12 步到达纳维——斯托克斯

Lorena A Barba 编写　刘尚懿 翻译

2018 年 5 月 9 日

目录

第 1 章　步骤 1

您好！欢迎来到 **12 步到达纳维——斯托克斯**。这是一个用于交互式计算流体力学 (CFD) 初级课程的实践模块，是由罗瑞娜巴尔巴教授自 2009 年春季起在波士顿大学讲授。本课程假定读者具有基本的编程知识 (任何语言均可)、偏微分方程和流体力学中的一些基础。本实践模块受到了巴尔巴实验室里的一个博士后力拓田博士的启发，并经过巴尔巴和她的学生在几个学期的教学过程中进行完善。本课程完全使用 Python，那些不了解 Python 的学生可以通过模块学习。

jupyter notebook 将引导您从头开始用 Python 编写纳维——斯托克斯解算器。我们要立刻开始了。不要担心，如果你不明白发生的一切，我们会在我们前进的时候详细介绍它。

要获得最佳效果，在遵循本笔记本后，请为步骤 1 准备自己的代码（作为 Python 脚本或在干净的 jupyter notebook 中）。

要执行此笔记本，我们假定您已使用以下方式调用了笔记本服务器：jupyter notebook

1.1　1 维线性对流

一维线性对流方程是可以用来学习有关 CFD 知识的最简单、最基本的模型。令人惊讶的是，这个小方程式能教会我们非常多的东西！这个就是：

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0$$

在给定初始状态下 (理解为一个波), 方程表示该初始波的传播速度为 c，形状不变。设初始状态为 $u(x,0) = u_0(x)$。然后方程的精确解是 $u(x,t) = u_0(x-ct)$。

我们使用前向差分格式的时间导数和后向差分格式的空间导数，在空间和时间上对该方程进行离散化。考虑把空间坐标 x 离散成下标从 $i = 0 \,\text{to}\, N$ 的点，并以 Δt 为离散时间间隔步长。

从导数的定义（并简单地去除极限），我们知道：

$$\frac{\partial u}{\partial x} \approx \frac{u(x+\Delta x) - u(x)}{\Delta x}$$

则我们的离散方程为：

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + c\frac{u_i^n - u_{i-1}^n}{\Delta x} = 0$$

其中 n 和 $n+1$ 在时间上是两个连续的步骤，而 $i-1$ 和 i 是离散的 x 坐标的两个相邻点。如果给定初始状态，那么在这种离散化过程中唯一的未知量是 u_i^{n+1}。我们可以得到一个方程来求出我们的未知量，如下所示：

$$u_i^{n+1} = u_i^n - c\frac{\Delta t}{\Delta x}(u_i^n - u_{i-1}^n)$$

现在让我们试试在 Python 中实现它。

我们将开始引进一些库来帮助我们。

- numpy 是一个库，它提供了类似于 MATLAB 的一组有用的矩阵运算
- matplotlib 是一个 2 维绘图库我们将用来绘制我们的结果
- time 和 sys 提供基本的计时功能，我们将用来减慢动画的观看速度

```
# 记住：在Python中的注释用英镑符号表示
import numpy                          # 这里，我们加载了numpy
from matplotlib import pyplot         # 这里，我们加载了matplotlib
import time, sys                      # 并加载一些实用程序
# 这让matplotlib在notebook中绘图 (而不是一个分离的窗口)
%matplotlib inline
```

现在让我们定义几个变量；我们希望在空间域内定义一个均匀间隔的点网格，该空间域宽 2 个单位，即 $x_i \in (0,2)$。我们将定义一个变量 nx 表示我们想要的网格点的数量，并且 dx 将是任何一对相邻网格点之间的距离。

```
nx = 41    # 尝试将此数字从41改变为81并运行..发生了什么？
dx = 2 / (nx-1)
nt = 25    # nt是我们要计算的时间步长数
dt = .025  # dt是每个时间步长覆盖的时间量(delta t)
c = 1      # 假定波速为c=1
```

我们还需要建立我们的初始状态。在时间间隔 $0.5 \leq x \leq 1$ 内，初速 u_0 被赋予了 $u = 2$ 并且在 $(0,2)$ 的其他地方 $u = 1$(即，帽子函数)。

在这里，我们使用函数 ones() 定义一个长度为 nx 每一个元素值都等于 1 的 numpy 数组。

```
u = numpy.ones(nx)      # numpy函数ones()
u[int(.5 / dx):int(1 / dx + 1)] = 2  # 设置在0.5与1之间的u=2
print(u)
[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  2.  2.  2.  2.  2.  2.  2.  2.
  2.  2.  2.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.]
```

现在让我们使用 matplotlib 图看看这些初始状态。我们已经导入了 matplotlib 绘制库 pyplot 和绘图函数 plot，所以我们调用 pyplot.plot。

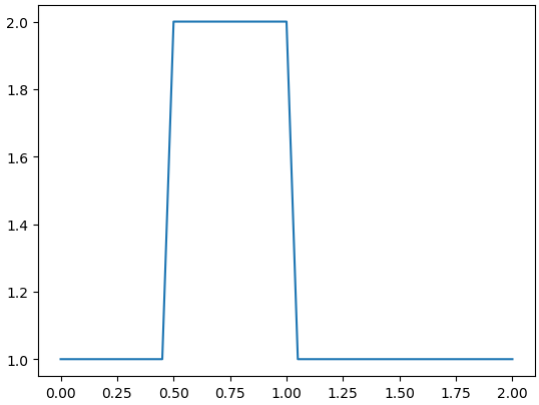
在这里，我们使用简单的 2 维绘图语法：plot(x,y)，其中 x 值是均匀分布的网格点：

```
pyplot.plot(numpy.linspace(0,2,nx),u)
pyplot.show()
```

为什么帽子函数没有完美的直边？想想吧。

现在是使用有限差分格式来实现对流方程离散化的时间。

对于阵列的每个元素 u 我们需要执行操作 $u_i^{n+1} = u^n - c\frac{\Delta t}{\Delta x}(u_i^n - u_{i-1}^n)$



我们将结果存储在一个新的 (临时) 数组 un 中，其将是下一个步进式的解 u。因为指定了许多步，所以我们将重复此操作，然后我们可以看到波的对流距离。

我们首先初始化占位符数组 un 来保持我们计算的第 n+1 步进的值，再次使用 numpy 函数 ones()。

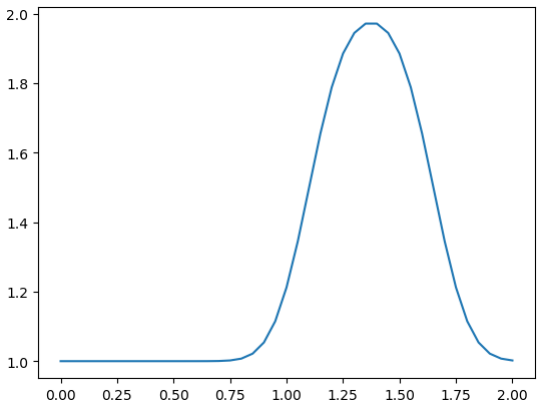
然后，我们可以认为我们有两个迭代操作：一个在空间中，一个在时间上（以后我们会学习其中的区别），所以我们将一个循环嵌套在另一个循环中来开始迭代操作。注意 range() 函数的使用。当我们写的时候：for i in range(1, nx) 时，我们将通过 u 数组迭代，但我们将跳过第一个元素（零元素）。为什么？

```
un = numpy.ones(nx)  # 初始化临时数组
for n in range(nt):  # 从0到nt循环n的值，因此它将运行nt次
    un = u.copy()    # 将u的现有值复制到un
    for i in range(1, nx): # 你可以尝试注释本行并且...
    #for i in range(nx): # ... 取消注释本行看看发生什么！
        u[i] = un[i] - c * dt / dx * (un[i] - un[i-1])
```

注释——稍后我们将知道上述代码是相当低效的，还有更好的方法来编写，使用 Python 风格。但让我们继续吧。

现在让我们尝试绘制出按时间推进后的数组 u。

```
pyplot.plot(numpy.linspace(0,2,nx),u)
pyplot.show()
```



好的！所以我们的帽子函数肯定移动到了右边，但它不再是帽子了。怎么了？

1.2 了解更多

为了更深入地解释有限差分法，包括诸如截断误差、收敛阶和其他细节的主题，在 YouTube 上观看 Barba 教授的**视频课程 2 和 3**。

```
from IPython.display import YouTubeVideo
YouTubeVideo('iz22_37mMkk')
YouTubeVideo('xq9YTcv-fQg')
```

对于具有有限差异的线性对流方程的离散化，请从头到尾仔细的走一遍 (也包括直到步骤 4 的这些步骤)，在 YouTube 上观看 Barba 教授的**视频课程 4**。

```
YouTubeVideo('y2WaK7_iMRI')
```

1.3 最后但不是最重要的

记住将步骤 1 重写为新 Python 脚本或者在你自己的 jupyter notebook 中，然后通过更改离散化参数进行实验。一旦你做到了这一点你就准备好学习步骤 2

第 2 章　步骤 2

本 jupyter notebook 继续演示 **12 步到达纳维——斯托克斯**，巴尔巴教授讲授的交互式 CFD 课程中的实际模块。在继续本模块之前，你应该已经完成了步骤 1，已经编写了自己的 Python 脚本或笔记本，已经尝试改变离散化的参数并观察发生了什么。

2.1　非线性对流

现在，我们将使用与步骤 1 相同的方法来实施非线性对流。一维对流方程是:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0$$

我们现在不再将常数因子 c 与第二项乘而是与解 u 乘。因此，方程的第二项现在是非线性的。我们将使用与步骤 1 中相同的离散化——时间上的前向差分和空间的后向差分。这里是离散方程。

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + u_i^n\frac{u_i^n - u_{i-1}^n}{\Delta x} = 0$$

求解唯一的未知项，u_i^{n+1}，得到：

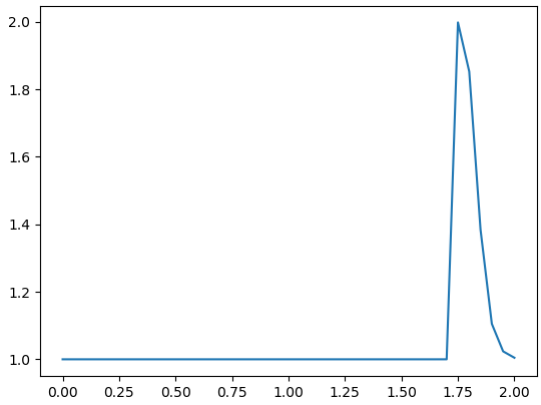
$$u_i^{n+1} = u_i^n - u_i^n\frac{\Delta t}{\Delta x}(u_i^n - u_{i-1}^n)$$

如前所述，在 Python 代码中开始加载必要的库。然后，我们声明一些确定空间和时间离散的变量（你应该通过更改这些参数来进行实验，看看会发生什么）。然后，我们通过将 $(0.5 \leq x \leq 1)$ 之间求解用的数组初始化为 $u = 2$，在 $(0, 2)$ 的其他地方初始化为 $u = 1$ 来创建初始状态 u_0（即一个帽子函数）。

```
import numpy                    # 我们导入numpy
from matplotlib import pyplot    # 和我们的2维绘图库，称为plt
nx = 41
dx = 2 / (nx - 1)
nt = 20    # nt是我们要计算的时间步长数
dt = .025  # dt是每个时间步长覆盖的时间量(Delta t)
u = numpy.ones(nx)       # 如前所述，我们用等于1的每个值初始化u。
u[int(.5 / dx) : int(1 / dx + 1)] = 2  # 然后将u=2设置在0.5和1之间

un = numpy.ones(nx) # 初始化我们的占位符数组un，以得到步进式的解
```

下面的代码片段为未完成的. 我们已从步骤 1 中复制了代码来执行时间步进更新。您可以编辑此代码以执行非线性对流吗?

```
    for n in range(nt):  # 迭代次数
    un = u.copy()        # 将u的现有值复制到un中
    for i in range(1, nx):  # 现在我们将遍历u数组
     # 这是从步骤1完全复制的代码。根据我们的新方程编辑它。
     # 然后取消注释并运行该单元以评估步骤2
        # #u[i] = un[i] - c * dt / dx * (un[i] - un[i-1])
        u[i] = un[i] - un[i]*dt / dx * (un[i] - un[i-1])
    pyplot.plot(numpy.linspace(0, 2, nx), u) # 打印结果
    pyplot.show()
```



在非线性对流方程下，对帽子函数的演变你观察到了什么？更改数值参数并再次运行时，会发生什么情况？

2.2 了解更多

为了仔细地对具有有限差分的对流方程进行离散化，(从 1 到 4 的所有步骤)，请在 YouTube 上观看 Barba 教授的**视频课程 4**。

```
    from IPython.display import YouTubeVideo
    YouTubeVideo('y2WaK7_iMRI')
```

第 3 章　CFL 条件

你在步骤中 1 和 2 中进行了不同参数的选择实验了吗？如果你做了，你可能会遇到意想不到的行为。你的解决方案曾经搞砸过吗？(在我的经验中，CFD 学生爱使事情搞砸。)

你可能正在考虑为什么更改离散化参数会影响您的解决方案。本笔记通过讨论 CFL 条件补充了我们的交互式 CFD 课程。并且通过观看巴尔巴教授的 YouTube 讲座来学习更多的内容。

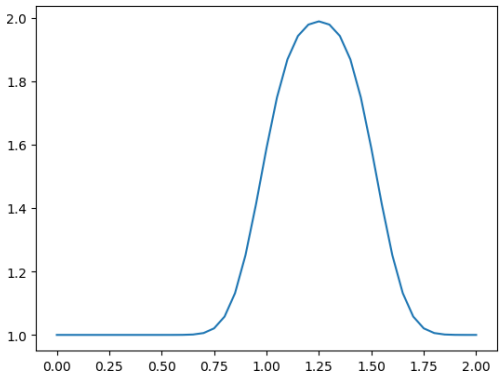
3.1　收敛与 CFL 条件

在前几个步骤中，我们使用了相同的初始和边界条件。根据我们最初提出的参数，网格有 41 个点，时间步长为 0.25 秒。现在，我们将通过增加网格的大小来进行实验。下面的代码与我们在步骤 1 中使用的代码相同，但这里它已被包装在一个函数中，这样我们就可以轻松地检查当我们调整一个变量：网格大小时发生的情况。

```
import numpy                    # numpy是类似于Matlab的矩阵操作库
from matplotlib import pyplot     # matplotlib是2维绘图库
def linearconv(nx):
    dx = 2 / (nx - 1)
    nt = 20     #nt是我们要计算的时间步长数
    dt = .025  #dt 是每个时间步长覆盖的时间量(delta t)
    c = 1
    # 定义一个numpy数组，该数组是nx元素，每个值等于1。
    u = numpy.ones(nx)
    # 设置u=2在0.5和1之间
    u[int(.5/dx):int(1 / dx + 1)] = 2
    # 初始化我们的占位符数组un，以保存我们为第n+1时间步长计算的值
    un = numpy.ones(nx)

    for n in range(nt):  # 遍历时间
        un = u.copy() # 将u的现有值复制到un中
        for i in range(1, nx):
            u[i] = un[i] - c * dt / dx * (un[i] - un[i-1])

    pyplot.plot(numpy.linspace(0, 2, nx), u);
    pyplot.show()
```

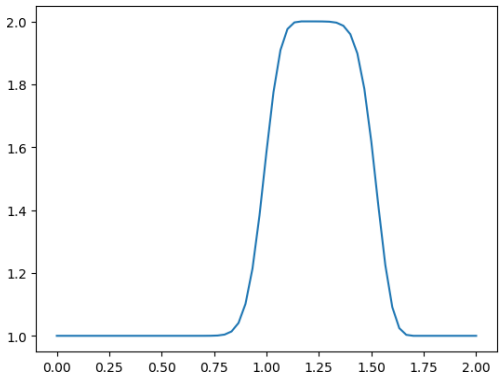
现在让我们用一个越来越细的网格检查我们的线性对流问题的结果。

```
linearconv(41)
```



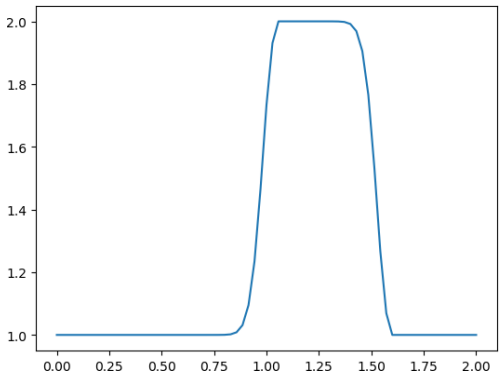
这与我们的第 1 步计算的结果相同，在此转载以供参考。

```
linearconv(61)
```



这里仍然存在数值扩散，但它不那么严重。

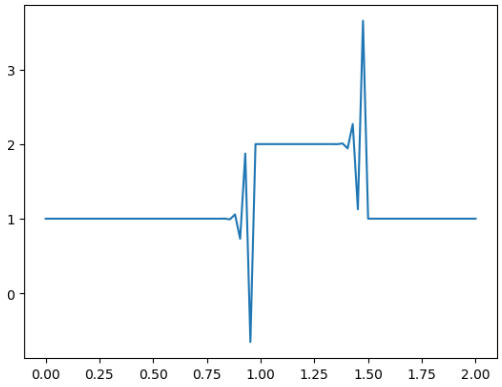
```
linearconv(71)
```



这里仍存在相同的模式——波比在前一运行中的波更小。

```
linearconv(85)
```

这并不像我们原来的帽子函数。



3.2 怎么了？

为了回答这个问题，我们必须考虑一下我们实际在代码中实现的内容。

在我们的每次时间循环迭代中，我们使用与我们的波有关的已知数据来估计随后时间步长中的波速。最初，网格点数量的增加返回了更准确的答案。在我们的第一个例子中，有较少的数值扩散且方形波看起来更像是方形波。

在时间循环的每次迭代中都包含一个时间步长 Δt，我们定义为 0.025

在此迭代过程中，我们评估在每一个我们创建的点 x 上的波速。在最后一个图中，有些事情显然是错误的。

所发生的情况是在 Δt 时间周期内，波传播的距离大于 dx。每个网格框的 dx 长度与 nx 的总数有关，所以如果 Δt 步长的计算与 dx 的尺寸有关，则可以保证运行的稳定性。

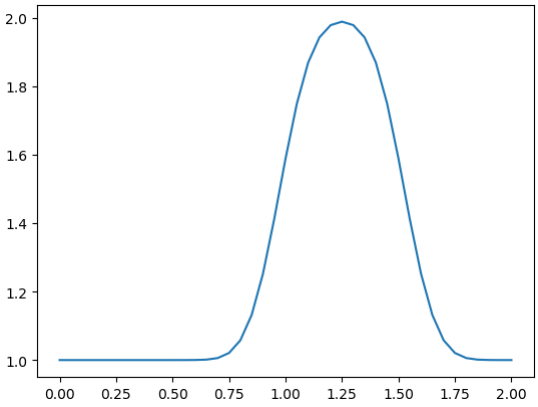
$$\sigma = \frac{u\Delta t}{\Delta x} \leq \sigma_{max}$$

其中 u 是波速；σ 被称为**库兰特数**，值 σ_{max} 保证运行的稳定性，该值的大小取决于所使用的离散化过程。

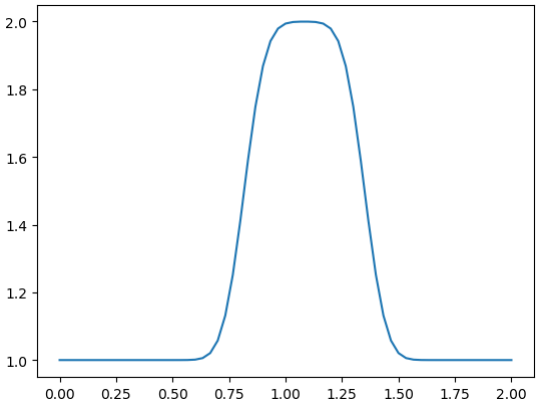
在我们的代码的新版本中，我们将使用 CFL 数来计算依赖于尺寸 dx 的时间步长 dt 的合适值。

```
import numpy
from matplotlib import pyplot
def linearconv(nx):
    dx = 2 / (nx - 1)
    nt = 20    # nt是我们想要计算的时间步长数
    c = 1
    sigma = .5
    dt = sigma * dx

    u = numpy.ones(nx)
    u[int(.5/dx):int(1 / dx + 1)] = 2
    un = numpy.ones(nx)

    for n in range(nt):  # 迭代时间
        un = u.copy() # 将u的现有值复制到un
```

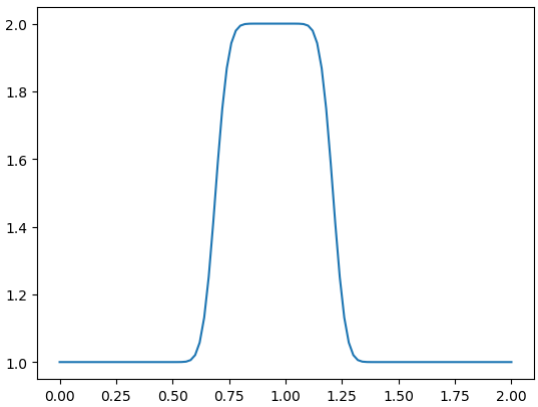
```
            for i in range(1, nx):
                u[i] = un[i] - c * dt / dx * (un[i] - un[i-1])
        pyplot.plot(numpy.linspace(0, 2, nx), u)
        pyplot.show()

    linearconv(41)
```



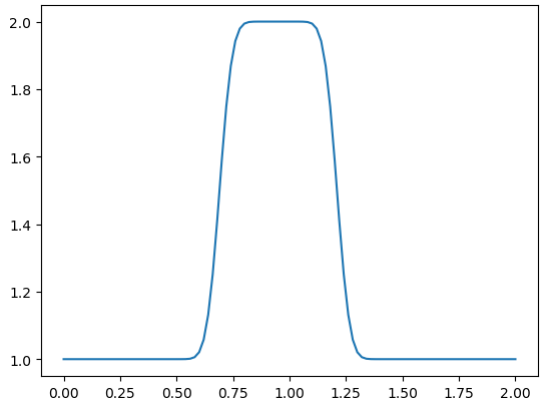

```
    linearconv(61)
```



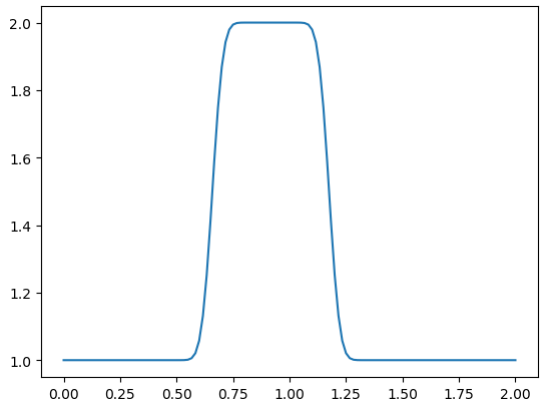

```
    linearconv(81)
```




```
    linearconv(101)
```




```
linearconv(121)
```



请注意，随着点 nx 数量的增加，波进行对流的距离越来越短。时间迭代次数在 nt = 20 时保持为常数，但取决于 nx 和相应的 dx 和 dt 的值，总体上来说检查的时间窗口更短。

3.3 了解更多

在某些情况下，有可能对数值方案的稳定性进行深入分析。在 YouTube 上的**视频讲座 9** 中观看巴尔巴教授对这一主题的介绍。

```
from IPython.display import YouTubeVideo
YouTubeVideo('Yw1YPBupZxU')
```

第 4 章　步骤 3

在继续之前，您应该已经完成了步骤 1 和 2。此 jupyter notebook 继续演示 **12 步到达纳维——斯托克斯**，此实践模块包括在罗瑞娜巴巴巴教授讲授的交互式 CFD 课程中。

4.1　1 维中的扩散方程

一维扩散方程是:

$$\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2}$$

你应该注意的第一件事是——与前面的两个简单的方程不同——这个方程具有二阶导数。我们首先需要学习如何处理它！

4.2　离散二阶导数

$\frac{\partial^2 u}{\partial x^2}$ 可在几何上表示为与一阶导数给出的曲线相切的线。我们将用一个中心差分格式对二阶导数进行离散化：一阶导数的前向差分和后向差分的组合。考虑 u_i 周围的 u_{i+1} 和 u_{i-1} 的泰勒展开：

$$u_{i+1} = u_i + \Delta x \frac{\partial u}{\partial x}\bigg|_i + \frac{\Delta x^2}{2!}\frac{\partial^2 u}{\partial x^2}\bigg|_i + \frac{\Delta x^3}{3!}\frac{\partial^3 u}{\partial x^3}\bigg|_i + O(\Delta x^4)$$

$$u_{i-1} = u_i - \Delta x \frac{\partial u}{\partial x}\bigg|_i + \frac{\Delta x^2}{2!}\frac{\partial^2 u}{\partial x^2}\bigg|_i - \frac{\Delta x^3}{3!}\frac{\partial^3 u}{\partial x^3}\bigg|_i + O(\Delta x^4)$$

如果我们将这两个展开式相加，你可以看到奇数编号的派生项将彼此抵消。如果我们忽略了 $O(\Delta x^4)$ 的项或者更高 (真的，那些是非常小的)，然后我们可以重新排列这两个展开式的和来求出我们的二阶导数。

$$u_{i+1} + u_{i-1} = 2u_i + \Delta x^2 \frac{\partial^2 u}{\partial x^2}\bigg|_i + O(\Delta x^4)$$

然后重新排序以求解 $\frac{\partial^2 u}{\partial x^2}\Big|_i$，其结果是：

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} + O(\Delta x^2)$$

4.3　返回步骤 3

现在我们可以写出 1 维扩散方程的离散版本：

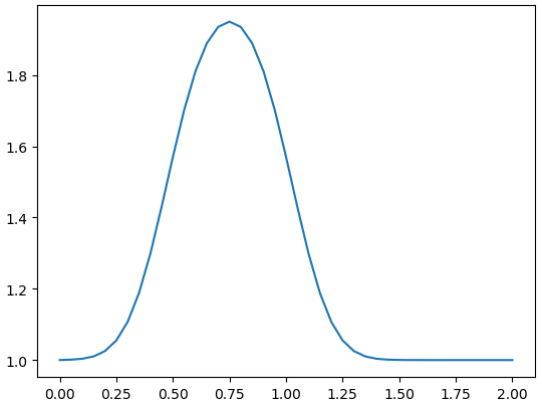
$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \nu \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}$$

就像以前一样，我们注意到，一旦我们有一个初始状态，唯一的未知数就是 u_i^{n+1}，因此我们重新安排了方程来求解：

$$u_i^{n+1} = u_i^n + \frac{\nu \Delta t}{\Delta x^2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$

上面的离散方程允许我们编写一个程序来求出一个时间上的解。但我们需要一个初始状态。让我们继续使用我们最喜欢的：帽子函数。于是，在 $t = 0$, 在区间 $0.5 \leq x \leq 1$ 中 $u = 2$ 其他情况 $u = 1$。我们准备好应对数字计算！

```
import numpy                    # 装载我们最喜欢的库
from matplotlib import pyplot   # 和有用的绘图库
%matplotlib inline
nx = 41
dx = 2 / (nx - 1)
nt = 20    # 我们要计算的时间步长数
nu = 0.3   # 粘度值
sigma = .2 # sigma是一个参数，我们稍后再学习
dt = sigma * dx**2 / nu # dt是使用sigma定义的..更晚些学！
u = numpy.ones(nx)      # 一个具有nx个元素的numpy数组，值均等于1。
u[int(.5 / dx):int(1 / dx + 1)] = 2  # 把在0.5和1之间的u设置成2。
un = numpy.ones(nx)  # 我们的占位符数组un，以得到时间上的解
for n in range(nt):  # 迭代时间
    un = u.copy()    # 将u的现有值复制到un
    for i in range(1, nx - 1):
        u[i] = un[i] + nu * dt / dx**2 * (un[i+1] - 2 * un[i] + un[i-1])
pyplot.plot(numpy.linspace(0, 2, nx), u)
pyplot.show()
```



4.4 了解更多

仔细演练有限差分扩散方程的离散化（以及从 1 到 4 的所有步骤），并请在 YouTube 上观看 Barba 教授的**视频课程 4**。

```
from IPython.display import YouTubeVideo
YouTubeVideo('y2WaK7_iMRI')
```

第 5 章　步骤 4

在步骤 4 中，我们继续我们的旅程来求解纳维 -斯托克斯方程。除非您完成了以前的步骤，否则不要继续！实际上，下一步将是前两个步骤的组合。代码重用的奇迹!

5.1　伯格斯方程

你可能在维基百科页面上阅读过伯格斯方程的内容。伯格斯方程是一维空间中的方程，如下所示：

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$$

正如你所看到的，它是非线性对流和扩散的组合。令人惊讶的是，你可从这个简洁的小方程式里学到很多东西！

我们可以使用我们已经在步骤 1 到 3 中详细说明的方法对其进行离散化。使用时间的前向差分，空间的后向差分和二阶导数的二阶方法得到：

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + u_i^n\frac{u_i^n - u_{i-1}^n}{\Delta x} = \nu\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}$$

就像以前一样，一旦我们有了一个初始状态，唯一的未知数就是 u_i^{n+1}。我们将在时间上进行以下步骤：

$$u_i^{n+1} = u_i^n - u_i^n\frac{\Delta t}{\Delta x}(u_i^n - u_{i-1}^n) + \nu\frac{\Delta t}{\Delta x^2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$

5.2　初始和边界条件

检验伯格斯方程的一些有趣性质，使用与我们之前步骤中不同的初始和边界条件有好处。

我们对这个问题的初始状态将是：

$$u = -\frac{2\nu}{\phi}\frac{\partial \phi}{\partial x} + 4$$
$$\phi = \exp\left(\frac{-x^2}{4\nu}\right) + \exp\left(\frac{-(x-2\pi)^2}{4\nu}\right)$$

其有一个解析解，给出如下：

$$u = -\frac{2\nu}{\phi}\frac{\partial \phi}{\partial x} + 4$$
$$\phi = \exp\left(\frac{-(x-4t)^2}{4\nu(t+1)}\right) + \exp\left(\frac{-(x-4t-2\pi)^2}{4\nu(t+1)}\right)$$

我们的边界条件是：

$$u(0) = u(2pi)$$

这叫做周期边界条件。注意！如果你不小心应对，这将使你感到头痛。

5.3 使用 SymPy 节省时间

我们在伯格斯方程中使用的初始状态在手工计算时你可能会感到些痛苦。导数 $\frac{\partial \phi}{\partial x}$ 也不是很困难，但是容易会在某处丢个符号，或忘记 x 的因子什么的，所以我们将使用 SymPy 来帮助我们搞定它。

SymPy 是一个 Python 的符号计算库。它具有许多与 MathMathica 相同的符号数学功能，附带的优点是，我们可以轻松地将其结果转换回我们的 Python 计算中（它也是免费和开源的）。

开始加载 SymPy 库，以及我们最喜欢的库，NumPy。

```
import numpy
import sympy
```

我们还将告诉 SymPy 我们希望使用它的所有输出转换成 LaTeX 格式. 这将使我们的笔记本很漂亮！

```
from sympy import init_printing
init_printing(use_latex=True)
```

通过在初始状态中设置三个变量的符号变量，然后写出完整的 ϕ 方程式。我们就会得到一个格式良好的 ϕ 方程式。

```
x, nu, t = sympy.symbols('x nu t')
phi = (sympy.exp(-(x - 4 * t)**2 / (4 * nu * (t + 1))) +
   sympy.exp(-(x - 4 * t - 2 * numpy.pi)**2 / (4 * nu * (t + 1))))
phi
```

$$e^{-\frac{(-4t+x-6.28318530717959)^2}{4\nu(t+1)}} + e^{-\frac{(-4t+x)^2}{4\nu(t+1)}}$$

可能有点小，但看起来是对的。现在求我们的偏导数 $\frac{\partial \phi}{\partial x}$ 是一件简单的事。

```
phiprime = phi.diff(x)
phiprime
```

$$-\frac{e^{-\frac{(-4t+x)^2}{4\nu(t+1)}}}{4\nu(t+1)}(-8t+2x) - \frac{1}{4\nu(t+1)}(-8t+2x-12.5663706143592)e^{-\frac{(-4t+x-6.28318530717959)^2}{4\nu(t+1)}}$$

如果要查看未渲染的版本，只使用 Python print 命令。

```
print(phiprime)
-(-8*t + 2*x)*exp(-(-4*t + x)**2/(4*nu*(t + 1)))/(4*nu*(t + 1)) -
(-8*t + 2*x - 12.5663706143592)*exp(-(-4*t + x - 6.28318530717959)**2
/(4*nu*(t + 1)))/(4*nu*(t + 1))
```

5.4 现在做什么？

现在，我们有了我们 Python 化版本的导数，我们可以完成写出完整的初始状态方程，然后将它转化为可用的 Python 表达式。为此，我们将使用 *lambdify* 函数，它需要一个 SymPy 的符号方程，并把它变成一个可调用的函数。

```
from sympy.utilities.lambdify import lambdify
u = -2 * nu * (phiprime / phi) + 4
print(u)
-2*nu*(-(-8*t + 2*x)*exp(-(-4*t + x)**2/(4*nu*(t + 1)))/(4*nu*(t + 1))
- (-8*t + 2*x - 12.5663706143592)*exp(-(-4*t + x - 6.28318530717959)**2
/(4*nu*(t + 1)))/(4*nu*(t + 1)))/(exp(-(-4*t + x - 6.28318530717959)**2
/(4*nu*(t + 1))) + exp(-(-4*t + x)**2/(4*nu*(t + 1)))) + 4
```

5.5 Lambdify 函数

要将此表达式 lambda 化为可用函数，我们会告诉 lambdify 函数哪些变量和函数进行需要带入其中。

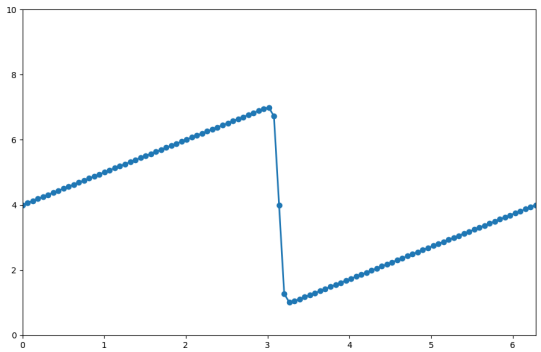
```
ufunc = lambdify((t, x, nu), u)
print(ufunc(1, 4, 3))
3.4917066420644494
```

5.6 回到伯格斯方程

既然我们有了最初的条件，我们就可以着手解决这个问题。我们可以使用我们的 lambdify 函数来生成初始状态的图形。

```
from matplotlib import pyplot
%matplotlib inline
# 变量声明
nx = 101
nt = 100
dx = 2 * numpy.pi / (nx - 1)
nu = .07
dt = dx * nu

x = numpy.linspace(0, 2 * numpy.pi, nx)
un = numpy.empty(nx)
t = 0

u = numpy.asarray([ufunc(t, x0, nu) for x0 in x])
u
```

```
    array([ 4.        ,  4.06283185,  4.12566371,  4.18849556,  4.25132741,
        4.31415927,  4.37699112,  4.43982297,  4.50265482,  4.56548668,
        4.62831853,  4.69115038,  4.75398224,  4.81681409,  4.87964594,
        4.9424778 ,  5.00530965,  5.0681415 ,  5.13097336,  5.19380521,
        5.25663706,  5.31946891,  5.38230077,  5.44513262,  5.50796447,
        5.57079633,  5.63362818,  5.69646003,  5.75929189,  5.82212374,
        5.88495559,  5.94778745,  6.0106193 ,  6.07345115,  6.136283  ,
        6.19911486,  6.26194671,  6.32477856,  6.38761042,  6.45044227,
        6.51327412,  6.57610598,  6.63893783,  6.70176967,  6.76460125,
        6.82742866,  6.89018589,  6.95176632,  6.99367964,  6.72527549,
        4.        ,  1.27472451,  1.00632036,  1.04823368,  1.10981411,
        1.17257134,  1.23539875,  1.29823033,  1.36106217,  1.42389402,
        1.48672588,  1.54955773,  1.61238958,  1.67522144,  1.73805329,
        1.80088514,  1.863717  ,  1.92654885,  1.9893807 ,  2.05221255,
        2.11504441,  2.17787626,  2.24070811,  2.30353997,  2.36637182,
        2.42920367,  2.49203553,  2.55486738,  2.61769923,  2.68053109,
        2.74336294,  2.80619479,  2.86902664,  2.9318585 ,  2.99469035,
        3.0575222 ,  3.12035406,  3.18318591,  3.24601776,  3.30884962,
        3.37168147,  3.43451332,  3.49734518,  3.56017703,  3.62300888,
        3.68584073,  3.74867259,  3.81150444,  3.87433629,  3.93716815,  4.
])

    \begin{lstlisting}
    pyplot.figure(figsize=(11, 7), dpi=100)
    pyplot.plot(x, u, marker='o', lw=2)
    pyplot.xlim([0, 2 * numpy.pi])
    pyplot.ylim([0, 10])
    pyplot.show()
```



这绝对不是我们一直在处理的帽子函数。我们称之为“锯齿函数”。让我们前进一步看看发生了什么。

5.7 周期边界条件

步骤 4 和以前的课程之间的一大区别是使用了周期性边界条件。如果您对步骤 1 和步骤 2 进行了实验，并使模拟运行时间更长（通过增加 nt 的值）您会注意到，波将一直向右移动，直到它不再显示在绘图窗口中为止。

在周期性边界条件下，当某个点到达画面的右侧时，它会绕回到画面的前面。

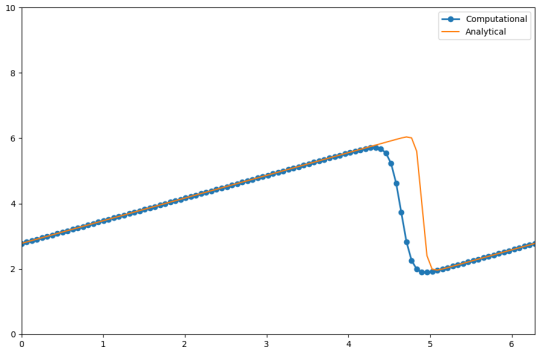
回顾我们在本笔记本开头所做的离散化：

$$u_i^{n+1} = u_i^n - u_i^n \frac{\Delta t}{\Delta x}(u_i^n - u_{i-1}^n) + \nu \frac{\Delta t}{\Delta x^2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$

当 i 已经在画面的尾部时，u_{i+1}^n 意味什么？

在继续之前考虑一下这个问题。

```
for n in range(nt):
un = u.copy()
for i in range(1, nx-1):
    u[i] = un[i] - un[i] * dt / dx *(un[i] - un[i-1]) + nu * dt / dx**2 *\
            (un[i+1] - 2 * un[i] + un[i-1])
u[0] = un[0] - un[0] * dt / dx * (un[0] - un[-2]) + nu * dt / dx**2 *\
            (un[1] - 2 * un[0] + un[-2])
u[-1] = u[0]

u_analytical = numpy.asarray([ufunc(nt * dt, xi, nu) for xi in x])
pyplot.figure(figsize=(11, 7), dpi=100)
pyplot.plot(x,u, marker='o', lw=2, label='Computational')
pyplot.plot(x, u_analytical, label='Analytical')
pyplot.xlim([0, 2 * numpy.pi])
pyplot.ylim([0, 10])
pyplot.legend()
pyplot.show()
```



5.8 接下来是什么？

随后的步骤 5 到 12 中将在二维中讨论。但很容易将 1 维有限差分公式扩展到 2 维或 3 维中的偏导数。仅应用定义——关于 x 的偏导数是在 x 方向为变量同时让 y 为常数——即可。

在移动到步骤 5 前，请确保你已经完成了步骤 1 到 4 的代码，并且你已经尝试了不同参数并考虑了所发生的事情。此外，我们建议你稍微休息一下来学习使用 NumPy 进行数组操作的内容

第 6 章　使用 NumPy 进行数组操作

对于计算更密集的程序，使用内置的 NumPy 函数可以使执行速度提高许多倍。作为一个简单的例子，请考虑以下等式：

$$u_i^{n+1} = u_i^n - u_{i-1}^n$$

现在，给定一个向量 $u^n = [0, 1, 2, 3, 4, 5]$ 我们可以通过使用一个 for 循环迭代 u^n 来计算 u^{n+1} 的值。

```
import numpy
u = numpy.array((0, 1, 2, 3, 4, 5))
for i in range(1, len(u)):
    print(u[i] - u[i-1])
```

```
1
1
1
1
```

这是预期的结果，并且执行时间几乎是瞬时的。如果我们执行与数组操作相同的操作，而不是单独计算 $u_i^n - u_{i-1}^n 5$ 次，我们可以使用一个命令执行切片 u 数组和计算每次运算。

```
u[1:] - u[0:-1]
```

array([1, 1, 1, 1, 1])

此命令将用 u 中第 1、2、3、4、5、6 元素减去 u 中第 0、1、2、3、4、5 元素。

6.1　速度增加

对于 6 个元素的数组，数组操作的好处是非常小的。执行时间不会有明显的差异，因为发生了很少的操作。但是，如果我们重新审视 2 维线性对流，我们可以看到一些实质性的速度增加。

```
nx = 81
ny = 81
nt = 100
c = 1
dx = 2 / (nx - 1)
dy = 2 / (ny - 1)
sigma = .2
dt = sigma * dx
```

```
x = numpy.linspace(0, 2, nx)
y = numpy.linspace(0, 2, ny)

u = numpy.ones((ny, nx)) # 创建一个值均为1的1xn的向量
un = numpy.ones((ny, nx))
# 分配初始状态
u[int(.5 / dy): int(1 / dy + 1), int(.5 / dx):int(1 / dx + 1)] = 2
```

在我们最初的条件下，让我们首先尝试运行我们最初的嵌套循环代码，使用 IPython“魔法”函数%%timeit 这将有助于我们评估我们的代码的性能。

注释：%%timeit 函数将多次运行代码，然后给出平均执行时间作为结果。如果在您运行了%%timeit 命令的单元格中绘制了任何图形，它将反复绘制那些图形，这会有点乱。

```
%%timeit
u = numpy.ones((ny, nx))
u[int(.5 / dy): int(1 / dy + 1), int(.5 / dx):int(1 / dx + 1)] = 2

for n in range(nt + 1): # 按时间步数循环
    un = u.copy()
    row, col = u.shape
    for j in range(1, row):
        for i in range(1, col):
            u[j, i] = (un[j, i] - (c * dt / dx * (un[j, i] - un[j, i - 1]))
                       - (c * dt / dy * (un[j, i] - un[j - 1, i])))
            u[0, :] = 1
            u[-1, :] = 1
            u[:, 0] = 1
            u[:, -1] = 1
1 loops, best of 3: 1.94 s per loop
```

有了上面“原始”的 Python 代码，实际的最佳执行时间为 1.94 秒。记住，在这三个嵌套的循环中，j 循环中的语句运行超过 650000 次。让我们比较一下数组操作实现的相同代码的性能：

```
%%timeit
u = numpy.ones((ny, nx))
u[int(.5 / dy): int(1 / dy + 1), int(.5 / dx):int(1 / dx + 1)] = 2

for n in range(nt + 1): # 按时间步数循环
    un = u.copy()
    u[1:, 1:] = un[1:, 1:] - ((c * dt / dx * (un[1:, 1:] - un[1:, 0:-1]))
                - (c * dt / dy * (un[1:, 1:] - un[0:-1, 1:])))
    u[0, :] = 1
    u[-1, :] = 1
    u[:, 0] = 1
```

```
        u[:, −1] = 1
    100 loops, best of 3: 5.09 ms per loop
```

正如你所看到的，速度的增加是相当大的。同样的计算从 1.94 秒至 5.09 毫秒。2 秒不是很多时间，但是这些速度增加随着所评估问题的大小和复杂性呈指数增长。

第 7 章　步骤 5

到目前为止，我们所有的工作都是在一个空间维度上进行的（步骤 1 至 4)。只在 1 维中我们可以学到很多，但让我们攀上平面：两个维度。

在以下练习中，您会将前四个步骤扩展到 2 维。为了将 1 维有限差分公式扩展到 2 维或 3 维中的偏导数，只使用定义：关于 x 的偏导数表示在 y 是常数时 x 方向的变化。

在二维空间中，矩形（统一）网格由坐标点定义：

$$x_i = x_0 + i\Delta x$$
$$y_i = y_0 + i\Delta y$$

现在，定义 $u_{i,j} = u(x_i, y_j)$ 并在分离为 i 和 j 索引的任意变量 x, y 上应用有限差分公式。所有的导数都是 $u_{i,j}$ 周围的网格点的二维泰勒展开。

因此，对于 x 方向的一阶偏导数，有限差分公式为：

$$\left.\frac{\partial u}{\partial x}\right|_{i,j} = \frac{u_{i+1,j} - u_{i,j}}{\Delta x} + \mathcal{O}(\Delta x)$$

并且在 y 方向类似。因此，我们可以为步骤 5 至 12 写出后向差分、前向差分或中心差公式。让我们开始吧！

7.1　2 维线性对流

控制 2 维线性对流的偏微分方程（PDE）可写成：

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} + c\frac{\partial u}{\partial y} = 0$$

除了我们现在有两个空间维度需要我们在时间上进行计算以外，方程与 1 维线性对流具有完全相同的形式。

同样，时间步长将离散为正向差分，并且两个空间步长都将离散为后向差异。

对于 1 维实现，我们使用下标 i 表示空间中的移动（例如，$u_i^n - u_{i-1}^n$)。现在我们有两个维度要考虑，我们需要添加第二个下标，j 以便考虑到该结构中的所有信息。

在这里，我们还会再用 i 作为我们 x 值的索引，我们将添加下标 j 来跟踪我们的 y 值。

考虑到这一点，我们的离散 PDE 方程则相对简单了。

$$\frac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} + c\frac{u_{i,j}^n - u_{i-1,j}^n}{\Delta x} + c\frac{u_{i,j}^n - u_{i,j-1}^n}{\Delta y} = 0$$

与以前一样，要求解的未知量是：

$$u_{i,j}^{n+1} = u_{i,j}^{n} - c\frac{\Delta t}{\Delta x}(u_{i,j}^{n} - u_{i-1,j}^{n}) - c\frac{\Delta t}{\Delta y}(u_{i,j}^{n} - u_{i,j-1}^{n})$$

我们将在以下初始状态下求解该方程：

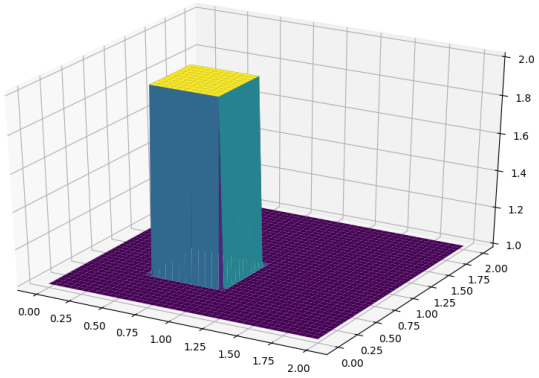
$$u(x,y) = \begin{cases} 2 \text{ for } & 0.5 \leq x, y \leq 1 \\ 1 \text{ for } & \text{everywhere else} \end{cases}$$

且边界条件为：

$$u = 1 \text{ for } \begin{cases} x = 0,\ 2 \\ y = 0,\ 2 \end{cases}$$

```
from mpl_toolkits.mplot3d import Axes3D    # 3维绘图所需的新库

import numpy
from matplotlib import pyplot, cm
%matplotlib inline

# 声明变量
nx = 81
ny = 81
nt = 100
c = 1
dx = 2 / (nx - 1)
dy = 2 / (ny - 1)
sigma = .2
dt = sigma * dx

x = numpy.linspace(0, 2, nx)
y = numpy.linspace(0, 2, ny)

u = numpy.ones((ny, nx)) # 创建一个值均为1的1xn的向量
un = numpy.ones((ny, nx))

# 分配初始状态

# 设置帽子函数初始状态:u(.5<=x<=1&&.5<=y<=1)为2
u[int(.5 / dy):int(1 / dy + 1),int(.5 / dx):int(1 / dx + 1)] = 2

# 绘制初始状态
# 图形尺寸参数可用于产生不同尺寸的图像
fig = pyplot.figure(figsize=(11, 7), dpi=100)
ax = fig.gca(projection='3d')
X, Y = numpy.meshgrid(x, y)
```

```
surf = ax.plot_surface(X, Y, u[:], cmap=cm.viridis)
pyplot.show()
```



7.2 3 维绘图说明

为了绘制 3 维投影图，请确保已添加 Axes3D 库。

```
from mpl_toolkits.mplot3d import Axes3D
```

正确的绘图命令比简单的 2 维绘图命令复杂些。

```
fig = pyplot.figure(figsize=(11, 7), dpi=100)
ax = fig.gca(projection='3d')
surf2 = ax.plot_surface(X, Y, u[:])
```

第一行初始化一个图形窗口。**figsize** 与 **dpi** 命令为可选项且只用来指定正在产生的图形大小和分辨率。你可以忽略它们，但你任然需要

```
fig = pyplot.figure()
```

下一行将绘图窗口分配给轴标签'ax' 并且还指定它是 3 维投影图。最后一行使用命令

```
plot_surface()
```

这相当于常规的绘图命令，但它为数据点位置设置了 X 和 Y 值构成的网格。

注释

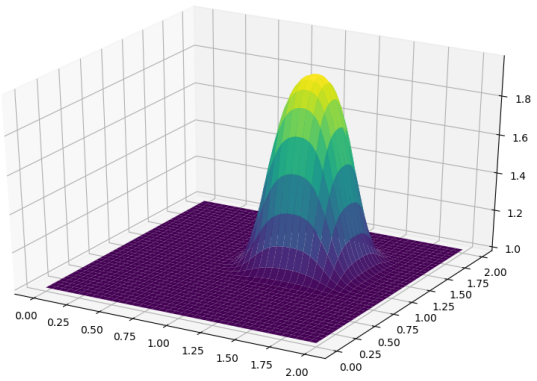
你传递到 plot_surface 的 X 和 Y 的值不是 1 维矢量 x 和 y。为了使用 matplotlib 的 3 维绘图函数，需要生成对应于绘图帧中的每个坐标点的由 x，y 值构成的网格。使用 NumPy 的 meshgrid 函数生成此坐标网格。

```
X, Y = numpy.meshgrid(x, y)
```

7.3 在两个维度上迭代

要对两个维度中的波进行评估，需要使用多个嵌套的 for 循环来覆盖所有的 i 和 j 循环。由于 Python 不是编译的语言，所以在执行具有多个 for 循环的代码时可能会出现明显的减慢。首先尝试评估二维对流代码并查看其产生的结果。

```
u = numpy.ones((ny, nx))
u[int(.5 / dy):int(1 / dy + 1), int(.5 / dx):int(1 / dx + 1)] = 2
for n in range(nt + 1): # 按时间步数循环
    un = u.copy()
    row, col = u.shape
    for j in range(1, row):
        for i in range(1, col):
            u[j, i] = (un[j, i] - (c * dt / dx * (un[j, i] - un[j, i - 1]))
                       - (c * dt / dy * (un[j, i] - un[j - 1, i])))
            u[0, :] = 1
            u[-1, :] = 1
            u[:, 0] = 1
            u[:, -1] = 1
fig = pyplot.figure(figsize=(11, 7), dpi=100)
ax = fig.gca(projection='3d')
X, Y = numpy.meshgrid(x, y)
surf2 = ax.plot_surface(X, Y, u[:], cmap=cm.viridis)
pyplot.show()
```

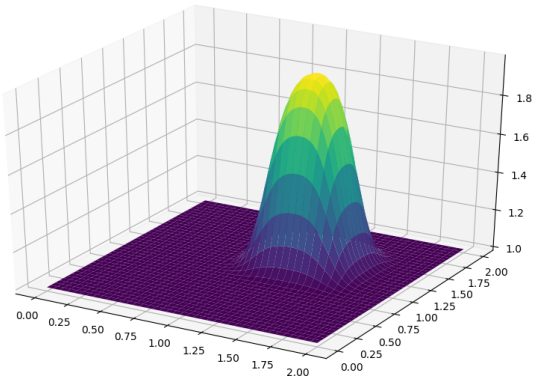


7.4 数组操作

在此实现了相同的 2 维对流代码，没有使用嵌套的 for 循环，而使用数组操作来评估相同的计算。

```
u = numpy.ones((ny, nx))
u[int(.5 / dy):int(1 / dy + 1), int(.5 / dx):int(1 / dx + 1)] = 2
for n in range(nt + 1): # 按时间步数循环
    un = u.copy()
    u[1:, 1:] = (un[1:, 1:] - (c * dt / dx * (un[1:, 1:] - un[1:, :-1]))
                 - (c * dt / dy * (un[1:, 1:] - un[:-1, 1:])))
    u[0, :] = 1
    u[-1, :] = 1
    u[:, 0] = 1
```

```
    u[:, -1] = 1
fig = pyplot.figure(figsize=(11, 7), dpi=100)
ax = fig.gca(projection='3d')
X, Y = numpy.meshgrid(x, y)
surf2 = ax.plot_surface(X, Y, u[:], cmap=cm.viridis)
pyplot.show()
```



7.5 了解更多

关于步骤 5（和步骤 8）详细内容的视频课程位于 YouTube 上的**视频课程 6**：

```
from IPython.display import YouTubeVideo
YouTubeVideo('tUg_dE3NXoY')
```

第 8 章　步骤 6

在继续本节课之前你自己应该已经完成了步骤 5 的代码。与步骤 1 到步骤 4 一样，我们将逐步构建，因此完成上一步是很重要的！

我们继续..

8.1　2 维对流

现在我们求解 2 维对流方程，由下面的一对耦合的偏微分方程表示：

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = 0$$
$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = 0$$

使用我们以前所应用的方法对这些方程进行离散化：

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} + u_{i,j}^{n}\frac{u_{i,j}^{n} - u_{i-1,j}^{n}}{\Delta x} + v_{i,j}^{n}\frac{u_{i,j}^{n} - u_{i,j-1}^{n}}{\Delta y} = 0$$
$$\frac{v_{i,j}^{n+1} - v_{i,j}^{n}}{\Delta t} + u_{i,j}^{n}\frac{v_{i,j}^{n} - v_{i-1,j}^{n}}{\Delta x} + v_{i,j}^{n}\frac{v_{i,j}^{n} - v_{i,j-1}^{n}}{\Delta y} = 0$$

重新排列两个方程，我们求解 $u_{i,j}^{n+1}$ 以及 $v_{i,j}^{n+1}$。注意，这些方程也是耦合的。

$$u_{i,j}^{n+1} = u_{i,j}^{n} - u_{i,j}\frac{\Delta t}{\Delta x}(u_{i,j}^{n} - u_{i-1,j}^{n}) - v_{i,j}^{n}\frac{\Delta t}{\Delta y}(u_{i,j}^{n} - u_{i,j-1}^{n})$$
$$v_{i,j}^{n+1} = v_{i,j}^{n} - u_{i,j}\frac{\Delta t}{\Delta x}(v_{i,j}^{n} - v_{i-1,j}^{n}) - v_{i,j}^{n}\frac{\Delta t}{\Delta y}(v_{i,j}^{n} - v_{i,j-1}^{n})$$

8.2　初始状态

初始状态与应用于 X 和 Y 方向上的 1 维对流方程的条件相同。

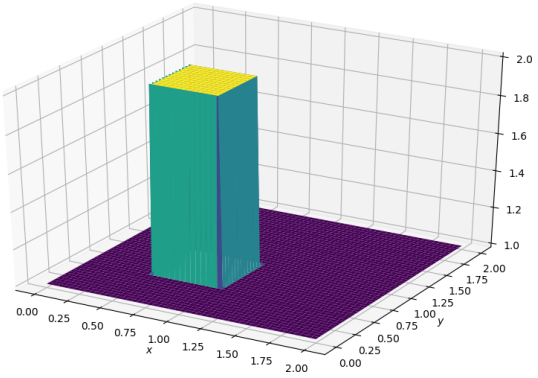
$$u,\ v\ = \begin{cases} 2 & \text{for } x, y \in (0.5, 1) \times (0.5, 1) \\ 1 & \text{everywhere else} \end{cases}$$

8.3 边界条件

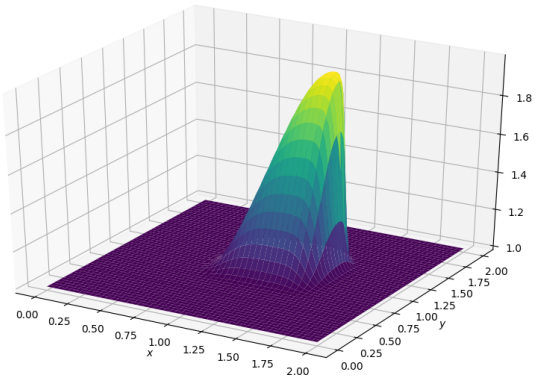
沿网格边界上边界条件保持 u 和 v 等于 1。

$$u = 1,\ v = 1 \text{ for } \begin{cases} x = 0, 2 \\ y = 0, 2 \end{cases}$$

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import pyplot, cm
import numpy
%matplotlib inline

# 声明变量
nx = 101
ny = 101
nt = 80
c = 1
dx = 2 / (nx − 1)
dy = 2 / (ny − 1)
sigma = .2
dt = sigma * dx

x = numpy.linspace(0, 2, nx)
y = numpy.linspace(0, 2, ny)

u = numpy.ones((ny, nx)) # 创建一个值均为1的1xn的向量
v = numpy.ones((ny, nx))
un = numpy.ones((ny, nx))
vn = numpy.ones((ny, nx))

# 分配初始状态
# 设置帽子函数初始状态 : u(.5<=x<=1 && .5<=y<=1 ) 为 2
u[int(.5 / dy):int(1 / dy + 1), int(.5 / dx):int(1 / dx + 1)] = 2
# 设置帽子函数初始状态 : v(.5<=x<=1 && .5<=y<=1 ) 为 2
v[int(.5 / dy):int(1 / dy + 1), int(.5 / dx):int(1 / dx + 1)] = 2
fig = pyplot.figure(figsize=(11, 7), dpi=100)
ax = fig.gca(projection='3d')
X, Y = numpy.meshgrid(x, y)

ax.plot_surface(X, Y, u, cmap=cm.viridis, rstride=2, cstride=2)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
pyplot.show()
```




```
for n in range(nt + 1): # 按时间步数循环
    un = u.copy()
    vn = v.copy()
    u[1:, 1:] = (un[1:, 1:] - (un[1:, 1:] * c * dt / dx * (un[1:, 1:] -
                 un[1:, :-1])) - vn[1:, 1:] * c * dt / dy * (un[1:, 1:]
                 - un[:-1, 1:]))
    v[1:, 1:] = (vn[1:, 1:] - (un[1:, 1:] * c * dt / dx * (vn[1:, 1:] -
                 vn[1:, :-1])) - vn[1:, 1:] * c * dt / dy * (vn[1:, 1:]
                 - vn[:-1, 1:]))
    u[0, :] = u[-1, :] = u[:, 0] = u[:, -1] = 1


    v[0, :] = v[-1, :] = v[:, 0] = v[:, -1] = 1
fig = pyplot.figure(figsize=(11, 7), dpi=100)
ax = fig.gca(projection='3d')
X, Y = numpy.meshgrid(x, y)
ax.plot_surface(X, Y, u, cmap=cm.viridis, rstride=2, cstride=2)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
pyplot.show()
```



8.4 了解更多

关于步骤 5 到步骤 8 详细内容的视频课程位于 YouTube 上的**视频课程 6:**

```
from IPython.display import YouTubeVideo
YouTubeVideo('tUg_dE3NXoY')
```

第 9 章　步骤 7

你知道怎么回事.. 我们现在将处理 2 维扩散方程，接下来我们将结合步骤 6 和 7 来解决伯格斯方程。因此，在继续进行之前请确保前期工作良好。

9.1　2 维扩散

下面是 2 维扩散方程：

$$\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2} + \nu \frac{\partial^2 u}{\partial y^2}$$

你会记得，在步骤 3 中当研究 1 维扩散时，我们提出了一种对二阶导数进行离散化的方法。我们将在这里使用具有前向差分和两个二阶导数的相同方案。

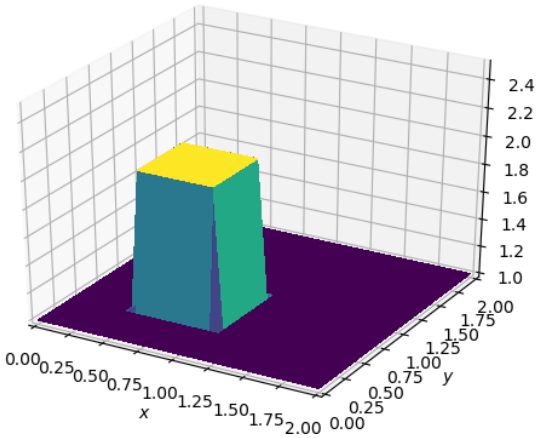
$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} = \nu \frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{\Delta x^2} + \nu \frac{u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}}{\Delta y^2}$$

再次，我们重新排列离散方程并求解 $u_{i,j}^{n+1}$

$$u_{i,j}^{n+1} = u_{i,j}^{n} + \frac{\nu \Delta t}{\Delta x^2}(u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}) + \frac{\nu \Delta t}{\Delta y^2}(u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n})$$

```
import numpy
from matplotlib import pyplot, cm
from mpl_toolkits.mplot3d import Axes3D # library for 3d projection plots
%matplotlib inline
# 声明变量
nx = 31
ny = 31
nt = 17
nu = .05
dx = 2 / (nx - 1)
dy = 2 / (ny - 1)
sigma = .25
dt = sigma * dx * dy / nu

x = numpy.linspace(0, 2, nx)
y = numpy.linspace(0, 2, ny)

u = numpy.ones((ny, nx))  # 创建一个值均为1的1xn的向量
```

```
un = numpy.ones((ny, nx))

# 分配初始状态
# 设置帽子函数的初始状态：u(.5<=x<=1 && .5<=y<=1 ) 为 2
u[int(.5 / dy):int(1 / dy + 1),int(.5 / dx):int(1 / dx + 1)] = 2
fig = pyplot.figure()
ax = fig.gca(projection='3d')
X, Y = numpy.meshgrid(x, y)
surf = ax.plot_surface(X, Y, u, rstride=1, cstride=1,
                       cmap=cm.viridis, linewidth=0, antialiased=False)

ax.set_xlim(0, 2)
ax.set_ylim(0, 2)
ax.set_zlim(1, 2.5)

ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
pyplot.show()
```

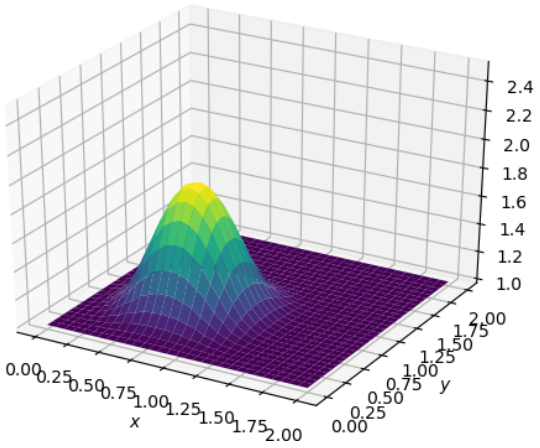


$$u_{i,j}^{n+1} = u_{i,j}^n + \frac{\nu\Delta t}{\Delta x^2}(u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n) + \frac{\nu\Delta t}{\Delta y^2}(u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n)$$

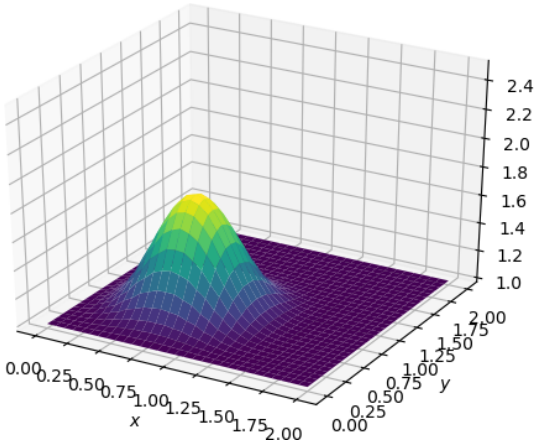
```
# 运行nt个时间步长
def diffuse(nt):
    u[int(.5 / dy):int(1 / dy + 1),int(.5 / dx):int(1 / dx + 1)] = 2

    for n in range(nt + 1):
        un = u.copy()
        u[1:-1,1:-1] = (un[1:-1,1:-1] + nu * dt / dx**2 * (un[1:-1,2:] -
                        2 * un[1:-1,1:-1] + un[1:-1,0:-2]) + nu * dt /
                        dy**2 * (un[2:,1:-1] - 2 * un[1:-1,1:-1] +
                        un[0:-2,1:-1]))
        u[0, :] = 1
        u[-1, :] = 1
```

```
        u[:, 0] = 1
        u[:, -1] = 1


    fig = pyplot.figure()
    ax = fig.gca(projection='3d')
    X, Y = numpy.meshgrid(x, y)
    surf = ax.plot_surface(X, Y, u[:], rstride=1, cstride=1,
                           cmap=cm.viridis, linewidth=0, antialiased=True)
    ax.set_zlim(1, 2.5)
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    pyplot.show()
```

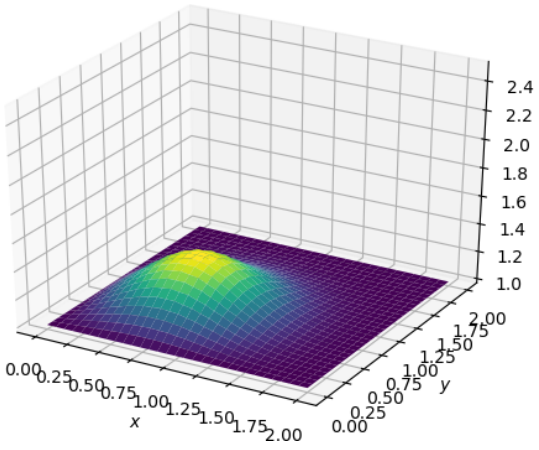
```
diffuse(10)
```




```
diffuse(14)
```




```
diffuse(50)
```



9.2 了解更多

关于步骤 5 到步骤 8 详细内容的视频课程位于 YouTube 上的**视频课程 6**：

```
from IPython.display import YouTubeVideo
YouTubeVideo('tUg_dE3NXoY')
```

第 10 章　步骤 8

这将是一个里程碑！我们现在进入第 8 步：伯格斯方程。从这个方程中我们可以学到更多的东西。它在流体力学中扮演非常重要的角色，因为它包含了流动方程的全对流非线性，同时也有许多已知的解析解。

10.1　2 维伯格斯方程

记住，伯格斯方程可以从光滑的初始状态产生不连续的解，例如，可以产生“冲击”。我们现在要在两个维度中看到它！

以下是我们耦合的 PDE 方程：

$$\frac{\partial u}{\partial t}+u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}=\nu\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)$$
$$\frac{\partial v}{\partial t}+u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}=\nu\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}\right)$$

我们知道如何离散每一项：在前面我们已经做过了！

$$\frac{u_{i,j}^{n+1}-u_{i,j}^n}{\Delta t}+u_{i,j}^n\frac{u_{i,j}^n-u_{i-1,j}^n}{\Delta x}+v_{i,j}^n\frac{u_{i,j}^n-u_{i,j-1}^n}{\Delta y}=\nu\left(\frac{u_{i+1,j}^n-2u_{i,j}^n+u_{i-1,j}^n}{\Delta x^2}+\frac{u_{i,j+1}^n-2u_{i,j}^n+u_{i,j-1}^n}{\Delta y^2}\right)$$

$$\frac{v_{i,j}^{n+1}-v_{i,j}^n}{\Delta t}+u_{i,j}^n\frac{v_{i,j}^n-v_{i-1,j}^n}{\Delta x}+v_{i,j}^n\frac{v_{i,j}^n-v_{i,j-1}^n}{\Delta y}=\nu\left(\frac{v_{i+1,j}^n-2v_{i,j}^n+v_{i-1,j}^n}{\Delta x^2}+\frac{v_{i,j+1}^n-2v_{i,j}^n+v_{i,j-1}^n}{\Delta y^2}\right)$$

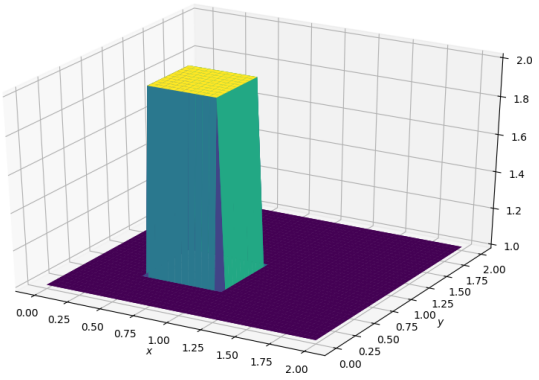
现在，我们将重新排列这些每个只有一个未知量的方程：即在下一步要求解的 u, v 分量：

$$u_{i,j}^{n+1}=u_{i,j}^n-\frac{\Delta t}{\Delta x}u_{i,j}^n(u_{i,j}^n-u_{i-1,j}^n)-\frac{\Delta t}{\Delta y}v_{i,j}^n(u_{i,j}^n-u_{i,j-1}^n)+$$
$$\frac{\nu\Delta t}{\Delta x^2}(u_{i+1,j}^n-2u_{i,j}^n+u_{i-1,j}^n)+\frac{\nu\Delta t}{\Delta y^2}(u_{i,j+1}^n-2u_{i,j}^n+u_{i,j-1}^n)$$

$$v_{i,j}^{n+1}=v_{i,j}^n-\frac{\Delta t}{\Delta x}u_{i,j}^n(v_{i,j}^n-v_{i-1,j}^n)-\frac{\Delta t}{\Delta y}v_{i,j}^n(v_{i,j}^n-v_{i,j-1}^n)+$$
$$\frac{\nu\Delta t}{\Delta x^2}(v_{i+1,j}^n-2v_{i,j}^n+v_{i-1,j}^n)+\frac{\nu\Delta t}{\Delta y^2}(v_{i,j+1}^n-2v_{i,j}^n+v_{i,j-1}^n)$$

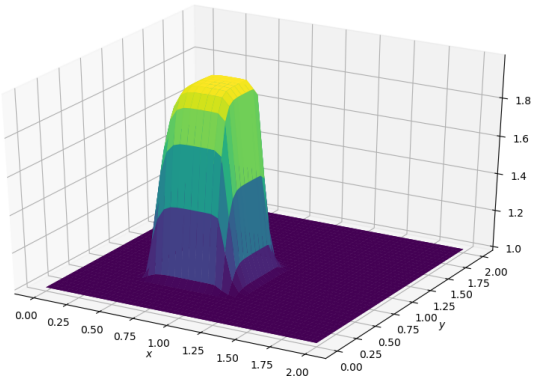
```
import numpy
from matplotlib import pyplot, cm
from mpl_toolkits.mplot3d import Axes3D
```

```
%matplotlib inline
# 声明变量
nx = ny = 41
nt = 120
c = 1
dx = dy = 2 / (ny − 1)
sigma = .0009
nu = 0.01
dt = sigma * dx * dy / nu
x = numpy.linspace(0, 2, nx)
y = numpy.linspace(0, 2, ny)

# 创建一个值均为1的1xn的向量
u = v = numpy.ones((ny, nx))
un = vn = numpy.ones((ny, nx))
comb = numpy.ones((ny, nx))

# 分配初始状态
# 设置帽子函数初始状态：u(.5<=x<=1 && .5<=y<=1 )为2
u[int(.5 / dy):int(1 / dy + 1),int(.5 / dx):int(1 / dx + 1)] = 2
# 设置帽子函数初始状态 ：v(.5<=x<=1 && .5<=y<=1 )为2
v[int(.5 / dy):int(1 / dy + 1),int(.5 / dx):int(1 / dx + 1)] = 2
# (绘制初始状态)
fig = pyplot.figure(figsize=(11, 7), dpi=100)
ax = fig.gca(projection='3d')
X, Y = numpy.meshgrid(x, y)
ax.plot_surface(X, Y, u[:], cmap=cm.viridis, rstride=1, cstride=1)
ax.plot_surface(X, Y, v[:], cmap=cm.viridis, rstride=1, cstride=1)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
pyplot.show()
```




```
for n in range(nt + 1): # 按时间步长循环
    un = u.copy()
```

```
    vn = v.copy()

    u[1:-1,1:-1] = (un[1:-1,1:-1] - dt / dx * un[1:-1,1:-1] * (un[1:-1,1:-1]
                   - un[1:-1,0:-2]) - dt / dy * vn[1:-1,1:-1] * (un[1:-1,1:-1]
                   - un[0:-2,1:-1]) + nu * dt / dx**2 * (un[1:-1,2:] -
                   2 * un[1:-1,1:-1] + un[1:-1,0:-2]) + nu * dt / dy**2 *
                   (un[2:,1:-1] - 2 * un[1:-1,1:-1] + un[0:-2,1:-1]))

    v[1:-1,1:-1] = (vn[1:-1,1:-1] - dt / dx * un[1:-1,1:-1] * (vn[1:-1,1:-1]
                   - vn[1:-1,0:-2]) - dt / dy * vn[1:-1,1:-1] * (vn[1:-1,1:-1]
                   - vn[0:-2,1:-1]) + nu * dt / dx**2 * (vn[1:-1,2:] -
                   2 * vn[1:-1,1:-1] + vn[1:-1,0:-2]) + nu * dt / dy**2 *
                   (vn[2:,1:-1] - 2 * vn[1:-1,1:-1] + vn[0:-2,1:-1]))

    u[0, :] = 1
    u[-1, :] = 1
    u[:, 0] = 1
    u[:, -1] = 1

    v[0, :] = 1
    v[-1, :] = 1
    v[:, 0] = 1
    v[:, -1] = 1
fig = pyplot.figure(figsize=(11, 7), dpi=100)
ax = fig.gca(projection='3d')
X, Y = numpy.meshgrid(x, y)
ax.plot_surface(X, Y, u, cmap=cm.viridis, rstride=1, cstride=1)
ax.plot_surface(X, Y, v, cmap=cm.viridis, rstride=1, cstride=1)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
pyplot.show()
```



10.2 了解更多

关于步骤 5 到步骤 8 详细内容的视频课程位于 YouTube 上的**视频课程 6**：

```
from IPython.display import YouTubeVideo
YouTubeVideo('tUg_dE3NXoY')
```

第 11 章　在 Python 中定义函数

本节课将补充由洛伦娜巴尔巴教授讲授的在线 CFDPython 课程，**12 步到达纳维——斯托克斯**的第一个交互模块。该交互模块从 1 维简单练习开始，没有使用 Python 的其他功能。我们现在提出了一些新的方法来做同样的事情，这些方法更有效并产生更漂亮的代码。

11.1　在 Python 中定义函数

在步骤 1 到 8 中，我们编写了 Python 代码，这意味着从上到下运行。我们能够重新使用代码（非常有效！通过复制和粘贴，逐步建立伯格斯方程的解算器。但是向前迈进有更有效的方法来编写我们的 Python 代码。在本课中，我们将介绍功能定义这将使我们能够更灵活地重用和组织我们的代码。

我们将从一个简单的例子开始：一个把两个数字相加的函数。

要在 Python 中创建函数，请执行以下操作：

```
def simpleadd(a,b):
```

此语句创建一个称为 simpleadd 的函数，其需要两个输入,a 以及 b。让我们执行此定义代码。

```
def simpleadd(a,b):
    return a+b
```

return 语句告诉 Python 调用后返回的是什么数据。现在我们可以尝试调用我们的 simpleadd 函数：

```
simpleadd(3,7)
7
```

当然，在 def 行和 return 行之间可以发生很多事情。按这种方式，用户可以用模块化的方式创建代码。尝试一个返回斐波那契数列第 n 个数的函数。

```
def fibonacci(n):
    a,b = 0,1
    for i in range(n):
        a,b = b,a+b
    return a
fibonacci(7)
13
```

一旦完成定义，可以像我们以前使用过的 Python 内置函数一样来调用 fibonacci 函数。例如，我们可以通过求第 n 个值的方式打印出 Fibonacci 数列来：

```
for n in range(10):
    print(fibonacci(n)
0
1
1
2
3
5
8
13
21
34
```

我们将使用 Python 自定义函数的功能来帮助我们构建更易复用、更易维护、更易分享的代码！

11.2 了解更多

还记得关于“使用 NumPy 进行数组操作”的捷径吗？

还有更多的方法使你的科学代码更快速地运行。我们推荐在“技术发现博客”上关于加速 python（http://technicaldiscovery.blogspot.com/2011/06/speeding-up-python-numpy-cython-and.html）的文章，它讨论 NumPy，Cython 和 Weave 的使用。该文章使用拉普拉斯方程作为示例（我们将在步骤 9 中求解）并让自定义函数的使用更简洁。

但是最近有一种新的方法 Numba 来获得快速运行的 python 代码。我们在结束 **12 步到达纳维——斯托克斯**后就会学到一点。

在高性能 Python 领域里正发生着许多令人兴奋的事情！

第 12 章　步骤 9

在上一步中，我们解决了 2 维伯格斯方程：一个流体力学研究中的重要方程，因为它包含了流体方程的全对流非线性。在这一练习中，我们还需要积累经验以增加一个纳维——斯托克斯解算器的代码。

在接下来的两个步骤中，我们将求解拉普拉斯和泊松方程。我们会把它们放在一起解决！

12.1　2 维拉普拉斯方程

下面是 2 维的拉普拉斯方程：

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} = 0$$

我们知道如何离散二阶导数。但是考虑到这一点——拉普拉斯方程具有扩散现象的典型特征。因此，必须使用中心差分进行离散，从而使离散结果与我们想要模拟的物理过程一致。离散方程为：

$$\frac{p^n_{i+1,j} - 2p^n_{i,j} + p^n_{i-1,j}}{\Delta x^2} + \frac{p^n_{i,j+1} - 2p^n_{i,j} + p^n_{i,j-1}}{\Delta y^2} = 0$$

请注意拉普拉斯方程没有时间相关性—不存在 p^{n+1}。拉普拉斯方程不是通过时间跟踪波 (如前面的步骤)，而是在所提供的边界条件下计算系统的平衡状态。

如果你完成了热传递过程中课程作业，你就会发现拉普拉斯方程是稳态的热方程。

我们会反复计算 $p^n_{i,j}$ 直到它符合我们指定的条件而不是计算某个时刻 t 系统将处于的状态。只有当迭代次数趋于 ∞ 时系统才会达到平衡，但是我们可以通过迭代到两次迭代之间的变化非常小来逼近平衡状态。

重新排列离散化的方程，求解 $p^n_{i,j}$:

$$p^n_{i,j} = \frac{\Delta y^2(p^n_{i+1,j} + p^n_{i-1,j}) + \Delta x^2(p^n_{i,j+1} + p^n_{i,j-1})}{2(\Delta x^2 + \Delta y^2)}$$

在两个方向上使用二阶中心差分格式是拉普拉斯算子最广泛的应用方法。众多周知的五点差分算子，指的就是这个模版。

我们将通过假设初始状态为处处 $p = 0$ 来数值求解拉普拉斯方程。然后我们添加如下的边界条件：

$$p = 0 \ at \ x = 0$$

$$p = y \ at \ x = 2$$

$$\frac{\partial p}{\partial y} = 0 \ at \ y = 0, \ 1$$

在这些条件下，拉普拉斯方程有一个解析解：

$$p(x,y) = \frac{x}{4} - 4\sum_{n=1,odd}^{\infty} \frac{1}{(n\pi)^2 \sinh 2n\pi} \sinh n\pi x \cos n\pi y$$

练习

按照我们在第一次课程中使用的编码样式，自己编写使用循环来求解泊松方程的代码。然后，考虑如何使用函数（如下）编写代码，并在该样式中修改代码。你能想到放弃旧样式并采用模块化编码的原因吗？

其他提示：

- 可视化迭代过程的每个步骤
- 想想边界条件在做什么
- 想想 PDE 在做什么

12.2 使用函数

还记得**使用 Python 编写函数**课程吗？我们在本练习中将使用其代码格式。

我们将定义两个函数：一个在 3 维投影图中绘制我们的数据，另一个通过多次迭代到 p 的 L1 范数的变化小于一个特定值的方式来求解 p 值。

```
import numpy
from matplotlib import pyplot, cm
from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline

def plot2D(x, y, p):
    fig = pyplot.figure(figsize=(11, 7), dpi=100)
    ax = fig.gca(projection='3d')

    X, Y = numpy.meshgrid(x, y)
    surf = ax.plot_surface(X, Y, p[:], rstride=1, cstride=1,
            cmap=cm.viridis, linewidth=0, antialiased=False)

    ax.set_xlim(0, 2)
    ax.set_ylim(0, 1)
    ax.view_init(30, 225)
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    pyplot.show()
```

plot2D 函数有 3 个参数，一个 x 向量，一个 y 向量与我们的 p 矩阵。给出 3 个值，该函数创建一个 3 维投影图，设置了绘图极限，并给出了一个良好的视角。

$$p_{i,j}^n = \frac{\Delta y^2(p_{i+1,j}^n + p_{i-1,j}^n) + \Delta x^2(p_{i,j+1}^n + p_{i,j-1}^n)}{2(\Delta x^2 + \Delta y^2)}$$

```
    def laplace2d(p, y, dx, dy, l1norm_target):
        l1norm = 1
        pn = numpy.empty_like(p)

        while l1norm > l1norm_target:
            pn = p.copy()
            p[1:-1, 1:-1] = ((dy**2 * (pn[1:-1, 2:] + pn[1:-1, 0:-2]) +
                              dx**2 * (pn[2:, 1:-1] + pn[0:-2, 1:-1])) /
                             (2 * (dx**2 + dy**2)))

            p[:, 0] = 0  # p = 0 @ x = 0
            p[:, -1] = y  # p = y @ x = 2
            p[0, :] = p[1, :]  # dp/dy = 0 @ y = 0
            p[-1, :] = p[-2, :]  # dp/dy = 0 @ y = 1
            l1norm = (numpy.sum(numpy.abs(p[:]) - numpy.abs(pn[:])) /
                      numpy.sum(numpy.abs(pn[:])))
        return p
```

laplace2d 函数有 5 个参数，p 矩阵，y 向量，dx，dy 与 l1norm_target 值。最后一个值定义了在循环终止并返回计算的 p 值之前的两次连续迭代中间距离 p 矩阵有多远。

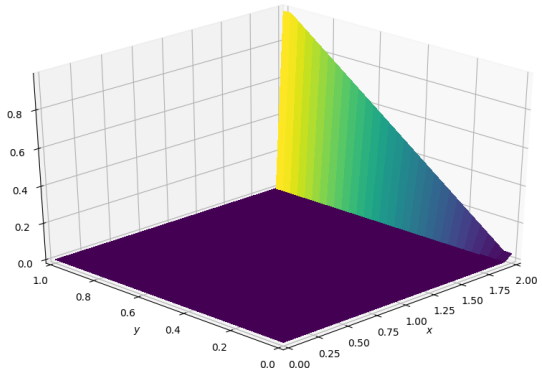
请注意，当在自己的笔记本中执行上面的代码单元时，没有输出。你已定义了函数但你还没有调用函数。现在你可以使用如下 numpy.linspace 函数或在我们的命名空间中的任何其他函数。

```
    # 声明变量
    nx = 31
    ny = 31
    c = 1
    dx = 2 / (nx - 1)
    dy = 2 / (ny - 1)

    # 初始状态
    p = numpy.zeros((ny, nx))  #创建一个值均为1的1xn的向量

    # 绘制点集
    x = numpy.linspace(0, 2, nx)
    y = numpy.linspace(0, 1, ny)

    # 边界条件
    p[:, 0] = 0  # p = 0 @ x = 0
    p[:, -1] = y  # p = y @ x = 2
    p[0, :] = p[1, :]  # dp/dy = 0 @ y = 0
    p[-1, :] = p[-2, :]  # dp/dy = 0 @ y = 1
```

现在让我们试着用我们的 plot2D 函数来查看我们的初始状态。如果函数已正确定义，你应该能

够开始键入 plot2D 并且使用 tab 键来自动补全。

```
plot2D(x,y,p)
```



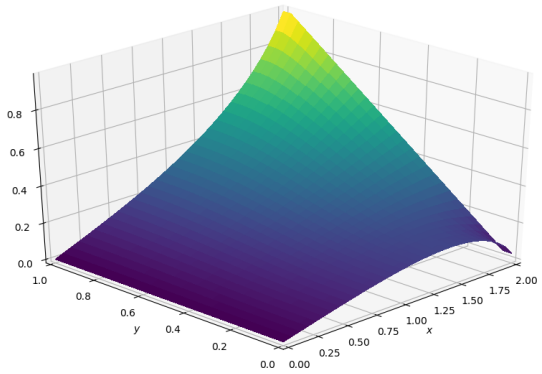
它奏效了！这是我们问题的初始状态，其中除了沿着 $x = 2$ 方向时 $p = y$ 之外，p 值均为 0。现在让我们试着运行具有指定 L1 目标的 laplace2d 函数，l1norm_target=1e-4。

[提示：如果你没有记住传入函数的参数顺序，你可以只键入 laplace2d(然后 jupyter Notebook 会弹出一个小的弹出框来提醒你]

```
p = laplace2d(p, y, dx, dy, 1e-4)
```

现在现在试着使用我们的绘图功能绘制 p 的这个新值。

```
plot2D(x,y,p)
```



12.3 了解更多

下一步会求解泊松方程。为了理解为什么在 CFD 中我们需要泊松方程，请在 YouTube 上观看**视频课程 11**。

```
from IPython.display import YouTubeVideo
YouTubeVideo('ZjfxA3qq2Lg')
```

第 13 章　步骤 10

现在，回忆不可压缩流体的纳维——斯托克斯方程，其中 $\vec{v}$ 表示速度字段：

$$\nabla \cdot \vec{v} = 0$$

$$\frac{\partial \vec{v}}{\partial t} + (\vec{v} \cdot \nabla)\vec{v} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \vec{v}$$

第一个方程表示恒定密度下的质量守恒。第二个方程是动量守恒方程。但是出现一个问题：不可压缩流的连续性方程不具有主导变量，并且没有明显的方法来耦合速度和压力。相反，在可压缩流动的情况下，质量连续性将提供密度 ρ 的演化方程，其耦合到与 ρ 和 p 相关的状态方程。

在不可压缩流中, 连续性方程 $\nabla \cdot \vec{v} = 0$ 提供了一个要求压力场演化以使得膨胀速率 $\nabla \cdot \vec{v}$ 处处都应该消失的**运动约束**。摆脱这种难题的方法是建造满足连续性的压力场；这种关系可以通过求动量方程的散度来获得。在这个过程中，压力的泊松方程出现了！

13.1　2 维泊松方程

泊松方程是通过向拉普拉斯方程的右侧添加源项而获得的：

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} = b$$

所以，不同于拉普拉斯方程，在影响解的场中存在一些有限值。在这样的场中泊松方程起到松弛初始源的作用。

在离散形式下，除了源项外，其几乎与步骤 9 一样：

$$\frac{p^n_{i+1,j} - 2p^n_{i,j} + p^n_{i-1,j}}{\Delta x^2} + \frac{p^n_{i,j+1} - 2p^n_{i,j} + p^n_{i,j-1}}{\Delta y^2} = b^n_{i,j}$$

与之前一样，我们对齐重新安排，以便我们得到一个在点 i, j 的关于 p 的方程。因此，我们获得：

$$p^n_{i,j} = \frac{(p^n_{i+1,j} + p^n_{i-1,j})\Delta y^2 + (p^n_{i,j+1} + p^n_{i,j-1})\Delta x^2 - b^n_{i,j}\Delta x^2 \Delta y^2}{2(\Delta x^2 + \Delta y^2)}$$

我们将通过假设初始状态为 处处$p = 0$ 来求解该方程。并应用如下边界条件：

$$在x = 0, 2及y = 0, 1时\ p = 0$$

源项包括域内的两个初始尖峰，如下所示：

$$\begin{aligned} 当i = nx/4, j = ny/4时 \qquad & b_{i,j} = 100 \\ 当i = 3 * nx/4, j = 3 * ny/4时 \qquad & b_{i,j} = -100 \\ 其他情况 \qquad & b_{i,j} = 0 \end{aligned}$$

在伪时间中推进迭代来松驰初始尖峰。泊松方程中的松弛随着迭代的进行而变得越来越慢。为什么？

让我们寻找一个编写泊松方程的代码可行方法来。一如既往地，我们加载我们最喜爱的 Python 库。我们还想在 3D 制作一些可爱的图形。让我们解决参数的定义和初始化问题。在下面的方法中你注意到了什么？

```
import numpy
from matplotlib import pyplot, cm
from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline
# 参数
nx = 50
ny = 50
nt  = 100
xmin = 0
xmax = 2
ymin = 0
ymax = 1

dx = (xmax - xmin) / (nx - 1)
dy = (ymax - ymin) / (ny - 1)
# 初始化
p  = numpy.zeros((ny, nx))
pd = numpy.zeros((ny, nx))
b  = numpy.zeros((ny, nx))
x  = numpy.linspace(xmin, xmax, nx)
y  = numpy.linspace(xmin, xmax, ny)
# 源
b[int(ny / 4), int(nx / 4)]  = 100
b[int(3 * ny / 4), int(3 * nx / 4)] = -100
```

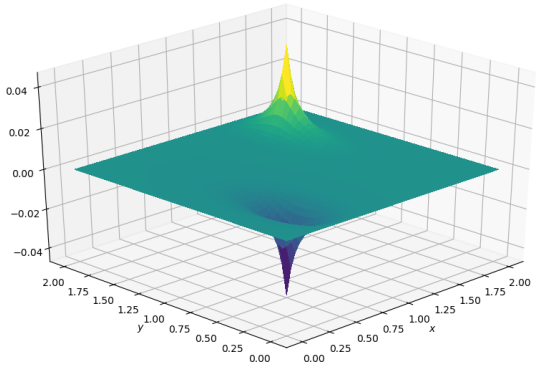
接着，我们准备在伪时间里进行初始预测。下面的代码与步骤 9 中使用的求解拉普拉斯方程的函数有什么样的不同？

```
for it in range(nt):
    pd = p.copy()
    p[1:-1,1:-1] = (((pd[1:-1, 2:] + pd[1:-1, :-2]) * dy**2 +
                    (pd[2:, 1:-1] + pd[:-2, 1:-1]) * dx**2 -
                    b[1:-1, 1:-1] * dx**2 * dy**2) /
                    (2 * (dx**2 + dy**2)))
    p[0, :] = 0
    p[ny-1, :] = 0
    p[:, 0] = 0
    p[:, nx-1] = 0
```

也许我们可以从步骤 9 中复用我们的绘图函数，你不觉得吗？

```
def plot2D(x, y, p):
    fig = pyplot.figure(figsize=(11, 7), dpi=100)
    ax = fig.gca(projection='3d')
    X, Y = numpy.meshgrid(x, y)
    surf = ax.plot_surface(X, Y, p[:], rstride=1, cstride=1, cmap=cm.viridis,
            linewidth=0, antialiased=False)
    ax.view_init(30, 225)
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    pyplot.show()
```

```
plot2D(x,y,p)
```



啊！代码复用的奇迹！现在，你可能会想：“嗯，如果我写的这个简单小函数如此有用的话，我要一遍又一遍地使用它“。如何在每次不用复制和粘贴做到这一点呢?——如果你对这个很好奇，你就必须学会了包装。但这超出了我们的 CFD 课程的范围。如果你真的想知道的话，那么你不得不使用谷歌搜索了。

13.2 了解更多

要了解更多有关 Poisson 方程在 CFD 中的作用，请观看 YouTube 上的**视频课程 11**。

```
from IPython.display import YouTubeVideo
YouTubeVideo('ZjfxA3qq2Lg')
```

第 14 章　用 Numba 优化循环

本笔记本补充了交互式 CFD 在线模块 **12 步到达纳维——斯托克斯**用 Python 解决高性能问题。

14.1　用 Numba 优化循环

回忆我们使用 NumPy 进行数组操作的内容，当使用 NumPy 优化的数组操作来代替多层嵌套的循环以实现我们的离散化时运行速度得到了很大的增加。

Numba 是一种工具，它提供了另一种优化 Python 代码的方法。Numba 是 Python 库，它使用 LLVM 将 Python 函数转化为 C 风格编译函数。根据原始代码和问题的大小，Numba 可以在 NumPy 优化代码上提供显著的加速。

让我们重新审视 2 维拉普拉斯方程：

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib import pyplot
import numpy
# 变量声明
nx = 81
ny = 81
c = 1
dx = 2.0/(nx-1)
dy = 2.0/(ny-1)
# 初始状态
p = numpy.zeros((ny,nx)) # create a XxY vector of 0's
# plotting aids
x = numpy.linspace(0,2,nx)
y = numpy.linspace(0,1,ny)
# 边界条件
p[:,0] = 0      # p = 0 @ x = 0
p[:,-1] = y     # p = y @ x = 2
p[0,:] = p[1,:]     # dp/dy = 0 @ y = 0
p[-1,:] = p[-2,:]   # dp/dy = 0 @ y = 1
```

下面是我们在步骤 9 中编写的用于迭代拉普拉斯方程的函数：

```
def laplace2d(p, y, dx, dy, l1norm_target):
```

```
    l1norm = 1
    pn = numpy.empty_like(p)

    while l1norm > l1norm_target:
        pn = p.copy()
        p[1:-1,1:-1] = (dy**2*(pn[2:,1:-1]+pn[0:-2,1:-1])+dx**2*(pn[1:-1,2:]
                        +pn[1:-1,0:-2]))/(2*(dx**2+dy**2))
        p[0,0] = (dy**2*(pn[1,0]+pn[-1,0])+dx**2*(pn[0,1]+pn[0,-1]))
                  /(2*(dx**2+dy**2))
        p[-1,-1] = (dy**2*(pn[0,-1]+pn[-2,-1])+dx**2*(pn[-1,0]+pn[-1,-2]))
                    /(2*(dx**2+dy**2))

        p[:,0] = 0      # p = 0 @ x = 0
        p[:,-1] = y     # p = y @ x = 2
        p[0,:] = p[1,:]     # dp/dy = 0 @ y = 0
        p[-1,:] = p[-2,:]   # dp/dy = 0 @ y = 1
        l1norm = (numpy.sum(np.abs(p[:])-np.abs(pn[:])))
                  /np.sum(np.abs(pn[:]))
    return p
```

让我们用%%timeit单元魔法命令了解它的运行速度：

```
%%timeit
laplace2d(p, y, dx, dy, .00001)
1 loops, best of 3: 206 us per loop
```

好！我们的函数 laplace2d 完成运行大约需要 206 微秒。这非常快，感谢数组运算。让我们看看“普通”Python 版本需要多长时间。

```
def laplace2d_vanilla(p, y, dx, dy, l1norm_target):
    l1norm = 1
    pn = numpy.empty_like(p)
    nx, ny = len(y), len(y)

    while l1norm > l1norm_target:
        pn = p.copy()
        for i in range(1, nx-1):
            for j in range(1, ny-1):
                p[i,j] = (dy**2*(pn[i+1,j]+pn[i-1,j])+dx**2
                          *(pn[i,j+1]-pn[i,j-1]))/(2*(dx**2+dy**2))

        p[0,0] = (dy**2*(pn[1,0]+pn[-1,0])+dx**2*(pn[0,1]+pn[0,-1]))
                  /(2*(dx**2+dy**2))
        p[-1,-1] = (dy**2*(pn[0,-1]+pn[-2,-1])+dx**2
                    *(pn[-1,0]+pn[-1,-2]))/(2*(dx**2+dy**2))
```

```
        p[:,0] = 0      # p = 0 @ x = 0
        p[:,-1] = y     # p = y @ x = 2
        p[0,:] = p[1,:]     # dp/dy = 0 @ y = 0
        p[-1,:] = p[-2,:]   # dp/dy = 0 @ y = 1
        l1norm = (numpy.sum(np.abs(p[:])-np.abs(pn[:])))
                 /np.sum(np.abs(pn[:]))
    return p
```

```
%%timeit
laplace2d_vanilla(p, y, dx, dy, .00001)
10 loops, best of 3: 32 ms per loop
```

简单的 Python 版本运行完成需要 32 毫秒。让我们计算在使用数组运算中获得的加速：

```
32*1e-3/(206*1e-6)
155.33980582524273
```

因此 Numpy 使我们比常规 Python 代码有 155 倍的速度增长。也就是说，有时在数组运算中实现我们的离散化可能有点棘手。

让我们看看 Numba 可以做什么。我们将通过从 numba 库中导入特殊函数修饰符 autojit 来开始：

```
from numba import autojit
```

为了把 Numba 与我们的现有函数集成起来，我们所需要做的是将`@autojit`函数修饰符放置在我们的`def`语句前：

```
@autojit
def laplace2d_numba(p, y, dx, dy, l1norm_target):
    l1norm = 1
    pn = numpy.empty_like(p)

    while l1norm > l1norm_target:
        pn = p.copy()
        p[1:-1,1:-1] = (dy**2*(pn[2:,1:-1]+pn[0:-2,1:-1])
                       +dx**2*(pn[1:-1,2:]+pn[1:-1,0:-2]))/(2*(dx**2+dy**2))
        p[0,0] = (dy**2*(pn[1,0]+pn[-1,0])+dx**2*(pn[0,1]
                 +pn[0,-1]))/(2*(dx**2+dy**2))
        p[-1,-1] = (dy**2*(pn[0,-1]+pn[-2,-1])+dx**2
                   *(pn[-1,0]+pn[-1,-2]))/(2*(dx**2+dy**2))

        p[:,0] = 0      # p = 0 @ x = 0
        p[:,-1] = y     # p = y @ x = 2
        p[0,:] = p[1,:]     # dp/dy = 0 @ y = 0
        p[-1,:] = p[-2,:]   # dp/dy = 0 @ y = 1
        l1norm = (numpy.sum(np.abs(p[:])-np.abs(pn[:])))
```

```
                    /np.sum(np.abs(pn[:]))
        return p
```

唯一改变的是@autojit行和更改的函数名以便于我们比较性能。现在让我们看看会发生什么？

```
%%timeit
laplace2d_numba(p, y, dx, dy, .00001)
1 loops, best of 3: 137 us per loop
```

好！虽然它不是我们在普通 Python 和 Numpy 之间看到一个 155 倍的速度增长，但是它是性能时间的一个非平凡的增益，特别是考虑到它是多么容易实现。Numba 的另一个很酷的特点是您也可以非数组运算函数上使用@autojit修饰符。让我们尝试把它添加到我们的普通版本上：

```
@autojit
def laplace2d_vanilla_numba(p, y, dx, dy, l1norm_target):
    l1norm = 1
    pn = numpy.empty_like(p)
    nx, ny = len(y), len(y)

    while l1norm > l1norm_target:
        pn = p.copy()
        for i in range(1, nx-1):
            for j in range(1, ny-1):
                p[i,j] = (dy**2*(pn[i+1,j]+pn[i-1,j])+dx**2
                          *(pn[i,j+1]-pn[i,j-1]))/(2*(dx**2+dy**2))

        p[0,0] = (dy**2*(pn[1,0]+pn[-1,0])+dx**2*(pn[0,1]+pn[0,-1]))
                  /(2*(dx**2+dy**2))
        p[-1,-1] = (dy**2*(pn[0,-1]+pn[-2,-1])+dx**2*(pn[-1,0]
                    +pn[-1,-2]))/(2*(dx**2+dy**2))

        p[:,0] = 0      # p = 0 @ x = 0
        p[:,-1] = y     # p = y @ x = 2
        p[0,:] = p[1,:]     # dp/dy = 0 @ y = 0
        p[-1,:] = p[-2,:]   # dp/dy = 0 @ y = 1
        l1norm = (numpy.sum(np.abs(p[:])-np.abs(pn[:])))
                  /np.sum(np.abs(pn[:]))
    return p
```

```
%%timeit
laplace2d_vanilla_numba(p, y, dx, dy, .00001)
1 loops, best of 3: 561 us per loop
```

561 微秒。不是我们看到的使用 Numpy 带来的 155 倍的增加，但是它很接近。我们所做的只是增加一行代码。

所以我们有：

- 普通 Python：32 毫秒
- Numpy Python：206 微秒
- 普通 Python+Numba：561 微秒
- NumPy+Numba：137 微秒

显然，Numpy+Numba 的组合是最快的，但是快速优化带有嵌套循环代码的能力也可以非常方便地用于在某些应用中。

第 15 章　步骤 11

在讲授如何使用 Python 开始 CFD 的交互模块的最后两步中，将会求解具有不同边界条件的 2 维纳维——斯托克斯方程。

用向量形式表示的速度场 $\vec{v}$ 中的动量方程为：

$$\frac{\partial \vec{v}}{\partial t} + (\vec{v} \cdot \nabla)\vec{v} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \vec{v}$$

这表示三个标量方程，每个速度分量 $(u,\ v,\ w)$ 一个。但是我们将在两个维度中求解它，因此将有两个标量方程。

请记住连续方程？这就是压力泊松方程的由来！

15.1　空腔流体的纳维——斯托克斯方程

一个由两个速度分量 u, v 的微分方程和一个压力的微分方程构成的体系:

$$\begin{aligned}
\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} &= -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \\
\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} &= -\frac{1}{\rho}\frac{\partial p}{\partial y} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \\
\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} &= -\rho\left(\frac{\partial u}{\partial x}\frac{\partial u}{\partial x} + 2\frac{\partial u}{\partial y}\frac{\partial v}{\partial x} + \frac{\partial v}{\partial y}\frac{\partial v}{\partial y}\right)
\end{aligned}$$

从前面的步骤，我们已经知道如何对所有微分项进行离散。只有最后一个方程式有点不熟悉。但有一点耐心，那就不难了！

15.2　离散方程组

首先，离散 u 方向的动量方程，如下：

$$\begin{aligned}
&\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} + u_{i,j}^{n}\frac{u_{i,j}^{n} - u_{i-1,j}^{n}}{\Delta x} + v_{i,j}^{n}\frac{u_{i,j}^{n} - u_{i,j-1}^{n}}{\Delta y} \\
&= -\frac{1}{\rho}\frac{p_{i+1,j}^{n} - p_{i-1,j}^{n}}{2\Delta x} + \nu\left(\frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{\Delta x^2} + \frac{u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}}{\Delta y^2}\right)
\end{aligned}$$

类似的 v 方向的动量方程为：

$$\begin{aligned}
&\frac{v_{i,j}^{n+1} - v_{i,j}^{n}}{\Delta t} + v_{i,j}^{n}\frac{v_{i,j}^{n} - v_{i-1,j}^{n}}{\Delta x} + v_{i,j}^{n}\frac{v_{i,j}^{n} - v_{i,j-1}^{n}}{\Delta y} \\
&= -\frac{1}{\rho}\frac{p_{i+1,j}^{n} - p_{i-1,j}^{n}}{2\Delta x} + \nu\left(\frac{v_{i+1,j}^{n} - 2v_{i,j}^{n} + v_{i-1,j}^{n}}{\Delta x^2} + \frac{v_{i,j+1}^{n} - 2v_{i,j}^{n} + v_{i,j-1}^{n}}{\Delta y^2}\right)
\end{aligned}$$

最终，离散的压力泊松方程可以写成如下形式：

$$\frac{p^n_{i+1,j}-2p^n_{i,j}+p^n_{i-1,j}}{\Delta x^2}+\frac{p^n_{i,j+1}-2p^n_{i,j}+p^n_{i,j-1}}{\Delta y^2}=\rho\left[\frac{1}{\Delta t}\left(\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}+\frac{v_{i,j+1}-v_{i,j-1}}{2\Delta y}\right)\right.$$
$$\left.-\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}-2\frac{u_{i,j+1}-u_{i,j-1}}{2\Delta y}\frac{v_{i+1,j}-v_{i-1,j}}{2\Delta x}-\frac{v_{i,j+1}-v_{i,j-1}}{2\Delta y}\frac{v_{i,j+1}-v_{i,j-1}}{2\Delta y}\right]$$

你应该手工把这些方程式写在你自己的笔记上，在你写下它们时应该理解其中每一项的含义。

如前所述，让我们重新排列公式，以便在代码中进行迭代。首先，是下一个时间步长中与速度有关的动量方程。

在 u 方向的动量方程：

$$u^{n+1}_{i,j}=u^n_{i,j}-u^n_{i,j}\frac{\Delta t}{\Delta x}(u^n_{i,j}-u^n_{i-1,j})-v^n_{i,j}\frac{\Delta t}{\Delta y}(u^n_{i,j}-u^n_{i,j-1})-\frac{\Delta t}{\rho 2\Delta x}(p^n_{i+1,j}-p^n_{i-1,j})$$
$$+\nu\left(\frac{\Delta t}{\Delta x^2}(u^n_{i+1,j}-2u^n_{i,j}+u^n_{i-1,j})+\frac{\Delta t}{\Delta y^2}(u^n_{i,j+1}-2u^n_{i,j}+u^n_{i,j-1})\right)$$

在 v 方向的动量方程：

$$v^{n+1}_{i,j}=v^n_{i,j}-u^n_{i,j}\frac{\Delta t}{\Delta x}(v^n_{i,j}-v^n_{i-1,j})-v^n_{i,j}\frac{\Delta t}{\Delta y}(v^n_{i,j}-v^n_{i,j-1})-\frac{\Delta t}{\rho 2\Delta x}(p^n_{i+1,j}-p^n_{i-1,j})$$
$$+\nu\left(\frac{\Delta t}{\Delta x^2}(v^n_{i+1,j}-2v^n_{i,j}+v^n_{i-1,j})+\frac{\Delta t}{\Delta y^2}(v^n_{i,j+1}-2v^n_{i,j}+v^n_{i,j-1})\right)$$

快到了！现在，我们重新排列压力 -泊松方程：

$$p^n_{i,j}=\frac{(p^n_{i+1,j}+p^n_{i-1,j})\Delta y^2+(p^n_{i,j+1}+p^n_{i,j-1})\Delta x^2}{2(\Delta x^2+\Delta y^2)}-\frac{\rho\Delta x^2\Delta y^2}{2(\Delta x^2+\Delta y^2)}\times$$
$$\left[\frac{1}{\Delta t}\left(\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}+\frac{v_{i,j+1}-v_{i,j-1}}{2\Delta y}\right)-\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}\right.$$
$$\left.-2\frac{u_{i,j+1}-u_{i,j-1}}{2\Delta y}\frac{v_{i+1,j}-v_{i-1,j}}{2\Delta x}-\frac{v_{i,j+1}-v_{i,j-1}}{2\Delta y}\frac{v_{i,j+1}-v_{i,j-1}}{2\Delta y}\right]$$

初始状态为 处处$u,\ v,\ p=0$，边界条件为：

$$\begin{aligned}
u&=1 && \text{当 y=2 时（盖子处）；}\\
u,v&=0 && \text{在其他边界处；}\\
\frac{\partial p}{\partial x}&=0 && \text{当 y=0 时；}\\
p&=0 && \text{当 y=2 时；}\\
\frac{\partial p}{\partial x}&=0 && \text{当 x=0, 2 时；}
\end{aligned}$$

15.3 实现空腔流体

```
import numpy
from matplotlib import pyplot, cm
from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline
```

```
nx = 41
ny = 41
nit = 50
c = 1
dx = 2 / (nx - 1)
dy = 2 / (ny - 1)
x = numpy.linspace(0, 2, nx)
y = numpy.linspace(0, 2, ny)
X, Y = numpy.meshgrid(x, y)

rho = 1
nu = .1
dt = .001
```

上述的压力泊松方程很难写出完全正确的。下面的函数 build_up_b 代表方程中方括号中的内容，这样使整个 PPE 稍微易于管理些。

```
def build_up_b(b, rho, dt, u, v, dx, dy):
    b[1:-1, 1:-1] = (rho * (1 / dt * ((u[1:-1, 2:] - u[1:-1, 0:-2]) /
                    (2 * dx) + (v[2:, 1:-1] - v[0:-2, 1:-1]) / (2 * dy))
                    - ((u[1:-1, 2:] - u[1:-1, 0:-2]) / (2 * dx))**2 - 2 *
                    ((u[2:, 1:-1] - u[0:-2, 1:-1]) / (2 * dy) * (v[1:-1, 2:]
                    - v[1:-1, 0:-2]) / (2 * dx)) - ((v[2:, 1:-1] -
                    v[0:-2, 1:-1]) / (2 * dy))**2))
    return b
```

也定义函数 pressure_poisson 帮助分离不同轮的计算。注意伪时间变量 nit 的存在。Poisson 计算中的子迭代有助于确保无发散场。

```
def pressure_poisson(p, dx, dy, b):
    pn = numpy.empty_like(p)
    pn = p.copy()

    for q in range(nit):
        pn = p.copy()
        p[1:-1, 1:-1] = (((pn[1:-1, 2:] + pn[1:-1, 0:-2]) * dy**2 +
                        (pn[2:, 1:-1] + pn[0:-2, 1:-1]) * dx**2) /
                        (2 * (dx**2 + dy**2)) - dx**2 * dy**2 /
                        (2 * (dx**2 + dy**2)) * b[1:-1,1:-1])

        p[:, -1] = p[:, -2] # dp/dy = 0 当x = 2时
        p[0, :] = p[1, :]  # dp/dy = 0 当y = 0时
        p[:, 0] = p[:, 1]    # dp/dx = 0 当x = 0时
        p[-1, :] = 0         # p = 0 当y = 2时
```

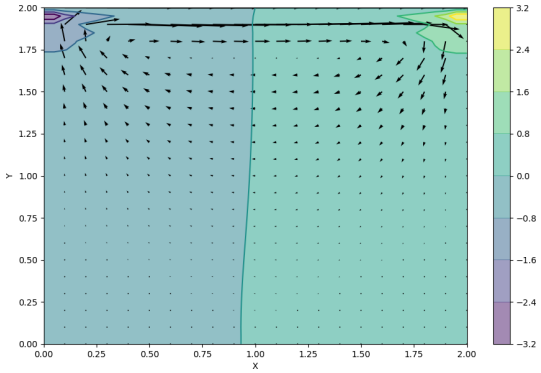
```
    return p
```

最后将其余的空腔流体方程包裹在 cavity_flow 函数内，这样我们更容易绘制不同时间长度的空腔流体求解的结果。

```
def cavity_flow(nt, u, v, dt, dx, dy, p, rho, nu):
    un = numpy.empty_like(u)
    vn = numpy.empty_like(v)
    b = numpy.zeros((ny, nx))

    for n in range(nt):
        un = u.copy()
        vn = v.copy()

        b = build_up_b(b, rho, dt, u, v, dx, dy)
        p = pressure_poisson(p, dx, dy, b)

        u[1:-1, 1:-1] = (un[1:-1, 1:-1] - un[1:-1, 1:-1] * dt / dx *
                         (un[1:-1, 1:-1] - un[1:-1, 0:-2]) -
                          vn[1:-1, 1:-1] * dt / dy *
                         (un[1:-1, 1:-1] - un[0:-2, 1:-1]) - dt /
                         (2 * rho * dx) * (p[1:-1, 2:] - p[1:-1, 0:-2])
                         + nu * (dt / dx**2 * (un[1:-1, 2:] - 2 *
                         un[1:-1, 1:-1] + un[1:-1, 0:-2]) + dt / dy**2 *
                         (un[2:,1:-1] - 2 * un[1:-1,1:-1] + un[0:-2,1:-1])))

        v[1:-1,1:-1] = (vn[1:-1, 1:-1] - un[1:-1, 1:-1] * dt / dx *
                        (vn[1:-1, 1:-1] - vn[1:-1, 0:-2]) -
                         vn[1:-1, 1:-1] * dt / dy *
                        (vn[1:-1, 1:-1] - vn[0:-2, 1:-1]) - dt /
                        (2 * rho * dy) * (p[2:, 1:-1] - p[0:-2, 1:-1])
                        + nu * (dt / dx**2 * (vn[1:-1, 2:] - 2 *
                        vn[1:-1, 1:-1] + vn[1:-1, 0:-2]) + dt / dy**2 *
                        (vn[2:,1:-1] - 2 * vn[1:-1,1:-1] + vn[0:-2,1:-1])))
        u[0, :] = 0
        u[:, 0] = 0
        u[:, -1] = 0
        u[-1, :] = 1    #在空腔盖子处令速度等于1。
        v[0, :] = 0
        v[-1, :] = 0
        v[:, 0] = 0
        v[:, -1] = 0

    return u, v, p
```

让我们从 nt=100 开始，看看解算器给出了什么：

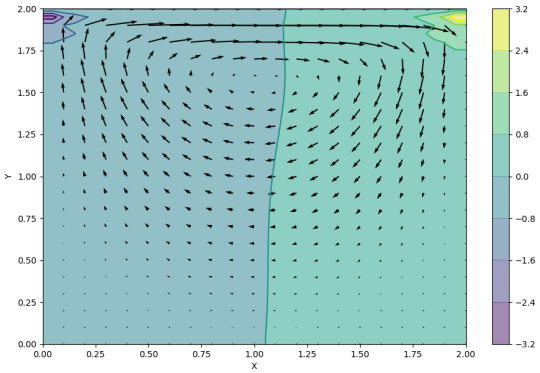
```
u = numpy.zeros((ny, nx))
v = numpy.zeros((ny, nx))
p = numpy.zeros((ny, nx))
b = numpy.zeros((ny, nx))
nt = 100
u, v, p = cavity_flow(nt, u, v, dt, dx, dy, p, rho, nu)
fig = pyplot.figure(figsize=(11,7), dpi=100)
# 以等值线的方式绘制压力场
pyplot.contourf(X, Y, p, alpha=0.5, cmap=cm.viridis)
pyplot.colorbar()
# plotting the pressure field outlines
pyplot.contour(X, Y, p, cmap=cm.viridis)
# 绘制速度场的轮廓线
pyplot.quiver(X[::2, ::2], Y[::2, ::2], u[::2, ::2], v[::2, ::2])
pyplot.xlabel('X')
pyplot.ylabel('Y')
pyplot.show()
```



您可以看到两个不同的压力区正在形成，并且根据盖驱动的空腔流所预期的螺旋图案开始形成。实验不同值的 nt 来看看系统要多长时间能够达到稳定。

```
u = numpy.zeros((ny, nx))
v = numpy.zeros((ny, nx))
p = numpy.zeros((ny, nx))
b = numpy.zeros((ny, nx))
nt = 700
u, v, p = cavity_flow(nt, u, v, dt, dx, dy, p, rho, nu)
fig = pyplot.figure(figsize=(11, 7), dpi=100)
pyplot.contourf(X, Y, p, alpha=0.5, cmap=cm.viridis)
pyplot.colorbar()
pyplot.contour(X, Y, p, cmap=cm.viridis)
pyplot.quiver(X[::2, ::2], Y[::2, ::2], u[::2, ::2], v[::2, ::2])
pyplot.xlabel('X')
pyplot.ylabel('Y')
```

```
    pyplot.show()
```



15.4 了解更多

本交互模块 **12 步到达纳维——斯托克斯**是洛伦娜 A. 巴尔巴教授从 2009 年至 2013 年在波士顿大学讲授的计算流体力学课的几个组成部分之一。

至于本课程的其他组件，你可以访问本课程 2013 春季版的资源网站（https://piazza.com/bu/spring2013/me702/resources）。

第 16 章　步骤 12

你走到这一步了吗？这是最后一步！学完此交互模块之后, 你用 Python 编写自己的纳维——斯托克斯解算器需要多长时间？让我们知道！

16.1　管道流体的纳维——斯托克斯方程

在最终一步与步骤 11 之间的差别是我们要在动量 u 方程添加一个源项，模拟压力驱动管道流体的效果。下面是我们修改的纳维——斯托克斯方程：

$$\frac{\partial u}{\partial t}+u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}=-\frac{1}{\rho}\frac{\partial p}{\partial x}+\nu\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+F$$
$$\frac{\partial v}{\partial t}+u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}=-\frac{1}{\rho}\frac{\partial p}{\partial y}+\nu\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}\right)$$
$$\frac{\partial^2 p}{\partial x^2}+\frac{\partial^2 p}{\partial y^2}=-\rho\left(\frac{\partial u}{\partial x}\frac{\partial u}{\partial x}+2\frac{\partial u}{\partial y}\frac{\partial v}{\partial x}+\frac{\partial v}{\partial y}\frac{\partial v}{\partial y}\right)$$

16.2　离散方程组

我们耐心并小心地写出了方程的离散形式。强烈建议你手工写这些，在你写下它们时应该理解其中每一项的含义。

u 方向的动量方程：

$$\frac{u_{i,j}^{n+1}-u_{i,j}^{n}}{\Delta t}+u_{i,j}^{n}\frac{u_{i,j}^{n}-u_{i-1,j}^{n}}{\Delta x}+v_{i,j}^{n}\frac{u_{i,j}^{n}-u_{i,j-1}^{n}}{\Delta y}$$
$$=-\frac{1}{\rho}\frac{p_{i+1,j}^{n}-p_{i-1,j}^{n}}{2\Delta x}+\nu\left(\frac{u_{i+1,j}^{n}-2u_{i,j}^{n}+u_{i-1,j}^{n}}{\Delta x^2}+\frac{u_{i,j+1}^{n}-2u_{i,j}^{n}+u_{i,j-1}^{n}}{\Delta y^2}\right)+F_{i,j}$$

v 方向的动量方程为：

$$\frac{v_{i,j}^{n+1}-v_{i,j}^{n}}{\Delta t}+v_{i,j}^{n}\frac{v_{i,j}^{n}-v_{i-1,j}^{n}}{\Delta x}+v_{i,j}^{n}\frac{v_{i,j}^{n}-v_{i,j-1}^{n}}{\Delta y}$$
$$=-\frac{1}{\rho}\frac{p_{i+1,j}^{n}-p_{i-1,j}^{n}}{2\Delta x}+\nu\left(\frac{v_{i+1,j}^{n}-2v_{i,j}^{n}+v_{i-1,j}^{n}}{\Delta x^2}+\frac{v_{i,j+1}^{n}-2v_{i,j}^{n}+v_{i,j-1}^{n}}{\Delta y^2}\right)$$

压力方程为：

$$\frac{p_{i+1,j}^{n}-2p_{i,j}^{n}+p_{i-1,j}^{n}}{\Delta x^2}+\frac{p_{i,j+1}^{n}-2p_{i,j}^{n}+p_{i,j-1}^{n}}{\Delta y^2}=\rho\left[\frac{1}{\Delta t}\left(\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}+\frac{v_{i,j+1}-v_{i,j-1}}{2\Delta y}\right)\right.$$
$$\left.-\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}-2\frac{u_{i,j+1}-u_{i,j-1}}{2\Delta y}\frac{v_{i+1,j}-v_{i-1,j}}{2\Delta x}-\frac{v_{i,j+1}-v_{i,j-1}}{2\Delta y}\frac{v_{i,j+1}-v_{i,j-1}}{2\Delta y}\right]$$

像往常一样，我们需要将这些方程重新排列到代码中需要的形式以便进行迭代过程。

对于 u和v 方向的动量方程，我们导出在时间步长 n+1 时的速度：

$$u_{i,j}^{n+1} = u_{i,j}^n - u_{i,j}^n \frac{\Delta t}{\Delta x}(u_{i,j}^n - u_{i-1,j}^n) - v_{i,j}^n \frac{\Delta t}{\Delta y}(u_{i,j}^n - u_{i,j-1}^n) - \frac{\Delta t}{\rho 2\Delta x}(p_{i+1,j}^n - p_{i-1,j}^n) + \nu\left(\frac{\Delta t}{\Delta x^2}(u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n) + \frac{\Delta t}{\Delta y^2}(u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n)\right) + F\Delta t$$

$$v_{i,j}^{n+1} = v_{i,j}^n - u_{i,j}^n \frac{\Delta t}{\Delta x}(v_{i,j}^n - v_{i-1,j}^n) - v_{i,j}^n \frac{\Delta t}{\Delta y}(v_{i,j}^n - v_{i,j-1}^n) - \frac{\Delta t}{\rho 2\Delta x}(p_{i+1,j}^n - p_{i-1,j}^n) + \nu\left(\frac{\Delta t}{\Delta x^2}(v_{i+1,j}^n - 2v_{i,j}^n + v_{i-1,j}^n) + \frac{\Delta t}{\Delta y^2}(v_{i,j+1}^n - 2v_{i,j}^n + v_{i,j-1}^n)\right)$$

对于压力方程，我们导出 $p_{i,j}^n$ 项以便于在伪时间中迭代：

$$p_{i,j}^n = \frac{(p_{i+1,j}^n + p_{i-1,j}^n)\Delta y^2 + (p_{i,j+1}^n + p_{i,j-1}^n)\Delta x^2}{2(\Delta x^2 + \Delta y^2)} - \frac{\rho\Delta x^2\Delta y^2}{2(\Delta x^2 + \Delta y^2)} \times \left[\frac{1}{\Delta t}\left(\frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y}\right) - \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x}\frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} - 2\frac{u_{i,j+1} - u_{i,j-1}}{2\Delta y}\frac{v_{i+1,j} - v_{i-1,j}}{2\Delta x} - \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y}\frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y}\right]$$

初始状态是 处处$u, v, p = 0$，边界条件为：

$$\begin{aligned} u,\ v,\ p\text{有周期性} &\quad \text{当 x=0, 2 时} \\ u,\ v = 0 &\quad \text{在 y=0, 2 时} \\ \frac{\partial p}{\partial x} = 0 &\quad \text{当 y=0, 2 时} \\ F = 1 & \end{aligned}$$

现在开始导入我们通常运行需要的库：

```
import numpy
from matplotlib import pyplot, cm
from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline
```

在步骤 11 中，我们分离出一部分转换方程以便于它更容易解析，在这里我们将做同样的事情。要注意的一点是，我们在整个网格中都有周期性边界条件, 因此我们需要显式地计算在我们的前缘和后缘处的矢量 u。

```
def build_up_b(rho, dt, dx, dy, u, v):
    b = numpy.zeros_like(u)
    b[1:-1, 1:-1] = (rho * (1 / dt * ((u[1:-1, 2:] - u[1:-1, 0:-2]) /
                    (2 * dx) + (v[2:, 1:-1] - v[0:-2, 1:-1]) / (2 * dy))
                    - ((u[1:-1, 2:] - u[1:-1, 0:-2]) / (2 * dx))**2 - 2
                    * ((u[2:, 1:-1] - u[0:-2, 1:-1]) / (2 * dy) *
                    (v[1:-1, 2:] - v[1:-1, 0:-2]) / (2 * dx))-
```

```
                    ((v[2:, 1:-1] - v[0:-2, 1:-1]) / (2 * dy))**2))

    # 周期性边界条件 压力 @ x = 2
    b[1:-1, -1] = (rho * (1 / dt * ((u[1:-1, 0] - u[1:-1,-2]) /
                  (2 * dx) + (v[2:, -1] - v[0:-2, -1]) / (2 * dy))
                  - ((u[1:-1, 0] - u[1:-1, -2]) / (2 * dx))**2 - 2
                  * ((u[2:, -1] - u[0:-2, -1]) / (2 * dy) *
                  (v[1:-1, 0] - v[1:-1, -2]) / (2 * dx)) -
                  ((v[2:, -1] - v[0:-2, -1]) / (2 * dy))**2))

    # 周期性边界条件 压力 @ x = 0
    b[1:-1, 0] = (rho * (1 / dt * ((u[1:-1, 1] - u[1:-1, -1]) /
                 (2 * dx) + (v[2:, 0] - v[0:-2, 0]) / (2 * dy))
                 - ((u[1:-1, 1] - u[1:-1, -1]) / (2 * dx))**2 - 2
                 * ((u[2:, 0] - u[0:-2, 0]) / (2 * dy) *
                 (v[1:-1, 1] - v[1:-1, -1]) / (2 * dx))-
                 ((v[2:, 0] - v[0:-2, 0]) / (2 * dy))**2))

    return b
```

我们还将定义一个压力泊松迭代函数，就像我们在步骤 11 中所做的那样。再次，请注意，我们必须将周期边界条件包含在前沿和后沿。我们还必须在网格的顶部和底部指定边界条件。

```
def pressure_poisson_periodic(p, dx, dy):
    pn = numpy.empty_like(p)

    for q in range(nit):
        pn = p.copy()
        p[1:-1, 1:-1] = (((pn[1:-1, 2:] + pn[1:-1, 0:-2]) * dy**2 +
                         (pn[2:, 1:-1] + pn[0:-2, 1:-1]) * dx**2) /
                         (2 * (dx**2 + dy**2)) - dx**2 * dy**2 /
                         (2 * (dx**2 + dy**2)) * b[1:-1, 1:-1])

        # 周期边界条件 压力 @ x = 2
        p[1:-1, -1] = (((pn[1:-1, 0] + pn[1:-1, -2])* dy**2 +
                        (pn[2:, -1] + pn[0:-2, -1]) * dx**2) /
                        (2 * (dx**2 + dy**2)) - dx**2 * dy**2 /
                        (2 * (dx**2 + dy**2)) * b[1:-1, -1])

        # 周期边界条件 压力 @ x = 0
        p[1:-1, 0] = (((pn[1:-1, 1] + pn[1:-1, -1])* dy**2 +
                       (pn[2:, 0] + pn[0:-2, 0]) * dx**2) /
                       (2 * (dx**2 + dy**2)) - dx**2 * dy**2 /
                       (2 * (dx**2 + dy**2)) * b[1:-1, 0])
```

```
        # 墙壁边界条件，压力
        p[-1, :] =p[-2, :]  # dp/dy = 0 at y = 2
        p[0, :] = p[1, :]  # dp/dy = 0 at y = 0

    return p
```

在开始之前，我们现在声明我们熟悉的变量和初始状态清单。

```
# 变量声明
nx = 41
ny = 41
nt = 10
nit = 50
c = 1
dx = 2 / (nx - 1)
dy = 2 / (ny - 1)
x = numpy.linspace(0, 2, nx)
y = numpy.linspace(0, 2, ny)
X, Y = numpy.meshgrid(x, y)
# 物理变量
rho = 1
nu = .1
F = 1
dt = .01
#初始状态
u = numpy.zeros((ny, nx))
un = numpy.zeros((ny, nx))

v = numpy.zeros((ny, nx))
vn = numpy.zeros((ny, nx))

p = numpy.ones((ny, nx))
pn = numpy.ones((ny, nx))

b = numpy.zeros((ny, nx))
```

因为是我们计算的重要部分，所以我们将回顾一下在第 9 步用于拉普拉斯方程的技巧。我们感兴趣的是，一旦我们达到了接近稳定的状态，我们的网格将是什么样的。我们可以指定一个时间步长数 nt 并且增加它直到我们对结果感到满意为止，或者我们可以告诉我们的代码一直运行到两次连续迭代之间的差非常小为止。

对每一次迭代我们必须管理 8 个独立边界条件。下面的代码将它们中的每一个明确地写入了进去。如果你喜欢挑战，你可以尝试写一个可以处理部分或全部边界条件的函数。如果你有兴趣解决这个问题，你应该可以阅读关于 Python 字典的内容

（https://docs.python.org/2/tutorial/datastructures.html#dictionaries）

```
udiff = 1
stepcount = 0

while udiff > .001:
    un = u.copy()
    vn = v.copy()

    b = build_up_b(rho, dt, dx, dy, u, v)
    p = pressure_poisson_periodic(p, dx, dy)

    u[1:-1, 1:-1] = (un[1:-1, 1:-1] - un[1:-1, 1:-1] * dt / dx *
                    (un[1:-1, 1:-1] - un[1:-1, 0:-2]) - vn[1:-1, 1:-1] *
                    dt / dy * (un[1:-1, 1:-1] - un[0:-2, 1:-1]) - dt /
                    (2 * rho * dx) * (p[1:-1, 2:] - p[1:-1, 0:-2]) +
                    nu * (dt / dx**2 * (un[1:-1, 2:] - 2 * un[1:-1, 1:-1] +
                    un[1:-1, 0:-2]) + dt / dy**2 * (un[2:, 1:-1] - 2 *
                    un[1:-1, 1:-1] + un[0:-2, 1:-1])) + F * dt)

    v[1:-1, 1:-1] = (vn[1:-1, 1:-1] - un[1:-1, 1:-1] * dt / dx *
                    (vn[1:-1, 1:-1] - vn[1:-1, 0:-2]) - vn[1:-1, 1:-1] *
                    dt / dy * (vn[1:-1, 1:-1] - vn[0:-2, 1:-1]) - dt /
                    (2 * rho * dy) * (p[2:, 1:-1] - p[0:-2, 1:-1]) +
                    nu * (dt / dx**2 * (vn[1:-1, 2:] - 2 * vn[1:-1, 1:-1] +
                    vn[1:-1, 0:-2]) + dt / dy**2 * (vn[2:, 1:-1] - 2 *
                    vn[1:-1, 1:-1] + vn[0:-2, 1:-1])))

    # 周期边界条件 u @ x = 2
    u[1:-1, -1] = (un[1:-1, -1] - un[1:-1, -1] * dt / dx *
                  (un[1:-1, -1] - un[1:-1, -2]) - vn[1:-1, -1] *
                  dt / dy * (un[1:-1, -1] - un[0:-2, -1]) - dt /
                  (2 * rho * dx) * (p[1:-1, 0] - p[1:-1, -2]) +
                  nu * (dt / dx**2 * (un[1:-1, 0] - 2 * un[1:-1,-1] +
                  un[1:-1, -2]) + dt / dy**2 * (un[2:, -1] - 2 *
                  un[1:-1, -1] + un[0:-2, -1])) + F * dt)

    # 周期边界条件 u @ x = 0
    u[1:-1, 0] = (un[1:-1, 0] - un[1:-1, 0] * dt / dx *
                 (un[1:-1, 0] - un[1:-1, -1]) - vn[1:-1, 0] * dt /
                 dy * (un[1:-1, 0] - un[0:-2, 0]) - dt / (2 * rho * dx) *
                 (p[1:-1, 1] - p[1:-1, -1]) + nu * (dt / dx**2 *
                 (un[1:-1, 1] - 2 * un[1:-1, 0] + un[1:-1, -1]) + dt /
                 dy**2 * (un[2:, 0] - 2 * un[1:-1, 0] + un[0:-2, 0]))
```

```
                + F * dt)

    # 周期边界条件 v @ x = 2
    v[1:-1, -1] = (vn[1:-1, -1] - un[1:-1, -1] * dt / dx *
                   (vn[1:-1, -1] - vn[1:-1, -2]) - vn[1:-1, -1] * dt /
                   dy * (vn[1:-1, -1] - vn[0:-2, -1]) - dt /
                   (2 * rho * dy) * (p[2:, -1] - p[0:-2, -1]) + nu *
                   (dt / dx**2 * (vn[1:-1, 0] - 2 * vn[1:-1, -1] +
                   vn[1:-1, -2]) + dt / dy**2 * (vn[2:, -1] - 2 *
                   vn[1:-1, -1] + vn[0:-2, -1])))

    # 周期边界条件 v @ x = 0
    v[1:-1, 0] = (vn[1:-1, 0] - un[1:-1, 0] * dt / dx *
                  (vn[1:-1, 0] - vn[1:-1, -1]) - vn[1:-1, 0] * dt / dy *
                  (vn[1:-1, 0] - vn[0:-2, 0]) - dt / (2 * rho * dy) *
                  (p[2:, 0] - p[0:-2, 0]) + nu * (dt / dx**2 *
                  (vn[1:-1, 1] - 2 * vn[1:-1, 0] + vn[1:-1, -1])
                  + dt / dy**2 * (vn[2:, 0] - 2 * vn[1:-1, 0] +
                  vn[0:-2, 0])))


    # 墙壁边界条件: u,v = 0 @ y = 0,2
    u[0, :] = 0
    u[-1, :] = 0
    v[0, :] = 0
    v[-1, :]=0

    udiff = (numpy.sum(u) - numpy.sum(un)) / numpy.sum(u)
    stepcount += 1
```

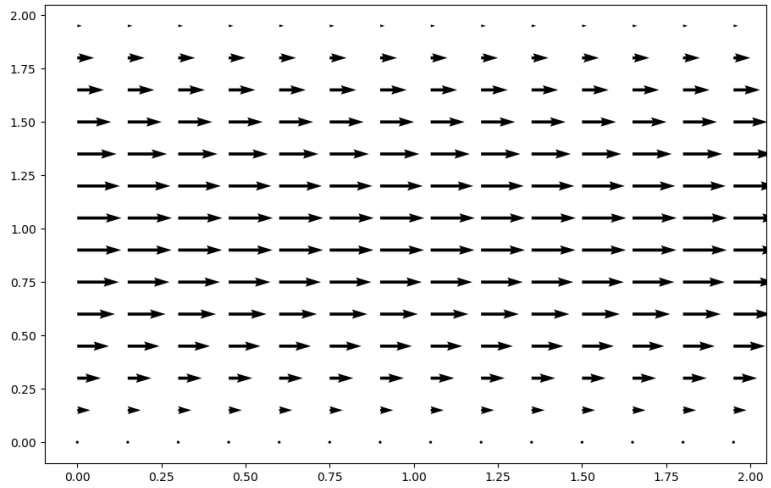
你可以看到我们还定义了一个变量 stepcount 用来观察在我们的停止条件满足之前，共经历了多少次循环迭代。

```
print(stepcount)
499
```

如果你想看看迭代的数目如何随着 udiff 条件变得越来越小而增加，可以尝试定义一个执行上述 while 循环的函数，输入 udiff 并输出函数运行过程中迭代的次数。

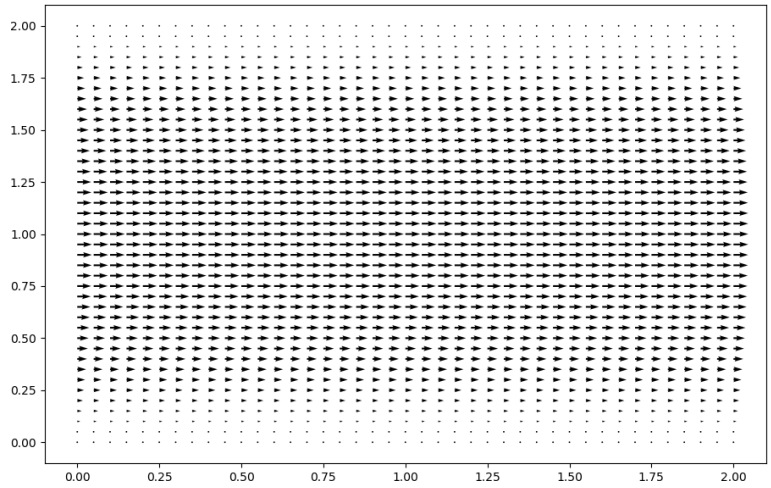
现在，让我们看看我们的结果。我们曾经使用了 quiver 函数来观察空腔流体的结果，它也很适合用于通道流体。

```
fig = pyplot.figure(figsize = (11,7), dpi=100)
pyplot.quiver(X[::3, ::3], Y[::3, ::3], u[::3, ::3], v[::3, ::3])
pyplot.show()
```



在 quiver 命令看到的类似结构 [::3, ::3] 在处理大量想要可视化数据时是有用的。上面使用的告诉 matplotlib 仅每 3 个数据绘制一个点。如果我们把它去掉，你就可以看到结果看起来有点拥挤。

```
fig = pyplot.figure(figsize = (11,7), dpi=100)
pyplot.quiver(X, Y, u, v)
pyplot.show()
```



16.3 了解更多

F 项的含义是什么？ 步骤 12 是演示在通道或管道中流体的问题。如果您回忆流体力学课程的话，指定的压力梯度是泊松流体的驱动器。

回忆 x 方向动量方程：

$$\frac{\partial u}{\partial t} + u \cdot \nabla u = -\frac{\partial p}{\partial x} + \nu \nabla^2 u$$

我们在步骤 12 中所做的是将压力分成稳定的和不稳定的部分 $p = P + p'$。所施加的稳定压力梯度是恒定的 $-\frac{\partial p}{\partial x} = F$(被解释为源项)，而非稳态成分是 $\frac{\partial p'}{\partial x}$。所以我们在步骤 12 中求解的压力实际上 p'，这对于稳定流体实际上是等于零的。

为什么我们要这么做？

请注意，我们对此流体使用了周期性边界条件。对于具有恒定压力梯度的流体而言，域的左边缘处的压力值必须与右边缘处的压力不同。因此，我们不能直接对压力施加周期性边界条件。较容易的是固定梯度，然后求解压力的扰动。

我们难道不该期望一个永远统一/不变的 p' 吗？

只有在层流稳定的情况下才是如此。在高雷诺数下，通道中的流体会变得紊乱，并且我们会看到在压力下的不稳定波动，这将导致非零的 p' 值。

在步骤 12 中，注意到压力场本身不是恒定的，但是它也是压力扰动场。压力场沿着具有等于压力梯度的斜率线性地变化。此外，对于不可压缩的流体，压力的绝对值是无关紧要的。

在线探索更多的 CFD 材料

本交互模块 **12 步到达纳维——斯托克斯**是洛伦娜 A. 巴尔巴教授从 2009 年至 2013 年在波士顿大学讲授的计算流体力学课的几个组成部分之一。

至于本课程的其他组件，你可以访问本课程 2013 春季版的资源网站（https://piazza.com/bu/spring2013/me702/resources）。

第 17 章　使用 NumbaPro 进一步优化

17.1　使用 NumbaPro 进一步优化

Continuum Analytics 新近推出的最令人兴奋的产品之一是 NumbaPro，它允许 Python 中编写的代码针对统一计算架构的 GPU 进行并行化计算。

本节不是关于如何在 GPU 上进行并行计算的快速入门指南。

现在，为了简单验证 NumbaPro 的能力，让我们返回到我们熟悉的问题，1 维非线性对流。

是的，这是一个微不足道的问题，但是它很好地演示了使用 NumbaPro 和 GPU 计算带来的速度增益的潜力。

我们将导入常用的库和 time 库，因此我们可以测量我们的性能增益，也可以测量 numbapro 中适当的库。

autojit 是与常规 numba 使用中相同的库，事实上，我们将以同样的方式使用它，来比较 Numba 和 NumbaPro 之间的区别。

cuda 是一个 NumbaPro 中的库，它提供了一些允许我们指定 GPU 进行计算的 CUDA 内嵌函数。

float32 是数据类型。Python 通常关心我们是否想要一个 int 或 str 类型，但是当我们开始深入内存管理的内部之后，对我们的数据格式有更具体的了解可能会有用（有时也是必需）。

```
import matplotlib.pyplot as plt
import numpy as np
import time
from numbapro import autojit, cuda, jit, float
```

我们正在尝试的第一个函数是一个使用 Numpy 中数组运算的简单实现。

```
# 使用Numpy实现的1维非线性对流
def NonLinNumpy(u, un, nx, nt, dx, dt):

    # 按nt时间步长运行并绘制每一步的动画
    for n in range(nt): # 循环时间步长次
        un = u.copy()
        u[1:] = -un[1:]*dt/dx*(un[1:]-un[:-1])+un[1:]

    return u
```

这是我们在步骤 2 使用过的”普通“版实现代码，两层嵌套循环，效率低。

```
# 使用普通Python实现的1维非线性对流
def NonLinVanilla(u, nx, nt, dx, dt):

    for n in range(nt):
        for i in range(1,nx-1):
            u[i+1] = -u[i]*dt/dx*(u[i]-u[i-1])+u[i]

    return u
```

这是我们已经实现的“普通”版本，但我们已经添加了 @autojit 修饰符，它将告诉 Numba 使用 JIT 编译这个函数以实现一个良好的速度提升。

```
# 使用Numba的JIT编译器实现的1维非线性对流(类似于LLVM)
@autojit
def NonLinNumba(u,un, nx, nt, dx, dt):

    for n in range(nt):
        for i in range(1,nx):
            un[i] = -u[i]*dt/dx*(u[i]-u[i-1])+u[i]

    return un
```

17.2 CUDA JIT

对你来说这里会有很多东西是新的，所以我们会一个一个的讲解。

```
@jit(argtypes=[float32[:],float32,float32,float32,float32[:]],target='gpu')
```

不同于 @autojit 自动确定了我们的数据类型，我们必须指定将被送到这个函数（实际上是个 CUDA”内核 “）的变量类型。上述`argtypes`参数告诉内核，将有五个变量，三个标量浮点和两个浮点数组。

```
# 使用NumbaPro的CUDA-JIT编译器实现的1维非线性对流
d@jit(argtypes=[float32[:],float32,float32,float32,float32[:]],target='gpu')
def NonLinCudaJit(u, dx, dt, nt, un):
    tid = cuda.threadIdx.x
    blkid = cuda.blockIdx.x
    blkdim = cuda.blockDim.x
    i = tid + blkid * blkdim

    if i >= u.shape[0]:
        return

    for n in range(nt):
```

```
            un[i] = -u[i]*dt/dx*(u[i]-u[i-1])+u[i]

            cuda.syncthreads()
```

```
    def main(nx):
        # 系统条件
        #nx = 500
        nt = 500
        c = 1
        xmax = 15.0
        dx = xmax/(nx-1)
        sigma = 0.25
        dt = sigma*dx

        # 波的初始状态
        ui = np.ones(nx) # 创建一个全部为1的1xn长度的向量
        ui[int(.5/dx):int(1/dx+1)]=2 # 设置帽子函数初始状态为: .5<=x<=1 为2
        un = np.ones(nx)

        if nx < 20001:
            t1 = time.time()
            u = NonLinVanilla(ui, nx, nt, dx, dt)
            t2 = time.time()
            print ("Vanilla version took: %.6f seconds" % (t2-t1))


        ui = np.ones(nx) # 创建一个全部为1的1xn长度的向量
        ui[int(.5/dx):int(1/dx+1)]=2 # 设置帽子函数初始状态为: .5<=x<=1 为2

        t1 = time.time()
        u = NonLinNumpy(ui, un, nx, nt, dx, dt)
        t2 = time.time()
        print("Numpy version took: %.6f seconds" % (t2-t1))
        numpytime = t2-t1
        #plt.plot(numpy.linspace(0,xmax,nx),u[:],marker='o',lw=2)


        ui = np.ones(nx) # 创建一个全部为1的1xn长度的向量
        ui[int(.5/dx):int(1/dx+1)]=2 # 设置帽子函数初始状态为: .5<=x<=1 为2

        t1 = time.time()
        u = NonLinNumba(ui, un, nx, nt, dx, dt)
        t2 = time.time()
        print ("Numbapro Vectorize version took: %.6f seconds" % (t2-t1))
```

```
    vectime = t2-t1
    #plt.plot(numpy.linspace(0,xmax,nx),u[:],marker='o',lw=2)

    u = np.ones(nx)
    u = ui.copy()
    griddim = 320, 1
    blockdim = 768, 1, 1
    NonLinCudaJit_conf = NonLinCudaJit[griddim, blockdim]
    t1 = time.time()
    NonLinCudaJit(u, dx, dt, nt, un)
    t2 = time.time()

    print("Numbapro Cuda version took: %.6f seconds" % (t2-t1))
    cudatime = t2-t1
```

```
main(500)
Vanilla version took: 0.581475 seconds
Numpy version took: 0.007635 seconds
Numbapro Vectorize version took: 0.000966 seconds
Numbapro Cuda version took: 0.002658 seconds

main(1000)
Vanilla version took: 1.140803 seconds
Numpy version took: 0.008878 seconds
Numbapro Vectorize version took: 0.001837 seconds
Numbapro Cuda version took: 0.002678 seconds

main(5000)
Vanilla version took: 5.336566 seconds
Numpy version took: 0.023648 seconds
Numbapro Vectorize version took: 0.009166 seconds
Numbapro Cuda version took: 0.002717 seconds

main(10000)
Vanilla version took: 10.719647 seconds
Numpy version took: 0.043988 seconds
Numbapro Vectorize version took: 0.018464 seconds
Numbapro Cuda version took: 0.002899 seconds

main(20000)
Vanilla version took: 21.414605 seconds
Numpy version took: 0.083821 seconds
Numbapro Vectorize version took: 0.036616 seconds
Numbapro Cuda version took: 0.002943 seconds
```

```
main(50000)
Numpy version took: 0.207808 seconds
Numbapro Vectorize version took: 0.093922 seconds
Numbapro Cuda version took: 0.003228 seconds

main(100000)
Numpy version took: 0.456931 seconds
Numbapro Vectorize version took: 0.189677 seconds
Numbapro Cuda version took: 0.004876 seconds

main(200000)
Numpy version took: 1.255342 seconds
Numbapro Vectorize version took: 0.393786 seconds
Numbapro Cuda version took: 0.005403 seconds
```

第 18 章　使用不同 CFD 方案评价伯格斯方程

我们已经在 1 维和 2 维空间中检验了伯格斯方程 (分别在步骤 4 及步骤 8)。在此，我们要再回到 1 维伯格斯方程中，并检验不同方案在离散非线性一阶双曲型方程中的作用。

考虑 1 维伯格斯方程：

$$\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial x}$$

我们想要以守恒形式表示该方程，所以我们最好处理潜在的冲击，我们有：

$$\frac{\partial u}{\partial t} = -\frac{\partial}{\partial x}(\frac{u^2}{2})$$

如果我们设 $E = \frac{u^2}{2}$，我们也可以写成如下形式：

$$\frac{\partial u}{\partial t} = -\frac{\partial E}{\partial x}$$

18.1　初始状态

对每一个方案而言，初始状态均是

$$u(x,\ 0) = \begin{cases} 1 & 0 \leq x \leq 2 \\ 0 & 2 \leq x \leq 4 \end{cases}$$

18.2　分配布置

首先，研究伯格斯方程在 $\Delta t = \Delta x = 1$ 时的行为，即库兰特数为 $\frac{\Delta t}{\Delta x} = 1$，然后将库兰特数改为 $\frac{\Delta t}{\Delta x} = 0.5$。

也以不同的网格大小进行实验，以查看每种方案在不同情况下呈现何种行为。

18.3　辅助代码

先导入 numpy 以及 matplotlib.pyplot。初始状态也被定义在下面，以及几个辅助函数来返回经常需要的值。

```
%matplotlib inline
import numpy
from matplotlib import pyplot
```

```
#基本初始状态参数
#定义网格数目，时间步长，CFL条件等...
nx = 81
nt = 70
sigma = 1
dx = 4.0/nx
dt = sigma*dx

#定义一个简单的函数来设置初始方形波状态
def u_ic():
    u = numpy.ones(nx)
    u[int((nx-1)/2):]=0
    return u

#定义2个lambda函数来帮助运算欧拉方程
utoE = lambda u: (u/2)**2
utoA = lambda u: u**2
```

18.4 莱克斯—弗里德里希斯方案

莱克斯—弗里德里希斯是一个使用时间上的前向差分和空间中的中心差分的显式 1 阶方案。然而，请注意，u_i^n 的值为其相邻单元格的平均值。

$$\frac{u_i^{n+1} - \frac{1}{2}(u_{i+1}^n + u_{i-1}^n)}{\Delta t} = -\frac{E_{i+1}^n - E_{i-1}^n}{2\Delta x}$$

变换等式求 u_i^{n+1} 的值，得到：

$$u_i^{n+1} = \frac{1}{2}(u_{i+1}^n + u_{i-1}^n) - \frac{\Delta t}{2\Delta x}(E_{i+1}^n - E_{i-1}^n)$$

在下面的函数中实现了该方案。本笔记本中的所有方案都以自己的函数进行包装，以帮助显示结果的动画。这也是很好的编码实践！

为了显示我们的动画，我们将每个时间步长的结果保存在 2 维数组中的 1 维 u 数组里。所以我们的数组 un 将具有 nt 行和 nx 列。

```
def laxfriedrichs(u, nt, dt, dx):
    # 初始化我们保存结果的数组，尺寸为ntxnx
    un = numpy.zeros((nt,len(u)))
    # 将初始数组u拷贝到我们新数组的每一行中
    un[:,:] = u.copy()

    # 现在，对于每一步时间步长而言，我们都将计算u^n+1,然后设置u的值等于u^n+1,
    # 这样我们能计算下一步迭代。对于每一步时间步长而言，整个向量u^n保存成
    # 我们结果数组un的一行。
```

```
    for i in range(1,nt):
        E = utoE(u)
        un[i,1:-1] = .5*(u[2:]+u[:-2]) - dt/(2*dx)*(E[2:]-E[:-2])
        un[i,0] = 1
        u = un[i].copy()

    return un
```

上述内容仅仅定义了一个能够执行莱克斯—弗里德里希斯方案的函数，现在需要调用它。

莱克斯—弗里德里希斯方案测试 1

现在，让我们试着展示我们莱克斯—弗里德里希斯方案的结果。首先，我们需要通过调用我们的 laxfriedrics 函数来生成结果。

```
u = u_ic()      #确保u被设置成我们期望的初始状态
un = laxfriedrichs(u,nt,dt,dx)
```

我们可以让 Numpy 告诉我们新数组 un 的大小并看看我们是否得到了我们期待的维度。我们应该得到 nt 行和 nx 列的结果。

```
print(un.shape)
(70,81)
```

看来不错！现在为了查看动画中的结果，我们将导入一些额外的库。JSAnimation 库还不是标准 iPython 安装的一部分，但您可以在下面的网站下载并了解如何安装它。

（https://github.com/jakevdp/JSAnimation）也可以直接使用命令`pip install JSAnimation`安装。

```
from matplotlib import animation
from JSAnimation.IPython_display import display_animation
```

为了看到我们结果的漂亮的交互式动画，我们需要定义几个参数。

- 首先，我们需要一个比默认大小稍微大一些，以便更容易查看的图形。
- 我们还希望在图形中定义具有某些限制的数轴。
- 然后，我们想在我们绘图中初始化一个行对象，但我们不需要给出任何数据，因此我们将只分配一个空列表给它的 x 和 y 坐标。

现在我们需要一个函数” 画出” 我们动画的每个帧：

- animate 取出我们结果数组 un 的一行然后绘制一条线。

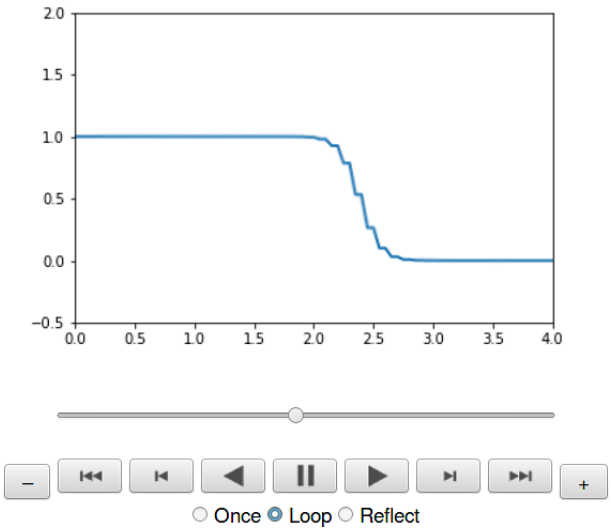
现在要设置 animation 对象，我们通过几个值来进行：

- fig 是我们已经定义的图形，我们要告诉 FuncAnimation 函数来使用它
- animate 是我们实际用来绘制我们结果的函数

- frames=un 告诉 FuncAnimation 函数它应该使用我们数组 un 中的每一行以生成动画的序列帧
- interval=50 设置帧之间毫秒的默认时间量

您会注意到动画下方的一组控件 –您可以使用 + 和 -按钮来加速或减慢动画。

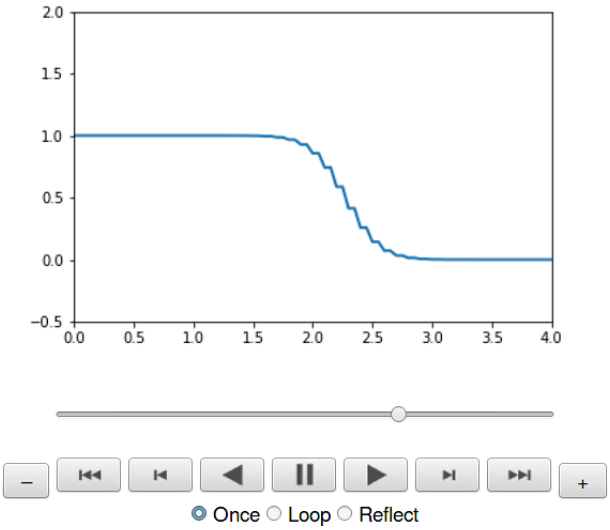
```
fig = pyplot.figure();
ax = pyplot.axes(xlim=(0,4),ylim=(-.5,2));
line, = ax.plot([],[],lw=2);

def animate(data):
    x = numpy.linspace(0,4,nx)
    y = data
    line.set_data(x,y)
    return line,

anim = animation.FuncAnimation(fig, animate, frames=un, interval=50)
display_animation(anim, default_mode='loop')
```



莱克斯—弗里德里希斯方案测试 2

现在我们将改变我们的 CFL 数，以便使 $\frac{\Delta t}{\Delta x} = 0.5$ 然后运行相同的方案。

```
dt = .5*dx
u = u_ic()  # 重新设置方形波的初始状态
un = laxfriedrichs(u,nt,dt,dx)
fig = pyplot.figure();
ax = pyplot.axes(xlim=(0,4),ylim=(-.5,2));
line, = ax.plot([],[],lw=2);

anim = animation.FuncAnimation(fig, animate, frames=un, interval=50)
display_animation(anim, default_mode='once')
```



在波的前缘注意到奇怪的” 步骤” 行为了吗？查看下一章笔记了解“奇偶解耦”。另外，看看 CFL 数的变化如何影响波浪。

18.5 莱克斯 -温德罗夫方案

莱克斯 -温德罗夫方案开始于 u^{n+1} 的泰勒级数展开：

$$u^{n+1} = u^n + u_t \Delta t + \frac{(\Delta t)^2}{2!} u_{tt} + \dots$$

现在用空间导数取代时间导数

$$E_t = - AE_x$$

$$u_t = - E_x$$

其中 $A = \frac{\partial E}{\partial u} = u$ 为伯格斯方程的雅克比矩阵。即，当带入完成的伯格斯方程中后，应该得到：

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{-E_{i+1}^n - E_{i-1}^n}{2\Delta x} + \frac{\Delta t}{2} \left(\frac{(A\frac{\partial E}{\partial x})_{i+\frac{1}{2}}^n - (A\frac{\partial E}{\partial x})_{i-\frac{1}{2}}^n}{\Delta x} \right)$$

上述方程的最后一项近似为：

$$\left(\frac{(A\frac{\partial E}{\partial x})_{i+\frac{1}{2}}^n - A\frac{\partial E}{\partial x})_{i-\frac{1}{2}}^n}{\Delta x} \right) \approx \frac{A_{i+\frac{1}{2}}^n \left(\frac{E_{i+1}^n - E_i^n}{\Delta x} \right) - A_{i-\frac{1}{2}}^n \left(\frac{E_i^n - E_{i-1}^n}{\Delta x} \right)}{\Delta x}$$

并在中心点处评估雅可比矩阵：

$$\frac{\frac{1}{2\Delta x}(A_{i+1}^n + A_i^n)(E_{i+1}^n - E_i^n) - \frac{1}{2\Delta x}(A_i^n + A_{i-1}^n)(E_i^n - E_{i-1}^n)}{\Delta x}$$

所以我们的方程变为:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{\frac{1}{2\Delta x}(A_{i+1}^n + A_i^n)(E_{i+1}^n - E_i^n) - \frac{1}{2\Delta x}(A_i^n + A_{i-1}^n)(E_i^n - E_{i-1}^n)}{\Delta x}$$

我们用 $A = u$ 进行代换然后求解 u_i^{n+1}，莱克斯 -温德罗夫方案的结果为：

$$u_i^{n+1} = u_i^n - \frac{\Delta t}{2\Delta x}(E_{i+1}^n - E_{i-1}^n) + \frac{\Delta t^2}{4\Delta x^2}\left[(u_{i+1}^n + u_i^n)(E_{i+1}^n - E_i^n) - (u_i^n + u_{i-1}^n)(E_i^n - E_{i-1}^n)\right]$$

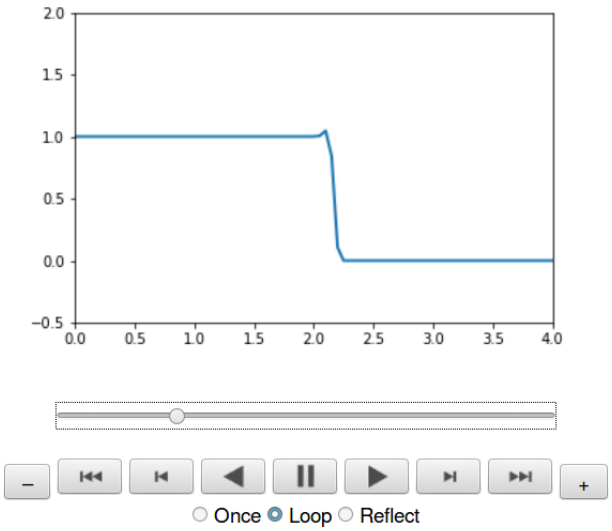
莱克斯 -温德罗夫方案有点长。记住，您可以使用 \ 斜杠来将语句拆分成多行。这使得代码更易解析（也更易调试!）。

```
def laxwendroff(u, nt, dt, dx):
    un = numpy.zeros((nt,len(u)))
    un[:] = u.copy()
    for i in range(1,nt):
        E = utoE(u)
        un[i,1:-1] = u[1:-1] - dt/(2*dx) * (E[2:]-E[:-2])
        + dt**2/(4*dx**2) * (u[2:]+u[1:-1])*(E[2:]-E[1:-1])
        - (u[1:-1]+u[:-2])*(E[1:-1]-E[:-2]))

        un[i,0]=1
        u = un[i].copy()
    return un
```

现在我们已经为莱克斯 -温德罗夫方案定义了一个函数，我们可以使用上面的过程来动画和查看我们的结果。

CFL=1 时

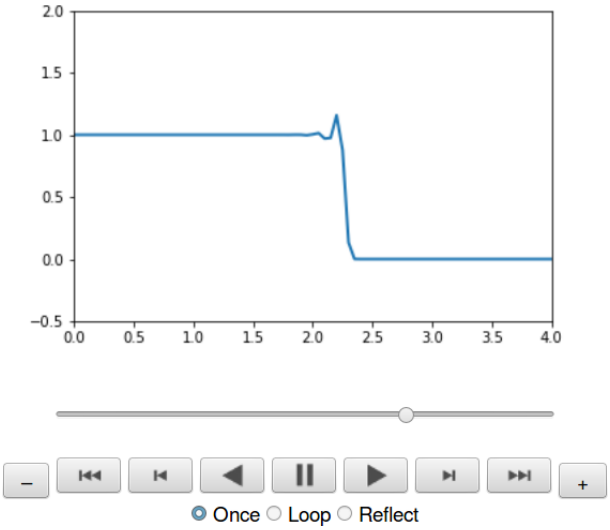
```
u = u_ic()
sigma = 1
dt = sigma*dx
un = laxwendroff(u,nt,dt,dx)
fig = pyplot.figure();
ax = pyplot.axes(xlim=(0,4),ylim=(-.5,2));
line, = ax.plot([],[],lw=2);
anim = animation.FuncAnimation(fig, animate, frames=un, interval=50)
display_animation(anim, default_mode='once')
```



CFL=0.5 时

```
u = u_ic()
```

```
sigma = .5
dt = sigma*dx
un = laxwendroff(u,nt,dt,dx)

fig = pyplot.figure();
ax = pyplot.axes(xlim=(0,4),ylim=(-.5,2));
line, = ax.plot([],[],lw=2);

anim = animation.FuncAnimation(fig, animate, frames=un, interval=50)
display_animation(anim, default_mode='once')
```



波前沿的振荡如何随着 CFL 条件的变化而变化？

18.6 麦科马克方案

$$u_i^* = u_i^n - \frac{\Delta t}{\Delta x}(E_{i+1}^n - E_i^n)(\quad)$$
$$u_i^{n+1} = \frac{1}{2}(u_i^n + u_i^* - \frac{\Delta t}{\Delta x}(E_i^* - E_{i-1}^*]))(\quad)$$

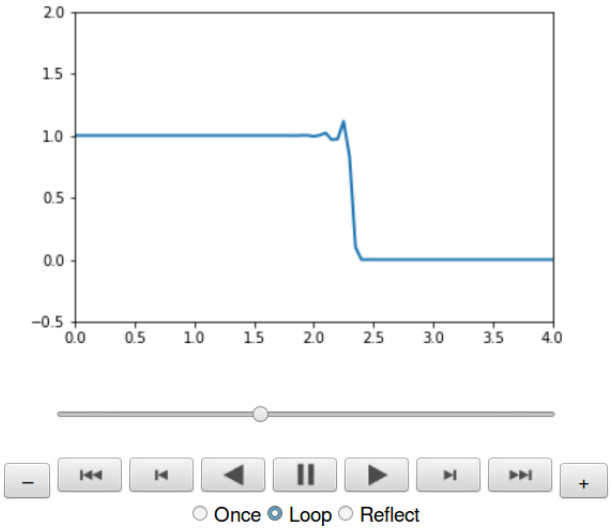
麦科马克方案是一种预测——校正方法。它首先计算下一个时间步长的粗略估计值，然后在第二步中平滑预测值。

```
def maccormack(u, nt, dt, dx):
    un = numpy.zeros((nt,len(u)))
    ustar = numpy.empty_like(u)
    un[:] = u.copy()
    ustar = u.copy()

    for i in range(1,nt):
        E = utoE(u)
```

```
        ustar[:-1] = u[:-1] - dt/dx * (E[1:]-E[:-1])
        Estar = utoE(ustar)
        un[i,1:] = .5 * (u[1:]+ustar[1:] - dt/dx * (Estar[1:] - Estar[:-1]))
        u = un[i].copy()

    return un
```

CFL=1 时

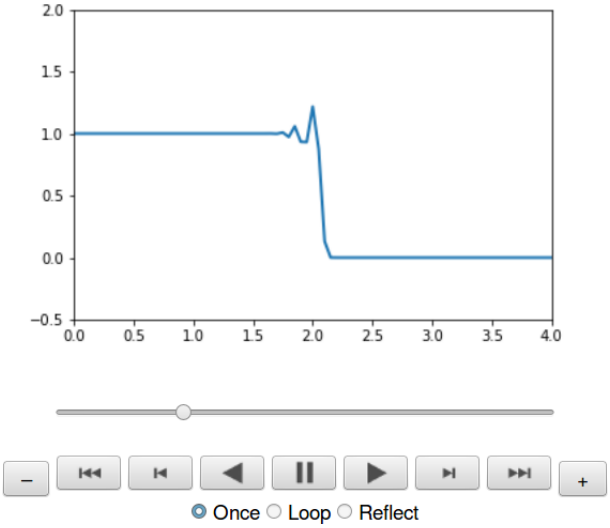
```
u = u_ic()
sigma = 1
dt = sigma*dx

un = maccormack(u,nt,dt,dx)

fig = pyplot.figure();
ax = pyplot.axes(xlim=(0,4),ylim=(-.5,2));
line, = ax.plot([],[],lw=2);

anim = animation.FuncAnimation(fig, animate, frames=un, interval=50)
display_animation(anim, default_mode='once')
```



CFL=0.5 时

```
u = u_ic()
sigma = 0.5
dt = sigma*dx

un = maccormack(u,nt,dt,dx)

fig = pyplot.figure();
ax = pyplot.axes(xlim=(0,4),ylim=(-.5,2));
```

```
line, = ax.plot([],[],lw=2);

anim = animation.FuncAnimation(fig, animate, frames=un, interval=50)
display_animation(anim, default_mode='once')
```



18.7 毕姆——沃明隐式方案

这是我们第一次看一个隐式方案。在先前使用的方案中，有可能分解出一些 u_i^{n+1} 因子并用 u_i^n 项求解。当使用的每一项来自先前的时间步骤时，即为一种显式的方法。

毕姆和沃明方案并不明确区分所有 u^{n+1} 项，它与其他示例的方式计算不同。

对 u_i^{n+1} 进行泰勒级数展开：

$$u_i^{n+1} = u_i^n + \frac{1}{2}\left[\left.\frac{\partial u}{\partial t}\right|_i^n + \left.\frac{\partial u}{\partial t}\right|_i^{n+1}\right]\Delta t + O(\Delta t^3)$$

对于伯格斯方程有 $\frac{\partial u}{\partial t} = -\frac{\partial E}{\partial x}$，所以：

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = -\frac{1}{2}\left[\left.\frac{\partial E}{\partial x}\right|_i^n + \left.\frac{\partial E}{\partial x}\right|_i^{n+1}\right] + O(\Delta t^2)$$

$$\therefore \frac{u_i^{n+1} - u_i^n}{\Delta t} = -\frac{1}{2}\left(\left.\frac{\partial E}{\partial x}\right|_i^n + \left.\frac{\partial E}{\partial x}\right|_i^n + \frac{\partial}{\partial x}\left[A(u_i^{n+1} - u_i^n)\right]\right)$$

对雅克比矩阵 A 使用 2 阶中心差分离散得到：

$$\frac{\partial}{\partial x}\left[A(u_i^{n+1} - u_i^n)\right] = \frac{1}{2\Delta x}\left(A_{i+1}^n u_{i+1}^{n+1} - A_{i-1}^n u_{i-1}^{n+1}\right) - \frac{1}{2\Delta x}\left(A_{i+1}^n u_{i+1}^n - A_{i-1}^n u_{i-1}^n\right)$$

得到一个三对角系统:

$$\begin{aligned}&\frac{\Delta t}{4\Delta x}\left(A_{i-1}^n u_{i-1}^{n+1}\right) + u_i^{n+1} + \frac{\Delta t}{4\Delta x}\left(A_{i+1}^n u_{i+1}^{n+1}\right)\\ &= u_i^n - \frac{1}{2}\frac{\Delta t}{\Delta x}\left(E_{i+1}^n - E_{i-1}^n\right) + \frac{\Delta t}{4\Delta x}\left(A_{i+1}^n u_{i+1}^n - A_{i-1}^n u_{i-1}^n\right)\end{aligned}$$

如果您不熟悉三对角系统，请查看维基百科页面上的简短解释。

```
    from scipy import linalg

    def beamwarming(u, nt, dt, dx):
        # Tridiagonal setup
        a = numpy.zeros_like(u)
        b = numpy.ones_like(u)
        c = numpy.zeros_like(u)
        d = numpy.zeros_like(u)

        un = numpy.zeros((nt,len(u)))
        un[:]=u.copy()

        for n in range(1,nt):
            u[0] = 1
            E = utoE(u)
            au = utoA(u)

            a[0] = -dt/(4*dx)*u[0]
            a[1:] = -dt/(4*dx)*u[0:-1]
            a[-1] = -dt/(4*dx)*u[-1]

            #b is all ones

            c[:-1] = dt/(4*dx)*u[1:]

            d[1:-1] = u[1:-1]-.5*dt/dx*(E[2:]-E[0:-2])+dt/(4*dx)*(au[2:]-au[:-2])

            # 通过减去a[0]与边界状态的乘积来修正托马斯算法
            d[0] = u[0] - .5*dt/dx*(E[1]-E[0])+dt/(4*dx)*(au[1]-au[0]) - a[0]

            ab = numpy.matrix([c,b,a])
            u = linalg.solve_banded((1,1), ab, d)
            u[0]=1
            un[n] = u.copy()
        return un
```

18.8 为什么托马斯算法需要“修正”?

在维基百科页面（https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm）上，解释了具有三对角形式的方程组

$$a_i x_{i-1} + b_i x_i + c_i x_{i-1} = d_i$$

其中 $a_1 = 0$，$c_n = 0$，所以方程组可以写成如下形式

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix}$$

但是如果 $a_1 \neq 0$ 会如何？边界条件是什么？

18.9 修正托马斯算法矩阵

为了利用托马斯算法，我们希望将算法只应用于我们问题的内部元素上，而不应用于边界元素上。设解为从 u_0 到 u_{n+1} 的向量

$$u_0 \quad u_1 \quad u_2 \quad \dots \quad u_n \quad u_{n+1}$$

其中 $u_0 = B_0$，$u_{n+1} = B_{n+1}$ 并且其中 B_0 以及 B_{n+1} 是一般边界条件。则 $[u_1 \ \dots \ u_n]$ 对应于 $[x_1 \ \dots \ x_n]$

现在我们能写出一个 $a_1 \neq 0$ 及边界条件的改进三对角方程组

$$\begin{bmatrix} 1 & 0 & 0 & \dots & & 0 \\ a1 & b_1 & c_1 & & & \vdots \\ 0 & a_2 & b_2 & c_2 & & \\ \vdots & & a_3 & b_3 & \ddots & \\ & & & \ddots & \ddots & c_n \\ 0 & & \dots & & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_n \\ u_{n+1} \end{bmatrix} = \begin{bmatrix} B_0 \\ d_1 \\ d_2 \\ \vdots \\ d_n \\ B_{n+1} \end{bmatrix}$$

检查此方程组时，你应该看到边界条件很明显，即 $u_0 = B_0$ 和 $u_{n+1} = B_{n+1}$。方程组中除了第一个和最后一个以外，其余的所有方程与维基百科中的示例相同。如果我们取 $a_1 \neq 0$ 和 $c_n \neq 0$ 的话则对应的方程为：

$$a_1 u_0 + b_1 u_1 + c_1 u_2 = d_1$$
$$a_n u_{n-1} + b_n u_n + c_n u_{n+1} = d_n$$

将 a_1 和 c_n 项移到它们各自的方程式的右边得到一个稍微变化的三对角矩阵，它可以使用现有的三对角解算器来求解。

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 - a_1 u_0 \\ d_2 \\ d_3 \\ \vdots \\ d_n - c_n u_{n+1} \end{bmatrix}$$

u_0 和 u_{n+1} 是指定的边界条件。对于伯格斯问题，有 $u_0 = B_0 = 1$ 和 $u_{n+1} = B_{n+1} = 0$，所以最终的矩阵为

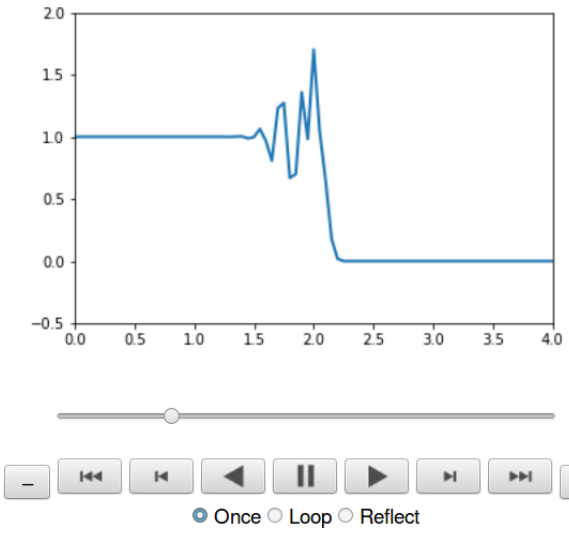
$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 - a_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix}$$

因此，对于这个的问题，当使用托马斯算法时我们必须修改 d 向量的第一元素为通常假设的值。

在上面的函数中，这是通过一行代码完成修改：

```
d[0] = u[0] − 0.5*dt/dx*(E[1]−E[0])+dt/(4*dx)*(au[1]−au[0]) − a[0]
```

```
u = u_ic()
sigma = .5
dt = sigma*dx
nt=60

un = beamwarming(u,nt,dt,dx)
fig = pyplot.figure();
ax = pyplot.axes(xlim=(0,4),ylim=(−.5,2));
line, = ax.plot([],[],lw=2);

anim = animation.FuncAnimation(fig, animate, frames=un, interval=50)
display_animation(anim, default_mode='once')
```



啊哟，你可能已经注意到我们使用的 nt 值比其他方案要小。在我们的 CFL 数等于 0.5 的情形下，这个隐式方法在 Python 无法处理之前大约可以执行 60 次迭代（然后被替换为 Inf，然后中断条带矩阵求解器)。

所以我们需要冷静一点，防止它变得这么快。

18.10 带阻尼的毕姆——沃明方案

隐式方法在许多情况下都可以很好地工作，但该情形不是其中之一。正如你所看到的，无阻尼的毕姆——沃明方案很快出错。我们可以增加阻尼来使之得到控制。

增加阻尼项‘D’：

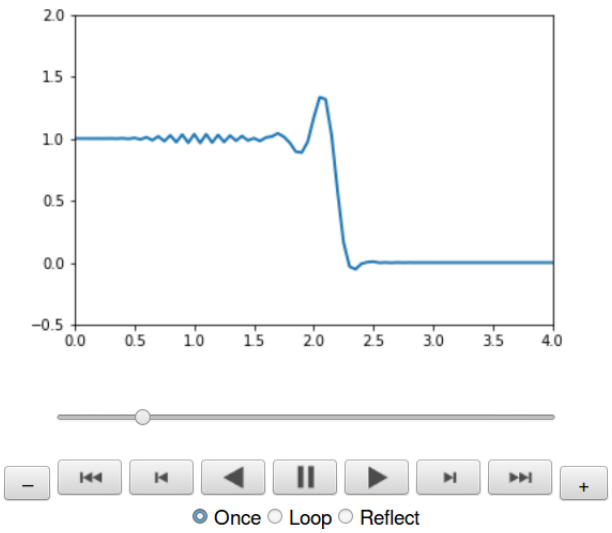
$$D = -\epsilon_e(u^n_{i+2} - 4u^n_{i+1} + 6u^n_i - 4^n_{i-1} + u^n_{i-2})$$

请注意，我们可以很容易地将大多数 d 向量包含进阻尼项中，d[0] 及 d[1] 需要单独注意一下。这里没有使用周期性边界条件，因此当 $i < 0$ 时，u_i 的任何值都被设置为等于 1。

```
    def dampit(u,eps,dt,dx):
        d = u[2]-.5*dt/dx*(u[3]**2/2-u[1]**2/2)+dt/(4*dx)*(u[3]**2-u[1]**2)\
            -eps*(u[4]-4*u[3]+6*u[2]-4*u[1]+u[0])
        return d

    def beamwarming_damp(u, nt, dt, dx):
        # 设置三对角矩阵
        a = numpy.zeros_like(u)
        b = numpy.ones_like(u)
        c = numpy.zeros_like(u)
        d = numpy.zeros_like(u)

        un = numpy.zeros((nt,len(u)))
        un[:] = u.copy()

        eps = .125

        for n in range(1,nt):
            u[0] = 1
            E = utoE(u)
            au = utoA(u)

            a[0] = -dt/(4*dx)*u[0]
            a[1:] = -dt/(4*dx)*u[0:-1]
            a[-1] = -dt/(4*dx)*u[-1]

            # b全部为1

            c[:-1] = dt/(4*dx)*u[1:]

            # 计算大多数u向量的阻尼系数
            d[2:-2] = u[2:-2]-.5*dt/dx*(E[3:-1]-E[1:-3])+dt/(4*dx)\
                                *(au[3:-1]-au[1:-3])\
                        -eps*(u[4:]-4*u[3:-1]+6*u[2:-2]-4*u[1:-3]+u[:-4])
```

```
        # 计算d[0]和d[1]的阻尼系数
        damp = numpy.concatenate((numpy.ones(2), u[:3]))
        d[0] = dampit(damp,eps,dt,dx)
        damp = numpy.concatenate((numpy.ones(1), u[:4]))
        d[1] = dampit(damp,eps,dt,dx)

        # 减去a[0]与边界条件的乘积来修正托马斯算法
        d[0] = d[0] - u[0] * a[0]

        ab = numpy.matrix([c,b,a])
        u = linalg.solve_banded((1,1), ab, d)
        u[0]=1
        un[n] = u.copy()
    return un
```

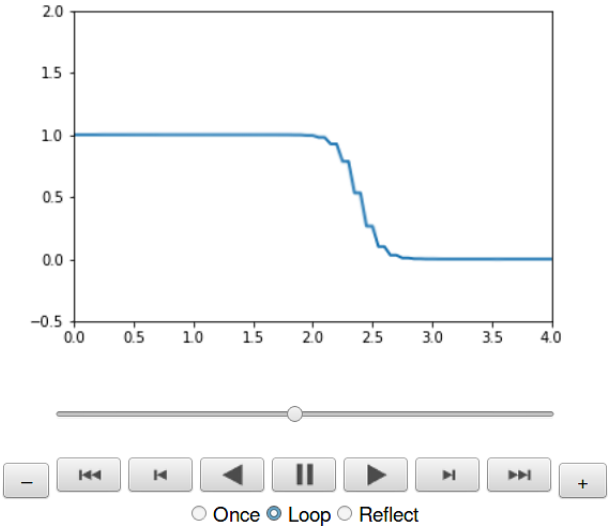
```
u = u_ic()
sigma = 0.5
dt = sigma*dx
nt = 120
un = beamwarming_damp(u,nt,dt,dx)

fig = pyplot.figure();
ax = pyplot.axes(xlim=(0,4),ylim=(-.5,2));
line, = ax.plot([],[],lw=2);

anim = animation.FuncAnimation(fig, animate, frames=un, interval=50)
display_animation(anim, default_mode='once')
```



这确实比以前更稳定。当然并不像我们的一些其他方案那样好，但至少保持了是有限的。此动画的阻尼系数为 $\epsilon = .125$。尝试用不同的阻尼系数，并查看发生了什么。

第 19 章　奇偶解耦

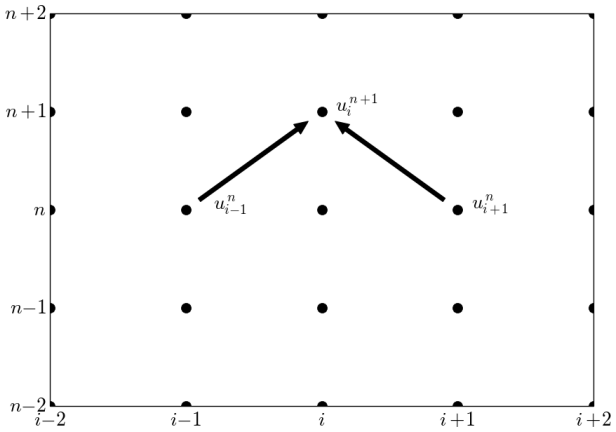
什么是奇偶解耦？让我们看看从伯格斯方程练习中提取的视频截图。



注意沿着波前缘的点是如何表现的？他们成对地级联起来。这不是渐逝性的行为，只要模拟运行就会一直持续，所以它为什么会发生呢？上面所示的特定视频为莱克斯—弗里德里希斯方案中的内容, 在该方案中

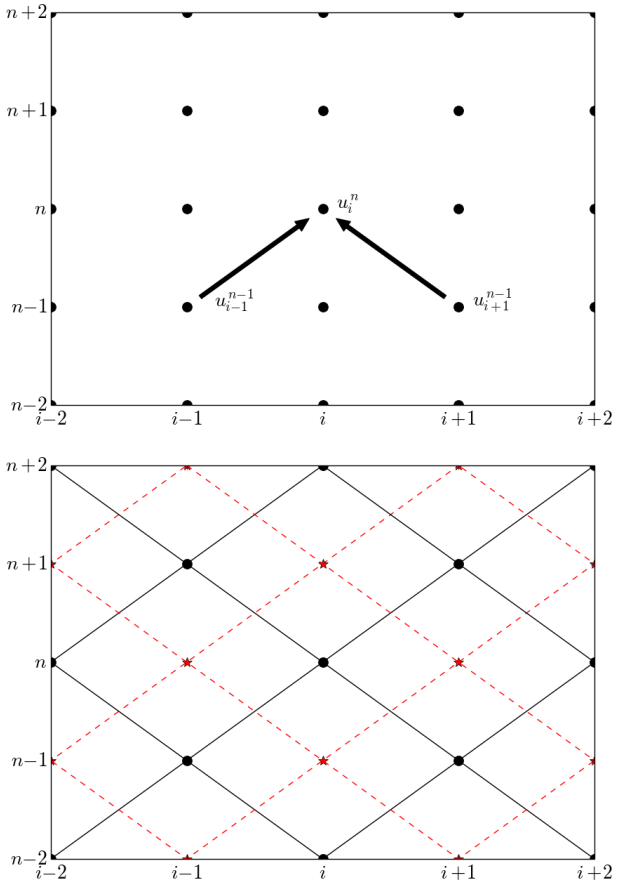
$$u_i^{n+1} = f\left(u_{i-1}^n, u_{i+1}^n\right)$$

网格上的每一个点 $u_{[}^{n+1}$ 都取决于这些父点 u_{i-1}^nh 和 u_{i+1}^n。



如果我们对 u_i^n 做同样的事会发生什么？按照经验，它的父母节点应该为 u_{i-1}^{n-1}h 和 u_{i+1}^{n-1}。在上面我们可以看到 u_i^{n+1} 不依赖与 u_i^n 的值。这意味着它也不依赖于 u_{i-1}^{n-1} 或是 u_{i+1}^{n-1}。

如果我们将网格上的所有点连接形成互相依赖的结构，你会得到这样的东西：



这些都意味着什么？

当使用依赖于 u_{i-1}^n 和 u_{i+1}^n 的某些方案时，该问题被一前一后地求解两次，其中每个解在空间中彼此偏移一步。这并不意味着它是对的或错的，但是当为某个特定问题选择方案时，需要注意这一点。