\nabla \cdot (\kappa \nabla u)$:

Python code

```
1 kappa = 1  
2 f = sym.diff(-kappa*sym.diff(u, x), x) + \  
3     sym.diff(-kappa*sym.diff(u, y), y)  
4 f = sym.simplify(f)  
5 f_code = sym.printing.ccode(f)  
6 f_code = f_code.replace('M_PI', 'pi')  
7 f = Expression(f_code, degree=1)
```

我们还需要一个 Python 函数来确定通量 $-\kappa \partial u / \partial x$:

Python code

```
1 flux_u_x_exact = sym.lambdify([x, y], -kappa*sym.diff(u, x),  
2                                 modules='numpy')
```

在调用之前，还要定义 $\kappa = \text{Constant}(1)$ 并设置 nx 和 ny solver 来计算这个问题的有限元解。

5.6 下一步

如果你来了这么远，你就学会了如何写简单一系列 PDE 的脚本式求解器，以及如何构造 Python 求解器使用功能和单元测试。解决更复杂的 PDE 并且编写一个更全功能的 PDE 求解器并不难第一步通常是写一个解析器进行剥离测试作为一个简单的 Python 脚本。随着脚本的成熟和变得越来越多复杂，现在是考虑设计的时候，特别是如何将代码模块化，并将其组织成可重复使用的部分用于构建灵活可扩展的求解器。

在 FEniCS 网站上，您将找到更多的文档，更多示例程序，以及高级解算器和应用程序的链接写在 FEniCS 之上。获得灵感并开发自己的求职者为您最喜欢的应用程序，发布您的代码并分享您的知识与 FEniCS 社区和世界！

PS: 请关注 FEniCS 教程第 2 卷！

