矩阵微积分
(用于机器学习及其它)

讲师：Alan Edelman 和 Steven G. Johnson　笔记：Paige Bright、Alan Edelman 和 Steven G. Johnson　基于2023年IAP期间MIT课程18.S096（现18.063）

内容

100

简介

这些笔记基于2023年1月第二次开设的课程，由麻省理工学院的Alan Edelman教授和Steven G. Johnson教授授课。该课程的上一版本，于2022年1月开设，可在OCW上找到。

Edelman和Johnson两位教授使用他/他的代词，并任职于麻省理工学院数学系；Edelman教授还担任MIT计算机科学与人工智能实验室（CSAIL）的Julia实验室主任，而Johnson教授则同时担任物理系教授。

这里是对该课程的描述。

> 我们都知道，典型的微积分课程序列分别从单变量和向量微积分开始。现代应用，如机器学习和大规模优化，需要下一步的大步，即“矩阵微积分”和任意向量空间的微积分。
>
> 本课程涵盖了一种连贯的矩阵微积分方法，展示了允许您从整体上（而不仅仅是作为标量数组）思考矩阵的技术，推广并计算重要矩阵分解的导数以及许多看似复杂的操作，并理解在大型计算中微分公式必须如何重新构想。我们将讨论“反向”（“伴随”，“反向传播”）微分以及现代自动微分如何比微积分更偏向于计算机科学（它既不是符号公式也不是有限差分）。

该课程涉及使用Julia语言进行大量示例数值计算，您可以根据以下说明在自己的计算机上安装Julia。本课程的材料也位于GitHub上，网址为https://github.com/mitmath/matrixcalc。

1 概述和动机

首先，矩阵微积分在麻省理工学院的课程目录中处于什么位置？嗯，有18.01（单变量微积分）和18.02（向量微积分），这是麻省理工学院学生必须选修的课程。但似乎这个材料序列正在被任意切断：{v*}

$$\text{Scalar} \rightarrow \text{Vector} \rightarrow \text{Matrices} \rightarrow \text{Higher-Order Arrays?}$$

毕竟，这是许多计算机编程语言，包括Julia，如何描述序列的方式！为什么微积分要在向量上停止呢?

在过去的十年中，线性代数在众多领域，如机器学习、统计学、工程等，扮演了越来越重要的角色。从这个意义上讲，线性代数已经逐渐占据了今天众多研究领域工具的更大一部分——现在每个人都需要线性代数。因此，我们想要在这些高阶数组上进行微积分是有道理的，而且这不会是一个简单/显然的推广（例如，对于非标量矩阵 A , $\frac{d}{dA}A^2 \neq 2A$）。

更普遍地，微分和敏感性分析的主题比从第一或第二学期的微积分中学到的简单规则所暗示的要深得多。对输入和/或输出在更复杂的向量空间（例如矩阵、函数等）中的函数进行微分是这个主题的一部分。另一个主题是高效评估涉及非常复杂计算的函数的导数，从神经网络到巨大的工程模拟——这导致了“伴随”或“反向模式”微分的话题，也称为“反向传播”。编译器通过计算机程序进行自动微分（AD）是另一个令人惊讶的话题，其中计算机所做的与典型的人类过程非常不同，即首先写出显式的符号公式，然后通过链式法则传递。这些只是几个例子：关键点是微分比你可能意识到的要复杂得多，而这些复杂性对于广泛的各类应用越来越相关。

让我们快速谈谈这些应用之一。

1.1 应用程序

应用：机器学习

机器学习有许多与之相关的热门词汇，包括但不限于：参数优化、随机梯度下降、自动微分和反向传播。在这整个拼贴画中，您可以看到矩阵微积分如何应用于机器学习的一部分。如果您对此感兴趣，建议您自己深入研究这些主题。

应用：物理建模

大型物理模拟，如工程设计问题，越来越多地以大量参数为特征，并且模拟输出对这些参数的导数对于评估对不确定性的敏感性以及应用大规模优化至关重要。

例如，飞机机翼的形状可能由数千个参数来表征，如果你能计算这些参数相对于阻力（来自大型流体流动模拟）的导数，那么你可以优化机翼形状以最小化给定升力或其他约束条件下的阻力。

一个此类参数化的极端版本被称为“拓扑优化”，在这种优化中，空间中的“每个点”的材料都可能是自由度，对这些参数进行优化不仅可以发现

最佳形状但最佳拓扑（材料在空间中的连接方式，例如有多少个孔）。例如，拓扑优化已应用于机械工程，以设计飞机机翼、人工髋关节等更复杂的金属支柱网格（例如，在给定强度下最小化重量）。

除了工程设计外，在将模型的未知参数拟合到实验数据以及评估具有不精确参数/输入的模型输出不确定性时，会出现复杂的高阶微分问题。这与下文所述的统计回归问题密切相关，只是在这里，模型可能是一组包含一些未知参数的巨大微分方程集。

A 应用：数据科学和多变量统计 tic

在多元统计分析中，模型通常用矩阵输入和输出（甚至更复杂的对象，如张量）来表述。例如，“简单”的线性多元矩阵模型可能是 $Y(X) = XB+U$，其中 B 是一个未知的系数矩阵（通过某种形式的拟合/回归来确定）和 U 是未知的随机噪声矩阵（防止模型精确拟合数据）。回归涉及最小化模型 XB 与数据 Y 之间误差 $U(B) = Y - XB$ 的某个函数；例如，矩阵范数 $\|U\|_F^2 = \text{tr}\, U^TU$，行列式 $\det U^TU$，或更复杂的函数。估计最佳拟合系数 B，分析不确定性以及许多其他统计分析需要对这些函数相对于 B 或其他参数进行微分。关于这个主题的最新综述文章是 Liu 等人（2022 年）：“矩阵微分计算及其在多元线性模型及其诊断中的应用”（https://doi.org/10.1016/j.sctalk.2023.100274）。

应用：自动微分

典型的微分学课程基于符号计算，学生实际上是在学习Mathematica或Wolfram Alpha能做的事情。即使你使用计算机进行符号求导，为了有效地使用它，你需要理解底层发生了什么。但是，类似地，一些数值方法可能只在这一课程的一小部分中出现（例如，使用差分商来近似导数），今天的自动微分既不是这两者中的任何一个。它更多地属于计算机科学领域的编译技术主题，而不是数学。然而，自动微分的底层数学很有趣，我们将在本课程中学习这一点！

甚至近似的计算机微分也比您想象的要复杂。对于单变量函数 $f(x)$，导数定义为差分 $[f(x + \delta x) - f(x)]\ /\delta x$ 当 $\delta x \to 0$ 时的极限。粗略的“有限差分”近似就是用一个小的 δx 来近似 $f'(x)$，但结果却引发了涉及截断误差和舍入误差平衡、高阶近似和数值外推的许多有趣问题。

1.2 一阶导数

函数一元导数本身也是一元函数——它简单地定义为函数的线性化。即，它具有$(f(x) - f(x_0)) \approx f'(x_0)(x - x_0)$的形式。在这个意义上，“一切对标量函数来说都很简单”（我们这里指的是，输入一个数字并输出一个数字的函数）。

有时还会使用其他符号表示这种线性化：

- $\delta y \approx f'(x)\delta x$,
- $dy = f'(x)dx$,
- $(y - y_0) \approx f'(x0)(x - x0)$,

- 并且 $df = f'(x)dx$。

这个最后一个是这个类别中上述选项的首选。可以将 dx 和 dy 视为“非常小的数”。在数学中，它们被称为无穷小量，通过极限的严格定义。请注意，我们在这里不想除以 dx。虽然用标量做这件事是完全可行的，但当我们到达向量和矩阵时，你并不总是可以除法！

这些导数的数值足够简单，可以随意操作。例如，考虑函数 $f(x) = x^2$ 和点 $(x_0, f(x_0)) = (3, 9)$。然后，我们在 $(3, 9)$ 附近的数值如下：

$$f(3.\mathbf{0001}) = 9.\mathbf{0006}0001$$
$$f(3.\mathbf{00001}) = 9.\mathbf{00006}00001$$
$$f(3.\mathbf{000001}) = 9.\mathbf{000006}000001$$
$$f(3.\mathbf{0000001}) = 9.\mathbf{0000006}0000001.$$

这里，左侧加粗的数字是 Δx，右侧加粗的数字是 Δy。注意 $\Delta y = 6\Delta x$。因此，我们有

$$f(3 + \Delta x) = 9 + \Delta y = 9 + 6\Delta x \implies f(3 + \Delta x) - f(3) = 6\Delta x \approx f'(3)\Delta x.$$

因此，我们有 x^2 在 $x = 3$ 处的线性化为函数 $f(x) - f(3) \approx 6(x - 3)$。

我们现在离开标量微积分的世界，进入向量/矩阵微积分的世界！Edelman教授邀请我们全面地思考矩阵——而不仅仅是作为一个数字表。

函数线性化的概念在我们定义接受/输出多个数字的函数的导数时在概念上会延续。当然，这意味着导数将与单个数字具有不同的“形状”。以下是第一阶导数形状的表格。函数的输入位于表格的左侧，函数的输出位于顶部。

input ↓ and output →	scalar	vector	matrix
scalar	scalar	vector (for instance, velocity)	matrix
vector	gradient = (column) vector	matrix (called the Jacobian matrix)	higher order array
matrix	matrix	higher order array	higher order array

您最终将在本课程中详细了解如何做这些事情！本表的目的在于建立微分作为线性化的概念。让我们来看一个例子。

示例 1
让 $f(x) = x^T x$，其中 x 是一个 2×1 矩阵，因此输出是一个 1×1 矩阵。确认 $2x_0^T dx$ 确实是 f 在 $x_0 = \begin{pmatrix} 3 & 4 \end{pmatrix}^T$ 处的微分。

首先，让我们计算 $f(x_0)$：

$$f(x_0) = x_0^T x_0 = 3^2 + 4^2 = 25.$$

然后，假设 $dx = [.001, .002]$。然后，我们就会得到

$$f(x + dx) = (3.001)^2 + (4.002)^2 = 25.\mathbf{022}005.$$

然后，注意$2x_0^T\ dx = 2\begin{pmatrix}3&4\end{pmatrix}^T\ dx = .022$。因此，我们有

$$f(x_0 + dx) - f(x_0) \approx 2x_0^T dx = .022.$$

我们现在就会看到，$2x_0^T dx$并非凭空而来！

1.3 简介：矩阵和向量乘积规则

对于矩阵，我们实际上仍然有一个乘积法则！我们将在后面的章节中更详细地讨论这个问题，但让我们从这里开始，先尝一小口。

定理 2（微分乘积法则）
设 A, B 为两个矩阵。然后，我们有 AB 的微分乘积法则：

$$d(AB) = (dA)B + A(dB).$$

通过矩阵 A 的微分，我们将其视为矩阵 A. 中的微小（无约束）变化。稍后，可能会对允许的扰动施加约束。

然而请注意，（根据我们的表格）矩阵的导数是一个矩阵！所以一般来说，乘积不会交换。

如果是一个向量，那么根据微分乘积法则我们有 ve

$$d(x^T x) = (dx^T)x + x^T(dx).$$

然而，请注意这是一个点积，点积是交换的（因为 $\sum a_i \cdot b_i = \sum b_i \cdot a_i$），所以我们有

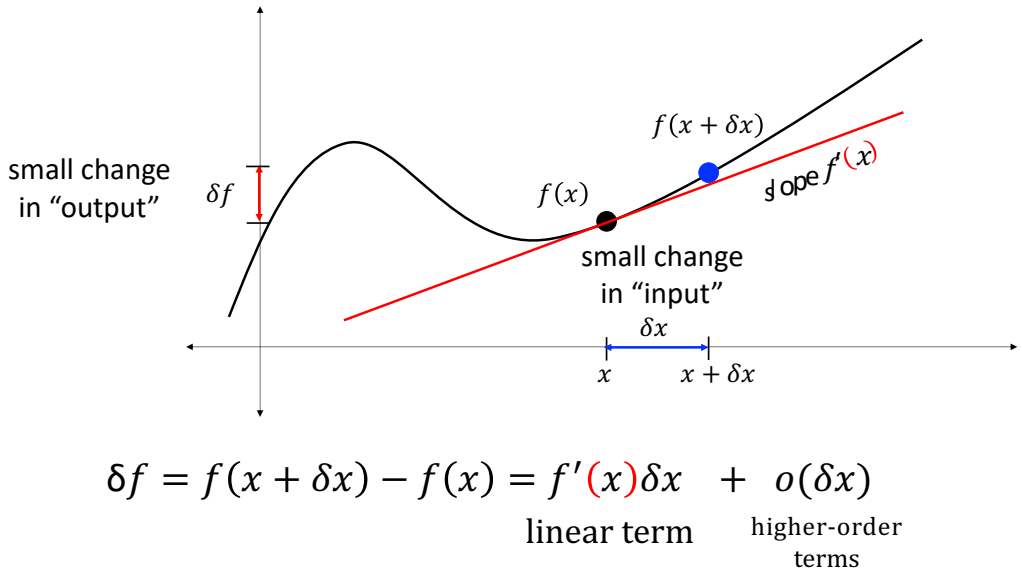
$$d(x^T x) = (2x)^T dx.$$

注意3. 向量作为矩阵时乘积法则的工作方式是转置“搭便车”。请参见下面的下一个例子。

示例 4
根据乘积法则，我们有

1. $d(u^T v) = (du)^T v + u^T(dv) = v^T du + u^T dv$ 由于点积交换律。2. $d(uv^T) = (du)v^T + u(dv)^T$.

备注5.这些陈述的证明方法可以在第2节中找到。



$$\delta f = f(x+\delta x) - f(x) = \underbrace{f'(x)\delta x}_{\text{linear term}} + \underbrace{o(\delta x)}_{\substack{\text{higher-order}\\ \text{terms}}}$$

图1：导数的本质是线性化：预测输出 $f(x)$ 中一个小的变化 δf 来自一个小的更改输入 x 中的 δx，使其在 δx 中的阶数为一次。

二 导数作为线性算子

我们现在将以一种可以推广到高阶数组和其他向量空间的方式重新审视导数的概念。我们将更详细地探讨作为线性算子的微分，特别是将深入研究我们迄今为止所陈述的一些事实。

2.1 重新审视单变量微积分

在第一学期单变量微积分课程（如MIT的18.01课程）中，导数$f'(x)$被引入为点$(x, f(x))$处切线的斜率，这也可以看作是f在x附近的线性近似。特别是，如图1所示，这相当于对$f(x)$的“输出”从“输入”的小变化δx进行一阶（线性）变化的预测：

$$\delta f = f(x+\delta x) - f(x) = f'(x)\,\delta x + \underbrace{(\text{higher-order terms})}_{o(\delta x)}.$$

我们可以更精确地表达这些高阶项，使用渐近的“小-o”符号“$o(\delta x)$”，它表示任何当$\delta x \to 0$时，其幅度比$|\delta x|$衰减得快的函数，因此对于足够小的δx，它相对于线性$f'(x)\,\delta x$项是可以忽略的。（这种记法的变体在计算机科学中常用，这里省略了其正式定义。[1]）此类高阶项的例子包括 $(\delta x)^2$，$(\delta x)^3$，$(\delta x)^{1.001}$，以及 $(\delta x)/\log(\delta x)$。

注意6。在这里，δx不是一个无穷小量，而是一个小数。请注意，我们的符号“δ”（希腊小写字母“delta”）与通常用来表示偏导数的符号“∂”不同。

这个导数的概念可能会让你想起泰勒级数的前两项 $f(x+\delta x) = f(x)+f'(x)\,\delta x+\cdots$（尽管实际上它比泰勒级数更基本！），并且这种表示法可以很好地推广到高维

[1] 简而言之，一个函数 $g(\delta x)$ 是 $o(\delta x)$ 当且仅当 $\lim_{\delta x\to 0} \frac{\|g(\delta x)\|}{\|\delta x\|} = = 0$。我们将在第5.2节回到这个主题。

并且其他向量空间。在微分符号中，我们可以用以下方式表达相同的概念：

$$df = f(x + dx) - f(x) = f'(x)\, dx.$$

在此记号中，我们隐含地省略了当 δx 趋近于无穷小时的极限消失的 $o(\delta x)$ 项 .

我们将使用这个作为导数的更一般定义。在这个公式中，我们避免除以 {v*}，因为很快我们将允许 x (以及因此 dx) 是除了数字以外的其他东西——如果 dx 是一个向量，我们就无法除以它了！

2.2 线性算子

从线性代数的角度来看，给定一个函数 f，我们考虑 f 的导数，将其视为线性算子 $f'(x)$，使得

$$df = f(x + dx) - f(x) = f'(x)[dx].$$

如上所述，你应该将微分符号 dx 视为 x 中的任意小变化，其中我们隐含地省略了任何 $o(dx)$ 项，即随 $dx \to 0$ 线性衰减更快的项。通常，我们会省略方括号，直接写 $f'(x)dx$ 而不是 $f'(x)[dx]$，但应理解为线性算子 $f'(x)$ 对 dx 的作用——不要写 $dx\, f'(x)$，这通常是没有意义的！

此定义将使我们能够将微分扩展到任意输入向量空间 x 和输出向量空间 $f(x)$。(更技术地说，我们将需要具有范数 $\|x\|$ 的向量空间，称为“Banach 空间”，以便精确地定义被省略的 $o(\delta x)$ 项。我们将在稍后回到 Banach 空间的话题。)

回忆 7（向量空间）
松散地说，向量空间（在 $\mathbb{R}$ 上）是一组元素，其中定义了元素之间的加法和减法，以及与实数标量的乘法。例如，虽然对 $\mathbb{R}^n$ 中的任意向量进行乘法没有意义，但我们当然可以相加它们，并且我们可以通过一个常数因子来缩放向量。

一些向量空间的例子包括：

- $\mathbb{R}^n$，如上所述。更一般地，$\mathbb{R}^{n\times m}$，具有实数元素的 $n \times m$ 矩阵空间。再次注意，如果 $n \neq m$，则元素之间的乘法未定义。
- $C0(\mathbb{R}^n)$，定义在$\mathbb{R}^n$上的连续函数集，加法按点定义。

回忆 8（线性算子）
回忆一下，线性算子是从向量空间 V中的一个向量 v 到另一个向量空间中的向量 $L[v]$（有时简单地表示为 Lv ）的映射 L。具体来说，L 是线性的，如果

$$L[v_1 + v_2] = Lv_1 + Lv_2 \quad \text{and} \quad L[\alpha v] = \alpha L[v]$$

对于标量 $\alpha \in \mathbb{R}$。

备注：在本课程中，f' 是一个映射，它接收一个 x 并输出一个线性算子 $f'(x)$ (，即 f 在 x) 处的导数。此外，$f'(x)$ 是一个线性映射，它接收一个输入方向 v 并给出一个输出向量 $f'(x)[v]$（我们将在后面将其解释为方向导数，参见第 2.2.1 节）。当方向 v 是无穷小 dx 时，输出 $f'(x)[dx] = df$ 是 f (的微分，即 f) 对应的无穷小变化。

符号9（导数算子和符号）

存在多种常用的导数表示法，以及与导数、微分和微分相关的多个概念。下表总结了这些表示法中的几个，并用方框圈出了本课程采用的表示法：

name	notations	remark
derivative	$\boxed{f'}$, also $\frac{df}{dx}$, Df, f_x, $\partial_x f$, ...	linear operator $f'(x)$ that maps a small change dx in the input to a small change $df = f'(x)[dx]$ in the output In single-variable calculus, this linear operator can be represented by a *single number*, the "slope," e.g. if $f(x) = \sin(x)$ then $f'(x) = \cos(x)$ is the number that we multiply by dx to get $dy = \cos(x)dx$. In multi-variable calculus, the linear operator $f'(x)$ can be represented by a *matrix*, the **Jacobian** J (see Sec. 3), so that $df = f'(x)[dx] = J\,dx$. But we will see that it is not always convenient to express f' as a matrix, even if we can.
differentiation	$\boxed{'}$, $\frac{d}{dx}$, D, ...	linear operator that maps a function f to its derivative f'
difference	$\boxed{\delta x}$ and $\boxed{\delta f} = f(x+\delta x) - f(x)$	small (but *not* infinitesimal) change in the input x and output f (depending implicitly on x and δx), respectively: an element of a vector space, *not* a linear operator
differential	$\boxed{dx}$ and $\boxed{df} = f(x+dx) - f(x)$	arbitrarily small ("infinitesimal"[a] — we drop higher-order terms) change in the input x and output f, respectively: an element of a vector space, *not* a linear operator
gradient	$\boxed{\nabla f}$	the vector whose inner product $df = \langle \nabla f, dx \rangle$ with a small change dx in the input gives the small change df in the output. The "transpose of the derivative" $\nabla f = (f')^T$. (See Sec. 2.3.)
partial derivative	$\boxed{\frac{\partial f}{\partial x}}$, f_x, $\partial_x f$	linear operator that maps a small change dx in a *single argument* of a multi-argument function to the corresponding change in output, e.g. for $f(x,y)$ we have $df = \frac{\partial f}{\partial x}[dx] + \frac{\partial f}{\partial y}[dy]$.

[a] 非正式地，可以将无穷小向量空间 dx 视为与 x (所在的同一空间，其中 x (被理解为向量的小变化，但仍然是一个向量)。正式地，可以通过各种方式定义一个独特的"无穷小向量空间"，例如在微分几何中作为余切空间，尽管我们在这里不会深入探讨。

一些线性算子的例子包括

- 乘以标量 {v*}，即 $Lv = \alpha v$。还包括列向量 v 与矩阵 A 的乘法，即 $Lv = Av$。
- 一些函数如 $f(x) = x^2$ 显然是非线性的。但 $f(x) = x + 1$ 呢？如果你绘制它，它可能看起来是线性的，但它不是线性运算，因为 $f(2x) = 2x + 1 \neq 2f(x)$——这种线性加上非零常数的函数被称为仿射函数。

- 存在许多其他线性运算的例子，这些运算不像矩阵-向量乘积那样方便或容易写出。例如，如果 {v*} 是一个 3x3 矩阵，那么 {v*}[{v*}] {v*} 是一个线性算子，给定 3x3 矩阵 {v*}。列向量的转置 {v*} 是线性的，但不是任何矩阵乘以 {v*} 得到的。或者，如果我们考虑函数向量空间，那么微分和积分的微积分运算也是线性算子！

2.2.1 方向导数

存在一种解释导数这种线性算子观点的等效方法，你可能之前在多元微积分中见过：作为方向导数。

如果我们有一个任意向量 x 的函数 $f(x)$，那么在 x 处沿一个方向（向量）的方向导数 v 定义为：

$$\frac{\partial}{\partial \alpha} f(x + \alpha v)\bigg|_{\alpha=0} = \lim_{\delta\alpha \to 0} \frac{f(x + \delta\alpha\, v) - f(x)}{\delta\alpha} \tag{1}$$

α 是一个标量。这把导数从任意向量空间转换回单变量微积分。它测量 f 在 v 方向上的变化率。但结果证明，这与我们上面的线性算子 $f'(x)$ 有一个非常简单的关系，因为（由于极限 $\delta\alpha \to 0$ 而忽略高阶项）：

$$f(x + \underbrace{d\alpha\, v}_{dx}) - f(x) = f'(x)[dx] = d\alpha\, f'(x)[v]\,,$$

在最后一步中，由于 $f'(x)$ 是一个线性算子，我们成功地提取了标量 $d\alpha$。与上述内容比较，我们立即发现方向导数为：

$$\boxed{\frac{\partial}{\partial \alpha} f(x + \alpha v)\bigg|_{\alpha=0} = f'(x)[v]}\,. \tag{2}$$

它与之前的$f'(x)$完全等价！（我们也可以将其视为2.5节中链式法则的一个实例。）从这个观点来看，将任意非无穷小向量v输入到$f'(x)[v]$中是完全合理的：结果不是一个df，而只是一个方向导数。

2.3 重新审视多元微积分，第1部分：标量值函数

设 f 为一个取“列”向量 $x \in \mathbb{R}^n$ 并产生标量（在 $\mathbb{R}$）的标量值函数。那么，

$$df = f(x + dx) - f(x) = f'(x)[dx] = \text{scalar}.$$

因此，由于 dx 是一个列向量（在任意方向上，表示 x 的任意小变化），产生标量 df 的线性算子 $f'(x)$ 必须是一个行向量（一个“1行矩阵”，或者更正式地，称为协向量或“对偶”向量或“线性形式”）！我们称这个行向量为梯度的转置

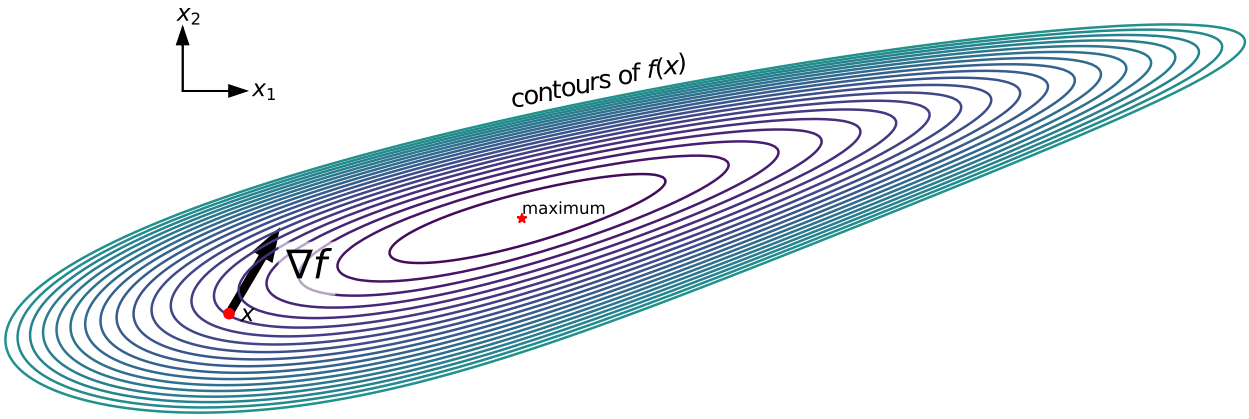


图2：对于实值 $f(x)$，梯度 ∇f 被定义为在点 x 处对应于“上升”方向，该方向垂直于 f 的等高线。尽管这不一定指向 f（的最近局部最大值，除非等高线是圆形），但“向上爬坡”仍然是许多计算优化算法寻找最大值的起点。

$(\nabla f)^T$，因此 df 是 dx 与梯度的点（“内”）积。所以我们有

$$df = \nabla f \cdot dx = \underbrace{(\nabla f)^T}_{f'(x)} dx \quad \text{where} \quad dx = \begin{pmatrix} dx_1 \\ dx_2 \\ \vdots \\ dx_n. \end{pmatrix}.$$

一些作者将梯度视为一个行向量（将其等同于 f' 或雅可比矩阵），但将其视为“列向量”（f' 的转置），正如我们在本课程中所做的那样，是一个常见且有用的选择。作为列向量，梯度可以被视为 x 空间中的“上升”方向（最陡上升方向），如图2所示。此外，它也更容易推广到其他向量空间的标量函数。在任何情况下，对于这门课程，我们都会始终将 ∇f 定义为与 x 具有相同的“形状”，以便 df 是 dx 与梯度的点积（内积）。

这与您可能从多元微积分中记住的梯度观点完全一致，其中梯度是一个由分量组成的向量

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix};$$

或者，等价地，

$$df = f(x + dx) - f(x) = \nabla f \cdot dx = \frac{\partial f}{\partial x_1} dx_1 + \frac{\partial f}{\partial x_2} dx_2 + \cdots + \frac{\partial f}{\partial x_n} dx_n \,.$$

虽然从分量角度观察有时可能方便，但我们希望鼓励您将向量 x 视为一个整体，而不仅仅是分量的集合，并了解通常在不逐个分量求导的情况下微分表达式更为方便和优雅，这是一种将更好地推广到更复杂的输入/输出向量空间的新方法。

让我们通过一个例子来看看我们是如何计算这个微分的。

示例 10
考虑 $f(x) = x^T Ax$，其中 $x \in \mathbb{R}^n$ 和 A 是一个方阵 $n \times n$，因此 $f(x) \in \mathbb{R}$。计算 df，$f'(x)$ 和 ∇f。

我们可以直接从定义中这样做。

$$\begin{aligned} df &= f(x+dx) - f(x) \\ &= (x+dx)^T A(x+dx) - x^T Ax \\ &= \cancel{x^T Ax} + dx^T\, Ax + x^T Adx + \cancel{dx^T Adx}^{\text{higher order}} - \cancel{x^T Ax} \\ &= \underbrace{x^T(A+A^T)}_{f'(x)=(\nabla f)^T} dx \implies \nabla f = (A+A^T)x\,. \end{aligned}$$

这里，我们删除了包含一个以上 dx 因子的项，因为这些项在渐近上可以忽略不计。另一个技巧是通过意识到这些是标量，因此等于它们自己的转置：$dx^T Ax = (dx^T Ax)^T = x^T A^T dx$。因此，我们发现 $f'(x) = x^T(A + A^T) = (\nabla f)^T$，或者等价地 $\nabla f = [x^T(A + A^T)]^T = (A + A^T)x$。

当然，也可以逐个分量地计算相同的梯度，就像你在多元微积分中学到的那样。首先，你需要将 $f(x)$ 明确地用 x 的分量表示，即 $f(x) = x^T Ax = \sum_{i,j} x_i A_{i,j} x_j$。然后，对于每个 k，你会计算 $\partial f/\partial x_k$，注意 x 在 f 的求和中出现两次。然而，这种方法很笨拙，容易出错，劳动密集，而且随着我们转向更复杂的功能，这种方法会迅速变得糟糕。我们认为，更好地习惯于将向量和矩阵作为一个整体来处理，而不是仅仅作为数字的集合，会更好。

2.4 重新审视多元微积分，第2部分：向量值函数

下次，我们将在第2部分再次回顾多元微积分（MIT的18.02），其中现在 f 将是一个向量值函数，接受向量 $x \in \mathbb{R}^n$ 并给出向量输出 $f(x) \in \mathbb{R}^m$。然后，df 将是一个 m 分量的列向量，dx 将是一个 n 分量的列向量，我们必须得到一个满足以下条件的线性算子 $f'(x)$：

$$\underbrace{df}_{m \text{ components}} = \underbrace{f'(x)}_{m\times n} \underbrace{dx}_{n \text{ components}}\,,$$

因此 $f'(x)$ 必须是一个称为 f 的雅可比矩阵 $m \times n$！

雅可比矩阵 J 表示将 dx 映射到 df 的线性算子：

$$df = J dx\,.$$

矩阵 J 有与 $J_{ij} = \frac{\partial f_i}{\partial x_j}$（对应的项，这些项对应于 J）的第 i 行和第 j 列。因此，现在假设 $f : \mathbb{R}^2 \to \mathbb{R}^2$。让我们了解我们如何计算 f 的微分：

$$df = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{pmatrix} \begin{pmatrix} dx_1 \\ dx_2 \end{pmatrix} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} dx_1 + \frac{\partial f_1}{\partial x_2} dx_2 \\ \frac{\partial f_2}{\partial x_1} dx_1 + \frac{\partial f_2}{\partial x_2} dx_2 \end{pmatrix}.$$

让我们计算一个例子。

示例 11
考虑函数 $f(x) = Ax$，其中 A 是一个常数 $m \times n$ 矩阵。然后，通过应用矩阵-向量乘积的分配律，我们有

$$\begin{aligned} df &= f(x+dx) - f(x) = A(x+dx) - Ax \\ &= \cancel{Ax} + Adx - \cancel{Ax} = Adx = f'(x)dx. \end{aligned}$$

因此，$f'(x) = A$。

注意，线性算子 A 是其自身的雅可比矩阵！现在让我们考虑一些
导数规则。

- 求和法则：给定 $f(x) = g(x) + h(x)$，我们得到

$$df = dg + dh \implies f'(x)dx = g'(x)dx + h'(x)dx.$$

因此，$f' = g' + h'$正如我们所期望的那样。这是线性算子$f'(x)[v] = g'(x)[v] + h'(x)[v]$，注意我们像求和矩阵一样可以求和线性算子（如$g'$和$h'$）！这样，线性算子构成一个向量空间。

- 产品法则：假设 $f(x) = g(x)h(x)$。那么，

$$\begin{aligned} df &= f(x+dx) - f(x) \\ &= g(x+dx)h(x+dx) - g(x)h(x) \\ &= (g(x) + \underbrace{g'(x)dx}_{dg})(h(x) + \underbrace{h'(x)dx}_{dh}) - g(x)h(x) \\ &= gh + dg\,h + g\,dh + \cancel{dg\,dh}^{0} - gh \\ &= dg\,h + g\,dh\,, \end{aligned}$$

在无穷小表示法中，由于 $dg\,dh$ 是高阶项，因此被省略。注意，像往常一样，现在 dg 和 h 可能不再交换，因为它们可能不再是标量了！

让我们看看一些如何优雅地应用乘积法则的简短示例。

示例 12
让 $f(x) = Ax$（映射 $\mathbb{R}^n \to \mathbb{R}^m$），其中 A 是一个常数 $m \times n$ 矩阵。然后，

$$df = d(Ax) = \cancel{dA}^{0}x + Adx = Adx \implies f'(x) = A.$$

我们有 $dA = 0$ 这里，因为当我们改变 x 时，A 不变。

示例 13
让 $f(x) = x^T Ax$（映射到 $\mathbb{R}^n \to \mathbb{R}$）。然后，

$$df = dx^T(Ax) + x^T d(Ax) = \underbrace{dx^T Ax}_{= x^T A^T dx} + x^T Adx = x^T(A + A^T)dx = (\nabla f)^T dx\,,$$

因此 $f'(x) = x^T(A + A^T)$。（在 A 对称的常见情况下，这简化为 $f'(x) = 2x^T A$。）请注意，在此我们应用了示例 12 来计算 $d(Ax) = Adx$。此外，f 是一个标量值函数，因此我们也可以像以前一样获得梯度 $\nabla f = (A + A^T)x$（如果 A 对称，则简化为 $2Ax$）。

示例 14（逐元素乘积）
给定 $x, y \in \mathbb{R}^m$，定义

$$x .* y = \begin{pmatrix} x_1 y_1 \\ \vdots \\ x_m y_m \end{pmatrix} = \underbrace{\begin{pmatrix} x_1 & & & \\ & x_2 & & \\ & & \ddots & \\ & & & x_m \end{pmatrix}}_{\operatorname{diag}(x)} \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix},$$

向量元素乘积（也称为Hadamard积），其中为了方便起见，以下我们还将 diag(x) 定义为对角线元素为 x 的 $m \times m$ 对角矩阵。然后，给定 $A \in \mathbb{R}^{m,n}$，定义 $f\colon \mathbb{R}^n \to \mathbb{R}^m$ 如下

$$f(x) = A(x .* x).$$

作为练习，可以验证以下内容：

(a) $x .* y = y .* x$，(b) $A(x .* y) = A\ \operatorname{diag}(x)\ y$。(c) $d(x .* y) = (dx) .* y + x .* (dy)$。因此，如果我们取 y 为常数并定义 $g(x) = y .* x$，其雅可比矩阵是 diag(y)。(d) $df = A(2x .* dx) = 2A\ \operatorname{diag}(x)\ dx = f'(x)[dx]$，所以雅可比矩阵是 $J = 2A\ \operatorname{diag}(x)$。(e) 注意，$f$ 在 x 处沿 v 方向的方向导数（见第2.2.1节）简单地给出为 $f'(x)[v] = 2A(x .* v)$。也可以对某些任意的 A, x, v 进行数值检查，$f(x + 10^{-8}v) - f(x) \approx 10^{-8}(2A(x .* v))$。

2.5 链式法则

一条微分学最重要的规则是链式法则，因为它允许我们对由简单函数复合而成的复杂函数进行微分。此链式法则也可以推广到我们的微分符号中，以便对任意向量空间上的函数进行操作：

- 链式法则：设 $f(x) = g(h(x))$。然后，

$$\begin{aligned} df = f(x+dx) - f(x) &= g(h(x+dx)) - g(h(x)) \\ &= g'(h(x))[h(x+dx) - h(x)] \\ &= g'(h(x))[h'(x)[dx]] \\ &= g'(h(x))h'(x)[dx] \end{aligned}$$

$g'(h(x))h'(x)$ 是 g' 和 h' 作为矩阵的组合。

换句话说，$f'(x) = g'(h(x))h'(x)$：雅可比（线性算子）f'仅仅是雅可比的乘积（复合），$g'h'$。顺序很重要，因为线性算子通常不交换：从左到右 $=$ 输出到输入。

让我们更仔细地观察这些雅可比矩阵的形状，在一个例子中，每个函数将一个列向量映射到另一个列向量：

示例 15
设 $x \in \mathbb{R}^n$，$h(x) \in \mathbb{R}^p$ 和 $g(h(x)) \in \mathbb{R}^m$。然后，设 $f(x) = g(h(x))$ 从 $\mathbb{R}^n$ 映射到 $\mathbb{R}^m$。链式法则随后表明

$$f'(x) = g'(h(x))h'(x),$$

这很有道理，因为 g' 是一个 $m \times p$ 矩阵，h' 是一个 $p \times n$ 矩阵，所以它们的乘积给出一个 $m \times n$ 矩阵 f'！然而，请注意，这不同于 $h'(x)g'(h(x))$，因为你不能（如果 $n \neq m$）将一个 $p \times n$ 矩阵和一个 $m \times p$ 矩阵相乘，即使 $n = m$ 你也会得到错误的结果，因为它们可能不会交换。

不仅乘法的顺序很重要，矩阵乘法的结合性在实际上也很重要。让我们考虑一个函数

$$f(x) = a(b(c(x)))$$

在 $c : \mathbb{R}^n \to \mathbb{R}^p$，$b : \mathbb{R}^p \to \mathbb{R}^q$ 和 $a : \mathbb{R}^q \to \mathbb{R}^m$ 的条件下。然后，根据链式法则，我们有

$$f'(x) = a'(b(c(x)))b'(c(x))c'(x).$$

请注意，这与以下相同{v*}

$$f' = (a'b')c' = a'(b'c')$$

通过结合律（为简洁起见省略函数参数）。左侧是左到右的乘法，右侧是右到左的乘法。

但是谁在乎呢？然而，结果证明结合律非常重要。重要到这两种顺序都有名字：从左到右乘法被称为“逆模式”，从右到左乘法被称为自动微分（AD）领域的“正向模式”。在某些情况下，逆模式微分也被称为“伴随方法”或“反向传播”，我们将在稍后更详细地探讨。这为什么重要呢？让我们思考一下矩阵乘法的计算成本。

2.5.1 矩阵乘法成本

如果你将一个 $m \times q$ 矩阵与一个 $q \times p$ 矩阵相乘，你通常通过计算 mp 个长度为 q (的点积或这些操作的某种等效重排来完成。要计算长度为 q 的点积需要 q 次乘法和 $q-$ 次标量加法。总的来说，这大约是 $2mpq$ 次标量操作。在计算机科学中，你会写成这样：“$\Theta(mpq)$”：对于大的 m, p, q，计算工作量与 mpq 是渐近成比例的。

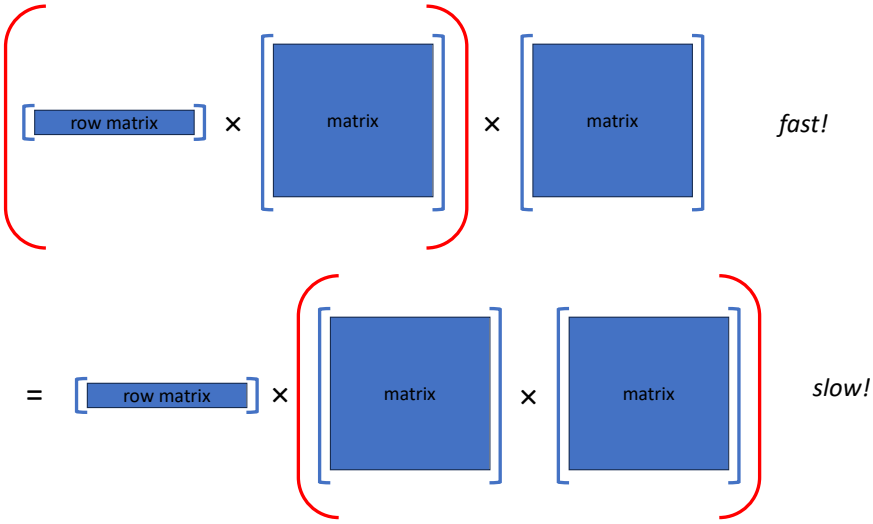


图3：矩阵乘法是结合律的——即对于所有A, B, C，有$(AB)C = A(BC)$——但如果最左边的矩阵只有一行（或很少的行），则从左到右乘法可能比从右到左更高效，如图所示。相应地，执行链式法则的顺序对所需的计算工作量有重大影响。从左到右被称为“逆模式”或“反向传播”，最适合输出比输入少得多的情况。

所以为什么链式法则的顺序很重要？考虑以下两个例子。

示例 16

假设你有很多输入$n \gg 1$，只有一个输出$m = 1$，以及许多中间值，即$q = p = n$。然后反向模式（从左到右）将花费$\Theta(n^2)$个标量运算，而正向模式（从右到左）将花费$\Theta(n^3)$！这是一个巨大的成本差异，如图3所示。

相反，假设你有很多输出$m \gg 1$和只有一个输入$n = 1$，以及很多中间值$q = p = m$。那么反向模式将需要$\Theta(m^3)$次操作，但正向模式只需$\Theta(m^2)$！

道德：如果你有很多输入和很少的输出（机器学习和优化的常见情况），则从左到右（反向模式）计算链式法则。如果你有很多输出和很少的输入，则从右到左（正向模式）计算链式法则。我们将在第8.4节中回到这个问题。

2.6 超越多变量导数

现在让我们计算一些超越一年级微积分的导数，其中输入和输出在更一般的向量空间中。例如，考虑以下示例：

示例 17
设 A 为一个 $n \times n$ 矩阵。您可能有以下矩阵值函数。例如：

- $f(A) = A3$,
- $f(A) = A^{-}1$ 如果 A 是可逆的,
- 或 U，其中 U 是对 A 应用高斯消元法后的结果矩阵！

您也可以有标量输出。例如：

- $f(A) = \det A$,
- $f(A) =$ 跟踪 A,
- 或 $f(A) = \sigma_1(A)$，A 的最大奇异值。

让我们关注这个讲座的两个更简单的例子。

示例 18
设 $f(A) = A^3$，其中 A 是一个方阵。计算 df。

这里，我们一步一步地应用链式法则：

$$df = dA\,A^2 + A\,dA\,A + A^2\,dA = f'(A)[dA].$$

注意，这不一定等于$3A^2$ (，除非dA和A交换，这通常不会成立，因为dA代表A)的任意小变化。右侧是一个作用在dA上的线性算子$f'(A)$，但将其简单地解释为乘以dA的“雅可比”矩阵并不容易！

示例 19
让 $f(A) = A^{-}\ 1$ 其中 A 是一个可逆的方阵。计算 $df = d$ $(A^{-}1)$。

这里，我们使用一个小技巧。注意 $AA^{-1} = \mathrm{I}$，单位矩阵。因此，我们可以使用乘积法则来计算微分（注意 $d\,\mathrm{I} = 0$，因为改变 A 不会改变 I）所以

$$d(AA^{-1}) = dA\,A^{-1} + A\,d(A^{-1}) = d(\mathrm{I}) = 0 \implies d(A^{-1}) = -A^{-1}\,dA\,A^{-1}.$$

3 矩阵函数的雅可比矩阵

当我们有一个以矩阵为输入和/或输出的函数时，我们在之前的讲座中已经看到，我们仍然可以通过一个公式来定义导数，即通过 f' 将输入的小变化映射到相应输出的小变化。然而，当你最初学习线性代数时，可能大多数线性运算都是通过矩阵乘以向量来表示的，可能需要一段时间才能习惯更广泛地思考线性运算。在本章中，我们讨论了即使在矩阵输入/输出情况下，仍然可以通过雅可比矩阵来表示 f'，以及最常见的实现这种表示的技术涉及矩阵“向量化”和一种新的矩阵运算，克罗内克积。这为我们提供了另一种思考我们的 f' 线性算子，偶尔很方便，但与此同时，我们也需要熟悉其他表示线性算子的方法——有时，显式的雅可比矩阵方法可能会掩盖关键结构，而且通常计算效率也不高。

对于本节笔记，我们参考链接的Pluto笔记本，以Julia中的计算演示来说明这一材料，展示了2×2矩阵 A 的平方 A^2 的多个视图的导数。

3.1 矩阵函数的导数：线性算子

我们已经强调过，导数 {v*} 是将输入的小变化映射到输出的小变化的线性算子。然而，当应用于将矩阵输入 A 映射到矩阵输出的函数 $f(A)$ 时，这个想法可能会以不熟悉的形式出现。例如，我们已经考虑了以下平方 $m \times m$ 矩阵上的函数：

- $f(A) = A3$，即给出 $df = f'(A)[dA] = dA\, A^2 + A\, dA\, A + A^2\, dA$。
- $f(A) = A^{-}1$，即给出 $df = f'(A)[dA] = -A^{-1}\, dA\, A^{-1}$

示例 20
一个更简单的例子是矩阵平方函数：

$$f(A) = A^2\,,$$

根据乘积法则得到

$$df = f'(A)[dA] = dA\, A + A\, dA\,.$$

您也可以从 $df = f(A+dA) - f(A) = (A+dA)^2 - A^2$ 中显式地解决这个问题，忽略 $(dA)^2$ 项。

在这些所有例子中，$f'(A)$ 由一个简单的公式描述 $f'(A)[dA]$，该公式将 A 中的任意变化 dA 与 f 中的变化 $df = f(A+dA) - f(A)$ 的一阶变化联系起来。如果微分方程让你感到困扰，请意识到我们可以将任何我们想要的矩阵 X 带入这个公式，而不仅仅是“无穷小”的变化 dA，例如，在我们的矩阵平方例子中，我们有

$$f'(A)[X] = XA + AX$$

对于任意 X (a 方向导数，从第 2.2.1 节)。这在 X 上是线性的：如果我们缩放或添加输入，则分别缩放或添加输出：

$$f'(A)[2X] = 2XA + A\,2X = 2(XA + AX) = 2f'(A)[X]\,,$$

$$f'(A)[X+Y] = (X+Y)A + A(X+Y) = XA + YA + AX + AY = XA + AX + YA + AY$$
$$= f'(A)[X] + f'(A)[Y]\,.$$

这是一个完美地定义线性运算的方法！我们在这里没有用熟悉的形式 $f'(A)[X] =$ (某个矩阵？) $\times$ (X向量？) 来表达它，这没关系！像 $XA + AX$ 这样的公式容易写出，容易理解，也容易计算。

但是有时你仍然可能想将 f' 视为一个单一的“雅可比”矩阵，使用线性代数最熟悉的语言，并且这样做是可能的！如果你上过足够抽象的线性代数课程，你可能已经学到，一旦你为输入和输出向量空间选择了一个基，任何线性算子都可以用一个矩阵来表示。然而，在这里，我们将更加具体，因为有一个传统的“笛卡尔”基矩阵 A 被称为“向量化”，在这个基中，一旦我们引入一种在“多维”线性代数中有广泛应用的新型矩阵乘法，线性算子如 $AX + XA$ 就特别容易用矩阵形式表示。

3.2 一个简单示例：二阶方阵平方函数

首先，让我们更详细地看看我们的矩阵平方函数

$$f(A) = A^2$$

对于2 $\times$ 2矩阵的简单情况，这些矩阵仅由四个标量描述，因此我们可以显式地查看导数中的每个项。特别是，

示例 21
对于一个2 $\times$ 2矩阵

$$A = \begin{pmatrix} p & r \\ q & s \end{pmatrix},$$

矩阵平方函数是

$$f(A) = A^2 = \begin{pmatrix} p & r \\ q & s \end{pmatrix}\begin{pmatrix} p & r \\ q & s \end{pmatrix} = \begin{pmatrix} p^2+qr & pr+rs \\ pq+qs & qr+s^2 \end{pmatrix}.$$

明确地用矩阵元素（p, q, r, s）表示，自然会想到我们的函数是将4个标量输入映射到4个标量输出。也就是说，我们可以将f视为与“向量化”函数$\tilde{f}$：$\mathbb{R}^4 \to \mathbb{R}^4$等价，该函数给出

$$\tilde{f}(\begin{pmatrix} p \\ q \\ r \\ s \end{pmatrix}) = \begin{pmatrix} p^2+qr \\ pq+qs \\ pr+rs \\ qr+s^2 \end{pmatrix}.$$

将矩阵以这种方式转换为列向量称为向量化，通常表示为 th e

操作“vec”：

$$\operatorname{vec} A = \operatorname{vec}\begin{pmatrix} p & r \\ q & s \end{pmatrix} = \begin{matrix} A_{1,1} \\ A_{2,1} \\ A_{1,2} \\ A_{2,2} \end{matrix}\begin{pmatrix} p \\ q \\ r \\ s \end{pmatrix},$$

$$\operatorname{vec} f(A) = \operatorname{vec}\begin{pmatrix} p^2+qr & pr+rs \\ pq+qs & qr+s^2 \end{pmatrix} = \begin{pmatrix} p^2+qr \\ pq+qs \\ pr+rs \\ qr+s^2 \end{pmatrix}.$$

关于vec，我们的“向量化”矩阵平方函数$\tilde{f}$ 定义为

$$\tilde{f}(\operatorname{vec} A) = \operatorname{vec} f(A) = \operatorname{vec}(A^2)\,.$$

更普遍地，

定义 22
任何 $m \times n$ 矩阵 $A \in \mathbb{R}^{m\times n}$ 的向量化 vec $A \in \mathbb{R}^{mn}$ 是通过简单地从左到右堆叠 A 的列来定义的，即列向量 vec A。也就是说，如果我们用 m 个分量向量 $\vec{a}_1, \vec{a}_2, \ldots \in \mathbb{R}^m$ 表示 A 的 n 列，那么

$$\operatorname{vec} A = \operatorname{vec}\underbrace{\begin{pmatrix} \vec{a}_1 & \vec{a}_2 & \cdots & \vec{a}_n \end{pmatrix}}_{A\in\mathbb{R}^{m\times n}} = \begin{pmatrix} \vec{a}_1 \\ \vec{a}_2 \\ \vdots \\ \vec{a}_n \end{pmatrix} \in \mathbb{R}^{mn}$$

是一个包含 mn-分量列向量的所有条目的 A。

在计算机上，矩阵元素通常以连续的内存位置序列存储，这可以看作是一种向量化的形式。实际上，vec A 正好对应于所谓的“列主序”存储，其中列元素是连续存储的；例如，在 Fortran、Matlab 和 Julia 中，这是默认格式，而 Fortran 的悠久传统意味着列主序在线性代数库中得到广泛应用。

.

问题 23
向量 vec A 对应于你将 $m \times n$ 矩阵 A 在矩阵基中表达时得到的系数。那个基是什么?

向量化将不熟悉的事物（如矩阵函数及其导数）转化为熟悉的事物（如向量函数及其雅可比或梯度）。这种方式可以是一个非常吸引人的工具，几乎是过于吸引人——如果你可以将一切转换回普通的多变量微积分，为什么还要进行“矩阵微积分”？然而，向量化有其缺点：从概念上讲，它可能会掩盖潜在的数学结构（例如，上面的$\tilde{f}$ 与矩阵平方 A^2 不太相似），并且在计算上，这种结构损失有时会导致严重的低效（例如，形成下面讨论的巨大的 $m^2 \times m^2$ 雅可比矩阵）。总的来说，我们相信

该研究此类矩阵函数的主要方法应该是将它们视为具有矩阵输入(A)和矩阵输出(A^2)，并且同样通常将导数视为矩阵上的线性算子，而不是其向量化的版本。然而，为了获得另一种视角的好处，仍然有必要熟悉向量化的观点。

3.2.1 矩阵平方四阶雅可比矩阵

T为了理解函数的雅可比矩阵（从矩阵到矩阵），让我们先考虑一个基本问题:

问题24. 矩阵平方函数的雅可比矩阵的大小是多少?

好吧，如果我们通过其向量化的等价函数 {v*} 来看待矩阵平方函数，将 $\mathbb{R}^4 \mapsto \mathbb{R}^4$ (4分量列向量映射到4分量列向量)，雅可比矩阵将是一个 4 × 4 矩阵（由每个输出分量对每个输入分量的导数组成）。现在让我们考虑一个更一般的平方矩阵 A：一个 $m \times m$ 矩阵。如果我们想找到 $f(A) = A^2$ 的雅可比矩阵，我们可以通过相同的过程（符号上）得到一个 $m^2 \times m^2$ 矩阵（因为有 m^2 个输入，A 的条目，以及 m^2 个输出，A^2 的条目）。即使对于小的 m，这些 m^4 偏导数的显式计算也很繁琐，但这是符号计算工具（例如 Julia 或 Mathematica）可以处理的任务。事实上，正如笔记本中所示，Julia 可以很容易地输出雅可比矩阵。对于我们在上面明确写出的 $m = 2$ 情况，你可以手动求 $\tilde{f}$ 的导数或使用 Julia 的符号工具来获得雅可比矩阵:

$$\tilde{f}' = \begin{array}{c} \\ (1,1) \\ (2,1) \\ (1,2) \\ (2,2) \end{array} \begin{array}{c} \begin{array}{cccc} (1,1) & (2,1) & (1,2) & (2,2) \end{array} \\ \begin{pmatrix} 2p & r & q & 0 \\ q & p+s & 0 & q \\ r & 0 & p+s & r \\ 0 & r & q & 2s \end{pmatrix} \end{array}. \tag{3}$$

例如，$\tilde{f}'$ 的第一行由 $p^2 + qr$ (的第一个输出) 对 4 个输入 $p, q, r,$ 和 s 的偏导数组成。在这里，我们通过“输出”矩阵 $d(A^2)$ 中元素的（行，列）索引 $(j_{\text{out}}, k_{\text{out}})$ 来标记行，并通过“输入”矩阵 A 中元素的索引 $(j_{\text{in}}, k_{\text{in}})$ 来标记列。虽然我们将雅可比矩阵 $\tilde{f}'$ 写作“2d”矩阵，但您也可以想象它是一个以 $j_{\text{out}}, k_{\text{out}}, j_{\text{in}}, k_{\text{in}}$ 为索引的“4d”矩阵。

然而，将导数 $f'(A)$ 视为矩阵上的线性变换的矩阵微积分方法（如我们上面所推导的），

$$f'(A)[X] = XA + AX\,,$$

似乎比逐个写出显式的“向量化”雅可比矩阵 $\tilde{f}'$ 更具揭示性，并为任何 $m \times m$ 矩阵提供了一个公式，而无需我们逐个费力地求取 m^4 偏导数。如果我们真的想追求向量化的视角，我们需要一种方法来恢复一些被繁琐的逐分量微分所掩盖的结构。在两种视角之间架起桥梁的关键工具是一种你可能不熟悉的矩阵运算：克罗内克积（表示为 $\otimes$ ）。

3.3 克罗内克积

一个线性运算如 $f'(A)[X] = XA + AX$ 可以被视为一个“高维矩阵”：普通的“2d”矩阵将“1d”列向量映射到1d列向量，而要将2d矩阵映射到2d矩阵，你可能想象一个“4d”矩阵（有时称为张量）。要将2d矩阵转换回1d向量，我们已经看到了向量化的概念（vec A）。一个与之密切相关工具有助于将“高维”线性运算

在矩阵中，对于向量化的输入/输出，返回“2d”矩阵，是克罗内克积 $A \otimes B$。尽管它们在初等线性代数课程中不常出现，但克罗内克积在许多涉及多维数据的数学应用中都会出现，例如多元统计学和数据科学或多维科学/工程问题。

定义 25
如果 A 是一个具有元素 a_{ij} 的 $m \times n$ 矩阵，并且 B 是一个 $p \times q$ 矩阵，那么它们的克罗内克积 $A \otimes B$ 定义为

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \Longrightarrow \underbrace{A}_{m\times n} \otimes \underbrace{B}_{p\times q} = \underbrace{\begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix}}_{mp\times nq},$$

因此，$A \otimes B$ 是通过“粘贴”B的副本并乘以A的每个元素形成的 $mp \times nq$ 矩阵。

例如，考虑2 × 2矩阵

$$A = \begin{pmatrix} p & r \\ q & s \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} a & c \\ b & d \end{pmatrix}.$$

然后 $A \otimes B$ 是一个 4×4 矩阵，包含所有可能的 A 条目与 B 条目乘积。注意 $A \otimes B \neq B \otimes A$ (，但这两个通过条目) 的重新排序相关联：

$$A \otimes B = \begin{pmatrix} pB & rB \\ qB & sB \end{pmatrix} = \begin{pmatrix} pa & pc & ra & rc \\ pb & pd & rb & rd \\ qa & qc & sa & sc \\ qb & qd & sb & sd \end{pmatrix} \quad \neq \quad B \otimes A = \begin{pmatrix} aA & cA \\ bA & dA \end{pmatrix} = \begin{pmatrix} ap & ar & cp & cr \\ aq & as & cq & cs \\ bp & br & dp & dr \\ bq & bs & dq & ds \end{pmatrix},$$

我们在图中用红色标出了一份 B 以便说明。请参阅笔记本以获取更多关于矩阵克罗内克积的示例（包括一些带有图片而不是数字的示例！）。

以上，我们看到了 $f(A) = A^2$ 在 $A = \begin{pmatrix} p & r \\ q & s \end{pmatrix}$ 可以被视为一个等价函数$\tilde{f}(\text{vec } A)$，将4个输入的列向量映射到4个输出（$\mathbb{R}^4 \mapsto \mathbb{R}^4$），具有一个4 × 4雅可比矩阵，我们（或计算机）费力地计算了16个逐元素偏导数。结果证明，一旦我们更好地理解克罗内克积，就可以更优雅地得到这个结果。我们将发现，4 × 4“向量化”的雅可比矩阵仅仅是

$$\tilde{f}' = \mathrm{I}_2 \otimes A + A^T \otimes \mathrm{I}_2\,,$$

在 I_2 是 2 × 2 的单位矩阵。也就是说，矩阵线性算子 $f'(A)[dA] = dA\,A + A\,dA$ 是等价的 t,
在向量化之后，变为：

$$\underbrace{\text{vec } f'(A)[dA]}_{dA\,A + A\,dA} = \underbrace{(\mathrm{I}_2 \otimes A + A^T \otimes \mathrm{I}_2)}_{\tilde{f}'} \text{vec } dA = \underbrace{\begin{pmatrix} 2p & r & q & 0 \\ q & p+s & 0 & q \\ r & 0 & p+s & r \\ 0 & r & q & 2s \end{pmatrix}}_{\tilde{f}'} \underbrace{\begin{pmatrix} dp \\ dq \\ dr \\ ds \end{pmatrix}}_{\text{vec } dA}.$$

为了理解为什么是这样，然而，我们首先必须对克罗内克积的代数建立一些理解。首先，一个很好的练习是让自己确信克罗内克积的一些简单性质。

产品：

问题 26
从克罗内克积的定义中，推导出以下恒等式：

1. $(A \otimes B)^T = A^T \otimes B^T$. 2. $(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$. 3. $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$. (由性质2得出。) 4. $A \otimes B$是正交的（其转置是其逆）当且仅当 A 和 B 是正交的。（由性质1和3得出。）

5. $\det(A \otimes B) = \det(A)^m \det(B)^n$，其中 $A \in \mathbb{R}^{n,n}$ 和 $B \in \mathbb{R}^{m,m}$。

6. $\mathrm{tr}(\{v^*\}) = (\mathrm{tr}\ A)(\mathrm{tr}\ B)$。7. 给定 A 和 B 的特征向量/值 $Au = \lambda u$ 和 $Bv = \mu v$，则 $\lambda\mu$ 是 $A \otimes B$ 的特征值，其特征向量为 $u \otimes v$。(此外，由于 $u \otimes v = \mathrm{vec}\ X$ 其中 $X = vu^T$，您可以通过下面的性质27将其与恒等式 $BXA^T = Bv(Au)^T = \lambda\mu X$ 相关联。)

3.3.1 关键克罗内克积恒等式

为了将线性运算如 $AX + XA$ 通过向量化为克罗内克积，关键恒等式为 i s:

命题27
给定（兼容尺寸）的矩阵 A, B, C，我们有

$$(A \otimes B)\,\mathrm{vec}(C) = \mathrm{vec}(BCA^T).$$

我们可以将 $A \otimes B$ 视为线性运算 $C \mapsto BCA^T$ 的向量等价形式。我们倾向于引入并行符号 $(A \otimes B)[C] = BCA^T$ 来表示此操作的“非向量”版本，尽管这种符号并不标准。

一个可能的记忆方法是，B 正在 C 的左边，而 A “环绕”到右边并进行了转置。

这个恒等式从哪里来？我们可以通过首先考虑以下情况将其分解成更简单的部分：要么 A 要么 B 是适当大小的单位矩阵 I。首先，假设 A = I，因此 $BCA^T = BC$。vec(BC) 是什么？如果我们让 $\vec{c}_1, \vec{c}_2, \ldots$ 表示 C 的列，那么回想一下 BC 简单地用 B 的每一列左乘 C 的列：

$$BC = B\begin{pmatrix} \vec{c}_1 & \vec{c}_2 & \cdots \end{pmatrix} = \begin{pmatrix} B\vec{c}_1 & B\vec{c}_2 & \cdots \end{pmatrix} \Longrightarrow \mathrm{vec}(BC) = \begin{pmatrix} B\vec{c}_1 \\ B\vec{c}_2 \\ \vdots \end{pmatrix}.$$

现在，我们如何将这个vec(BC)向量变成某个乘以vec C的结果？这应该立即很明显， t

$$\mathrm{vec}(BC) = \begin{pmatrix} B\vec{c}_1 \\ B\vec{c}_2 \\ \vdots \end{pmatrix} = \underbrace{\begin{pmatrix} B & & \\ & B & \\ & & \ddots \end{pmatrix}}_{\mathrm{I} \otimes B} \underbrace{\begin{pmatrix} \vec{c}_1 \\ \vec{c}_2 \\ \vdots \end{pmatrix}}_{\mathrm{vec}\, C},$$

但是这个矩阵恰好是克罗内克积 $I \otimes B$！因此，我们推导出

$$(\mathrm{I} \otimes B)\,\mathrm{vec}\,C = \mathrm{vec}(BC).$$

关于 A^T 项呢？这有点棘手，但同样，让我们简化到 $B = \mathrm{I}$ 的情况，在这种情况下 $BCA^T = CA^T$。为了向量化这个，我们需要查看 CA^T 的列。CA^T 的第一列是什么？它是 C 的列的线性组合，其系数由 A^T (= 的第一列和 A) 的第一行给出：

$$\text{column 1 of } CA^T = \sum_j a_{1j}\vec{c}_j\ .$$

类似地，对于第2列等，然后我们将这些列“堆叠”以获得vec(CA^T）。但这正是将矩阵A乘以向量的公式，如果向量的“元素”是列$\vec{c}_j$。明确写出，这成为：

$$\mathrm{vec}(CA^T) = \begin{pmatrix} \sum_j a_{1j}\vec{c}_j \\ \sum_j a_{2j}\vec{c}_j \\ \vdots \end{pmatrix} = \underbrace{\begin{pmatrix} a_{11}\,\mathrm{I} & a_{12}\,\mathrm{I} & \cdots \\ a_{21}\,\mathrm{I} & a_{22}\,\mathrm{I} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}}_{A\otimes\mathrm{I}} \underbrace{\begin{pmatrix} \vec{c}_1 \\ \vec{c}_2 \\ \vdots \end{pmatrix}}_{\mathrm{vec}\,C},$$

因此我们推导出

$$(A \otimes \mathrm{I})\,\mathrm{vec}\,C = \mathrm{vec}(CA^T).$$

The full identity ($A \otimes B$) vec(C) = vec(BCA^T) can then be obtained by straightforwardly combining these two derivations : replace CA^T with BCA^T in the second derivation, which replaces $\vec{c}_j$ with $B\vec{c}_j$ and hence I with B.

3.3.2 克罗内克积表示法中的雅可比矩阵

因此现在我们想使用命题27来计算$f(A) = A^2$的雅可比矩阵，以克罗内克积的形式。让dA成为命题27中的C。我们现在可以立即看出

$$\mathrm{vec}(A\,dA + dA\,A) = \underbrace{(\mathrm{I}\otimes A + A^T \otimes \mathrm{I})}_{\text{Jacobian } \tilde{f}'(\mathrm{vec}\,A)}\,\mathrm{vec}(dA)\,,$$

在 I 是与 A 同大小的单位矩阵。我们也可以用我们的“非向量化”线性运算来表示这个 rTranslated Tex
符号：

$$A\,dA + dA\,A = (\mathrm{I}\otimes A + A^T \otimes \mathrm{I})[dA]\,.$$

在2 × 2示例中，这些克罗内克积可以显式计算：

$$\underbrace{\begin{pmatrix} 1 & \\ & 1 \end{pmatrix}}_{\mathrm{I}} \otimes \underbrace{\begin{pmatrix} p & r \\ q & s \end{pmatrix}}_{A} + \underbrace{\begin{pmatrix} p & q \\ r & s \end{pmatrix}}_{A^T} \otimes \underbrace{\begin{pmatrix} 1 & \\ & 1 \end{pmatrix}}_{\mathrm{I}} = \underbrace{\begin{pmatrix} p & r & & \\ q & s & & \\ & & p & r \\ & & q & s \end{pmatrix}}_{\mathrm{I}\otimes A} + \underbrace{\begin{pmatrix} p & & q & \\ & p & & q \\ r & & s & \\ & r & & s \end{pmatrix}}_{A^T\otimes\mathrm{I}}$$

$$= \begin{pmatrix} 2p & r & q & 0 \\ q & p+s & 0 & q \\ r & 0 & p+s & r \\ 0 & r & q & 2s \end{pmatrix} = \tilde{f}'\,,$$

与我们在之前费力计算的雅可比矩阵$\tilde{f}'$完全匹配！

示例 28
对于矩阵立方函数 A^3，其中 A 是一个 $m \times m$ 方阵，计算向量化的函数 vec(A^3) 的 $m^2 \times m^2$ 雅可比。

让我们为矩阵-立方函数使用同样的技巧。当然，我们可以通过逐元素偏导数（在笔记本中通过符号计算做得很好）费力地计算雅可比矩阵，但使用克罗内克积要容易得多，也更优雅。回想一下，我们“非向量化的”矩阵微积分导数是线性算子：

$$(A^3)'[dA] = dA\,A^2 + A\,dA\,A + A^2\,dA,$$

现在通过三次应用克罗内克恒等式进行向量化：

$$\operatorname{vec}(dA\,A^2 + A\,dA\,A + A^2\,dA) = \underbrace{\left((A^2)^T \otimes \mathrm{I} + A^T \otimes A + \mathrm{I} \otimes A^2\right)}_{\text{vectorized Jacobian}} \operatorname{vec}(dX)\,.$$

您可以继续找到 A^4、A^5 以及等等的雅可比矩阵，或者任何矩阵幂的线性组合。实际上，您可以想象将类似的过程应用于任何（解析）矩阵函数 $f(A)$ 的泰勒级数，但这开始变得尴尬。稍后（以及在作业中），我们将讨论更优雅的微分其他矩阵函数的方法，不是作为向量化的雅可比矩阵，而是作为矩阵上的线性算子。

3.3.3 克罗内克积的计算成本

必须谨慎使用克罗内克积作为计算工具，而不仅仅是作为一种概念工具，因为它们很容易导致矩阵问题的计算成本远远超出必要的范围。

假设 A、B 和 C 都是 $m \times m$ 矩阵。两个 $m \times m$ 矩阵（通过常规方法）相乘的成本与 $\sim m^3$ 成正比，计算机科学家称之为 $\Theta(m^3)$ “复杂度”。因此，线性运算 $C \mapsto BCA^T$ 的成本与 $\sim m^3$ (两次 $m \times m$ 乘法) 成正比。然而，如果我们通过 vec(BCA^T) $= (A \otimes B)$ vec C 计算相同的答案，那么我们必须：

1. 构造 $m^2 \times m^2$ 矩阵 $A \otimes B$。这需要 m^4 次乘法（A 的所有项乘以 B 的所有项），以及 $\sim m^4$ 内存存储。（与存储 A 或 B 需要的 $\sim m^2$ 内存进行比较。如果 m 是 1000，这将是一百万倍更多的存储，是太字节而不是兆字节！）2. 将 $A \otimes B$ 乘以具有 m^2 个元素的向量 vec C。将 $n \times n$ 矩阵乘以一个向量需要 $\sim n^2$ 次操作，这里 $n = m^2$，所以这又是 $\sim m^4$ 次算术运算。

因此，而不是 $\sim m^3$ 操作和 $\sim m^2$ 存储来计算 BCA^T，使用 $(A \otimes B)$ vec C 需要 $\sim m^4$ 操作和 $\sim m^4$ 存储，差得多！本质上，这是因为 $A \otimes B$ 有很多结构我们没有利用（它是一个非常特殊的 $m^2 \times m^2$ 矩阵）。

存在许多此类例子。另一个著名的例子涉及求解线性矩阵方程 {v*}

$$AX + XB = C$$

对于一个未知的矩阵 X，给定 A, B, C，其中这些全都是 $m \times m$ 矩阵。这被称为“西尔维斯特方程”。这些是我们未知 X 中的线性方程，我们可以通过克罗内克积将它们转换为一个普通的 m^2 线性方程组：

$$\operatorname{vec}(AX + XB) = (\mathrm{I} \otimes A + B^T \otimes \mathrm{I}) \operatorname{vec} X = \operatorname{vec} C,$$

您可以使用高斯消元法求解未知向量 m^2 vec X 的未知量。但是，使用高斯消元法求解 $m^2 \times m^2$ 个方程组的成本是 $\sim (m^2)^3 = m^6$。然而，实际上存在聪明的算法，只需 $\sim m^3$ 次操作（使用 $\sim m^2$ 内存）即可求解 $AX + XB = C$ —对于 $m = 1000$，这节省了 10^9 (a billion) 的计算工作量。

克罗内克积可以成为稀疏矩阵（例如每行只有少数非零项的矩阵）更实用的计算工具：因为两个稀疏矩阵的克罗内克积仍然是稀疏的，避免了非稀疏“密集”矩阵克罗内克积的巨大存储需求。这对于组装大型稀疏方程组，如多维偏微分方程（PDEs）等，是一种方便的方法。

4 有限差分近似

在这个部分，我们将 l 将参考这个 Julia 笔记本进行计算 未在此处包含。

4.1 为什么近似计算导数而不是精确计算?

手动求导是一个众所周知容易出错的复杂函数过程。即使每个单独的步骤都很简单，仍然有很多机会出错，无论是在推导过程中还是在计算机上的实现中。每次实现导数时，都应该通过将其与独立计算进行比较来仔细检查错误。最简单的检查方法是有限差分近似，其中我们通过比较一个或多个“有限”（非无穷小）扰动 δx 的 $f(x)$ 和 $f(x+\delta x)$ 来估计导数（s）。

存在许多不同复杂程度的有限差分技术，我们将在下面讨论。它们都因为 δx 不是无穷小而引入内在截断误差。（我们还将看到，δx 也不能太小，否则舍入误差会爆炸！）此外，对于需要为每个输入维度计算完整雅可比矩阵) 的高维 x (，有限差分变得昂贵。这使得它们成为计算导数的最后手段。另一方面，它们通常是您首先采用的方法来检查导数：如果您在解析导数计算中有一个错误，通常答案是完全错误的，因此在高维) 中随机选择一个小的 δx (进行粗略的有限差分近似通常可以揭示问题。

另一种选择是自动微分（AD），软件/编译器为您执行解析导数。这非常可靠，并且，使用现代AD软件，可以非常高效。不幸的是，仍然有很多代码，例如调用其他语言中外部库的代码，AD工具无法理解。还有其他情况下AD效率低下，通常是因为它遗漏了问题的某些数学结构。即使在这样的情况下，您通常也可以通过手动定义程序的一小部分导数来修复AD，[2]这比微分整个程序要容易得多。在这种情况下，您仍然通常会想要进行有限差分检查，以确保您没有犯错误。

有限差分近似是一个出人意料复杂的主题，与许多数值分析领域有着丰富的联系；在本讲座中，我们仅将触及表面。

4.2 有限差分近似：简单版本

最简单检查导数的方法是回忆微分定义：

$$df = f(x+dx) - f(x) = f'(x)dx$$

来自 m 从一个小但有限的差分中去除高阶项 参考：

$$\delta f = f(x+\delta x) - f(x) = f'(x)\delta x + o(\|\delta x\|)\,.$$

因此，我们只需将有限差分 $f(x+\delta x) - f(x)$ 与我们的（方向）导数算子 $f'(x)\delta x$ (进行比较，即 δx) 方向的导数。$f(x+\delta x)-f(x)$ 也被称为前向差分近似。前向差分的反义词是后向差分近似 $f(x) - f(x-\delta x) \approx f'(x)\delta x$。如果你只想计算导数，前向和后向差分之间没有太多实际区别。

[2] In some Julia AD software, this is done with by defining a “ChainRule”, and in Python autograd/JAX it is done by defining a custom “vJp” (row-vector—Jacobian product) and/or “Jvp” (Jacobian–vector product).

当离散化（近似）微分方程时，这种区别变得更加重要。我们将在下面探讨其他可能性。

注意，这个前向和后向差分的定义与前向和后向模式微分不同——这些是无关的概念。

如果 x 是一个标量，我们也可以将两边都除以 δx 来得到 $f'(x)$ 的近似值，而不是 df 的近似值：

$$f'(x) \approx \frac{f(x+\delta x) - f(x)}{\delta x} + (\text{higher-order corrections}) .$$

这是一个更常见的正向差分近似写法，但它仅适用于标量 x，其中 s
在这个课程中，我们希望将 x 视为可能属于某个其他向量空间。

有限差分近似有多种形式，但它们通常是在无法计算出解析导数且AD失败的情况下的一种最后手段。但它们也有助于检查你的解析导数，并快速探索。

4.3 示例：矩阵平方

让我们尝试平方函数 $f(A) = A^2$ 的有限差分近似，其中这里 A 是 $\mathbb{R}^{m,m}$ 中的一个方阵。用手计算，我们得到乘积法则

$$df = A\,dA + dA\,A,$$

即 $f'(A)$ 是线性算子 $f'(A)[\delta A] = A\,\delta A + \delta A\,A.$ 这不等于 $2A\,\delta A$，因为在一般情况下 A 和 δA 不交换。所以让我们检查这个差值与有限差分。我们将对随机输入 A 和随机小的扰动 δA 进行尝试。

使用随机矩阵 A，设 $dA = A \cdot 10^{-8}$。然后，您可以比较 $f(A + dA) - f(A)$ 和 $A\,dA + dA\,A$。如果您选择的矩阵确实是随机的，您会发现从乘积规则得到的近似值与精确等式之差具有约 10^{-16} 的数量级！然而，与 $2AdA$ 相比，您将获得数量级为 10^{-8} 的项。

为了更量化，我们可能计算“范数”{v*} ≈ − exact||，我们希望它很小。但相对于什么来说很小？自然的答案是相对于正确答案来说很小。这被称为相对误差（或“分数误差”），并且通过以下方式计算：

$$\text{relative error} = \frac{\|\text{approx} - \text{exact}\|}{\|\text{exact}\|}.$$

这里，$\|\cdot\|$是一个范数，就像向量的长度。这使我们能够理解有限差分近似中的误差大小，即它允许我们回答这个近似有多准确（回忆第4.1节）。

因此，如上所述，您可以计算出近似值与精确答案之间的相对误差约为10^{-8}，而$2AdA$与精确答案之间的相对误差约为10^{0}。这表明我们的精确答案很可能是正确的！当然，随机输入与微小位移之间的良好匹配并不能证明正确性，但检查总是好事。这种随机比较几乎总是能捕捉到您在计算符号导数时犯的重大错误，就像我们在$2AdA$示例中那样。

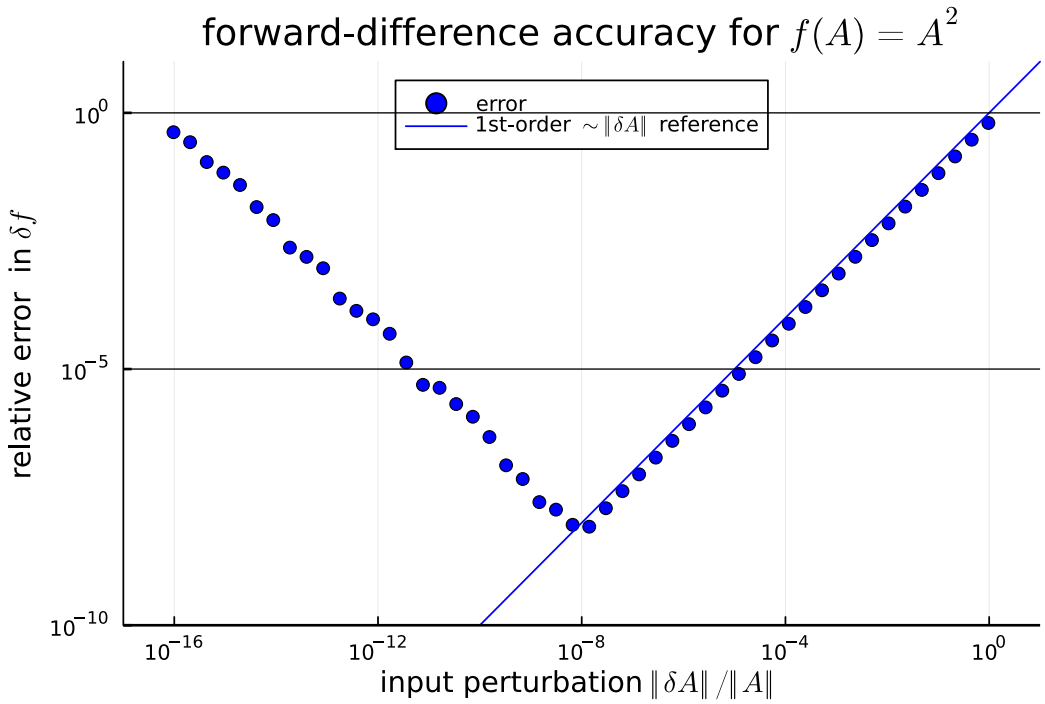


图4：$f(A) = A^2$的前向差分精度，显示$\delta f = f(A + \delta A) - f(A)$相对于线性化$f'(A)\delta A$的相对误差，作为幅度$\|\delta A\|$的函数。$A$是一个4×4矩阵，具有单位方差高斯随机项，而$\delta A$则是一个类似地具有单位方差高斯随机扰动，其缩放因子s在1到10^{-16}之间。

定义 30
请注意，我们使用的矩阵范数，在Julia中通过norm(A)计算，只是熟悉的欧几里得范数在向量情况下的直接类比。它仅仅是矩阵各项平方和的平方根：

$$\|A\| := \sqrt{\sum_{i,j} |A_{ij}|^2} = \sqrt{\mathrm{tr}(A^T A)}\,.$$

这被称为Frobenius范数。

4.4 有限差分的精度

现在我们上面的有限差分近似有多准确？我们应该如何选择 δx 的大小？让我们再次考虑示例 $f(A) = A^2$，并绘制相对误差作为 $\|\delta A\|$ 的函数图。这个图将以对数方式（对数-对数尺度）绘制，这样我们就可以看到幂律关系作为直线。

我们注意到随着 δA 的减小，有两个主要特征：

1. 首次相对误差随着 $\|\delta A\|$ 线性降低。这被称为一阶精度。为什么？
2. 当δA变得太小时，误差会增加。为什么？

4.5 精度阶

截断误差是由于输入扰动 δx 不是无穷小的事实而产生的误差：我们计算的是差分，而不是导数。如果导数的截断误差按 $\|\delta x\|^n$ 比例缩放，我们称该近似为 n 阶精确。对于前向差分，这里的阶数是 n=1。为什么?

对于任何具有非零二阶导数（想想泰勒级数）的 $f(x)$，我们有

$$f(x+\delta x) = f(x) + f'(x)\delta x + (\text{terms proportional to } \|\delta x\|^2) + \underbrace{o(\|\delta x\|^2)}_{\text{i.e. higher-order terms}}$$

这意味着我们在前向差分近似中省略的项与 $\|\delta x\|^2$ 成正比。但这意味着相对误差是线性的:

$$\begin{aligned}\text{relative error} &= \frac{\|f(x+\delta x) - f(x) - f'(x)\delta x\|}{\|f'(x)\delta x\|} \\ &= \frac{(\text{terms proportional to } \|\delta x\|^2) + o(\|\delta x\|^2)}{\text{proportional to } \|\delta x\|} = (\text{terms proportional to } \|\delta x\|) + o(\|\delta x\|)\end{aligned}$$

这是一阶精度。在有限差分近似中的截断误差是非无穷小 δx 公式中的固有误差。这意味着我们是否应该尽可能使 δx 尽可能小?

4.6 四舍五入误差

错误对于非常小的 δA 增加的原因是由于舍入误差。计算机只为每个实数存储有限个有效数字（大约15个十进制数字），并在每次操作中舍入其余部分——这被称为浮点运算。如果 δx 太小，那么差 $f(x+\delta x) - f(x)$ 就会被舍入为零（一些或所有有效数字相消）。这被称为灾难性消去。

浮点运算与科学记数法类似 $*.***** \times 10^e$：一个有限精度的系数 $*.*****$ 通过10的幂（或在计算机上，2的幂）进行缩放。系数中的数字位数（即“有效数字”）是“精度”，在通常的64位浮点运算中，由一个量 $\epsilon = 2^{-52} \approx 2.22 \times 10^{-16}$ 表示，称为机器精度。当一个任意实数 $y \in \mathbb{R}$ 被四舍五入到最接近的浮点值 $\tilde{y}$ 时，舍入误差被限制在 $|\tilde{y} - y| \le \epsilon|y|$。等价地，计算机只保留大约15-16 $\approx -\log_{10}\epsilon$ 十进制数字，或者实际上每个数字53 $= 1 - \log_2\epsilon$ 二进制数字。

在我们的有限差分示例中，对于大约$10^{-8} \approx \sqrt{\epsilon}\|A\|$或更大的$\|\delta A\|/\|A\|$，$f'(A)$的近似主要由截断误差主导，但如果我们小于这个值，由于舍入误差，相对误差开始增加。经验表明，$\|\delta x\| \approx \sqrt{\epsilon}\|x\|$通常是一个很好的经验法则——大约一半的有效数字是最合理的安全依据，但最小误差的确切交叉点取决于函数f和有限差分方法。但是，像所有经验法则一样，这并不总是完全可靠的。

4.7 其他有限差分法

存在更复杂的有限差分方法，例如Richardson外推法，它考虑一系列越来越小的δx值，以便自适应地确定对f' (进行外推到$\delta x \to 0$的最佳可能估计，使用越来越高的多项式)。还可以使用比简单前向差分法更高阶的差分公式，这样截断误差比线性地随着δx减小得更快。最著名的更高阶公式是“中心差分”$f'(x)\delta x \approx [f(x+\delta x) - f(x-\delta x)]/2$，它具有二阶精度（相对截断误差与$\|\delta x\|^2$成比例）。

高维输入 {v*} 对有限差分技术构成了基本的计算挑战，因为如果你想知道每个可能的方向 δx 发生了什么，那么你需要很多有限差分：每个 δx 的维度一个。例如，假设 $x \in \mathbb{R}^n$ 和 $f(x) \in \mathbb{R}$，因此你正在计算 $\nabla f \in \mathbb{R}^n$；如果你想知道整个梯度，你需要 n 个独立的有限差分。结果是，高维中的有限差分很昂贵，很快就会变得不切实际，尤其是在高维优化（例如神经网络）中，其中 n 可能非常大。另一方面，如果你只是将有限差分用作检查代码中错误的手段，通常只需要在几个随机方向上比较 $f(x + \delta x) - f(x)$ 和 $f'(x)[\delta x]$，即对于几个随机的小 δx。

5 一般向量空间中的导数

矩阵微积分要求我们将导数和梯度的概念进一步推广到输入和/或输出不是简单标量或列向量的函数。为了实现这一点，我们将普通向量点积和普通欧几里得向量“长度”的概念扩展到向量空间上的通用内积和范数。我们的第一个例子将从这一角度考虑熟悉的矩阵。

从线性代数中回忆起，我们可以称任何集合 {v*} 为“向量空间”，如果其元素可以加/减 $x \pm y$ 和由标量 αx (乘以，并且满足一些基本的算术公理，例如分配律)。例如，矩阵的集合 $m \times n$ 本身形成一个向量空间，或者甚至是连续函数的集合 $u(x)$ (映射 $\mathbb{R} \to \mathbb{R}$)——关键的事实是我们可以加/减/缩放它们并得到相同集合的元素。将微分扩展到这样的空间变得非常有用，例如对于将矩阵映射到矩阵或函数映射到数字的函数。这样做关键依赖于我们的输入/输出向量空间 V 具有范数，理想情况下，具有内积。

5.1 简单矩阵点积与范数

回忆一下，对于具有向量输入 $x \in \mathbb{R}^n$ (的标量值函数 $f(x) \in \mathbb{R}$，即 n 分量的“列向量”)
我们有{v*}

$$df = f(x + dx) - f(x) = f'(x)[dx] \in \mathbb{R}.$$

因此，$f'(x)$ 是一个线性算子，它接受向量 dx 并输出一个标量值。另一种看法是 $f'(x)$ 是行向量 [3] $(\nabla f)^T$。从这个角度来看，可以得出 df 是点积（或“内积”）：

$$df = \nabla f \cdot dx$$

我们可以将此推广到任何具有内积的向量空间 V! 给定 $x \in V$和一个标量值函数 f，我们得到线性算子 $f'(x)[dx] \in \mathbb{R}$，称为“线性形式”。为了定义梯度 ∇f，我们需要 V的内积，这是熟悉的点积的向量空间推广!

给定 $x, y \in V$，内积 $\langle x, y \rangle$ 是一个映射 $(\cdot)$，使得 $\langle x, y \rangle \in \mathbb{R}$。这通常也记作 $x \cdot y$ 或 $\langle x \mid y \rangle$。更技术地说，内积是一个映射，它是

1. 对称：即 $\langle x, y \rangle = \langle y, x \rangle$ (或共轭对称 [4] $\langle x, y \rangle = \overline{\langle y, x \rangle}$，如果我们使用复数)，
2. 线性：即 $\langle x, \alpha y + \beta z \rangle = \alpha \langle x, y \rangle + \beta \langle x, z \rangle$，和
3. 非负：即 $\langle x, x \rangle := \|x\|^2 \geq 0$，并且 $= 0$ 当且仅当 $x = 0$。

请注意，前两个属性的组合意味着它也必须在左向量（或如果我们使用复数，则为共轭线性）上是线性的。这三个属性的另一个有用后果，这是一个稍微复杂一些的推导，是柯西-施瓦茨不等式 $|\langle x, y \rangle| \leq \|x\| \, \|y\|$。

[3] The concept of a “row vector” can be formalized as something called a “covector,” a “dual vector,” or an element of a “dual space,” not to be confused with the *dual numbers* used in automatic differentiation (Sec. 8).

[4] Some authors distinguish the “dot product” from an “inner product” for complex vector spaces, saying that a dot product has no complex conjugation $x \cdot y = y \cdot x$ (in which case $x \cdot x$ need not be real and does not equal $\|x\|^2$), whereas the inner product must be conjugate-symmetric, via $\langle x, y \rangle = \bar{x} \cdot y$. Another source of confusion for complex vector spaces is that some fields of mathematics define $\langle x, y \rangle = x \cdot \bar{y}$, i.e. they conjugate the *right* argument instead of the left (so that it is linear in the left argument and conjugate-linear in the right argument). Aren’t you glad we’re sticking with real numbers?

定义 31（希尔伯特空间）
一个具有内积的（完备）向量空间被称为希尔伯特空间。（“完备性”的技术要求本质上意味着你可以在空间中取极限，这对于严格的证明很重要。[a]）

[a] 完整性意味着向量空间中任意一个柯西序列（即任意一个越来越接近的点序列）都有一个在向量空间内的极限。这个标准在实数或复数标量上的向量空间中通常成立，但当讨论函数向量空间时可能会变得复杂，例如，连续函数序列的极限可能是一个不连续的函数。

一旦我们有一个希尔伯特空间，我们就可以为标量值函数定义梯度。给定 $x \in V$一个希尔伯特空间，和 $f(x)$ 标量，那么我们就有线性形式 $f'(x)[dx] \in \mathbb{R}$。然后，在这些假设下，有一个被称为“里兹表示定理”的定理，表明任何线性形式（包括 f'）都必须与某个东西的内积：

$$f'(x)[dx] = \Big\langle \underbrace{(\text{some vector})}_{\text{gradient } \nabla f\big|_x}, dx \Big\rangle = df.$$

这意味着梯度 ∇f 被定义为与 dx 取内积以得到 df 的那个东西。请注意，∇f 总是与 x 具有相同的形状。

我们考察的前几个例子涉及通常的希尔伯特空间 $V = \mathbb{R}^n$ 以及不同的内积。

示例 32
给定 $V = \mathbb{R}^n$ 与 n-列向量，我们有熟悉的欧几里得点积 $\langle x, y\rangle = x^T y$。这导致通常的 ∇f。

示例 33
我们可以有 $\mathbb{R}^n$ 上的不同内积。例如，

$$\langle x, y\rangle_W = w_1 x_1 y_1 + w_2 x_2 y_2 + \ldots w_n x_n y_n = x^T \underbrace{\begin{pmatrix} w_1 & & \\ & \ddots & \\ & & w_n \end{pmatrix}}_{W} y$$

对于权重 $w_1, \ldots, w_n > 0$.

更一般地，我们可以为任何对称正定矩阵 W ($W = W^T$ 定义一个加权点积 $\langle x, y\rangle_W = x^T W y$，并且 W 是正定的，这对于它成为一个有效的内积) 是足够的。

如果我们改变内积的定义，那么我们也改变了梯度的定义！例如，使用 $f(x) = x^T A x$ 我们之前发现 $df = x^T(A + A^T)dx$。在普通的欧几里得内积下，这给出了一个梯度 $\nabla f = (A + A^T)x$。然而，如果我们使用加权内积 $x^T W y$，那么我们将获得一个不同的“梯度” $\nabla^{(W)} f = W^{-1}(A + A^T)x$ 以便 $df = \langle \nabla^{(W)} f, dx\rangle$。

在这些笔记中，我们将使用欧几里得内积来表示 $x \in \mathbb{R}^n$，因此通常表示为 ∇f，除非另有说明。然而，加权内积在许多情况下很有用，尤其是在 x 的分量具有不同的尺度/单位时。

我们也可以考虑 $m \times n$ 矩阵的空间 $V = \mathbb{R}^{m \times n}$。在那里，当然存在从 $V \ni A \to \text{vec}(A) \in \mathbb{R}^{mn}$ 到向量空间的同构。因此，在这个空间中，我们有熟悉的（“Frobenius”）欧几里得内积的类似物，这可以通过迹来方便地用矩阵运算来重写：

定义34（Frobenius内积）
两个 $m \times n$ 矩阵 A 和 B 的Frobenius内积是：

$$\langle A, B\rangle_F = \sum_{ij} A_{ij} B_{ij} = \mathrm{vec}(A)^T \mathrm{vec}(B) = \mathrm{tr}(A^T B)\,.$$

给定这个内积，我们也有相应的Frobenius范数：

$$\|A\|_F = \sqrt{\langle A, A\rangle_F} = \sqrt{\mathrm{tr}(A^T A)} = \|\mathrm{vec} A\| = \sqrt{\sum_{i,j} |A_{ij}|^2}\,.$$

使用这个，我们现在可以定义具有矩阵输入的标量函数的梯度！这将是这些笔记中的默认矩阵内积（因此定义了我们的默认矩阵梯度）（有时省略F下标）。

示例 35
考虑函数

$$f(A) = \|A\|_F = \sqrt{\mathrm{tr}(A^T A)}.$$

什么是 *df*?

首先，通过熟悉的标量微分链和幂规则，我们有

$$df = \frac{1}{2\sqrt{\mathrm{tr}(A^T A)}} d(\mathrm{tr}\, A^T A).$$

然后，注意（由迹的线性性质）{v*}

$$d(\mathrm{tr}\, B) = \mathrm{tr}(B + dB) - \mathrm{tr}(B) = \mathrm{tr}(B) + \mathrm{tr}(dB) - \mathrm{tr}(B) = \mathrm{tr}(dB).$$

因此，

$$\begin{aligned} df &= \frac{1}{2\|A\|_F} \mathrm{tr}(d(A^T A)) \\ &= \frac{1}{2\|A\|_F} \mathrm{tr}(dA^T\, A + A^T\, dA) \\ &= \frac{1}{2\|A\|_F} (\mathrm{tr}(dA^T\, A) + \mathrm{tr}(A^T\, dA)) \\ &= \frac{1}{\|A\|_F} \mathrm{tr}(A^T\, dA) = \Big\langle \frac{A}{\|A\|_F}, dA \Big\rangle. \end{aligned}$$

这里，我们使用了 tr $B =$ tr B^T 的性质，并在最后一步将 *df* 与 Frobenius 内积连接起来。换句话说，

$$\nabla f = \nabla \|A\|_F = \frac{A}{\|A\|_F}.$$

请注意，对于列向量 x，即 $\nabla \|x\| = x/\|x\|$ (，可以得到完全相同的结果，实际上这通过 $x =$ vec A) 是等价的。

让我们考虑另一个简单的例子：

示例 36
修复一些常数 $x \in \mathbb{R}^m$，$y \in \mathbb{R}^n$，并考虑函数 f：$\mathbb{R}^{m\times n} \to \mathbb{R}$，由以下公式给出

$$f(A) = x^T A y.$$

什么是 ∇f?

我们有以下结果

$$\begin{aligned} df &= x^T\, dA\, y \\ &= \operatorname{tr}(x^T\, dA\, y) \\ &= \operatorname{tr}(yx^T\, dA) \\ &= \langle \underbrace{xy^T}_{\nabla f}, dA\rangle. \end{aligned}$$

更一般地，对于任何标量值函数 $f(A)$，根据Frobenius内积的定义，可以得出：

$$df = f(A + dA) - f(A) = \langle \nabla f, dA\rangle = \sum_{i,j} (\nabla f)_{i,j}\, dA_{i,j}\,,$$

因此，梯度的分量恰好是逐元素导数

$$(\nabla f)_{i,j} = \frac{\partial f}{\partial A_{i,j}}\,,$$

类似于多元微积分中梯度向量的分量定义！但对于非平凡矩阵输入函数 $f(A)$，对 A 的每个元素分别求导可能非常尴尬。使用“整体”的矩阵内积定义，我们很快就能计算更复杂的矩阵值梯度，包括 $\nabla(\det A)$!

5.2 导数、范数和Banach空间

我们一直在本课程中使用“范数”这个术语，但技术上什么是范数呢? 当然，有熟悉的例子，例如欧几里得范数（“ℓ^2”）$\|x\| = \sqrt{\sum_k x_k^2}$ 对于 $x \in \mathbb{R}^n$，但考虑这个概念如何推广到其他向量空间是有用的。事实上，范数对于导数的定义至关重要!

给定一个向量空间 V，V 上的范数 $\|\cdot\|$ 是一个满足以下三个性质的映射 $\|\cdot\|$：$V \to \mathbb{R}$：

1. 非负：即 $\|v\| \geq 0$ 和 $\|v\| = 0 \iff v = 0$,
2. 均匀性：对于任何 $\|\alpha v\| = |\alpha|\|v\|$，和
3. 三角不等式：$\|u + v\| \leq \|u\| + \|v\|$。

一个具有范数的向量空间被称为赋范向量空间。通常，数学家在技术上希望有一种稍微更精确的赋范向量空间类型，但名称不那么明显：巴拿赫空间。

定义37（Banach空间）
一个带有范数的（完备）向量空间称为Banach空间。（与Hilbert空间一样，“完备性”是某些类型严格分析的技术要求，本质上允许你取极限。）

例如，给定任何内积 $\langle u, v\rangle$，存在相应的范数 $\|u\| = \sqrt{\langle u, u\rangle}$。（因此，每个希尔伯特空间也是巴拿赫空间.[5]）

为了定义导数，我们在技术上需要输入和输出都是Banach空间。为了看到这 s,
回忆我们的公理化体系

$$f(x + \delta x) - f(x) = \underbrace{f'(x)[\delta x]}_{\text{linear}} + \underbrace{o(\delta x)}_{\text{smaller}} .$$

为了精确地定义 $o(\delta x)$ 项在“较小”或“高阶”意义上的含义，我们需要规范。特别是 r,
“小-o”表示法 $o(\delta x)$ 表示任何满足以下条件的函数

$$\lim_{\delta x \to 0} \frac{\|o(\delta x)\|}{\|\delta x\|} = 0 ,$$

即，它比线性更快地趋于零在 δx 中。这要求输入 δx 和输出（函数）都必须有范数。这种将微分扩展到任意赋范/Banach空间的做法有时被称为弗雷歇导数。

[5]Proving the triangle inequality for an arbitrary inner product is not so obvious; one uses a result called the Cauchy–Schwarz inequality.

6 非线性根查找、优化和伴随微分

下一部分基于这些幻灯片。今天，我们想谈谈为什么我们在fir中计算导数。 st
位置。特别是，我们将对此进行一点深入探讨，然后讨论导数的计算。

6.1 牛顿法

一个常见的应用 导数的应用是通过对线性方程进行求解来解决非线性方程r化。

6.1.1 矢量函数

例如，假设我们有一个标量函数 $f\colon \mathbb{R} \to \mathbb{R}$，并且我们想要求解 $f(x) = 0$ 的根 x。当然，在简单的情况下，例如当 f 是线性或二次的，我们可以显式地求解这样的方程，但如果函数是像 $f(x) = x^3 - \sin(\cos x)$ 这样更任意的函数，你可能无法得到封闭形式的解。然而，有一种方法可以近似地获得任何你想要的精度，只要你知道根的大致位置。我们正在讨论的方法被称为牛顿法，它实际上是一种线性代数技术。它接受一个函数和一个根的猜测，通过一条直线（其根容易找到）来近似它，然后这个近似根就可以用作新的猜测。特别是，该方法（如图5所示）如下：

- 线性化 $f(x)$ 在某些 x 附近使用近似

$$f(x + \delta x) \approx f(x) + f'(x)\delta x,$$

- 解线性方程 $f(x) + f'(x)\delta x = 0 \implies \delta x = -\frac{f(x)}{f'(x)}$，
- 然后使用此来更新我们近似的 x 的值——即，让新的 x 为

$$x_{\text{new}} = x - \delta x = x + \frac{f(x)}{f'(x)}\,.$$

一旦接近根，牛顿法收敛得非常快。如下所述，它是渐近的 ly
将每一步的正确数字数量翻倍！

可以问当 $f'(x)$ 不可逆时会发生什么，例如这里 $f'(x) =$ 为 0。如果发生这种情况，那么牛顿法可能会失效！请参见牛顿法失效的例子。

6.1.2 多维函数

我们可以将牛顿法推广到多维函数！设 $f\colon \mathbb{R}^n \to \mathbb{R}^n$ 是一个函数，它接收一个向量并输出一个同样大小的向量 n。然后我们可以在更高维度应用牛顿方法：

- 线性化 $f(x)$ 在某些 x 附近使用一阶导数近似

$$f(x + \delta x) \approx f(x) + \underbrace{f'(x)}_{\text{Jacobian}}\ \delta x,$$

- 解线性方程 $f(x) + f'(x)\delta x = 0 \implies \delta x = -\underbrace{f'(x)^{-1}}_{\text{inverse Jacobian}}\ f(x)$,

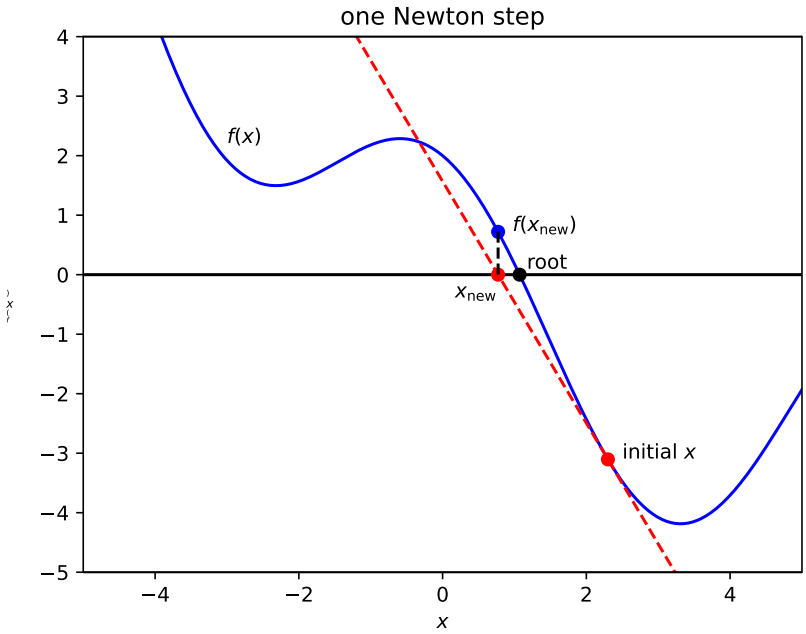


图5：求解非线性函数$f(x) = 2\cos(x) - x + x^2/10$的标量牛顿法单步。给定一个起始猜测($x = 2.3$在此例中)，我们使用$f(x)$和$f'(x)$来形成$f$的线性（仿射）近似，然后我们的下一步$x_{\text{new}}$是这个近似的根。只要初始猜测不是离根太远，牛顿法就会非常迅速地收敛到精确根（黑色点）。

- 然后使用此来更新我们近似的 x 的值，即让新的 x 为

$$x_{\text{new}} = x_{\text{old}} - f'(x)^{-1} f(x)\,.$$

这就是了！一旦我们有了雅可比矩阵，我们就可以在每一步解一个线性系统。这又以惊人的速度收敛，每一步的精度数字翻倍。（这被称为“二次收敛”。）然而，有一个注意事项：我们需要一个关于 x 的起始猜测，并且这个猜测需要足够接近根，以便算法能够可靠地进步。（如果你从一个远离根的初始 x 开始，牛顿法可能无法收敛，或者它可能会以复杂和令人惊讶的方式跳跃——通过谷歌搜索“牛顿分形”可以找到一些迷人的例子。）这是雅可比矩阵和导数的一个广泛使用且非常实用的应用！

6.2 优化

6.2.1 非线性优化

大规模微分的可能更著名的应用是非线性优化。假设我们有一个标量值函数 $f\colon \mathbb{R}^n \to \mathbb{R}$，并且假设我们想要最小化（或最大化）$f$。例如，在机器学习中，我们可能有一个包含一百万个参数的大神经网络（NN），并且试图最小化一个“损失”函数 f，该函数比较NN输出与“训练”数据上的期望结果。优化的最基本思想是“向下走坡”（见图表）以使 f 尽可能小。如果我们能取这个函数 f 的梯度，为了“向下走坡”，我们考虑 $-\nabla f$，即最陡下降的方向，如图6所示。

然后，即使我们有百万个参数，我们也可以同时将它们向下山方向进化。结果是计算所有百万个导数所需的成本与在某个点评估函数所需的成本大致相同。

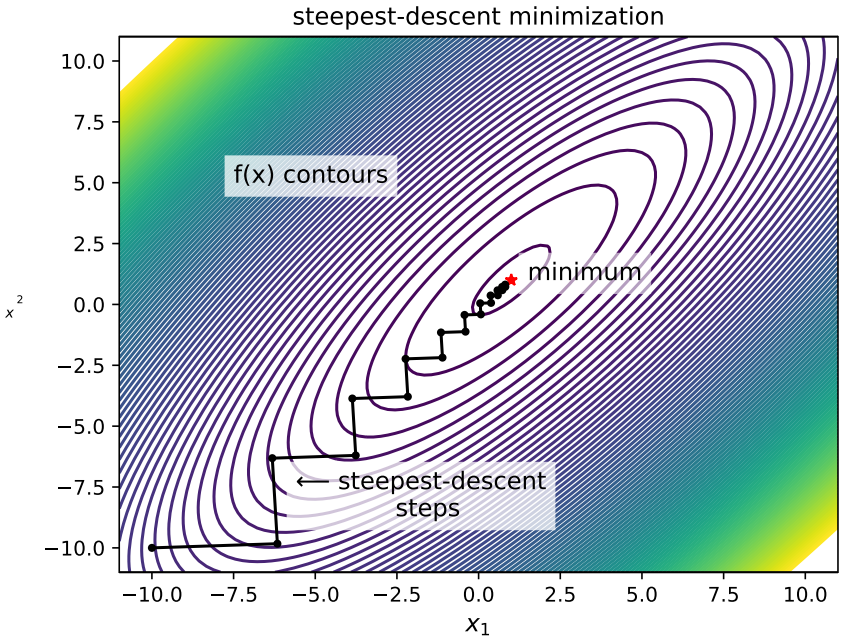


图6：最速下降算法通过在方向$-\nabla f$上采取连续的“下山”步骤来最小化函数$f(x)$。（在此例中，我们最小化的是二维空间中的二次函数$x \in \mathbb{R}^2$，对每个步骤在下山方向进行精确的1维最小化。）最速下降算法有时会在狭窄的山谷中“之字形”前进，从而减慢收敛速度（这可以通过更复杂的算法中的“动量”项、二阶导数信息等来抵消）。

一次点（使用反向模式/伴随/从左到右/反向传播方法）。最终，这使得大规模优化对于训练神经网络、优化飞机机翼形状、优化投资组合等成为可能。

当然，有许多实际复杂性使得非线性优化变得棘手（远超单个讲座或整个课程所能涵盖的内容！），但我们在这里给出一些例子。

- 例如，尽管我们可以计算“下山方向”，但我们需要在这个方向上走多远？（在机器学习中，这有时被称为“学习率”。）通常，你希望“尽可能大地迈出一步”以加快收敛，但你又不希望步子迈得太大，因为 ∇f 只告诉你 f 的局部近似。关于如何确定这一点，有许多不同的想法：

  - 线搜索：使用一维最小化来确定步长。

  – 一个“信任域”，限制步长（在这里我们信任基于导数的 f 近似）。随着优化的进行，有许多技术可以演化信任域的大小。

- 我们可能还需要考虑约束，例如在 $g_k(x) \leq 0$ 或 $h_k(x) = 0$ 的条件下最小化 $f(x)$，这被称为不等式/等式约束。满足约束的点 x 被称为“可行点”。通常，人们会结合使用 ∇f 和 ∇g_k 来近似（例如线性化）问题，并朝着最佳可行点迈进。

- 如果您直接向下走，可能会在狭窄的山谷中“之字形”前进，使收敛非常缓慢。有几个选项可以对抗这种情况，例如“动量”项和共轭梯度。甚至比这些技术更复杂的是，可以从一系列 ∇f 值中估计二阶导数“海森矩阵”——这种方法的著名版本被称为BFGS算法——并使用海森矩阵来近似牛顿

步骤（对于根 $\nabla f = 0$）。（我们将在以后的讲座中回到Hessian。）

- 最终，有许多技术和一大群相互竞争的算法，你可能需要尝试以找到给定问题的最佳方法。（关于优化算法有很多书籍，甚至整本书也只能涵盖其中的一小部分！）

一些告别建议：通常，主要技巧与其说是算法的选择，不如说是找到你问题的正确数学公式——例如，应该考虑什么函数、什么约束和什么参数——以将你的问题与一个好的算法相匹配。然而，如果你有很多（$\gg 10$）参数，请尽力使用分析梯度（而不是有限差分），在反向模式中高效计算。

6.2.2 工程物理优化

优化除了机器学习（将模型拟合到数据）之外还有许多许多应用。考虑工程/物理优化也很有趣。（例如，假设你想制造一个尽可能坚固的飞机机翼。）这类问题的一般轮廓通常是：

1. 您从一些设计参数 $\mathbf{p}$ 开始，例如描述几何形状、材料、力或其他自由度。2. 这些 $\mathbf{p}$ 然后被用于某些物理模型（如固体力学、化学反应、热传输、电磁学、声学等）。例如，您可能有一个形式为 $A(\mathbf{p})x = b(\mathbf{p})$ 的线性模型，对于某些矩阵 A (通常非常大且稀疏)。3. 物理模型的解是一个解 $x(\mathbf{p})$。例如，这可能包括机械应力、化学浓度、温度、电磁场等。4. 物理解 $x(\mathbf{p})$ 是您想要改进/优化的某些设计目标 $f(x(\mathbf{p}))$ 的输入。例如，强度、速度、功率、效率等。5. 为了最大化/最小化 $f(x(\mathbf{p}))$，使用反向模式/“伴随”方法计算的梯度 $\nabla_{\mathbf{p}} f$ 来更新参数 $\mathbf{p}$ 并改进设计。

作为一个有趣的例子，研究人员甚至将“拓扑优化”应用于设计椅子，优化设计的每个体素——参数 $\mathbf{p}$ 代表每个体素中存在的（或不存在）材料，以便优化不仅发现最佳形状，还发现最佳拓扑（材料在空间中的连接方式，孔的数量等等）——以最小材料支撑给定的重量。要看到它的实际应用，请观看这个椅子优化视频。（人们还将此类技术应用于许多更实际的问题，从飞机机翼到光通信。）

6.3 反向模式“伴随”微分

但是伴随微分是什么——使这些应用实际上可行求解的微分方法? 最终，它又是左到右/反向模式微分的一个例子，本质上是从输出到输入应用链式法则。例如，考虑尝试计算标量值函数的梯度 ∇g。

$$g(p) = f(\underbrace{A(p)^{-1}b}_{x}).$$

在 x 解 $A(p)x = b$ (的地方，例如，如前节所述的参数化物理模型) 和 $f(x)$ 是 x (的标量值函数，例如，一个依赖于我们的物理解的优化目标)。例如，这可能在

一个优化问题

$$\min_p g(p) \iff \min_{\text{subject to } A(p)x=b} f(x),$$

对于其中梯度 ∇g 将有助于寻找局部最小值。g 的链式法则对应以下概念依赖链：

$$\begin{aligned}\text{change } dg \text{ in } g &\longleftarrow \text{change } dx \text{ in } x = A^{-1}b \\ &\longleftarrow \text{change } d(A^{-1}) \text{ in } A^{-1} \\ &\longleftarrow \text{change } dA \text{ in } A(p) \\ &\longleftarrow \text{change } dp \text{ in } p\end{aligned}$$

这是由以下方程表示的：

$$\begin{aligned} dg &= f'(x)[dx] && dg \longleftarrow dx \\ &= f'(x)[d(A^{-1})b] && dx \longleftarrow d(A^{-1}) \\ &= -\underbrace{f'(x)A^{-1}}_{v^T} dA\, A^{-1}b && dA^{-1} \longleftarrow dA \\ &= -v^T \underbrace{A'(p)[dp]}_{dA} A^{-1}b && dA \longleftarrow dp\,. \end{aligned}$$

这里，我们定义了行向量 $v^T = f'(x)A^{-1}$，并且使用了第7.3节中的矩阵逆的微分 $d(A^{-1}) = -A^{-1}\, dA\, A^{-1}$。

将项从左到右分组，我们首先解“伴随”方程 $A^T v = f'(x)^T = \nabla_x f$ 对 v，然后得到 $dg = -v^T dA\, x$。因为矩阵相对于向量的导数 $A'(p)$ 显式地写起来很麻烦，所以逐个参数检查这个对象是方便的。对于任何给定的参数 p_k，$\partial g/\partial p_k = -v^T(\partial A/\partial p_k)x$ (以及在许多应用中 $\partial A/\partial p_k$ 非常稀疏)；在这里，“除以” ∂p_k 是可行的，因为这是一个与其它线性运算交换的标量因子。也就是说，只需两次求解就能得到 g 和 ∇g：一次是求解 $Ax = b$ 以找到 $g(p) = f(x)$，另一次是使用 A^T 对 v 进行求解，之后所有的导数 $\partial g/\partial p_k$ 只是些廉价的点积。

请注意，您不应使用具有许多参数的从右到左“正向模式”导数，因为

$$\frac{\partial g}{\partial p_k} = -f'(x)\left(A^{-1}\frac{\partial A}{\partial p_k}x\right)$$

表示每个参数 p_k 一次求解！如第8.4节所述，当有一个（或少数）输入参数 p_k 和许多输出时，从右到左（即正向模式）更好，而当有一个（或少数）输出值和许多输入参数时，从左到右“伴随”微分（即反向模式）更好。（在第8.1节中，我们将讨论使用双数进行微分，这也对应于正向模式。）

另一种可能的想法是使用有限差分法（如第4节所述），但如果您有很多参数，则不应使用这种方法！有限差分法将涉及计算类似以下内容的东西

$$\frac{\partial g}{\partial p_k} \approx [g(p + \epsilon e_k) - g(p)]/\epsilon,$$

在 k-th 方向上是单位向量，ϵ 是一个小数。然而，这需要为每个参数 p_k 求解，就像前向模式微分一样。（如果你使用更高级的更高阶有限差分近似来获得更高的精度，这会变得更加昂贵。）

6.3.1 Nonlinear equations

您也可以将伴随/反向微分应用于非线性方程。例如，考虑标量函数 $g(p) = f(x(p))$ 的梯度，其中 $x(p) \in \mathbb{R}^n$ 解决某个 n 个方程组 $h(p, x) = 0 \in \mathbb{R}^n$。根据链式法则，

$$h(p,x) = 0 \implies \frac{\partial h}{\partial p}dp + \frac{\partial h}{\partial x}dx = 0 \implies dx = -\left(\frac{\partial h}{\partial x}\right)^{-1}\frac{\partial h}{\partial p}dp\,.$$

(这是一个隐函数定理的例子：只要 $\frac{\partial h}{\partial x}$ 是非奇异的，我们就可以在局部从隐函数方程 $h = 0$ 定义一个函数 $x(p)$，这里是通过线性化。因此，

$$dg = f'(x)dx = -\underbrace{f'(x)\left(\frac{\partial h}{\partial x}\right)^{-1}}_{v^T}\frac{\partial h}{\partial p}dp\,.$$

将左到右再次关联会导致一个单一的“伴随”方程：$(\partial h/\partial x)^T v = f'(x)^T = \nabla_x f$。换句话说，它再次只需要两次求解就能得到$g$和$\nabla g$——一次非线性“正向”求解$x$和一次线性“伴随”求解$v$！此后，所有导数$\partial g/\partial p_k$都是廉价的点积。（注意，线性“伴随”求解涉及转置雅可比矩阵$\partial h/\partial x$。除了转置之外，这与求解$h = 0$以获得x的单次牛顿步的成本非常相似。因此，伴随问题应该比正向问题便宜。）

6.3.2 伴随方法和AD

如果您使用自动微分（AD）系统，为什么还需要学习这些内容？AD难道不是已经为您做了一切吗？然而，在实践中，即使您使用自动微分，了解伴随方法也是有帮助的。首先，它有助于您理解何时使用前向模式与反向模式自动微分。其次，许多物理模型调用了几十年间用各种语言编写的庞大软件包，这些软件包无法通过AD自动微分。您通常可以通过仅为此物理现象提供“向量-雅可比乘积”$y^T dx$来纠正这一点，或者甚至只是部分物理现象，然后AD将微分其余部分并为您应用链式法则。最后，通常模型涉及近似计算（例如，用于线性或非线性方程的迭代求解、数值积分等），但AD工具通常“不知道”这一点，并额外努力尝试微分近似中的误差；在这种情况下，手动编写的导数规则有时可以更有效率。例如，假设您的模型涉及通过牛顿法等迭代方法求解非线性系统$h(x, p) = 0$。简单的AD将非常低效，因为它将尝试通过您的所有牛顿步骤进行微分。假设您将牛顿求解器收敛到足够的精度，以至于误差可以忽略不计，那么通过上述隐函数定理进行微分要高效得多，从而只需进行一次线性伴随求解。

6.3.3 伴随方法示例

要完成笔记的这一部分，我们用一个示例来说明如何高效地使用这种“伴随方法”来计算导数。在分析示例之前，我们首先陈述问题，并强烈建议在阅读解决方案之前尝试一下。

问题 38

假设 $A(p)$ 接收一个向量 $p \in \mathbb{R}^{n-1}$ 并返回一个 $n \times n$ 三对角实对称矩阵

$$A(p) = \begin{pmatrix} a_1 & p_1 & & & \\ p_1 & a_2 & p_2 & & \\ & p_2 & \ddots & \ddots & \\ & & \ddots & a_{n-1} & p_{n-1} \\ & & & p_{n-1} & a_n \end{pmatrix},$$

在 $a \in \mathbb{R}^n$ 是某个常量向量的情况下。现在，定义一个标量值函数 $f(p)$ 如下：

$$g(p) = \left(c^T A(p)^{-1} b\right)^2$$

对于某些常数向量 $b, c \in \mathbb{R}^n$（，假设我们选择 p 和 a 使得 A 是可逆的）。请注意，在实践中，$A(p)^{-1}b$ 不是通过显式求逆矩阵 A 来计算的——相反，它可以在 $\Theta(n)$（中计算，即大致与 n）算术运算成比例，使用利用“稀疏性”的 A（零值项模式）的高斯消元法，即“三对角求解”。

(a) 写出一个公式，用矩阵-向量乘法和矩阵逆来计算 {v*}。(提示：一旦你知道 dg 与 dA 的关系，你可以通过“除以” ∂p_1 两边来得到 $\partial g/\partial p_1$，使得 dA 成为 $\partial A/\partial p_1$。) (b) 概述一个计算 g 和 ∇g（的步骤序列，相对于 p），只使用两个三对角解 $x = A^{-1}b$ 和一个“伴随”解 $v = A^{-1}$(，以及 $\Theta(n)$（，即大致与 n）相当的额外算术运算。(c) 编写一个程序实现你之前部分中的 ∇g 程序（在 Julia、Python、Matlab 或任何你想要的编程语言中）。(如果你不知道如何在你的语言中实现这个，你不需要使用花哨的三对角解；如果需要，你可以使用你喜欢的矩阵库以低效的方式解决 A^{-1}(向量）。) 实现一个有限差分测试：随机选择 a, b, c, p，并检查对于随机选择的小 δp，$\nabla g \cdot \delta p \approx g(p + \delta p) - g(p)$（到几位数字）。

问题38(a) 解答：从链式法则和矩阵逆的微分公式，我们有 $dg = -\text{^}2(c^T A^{-1}b)c^T A^{-1} dA\, A^{-1}b$（，注意到 $c^T A^{-1}b$ 是一个标量，因此我们可以按需交换它）。因此

$$\frac{\partial g}{\partial p_1} = \underbrace{-2(c^T A^{-1}b)c^T A^{-1}}_{v^T} \frac{\partial A}{\partial p_1} \underbrace{A^{-1}b}_{x}$$

$$= v^T \underbrace{\begin{pmatrix} 0 & 1 & & & \\ 1 & 0 & 0 & & \\ & 0 & \ddots & \ddots & \\ & & \ddots & 0 & 0 \\ & & & 0 & 0 \end{pmatrix}}_{\frac{\partial A}{\partial p_1}} x = \boxed{v_1 x_2 + v_2 x_1},$$

在下一部分中，我们已将结果简化为 x 和 v。

问题38(b) 解答：使用前一部分的符号，利用事实 $A^T = A$，我们

可以选择 $v = A^{-1}[-2(c^T x)c]$，这是一个单三对角求解。给定 x 和 v，我们两个 $\Theta(n)$ 三对角求解的结果，可以通过 $\partial g/\partial p_k = v_k x_{k+1} + v_{k+1} x_k$ 对 $k = 1, \ldots, n-1$ 进行类似计算，每个分量的梯度，这需要 $\Theta(1)$ 次算术运算，因此总共需要 $\Theta(n)$ 次算术运算来获得所有 ∇g。

问题38(c) 解答：参见我们IAP 2023课程的Julia解决方案笔记本（问题1）（其中调用函数f而不是g）。

7 矩阵行列式和逆的导数

7.1 两个推导

本节笔记遵循此Julia笔记本。这个笔记本有点短，但很重要且很有用 无序列表 计算。

定理39
给定 A 是一个方阵，我们有

$$\nabla(\det A) = \text{cofactor}(A) = (\det A)A^{-T} := \text{adj}(A^T) = \text{adj}(A)^T$$

adj是“伴随矩阵”。（你可能没有听说过矩阵的伴随矩阵，但这个公式告诉我们它仅仅是 adj(A) = det(A)A^{-1}，或者 cofactor(A) = adj(A^T)。）。此外，

$$d(\det A) = \text{tr}(\det(A)A^{-1}dA) = \text{tr}(\text{adj}(A)dA) = \text{tr}(\text{cofactor}(A)^T dA).$$

您可能记得，余子式矩阵中的每个元素(i, j)是删除A中的第i行和第j列后得到的行列式的$^{i+j}$−1倍。以下是一些2×2的计算，以获得对这些函数的直观理解：

$$M = \begin{pmatrix} a & c \\ b & d \end{pmatrix} \tag{4}$$

$$\implies \text{cofactor}(M) = \begin{pmatrix} d & -c \\ -b & a \end{pmatrix} \tag{5}$$

$$\text{adj}(M) = \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} \tag{6}$$

$$(M)^{-1} = \frac{1}{ad-bc}\begin{pmatrix} d & -b \\ -c & a \end{pmatrix}. \tag{7}$$

数值上，正如笔记本中所做的那样，你可以构造一个随机 $n \times n$ 矩阵 A (，例如，9×9)，例如考虑 dA = .000 01A，并数值上看到

$$\det(A + dA) - \det(A) \approx \text{tr}(\text{adj}(A)dA),$$

该文本数值上支持我们对该定理的断言。

我们现在以两种方式证明该定理。首先，存在一种直接证明方法，只需对每个输入使用基于 i-行的行列式余子式展开对标量进行微分。回忆一下

$$\det(A) = A_{i1}C_{i1} + A_{i2}C_{i2} + \cdots + A_{in}C_{in}.$$

Thus,Thus,

$$\frac{\partial \det A}{\partial A_{ij}} = C_{ij} \implies \nabla(\det A) = C,$$

系数矩阵。 (在计算这些偏导数时，重要的是要记住系数 C_{ij} 不包含来自行 i 或列 j 的 A 的任何元素。因此，例如，A_{i1} 只在第一个项中明确出现，而不隐藏在这个展开的任何 C 项中。)

定理还有一个使用近似的线性化证明，该证明更为复杂。首先，注意到从行列式的性质中很容易看出

$$\det(I + dA) - 1 = \operatorname{tr}(dA),$$

因此

$$\begin{aligned}\det(A + A(A^{-1}dA)) - \det(A) &= \det(A)(\det(I + A^{-1}dA) - 1) \\ &= \det(A)\operatorname{tr}(A^{-1}dA) = \operatorname{tr}(\det(A)A^{-1}dA) \\ &= \operatorname{tr}(\operatorname{adj}(A)dA).\end{aligned}$$

这也意味着该定理。

7.2 应用程序

7.2.1 特征多项式

我们现在将此用作一个应用来找到在 x 处评估的特征多项式的导数。令 $p(x) = \det(xI - A)$，是 x 的一个标量函数。回忆一下，通过因式分解，$p(x)$ 可以用特征值 λ_i 表示。因此，我们可以问：$p(x)$ 的导数是什么，即 x 处的特征多项式？使用大学一年级微积分，我们可以简单地计算

$$\frac{d}{dx}\prod_i (x - \lambda_i) = \sum_i \prod_{j \neq i} (x - \lambda_j) = \prod (x - \lambda_i)\{\sum_i (x - \lambda_i)^{-1}\},$$

只要 $x \neq \lambda_i$。

这是一个完美的简单证明，但有了我们新的技术，我们有一个新的证明：

$$\begin{aligned}d(\det(xI - A)) &= \det(xI - A)\operatorname{tr}((xI - A)^{-1}d(xI - A)) \\ &= \det(xI - A)\operatorname{tr}(xI - A)^{-1}dx.\end{aligned}$$

请注意，在此我们使用了当 A 为常数时 $d(xI - A) = dx\, I$，以及 $\operatorname{tr}(Adx) = \operatorname{tr}(A)dx$，因为 dx 是一个标量。

我们可能再次像在笔记本中那样进行计算来检查这一点。

7.2.2 对数导数

我们可以类似地使用链式法则计算，即

$$d(\log(\det(A))) = \frac{d(\det A)}{\det A} = \det(A^{-1})d(\det(A)) = \operatorname{tr}(A^{-1}dA).$$

对数导数在应用数学中经常出现。注意，在这里我们使用 $\frac{1}{\det A} = \det(A^{-1})$ 作为 $1 = \det(I) = \det(AA^{-1}) = \det(A)\det(A^{-1})$.

例如，回忆牛顿法通过一系列步骤 $x \to x + \delta x$ 寻找单变量实值函数 $f(x)$ 的根 $f(x) = 0$。牛顿法的关键公式是 $\delta x = f'(x)^{-1}f(x)$，但这与 $\frac{1}{(\log f(x))'}$ 相同。因此，对数行列式的导数出现在行列式根的寻找中，即对于 $f(x) = \det M(x)$。当 $M(x) = A - xI$ 时，行列式的根是 A 的特征值。对于更一般的函数 $M(x)$，求解 $\det M(x) = 0$ 因此被称为非线性特征值问题。

7.3 逆函数的雅可比矩阵

最后，我们计算矩阵逆的导数（作为线性算子和显式雅可比矩阵）。有一个巧妙的方法来获得这个导数，只需从逆的性质 $A^{-1}A = I$ 出发。根据乘积法则，这意味着

$$
\begin{aligned}
d(A^{-1}A) = d(I) = 0 &= d(A^{-1})A + A^{-1}dA \\
\implies & \boxed{d(A^{-1}) = (A^{-1})'[dA] = -A^{-1}\,dA\,A^{-1}}.
\end{aligned}
$$

这里，第二行定义了一个完美的线性算子用于导数$(A^{-1})'$，但如果我们愿意，我们可以通过使用克罗内克积在“向量化”矩阵上作用，就像我们在第3节中所做的那样，将其重写为一个显式的雅可比矩阵：

$$
\operatorname{vec}\left(d(A^{-1})\right) = \operatorname{vec}\left(-A^{-1}(dA)A^{-1}\right) = \underbrace{-(A^{-T}\otimes A^{-1})}_{\text{Jacobian}}\operatorname{vec}(dA)\,,
$$

A^{-T} 表示 $(A^{-1})^T = (A^T)^{-1}$。可以通过笔记本中的方法进行数值检验。

实际上，然而，你可能发现操作符表达式 $-A^{-1}\,dA\,A^{-1}$ 在涉及矩阵逆的求导中比显式的雅可比矩阵更有用。例如，如果你有一个关于标量参数 $t \in \mathbb{R}$ 的矩阵值函数 $A(t)$，你将立即获得 $\frac{d(A^{-1})}{dt} = -A^{-1}\frac{dA}{dt}A^{-1}$。更复杂的应用在6.3节中讨论。

8 前向和反向模式自动微分

第一次听到自动微分（AD）时，Edelman教授很容易想象它是什么……但他想象的是错误的！在他脑海中，他认为它是将符号微分直接应用于代码——有点像执行Mathematica或Maple，甚至只是自动做他在微积分课上学到的事情。例如，只需将类似以下一年级微积分表中的函数及其定义域插入即可：

Derivative	Domain
$(\sin x)' = \cos x$	$-\infty < x < \infty$
$(\cos x)' = -\sin x$	$-\infty < x < \infty$
$(\tan x)' = \sec^2 x$	$x \neq \frac{\pi}{2} + \pi n, n \in \mathbb{Z}$
$(\cot x)' = -\csc^2 x$	$x \neq \pi n, n \in \mathbb{Z}$
$(\sec x)' = \tan x \sec x$	$x \neq \frac{\pi}{2} + \pi n, n \in \mathbb{Z}$
$(\csc x)' = -\cot x \csc x$	$x \neq \pi n, n \in \mathbb{Z}$

在任何情况下，如果它不像执行 Mathematica 或 Maple 那样，那么它必须是有限差分，就像在数值计算课程中学到的那样（或者就像我们在第 4 节中所做的那样）。

结果显示这绝对不是有限差分——AD算法通常是精确的（在精确算术中，忽略舍入误差），而不是近似的。但它看起来也不像传统的符号代数：计算机并不真正构建一个大的“展开”符号表达式，然后再对其进行微分，就像你可能想象的手动或通过计算机代数软件进行的那样。例如，想象一个计算机程序计算一个$n \times n$矩阵的det A——只有在程序运行并且n已知（例如，由用户输入）之后，才能写下“整个”符号表达式，而且在任何情况下，一个简单的符号表达式都需要$n!$项。因此，AD系统必须处理在程序运行之前才知的计算机编程结构，如循环、递归和问题大小n，同时避免构建符号表达式的大小变得过大。（参见第8.1.1节，其中有一个例子与你在第一年微积分中微分的公式看起来非常不同。）AD系统的设计通常最终更多地关于编译器，而不是微积分！

8.1 通过双数自动微分

（本讲座附有Julia“笔记本”，展示了各种计算实验的结果，可在课程网页上找到。以下包含实验摘录。）

一种可以相对简单地解释的AD方法是“前向模式”AD，它通过同时进行 f' 和 f 的计算来实现。在计算机程序中，对每个中间值 a 增加另一个表示其导数的值 b，并使用链式法则将这些导数通过程序中的值计算传播。结果，这可以被视为用一种新的“双数”$D(a, b)$（（值 & 导数））和相应的算术规则来替换实数（值 a），如下所述。

8.1.1 示例：巴比伦平方根

我们从简单的例子开始，一个平方根函数的算法，其中自动微分的一个实用方法既给Edelman教授带来了数学上的惊喜，也给计算机带来了奇迹。特别是，我们考虑了计算$\sqrt{x}$的“巴比伦”算法，这个算法已流传千年（后来被揭示为一种特殊情况

牛顿法应用于 $t^2 - x = 0$)：简单地重复 $t \leftarrow (t + x/t)/2$，直到 t 收敛到 $\sqrt{x}$，达到任何所需的精度。每次迭代有一个加法和两个除法。为了说明目的，10 次迭代就足够了。以下是一个用 Julia 实现此算法的简短程序，从猜测值 1 开始，然后执行 N 步（默认为 $N = 10$）：

julia>函数巴比伦(x; N = 10)
t =（1+x)/2 # 从 t=1 的一步 i = 2:N # 剩余
N-1 步 t =（t + x/t）/ 2 end return t

zhd

如果我们运行这个函数来计算 $x = 4$ 的平方根，我们会看到它收敛得非常快：只需 $N = 3$ 步，它就获得了几乎精确到小数点后 3 位的正确答案（2），并且在大约 $N = 10$ 步之前，它就已经收敛到 2，在计算机算术精度（大约 16 位）的范围内。事实上，它在每一步都大致将正确数字的数量翻倍：

julia> 巴比伦（4， N=1）
2.5 julia> 巴比伦(4, N=2) 2.05

julia> 巴比伦(4, N=3) 2.000609
756097561

julia> 巴比伦(4, N=4) 2.000000
0929222947

julia> 巴比伦(4, N=10)
2.0

当然，任何一年级微积分学生都知道平方根的导数，$(\sqrt{x})' = 0.5/\sqrt{x}$，我们可以通过0.5 / 巴比伦算法(x)在这里计算它，但我们要知道如何自动地从巴比伦算法本身获得这个导数。如果我们能找出如何轻松高效地通过这个算法传递链式法则，那么我们就会开始理解AD如何也能对那些没有已知简单导数公式的更复杂的计算机程序进行微分。

8.1.2 简单的前向模式AD

计算机程序中传递链式法则的基本思想非常简单：将每个数字替换为两个数字，一个用于跟踪值，另一个用于跟踪该值的导数。值的计算方式与之前相同，而导数的计算是通过执行类似 $+$ 和 $/$ 的基本运算的链式法则来完成的。

在 Julia 中，我们可以通过定义一种新的数字类型来实现这个想法，我们将称之为 D，它封装了一个值 val 和一个导数 deriv。

julia> struct D <: Number val::Fl
oat64 deriv::Float64

结束

(关于Julia语法的详细解释可以在其他地方找到，但希望即使你不理解每个标点符号，也能理解基本思想。) 这种新类型的量x = D(a,b)有两个组成部分x.val = a和x.deriv = b，我们将分别用它们来表示值和导数。巴比伦代码只使用两种算术运算，+和/，所以我们只需要在Julia中覆盖（“Base”）这些运算的定义，以包括我们D类型的新规则：

```
julia> Base.:+(x::D, y::D) = D(x.val+y.val, x.deriv+y.deriv)
       Base.:/(x::D, y::D) = D(x.val/y.val, (y.val*x.deriv - x.val*y.deriv)/y.val^2)
```

如果您仔细观察，会发现值只是以普通方式相加和相除，而导数则是使用求和法则（相加输入的导数）和商法则分别计算。我们还需要一个其他的技术技巧：我们需要定义“转换”和“提升”规则，告诉 Julia 如何将 D 值与普通实数组合，例如在 $x + 1$ 或 $x/2$ 这样的表达式中：

julia> 基础转换函数 ::Type{D}, r::Real) = D(r,0) 基础提升规则函数 :
:Type{D}, ::Type{<:Real}) = D

这仅仅表示一个普通实数 r 通过先将 r 转换为 D(r,0) 与 D 值相结合：值 e
是 r，其导数为0（任何常数的导数都是零）。

给定这些定义，我们现在可以将一个D值插入到我们的未修改的巴比伦函数中，并且它将 illTranslated Tex
“神奇地”计算平方根的导数。让我们尝试一下 $x = 49 = 7^2$：

julia> x = 49
49

julia> 巴比伦(D(x,1))
D(7.0, 0.07142857142857142)

我们可以看到它正确地返回了7.0的值和0.07142857142857142的导数，这确实与平方根$\sqrt{49}$及其导数$0.5/\sqrt{49}$相匹配：

julia> （$\sqrt$x, 0.5/$\sqrt$x）
（7.0, 0.07142857142857142）

为什么我们输入了 D(x,1)？1 从哪里来的？这仅仅是输入的导数的事实 x
关于自身是 $(x)' = 1$，因此这是链式法则的起点。

在实践中，所有这些（以及更多）已经在Julia的ForwardDiff.jl包中实现（以及各种语言中许多类似的软件包中）。该包在底层隐藏了实现细节，并明确提供了一个计算导数的函数。例如：

julia> 使用 ForwardDiff

```
julia> ForwardDiff.derivative(Babylonian, 49)
0.07142857142857142
```

本质上，然而，这与我们的简单D实现相同，但以更一般和复杂的方式实现（例如，为更多操作使用链式法则，支持更多数值类型，对多个变量的偏导数等）：就像我们做的那样，ForwardDiff为每个值增加一个跟踪导数的第二个数字，并通过计算传播这两个数量。

我们也可以为巴比伦算法具体实现同样的想法，通过编写一个新的函数 dBabylonian 来跟踪计算过程中的变量 t 及其导数 $t' = dt/dx$:

julia>函数dBabylonian(x; N = 10)
t = （1+x）/2 t′ = 1/2 for i = 1:N t
= （t+x/t）/2 t′= （t′+(t-x*t′)/t2）/2 end
return t′ end

julia> dB巴比伦(49) 0.0714
2857142857142

这是与调用 Babylonian(D(x,1)) 或 ForwardDiff.derivative(Babylonian, 49) 完全相同的计算，但需要更多的人工努力——我们得为每个编写的计算机程序都这样做，而不是只实现一种新的数字类型。

8.1.3 双数

有一个令人愉悦的代数方法来思考我们新的数字类型 $D(a,b)$，而不是上面的“值与导数”观点。还记得一个复数 $a+bi$ 是如何由两个实数 (a,b) 通过定义一个特殊的新量 i (虚数单位) 形成的，它满足 $i^2=-1$，并且所有其他复数算术规则都由此而来吗？同样，我们可以将 $D(a,b)$ 视为 $a+b\epsilon$，其中 ϵ 是一个满足 $\epsilon^2=0$ 的新“无穷小单位”量。这种观点被称为双数。

给定基本规则 $\epsilon^2=0$，对偶数的其他代数规则立即得出:

$$\begin{aligned}(a+b\epsilon)\pm(c+d\epsilon) &= (a\pm c)+(b\pm d)\epsilon\\ (a+b\epsilon)\cdot(c+d\epsilon) &= (ac)+(bc+ad)\epsilon\\ \frac{a+b\epsilon}{c+d\epsilon} &= \frac{a+b\epsilon}{c+d\epsilon}\cdot\frac{c-d\epsilon}{c-d\epsilon} = \frac{(a+b\epsilon)(c-d\epsilon)}{c^2} = \frac{a}{c}+\frac{bc-ad}{c^2}\epsilon.\end{aligned}$$

这些规则的 ϵ 系数对应于微分学的和/差、乘积和商的规则！实际上，这些正是我们上面为我们的 D 类型实现的规则。我们只是缺少减法和乘法的规则，现在我们可以包括它们:

```
julia> Base.:-(x::D, y::D) = D(x.val - y.val, x.deriv - y.deriv)
       Base.:*(x::D, y::D) = D(x.val*y.val, x.deriv*y.val + x.val*y.deriv)
```

它也很不错添加一个“美化打印”规则，以便Julia将双数显示为+ bϵ而不是D(a,b)

julia> Base.show(io::IO, x::D) = 打印(io, x.val, " + ", x.deriv, "ϵ")

一旦我们在Julia中实现了双数的乘法规则，那么 $\epsilon^2 = 0$ 就可以从特殊情况 $a = c = 0$ 和 $b = d = 1$ 得出：

```
julia> ϵ = D(0,1)
0.0 + 1.0ϵ

julia> ϵ * ϵ
0.0 + 0.0ϵ

julia> ϵ^2
0.0 + 0.0ϵ
```

(我们没有为幂 $D(a, b)^n$ 定义规则，那么它是如何计算 ϵ^2 的呢？答案是 Julia 默认通过重复乘法实现 x^n，因此只需要定义 $*$ 规则即可.) 现在，我们可以像上面一样计算巴比伦算法在 $x = 49$ 的导数：

```
julia> 巴比伦(x + ϵ)
7.0 + 0.07142857142857142ϵ
```

与“无穷小部分”是导数 $0.5/\sqrt{49} = 0.0714\cdots$。

关于这个双重数观点的一个优点是它与我们的导数概念直接对应 a s 线性化：

$$f(x + \epsilon) = f(x) + f'(x)\epsilon + (\text{higher-order terms}),$$

与双数规则 $\epsilon^2 = 0$ 对应于省略高阶项。

8.2 简单符号微分

前向模式AD通过传播链式法则实现精确的解析导数，但它与许多人想象的前向差分法完全不同：通过符号评估程序以获得巨大的符号表达式，然后对这一巨大表达式进行微分以获得导数。这种方法的一个基本问题是，随着程序的运行，这些符号表达式的大小会迅速膨胀。让我们看看巴比伦算法会是什么样子。

想象将一个“符号变量”x输入到我们的巴比伦代码中，运行算法，并写出结果的一个大代数表达式。例如，在第一步之后，我们会得到 $(x + 1)/2$。在第二步之后，我们会得到 $((x + 1)/2 + 2x/(x + 1))/2$，这简化为两个多项式的比（一个“有理函数”）：

$$\frac{x^2 + 6x + 1}{4(x + 1)}.$$

手动继续这个过程相当繁琐，但幸运的是，计算机可以为我们完成它（如随附的Julia笔记本所示）。三次巴比伦迭代产生：

$$\frac{x^4 + 28x^3 + 70x^2 + 28x + 1}{8\left(x^3 + 7x^2 + 7x + 1\right)},$$

四次迭代给出

$$\frac{x^8 + 120x^7 + 1820x^6 + 8008x^5 + 12870x^4 + 8008x^3 + 1820x^2 + 120x + 1}{16\left(x^7 + 35x^6 + 273x^5 + 715x^4 + 715x^3 + 273x^2 + 35x + 1\right)},$$

并且五次迭代产生了一个巨大的表达式:

$$\frac{x^{16}+496x^{15}+35960x^{14}+906192x^{13}+10518300x^{12}+64512240x^{11}+225792840x^{10}+471435600x^{9}+601080390x^{8}+471435600x^{7}+225792840x^{6}+64512240x^{5}+10518300x^{4}+906192x^{3}+35960x^{2}+496x+1}{32(x^{15}+155x^{14}+6293x^{13}+105183x^{12}+876525x^{11}+4032015x^{10}+10855425x^{9}+17678835x^{8}+17678835x^{7}+10855425x^{6}+4032015x^{5}+876525x^{4}+105183x^{3}+6293x^{2}+155x+1)}.$$

注意这些增长得多快——实际上，多项式的次数在每次迭代中都翻倍！现在，如果我们使用我们的普通微积分规则对这些函数进行符号微分，并在计算机的帮助下简化，那么一次迭代的导数是 $\frac{1}{2}$，两次迭代的导数是

$$\frac{x^2+2x+5}{4\left(x^2+2x+1\right)},$$

三次迭代的公式是

$$\frac{x^6+14x^5+147x^4+340x^3+375x^2+126x+21}{8\left(x^6+14x^5+63x^4+100x^3+63x^2+14x+1\right)},$$

四次迭代的公式是

$$\frac{x^{14}+70x^{13}+3199x^{12}+52364x^{11}+438945x^{10}+2014506x^{9}+5430215x^{8}+8836200x^{7}+8842635x^{6}+5425210x^{5}+2017509x^{4}+437580x^{3}+52819x^{2}+3094x+85}{16(x^{14}+70x^{13}+1771x^{12}+20540x^{11}+126009x^{10}+440986x^{9}+920795x^{8}+1173960x^{7}+920795x^{6}+440986x^{5}+126009x^{4}+20540x^{3}+1771x^{2}+70x+1)},$$

并且经过五次迭代后，它变成了一个只有通过放大才能阅读的怪物:

$$\frac{x^{30}+310x^{29}+59799x^{28}+4851004x^{27}+215176549x^{26}+5809257090x^{25}+102632077611x^{24}+1246240871640x^{23}+10776333438765x^{22}+68124037776390x^{21}+321156247784955x^{20}+1146261110726340x^{19}+3133113888931089x^{18}+6614351291211874x^{17}+10850143060249839x^{16}+13883516068991952x^{15}+13883516369532147x^{14}+10850142795067314x^{13}+6614351497464949x^{12}+3133113747810564x^{11}+1146261195398655x^{10}+321156203432790x^{9}+68124057936465x^{8}+10776325550040x^{7}+1246243501215x^{6}+102631341330x^{5}+5809427001x^{4}+215145084x^{3}+4855499x^{2}+59334x+341}{32(x^{30}+310x^{29}+36611x^{28}+2161196x^{27}+73961629x^{26}+1603620018x^{25}+23367042639x^{24}+238538538360x^{23}+1758637118685x^{22}+9579944198310x^{21}+39232152623175x^{20}+122387258419860x^{19}+293729420641881x^{18}+546274556891506x^{17}+791156255418003x^{16}+894836006026128x^{15}+791156255418003x^{14}+546274556891506x^{13}+293729420641881x^{12}+122387258419860x^{11}+39232152623175x^{10}+9579944198310x^{9}+1758637118685x^{8}+238538538360x^{7}+23367042639x^{6}+1603620018x^{5}+73961629x^{4}+2161196x^{3}+36611x^{2}+310x+1)}.$$

这是一个计算导数的糟糕方法！（然而，存在更复杂的有效符号微分方法，例如“D^*”算法，通过利用重复的子表达式来避免显式的巨大公式。）为了清楚起见，双重数方法（不考虑舍入误差）精确地计算出答案，就像它在某个特定的 x 上评估这些疯狂的表达式一样，但这里的“就像”这个词非常重要。正如你所看到的，我们并没有形成这些表达式，更不用说评估它们了。我们只是计算出结果，这些结果与我们本应得到的结果相等。

8.3 通过计算图进行自动微分

让我们现在通过计算图来探讨自动微分。对于本节，我们考虑以下简单激励示例。 g

示例 40
定义以下函数:

$$\begin{cases} a(x,y) = \sin x \\ b(x,y) = \frac{1}{y} \cdot a(x,y) \\ z(x,y) = b(x,y) + x. \end{cases}$$

计算 $\frac{\partial z}{\partial x}$ 和 $\frac{\partial z}{\partial y}$.

有几种方法可以解决这个问题。首先，当然，可以符号地计算这个符号，注意到

$$z(x,y) = b(x,y) + x = \frac{1}{y}a(x,y) + x = \frac{\sin x}{y} + x,$$

这表示

$$\frac{\partial z}{\partial x} = \frac{\cos x}{y} + 1 \quad \text{and} \quad \frac{\partial z}{\partial y} = -\frac{\sin x}{y^2}.$$

然而，也可以使用计算图（见下方的计算图图示）其中从节点 A 到节点 B 的边被标记为 $\frac{\partial B}{\partial A}$。

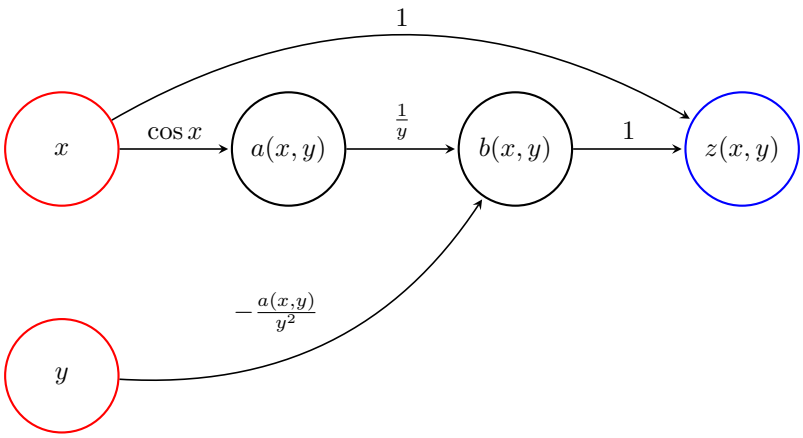


图7：对应示例40的计算图，表示从两个输入x, y计算输出$z(x, y)$，其中包含中间量$a(x, y)$和$b(x, y)$。节点按值标记，边按值相对于前一个值的导数标记。

现在我们如何使用这个有向无环图（DAG）来求导数？嗯，一个观点（称为“正向视图”）是通过跟随从输入到输出的路径，并在过程中（左）相乘，将多个路径相加得到的。例如，对于从 x 到 $z(x,y)$ 的路径，我们得到

$$\frac{\partial z}{\partial x} = 1 \cdot \frac{1}{y} \cdot \cos x + 1 = \frac{\cos x}{y} + 1.$$

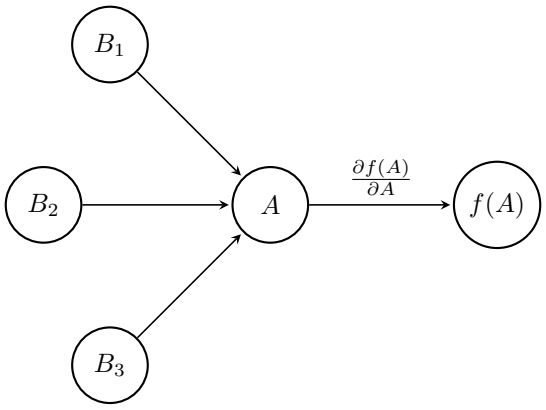
同样，对于从 y 到 $z(x,y)$ 的路径，我们有

$$\frac{\partial z}{\partial y} = 1 \cdot \frac{-a(x,y)}{y^2} = \frac{-\sin x}{y^2},$$

并且如果你在边缘上有数值导数，这个算法是有效的。或者，你可以采取反向视图并反向追踪路径（从右到左相乘），并得到相同的结果。请注意，这里并没有什么神奇之处——你可以想象这些函数是我们在这个课程中看到的那种类型，并执行相同的计算！这里根本重要的是结合律。然而，当考虑向量值函数时，乘以边缘权重顺序至关重要（因为向量/矩阵值函数通常不是交换的）。

图论中思考这种方法是考虑“路径乘积”。路径乘积是在遍历路径时边权的乘积。这样，我们感兴趣的是从输入到输出的路径乘积之和，以使用计算图来计算导数。显然，只要我们取乘积的顺序正确，我们并不特别关心遍历路径的顺序。因此，前向和反向模式的自动微分并不那么神秘。

让我们更仔细地看看前向模式自动微分的具体实现。假设我们在计算计算图导数的过程中处于一个节点 A，如图所示：



假设我们知道所有边直到并包括从 B_2 到 A 的路径乘积 P。那么当我们从 A 向右移动时，新的路径乘积是什么？它是 $f'(A) \cdot P$！因此我们需要一种数据结构，其映射方式如下：

$$(\text{value}, \text{path product}) \mapsto (f(\text{value}), f' \cdot \text{path product}).$$

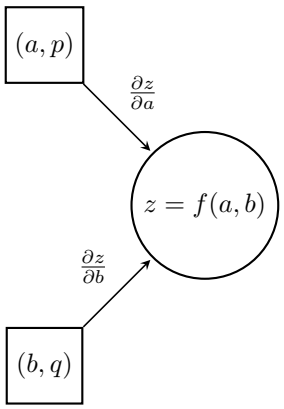
在某些意义上，这是看待双数的一种另一种方式——接受我们的路径积并输出值。无论如何，我们重载我们的程序，可以轻松计算 f(值) 并附加 $f' \cdot$(路径积)。

可能有人会问我们的程序是如何开始的——这是程序在“中间”的工作方式，但我们的起始值应该是多少呢？嗯，为了让这种方法起作用，它只能是 $(x, 1)$。然后，在每一步，你都要执行上面列出的以下映射：

$$(\text{value}, \text{path product}) \mapsto (f(\text{value}), f' \cdot \text{path product}),$$

在最后我们得到我们的导数。

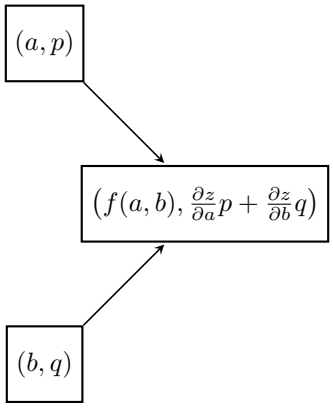
现在我们如何组合箭头？换句话说，假设在LHS的两个音符上我们有值(a, p) ）并且 (b, q)，如以下图中所示：所以在这里，我们不是将 a, b 视为数，而是将其视为变量。Wha t



应该新的输出值是多少？我们想将两个路径积相加，得到

$$\left(f(a,b), \frac{\partial z}{\partial a}p + \frac{\partial z}{\partial b}q\right).$$

所以实际上，我们的超载数据结构看起来是这样的：



此图如果可以，则概括了图中左侧的许多不同节点。

如果我们为所有简单计算（加法/减法、乘法和除法）设计出这样的数据结构，并且这正好是我们计算机程序所需要的，那么我们就成功了！以下是加法/减法、乘法和除法的结构定义方式。

加法/减法：见图。

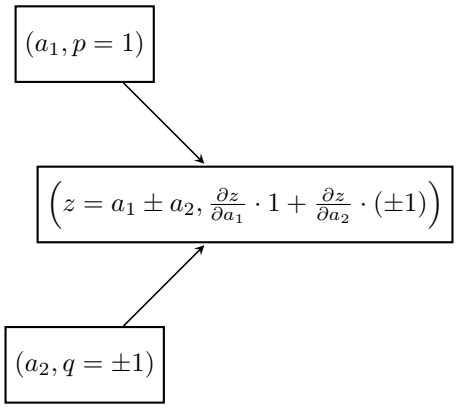


图8：加/减计算图示

乘法：见图。

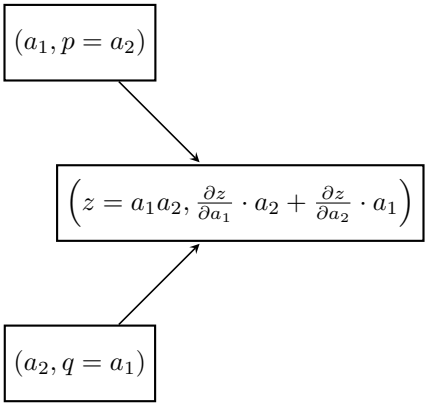


图9：乘法计算图

除法：参见图。

在理论上，这三个图就足够了，我们可以使用泰勒级数展开更复杂的函数。但在实践中，我们会加入更复杂函数的导数，这样我们就不浪费时间去计算已知的东西，比如正弦或对数的导数。

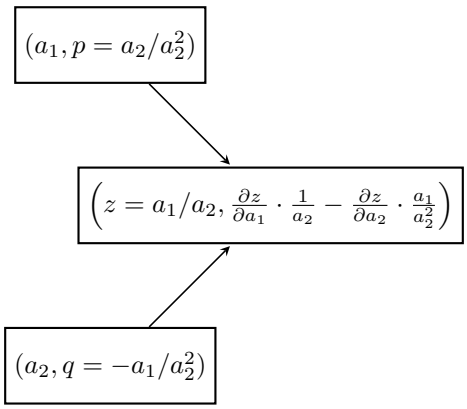
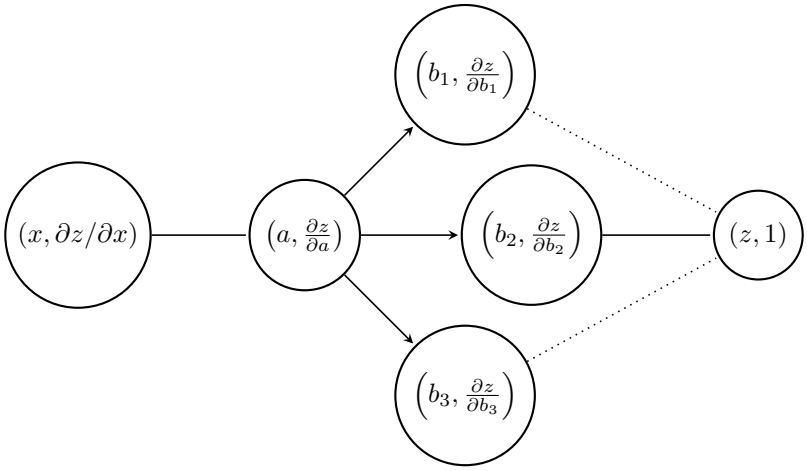


图10：除法计算图示意图

8.3.1 逆模式自动微分在Gr上的应用 aphs

当我们进行反向模式时，箭头会指向相反的方向，这一点我们将在本节笔记中理解。在正向模式中，一切都是关于“我们依赖于什么”，即使用左侧节点中的函数计算上述图中的导数。在反向模式中，问题实际上是“我们受到什么影响？”或者“我们后来又影响了什么？”

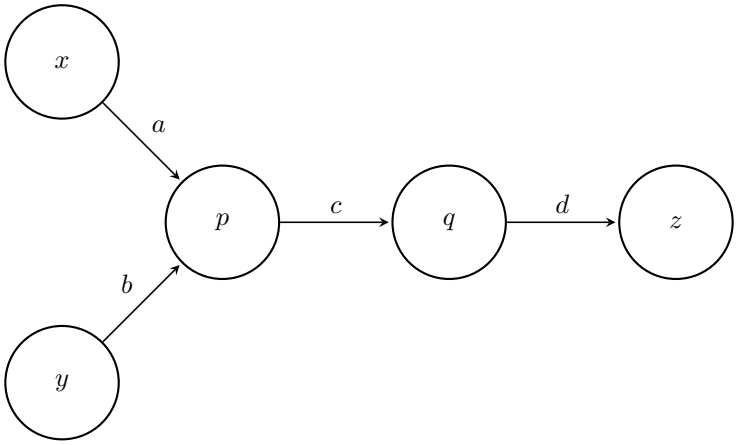
当“向后”时，我们需要知道给定节点影响哪些节点。例如，给定一个节点A，我们想知道受节点A影响或依赖的节点B_i。因此，现在我们的图看起来是这样的：



因此，我们现在最终有一个最终节点（$z, 1$）（在右侧远处）一切从这里开始。这次，由于我们处于反向模式，所有的乘法都是从右到左进行的。我们的目标是能够计算出节点（$x, \partial z/\partial x$）。所以如果我们知道如何填写 $\frac{\partial z}{\partial a}$ 项，我们就能从右到左在这些计算图中进行（即在反向模式下）。实际上，获取 $\frac{\partial z}{\partial a}$ 的公式如下

$$\frac{\partial z}{\partial a} = \sum_{i=1}^{s} \frac{\partial b_i}{\partial a} \frac{\partial z}{\partial b_i}$$

在节点 A 影响下的节点中，b_i 来自于此。这又是一个类似于微积分中的链式法则，但你也可以将其视为乘以受 A 影响的图中所有权重的总和。为什么反向模式比正向模式更高效？一个原因是因为它可以节省数据并使用它。



稍后。例如，以下是一个源/汇计算图。

如果 x, y 是我们的源，而 z 是我们的汇，我们想要计算从源到汇的路径上权重的乘积之和。如果我们使用正向模式，我们需要计算路径 dca 和 dcb，这需要四次乘法（然后你会将它们相加）。如果我们使用反向模式，我们只需要计算 $a\underline{cd}$ 和 $b\underline{cd}$ 并将它们相加；注意反向模式（因为我们只需要计算 cd 一次），只需要三次乘法。一般来说，这可以更有效地解决某些类型的问题，例如源/汇问题。

8.4 前向-与-反向模式微分

在这个部分，我们简要总结了这两种求导方法（无论是手工计算还是使用AD软件）的相对优缺点。从数学角度来看，这两种方法互为镜像，但从计算角度来看，它们相当不同，因为计算机程序通常从输入到输出按“正向”时间顺序进行。

假设我们正在对函数 $f\colon \mathbb{R}^n \mapsto \mathbb{R}^m$ 进行微分，该函数将 n 个标量输入（一个 n 维输入）映射到 m 个标量输出（一个 m 维输出）。正向模式与反向模式的第一关键区别在于计算成本如何随着输入和输出的数量/维度而缩放：

- 前向模式求导（输入到输出）的成本与输入数量 n 成正比。这对于输入少、输出多的函数来说很理想。
- 反向模式求导（输出到输入）的成本与输出数量 m 成正比。这对于输出少、输入多的函数来说很理想。

在本章之前，我们首先在2.5.1节中看到了这些缩放，然后在6.3节中再次看到；在未来的讲座中，我们将在9.2节中再次看到。在大量优化（无论是机器学习、工程设计或其他应用）中，少量输出的情况极为常见，因为那时有多个优化参数($n \gg 1$)，但只有一个与目标（或“损失”）函数相对应的输出($m = 1$)，或者有时有少量与目标函数和约束函数相对应的输出。因此，反向模式微分（“反向传播”）是大规模优化和训练神经网络等应用的主导方法。

存在其他值得考虑的实用问题，然而 er：

- 前向模式微分与函数本身的计算顺序相同，从输入到输出。这似乎使得前向模式自动微分更容易实现（例如，第8.1节中的我们的示例实现）且高效。

- 反向模式求导与普通计算的方向相反。这使得反向模式自动微分（AD）的实现更加复杂，并为函数计算增加了大量的存储开销。首先，您从输入评估函数到输出，但您（或AD系统）会记录（一个“带子”）所有计算的中介步骤；然后，您以相反的方向运行计算（“倒带”）以反向传播导数。

由于这些实际优势，即使对于许多 ($n > 1$) 输入和单个 ($m = 1$) 输出的情况，实践者告诉我们，他们发现前向模式在 n 足够大时（也许甚至直到 $n > 100$，具体取决于被微分的函数和自动微分实现）更有效率。（您可能还对克里斯·拉卡乌卡斯（Chris Rackauckas）关于自动微分工程权衡的博客文章感兴趣，该文章主要关于反向模式实现。）

如果 $n = m$，其中两种方法都没有缩放优势，通常人们更倾向于选择前向微分法的低开销和简单性。这种情况出现在计算非线性根查找的显式雅可比矩阵（第6.1节）或二阶导数的海森矩阵（第12节）时，对于这些情况，人们通常使用前向模式……或者甚至使用前向和反向模式的组合，如下文所述。

当然，正向和反向并不是唯一的选择。链式法则具有结合律，因此存在许多可能的顺序（例如，从两端开始并在中间相遇，反之亦然）。一个可能经常需要混合方案来计算雅可比（或海森）矩阵的困难问题是在最小操作次数内进行计算，利用任何特定问题的结构（例如稀疏性：许多条目可能为零）。关于这一点和其他AD主题的讨论，可以在Griewank和Walther（2008年）的《评估导数》（第2版）一书中找到，比这些笔记中详细得多。

8.4.1 前向-反向模式：二阶导数

通常，在计算许多实际应用中出现的二阶导数时，正向和反向模式的微分结合使用是有利的。

海森矩阵计算：例如，让我们考虑一个函数 $f(x)$：$\mathbb{R}^n \to \mathbb{R}$，它将 n 输入 x 映射到单个标量。如果 $n \gg 1$（许多输入），则一阶导数 $f'(x) = (\nabla f)^T$ 最好通过逆模式计算。然而，现在考虑二阶导数，它是 $g(x) = \nabla f$ 的导数，将 n 输入 x 映射到 n 输出 ∇f。因此，$g'(x)$ 是一个 $n \times n$ 雅可比矩阵，称为 f 的海森矩阵，我们将在第 12 节中更一般地讨论它。由于 $g(x)$ 有相同数量的输入和输出，因此前向模式和逆模式都没有固有的缩放优势，所以通常选择前向模式进行 g'，因为它在实际中简单易行，同时仍然通过逆模式计算 ∇f。也就是说，我们通过逆模式计算 ∇f，然后通过将前向模式微分应用于 ∇f 算法来计算 $g' = (\nabla f)'$。这被称为前向-逆模式算法。

一个更明确的正向-反向微分应用是到Hessian-向量乘积。在许多应用中，结果是只需要Hessian（∇f）′与任意向量v的乘积（∇f）。在这种情况下，可以完全避免显式计算（或存储）Hessian矩阵，并且计算成本仅与单个函数评估（$f(x)$）成比例。诀窍是回想一下（从第2.2.1节），对于任何函数g，线性操作$g'(x)$[v]是一个方向导数，相当于在$\alpha = 0$处评估的单变量导数$\frac{\partial}{\partial \alpha} g(x + \alpha v)$。在这里，我们只需将此规则应用于函数$g(x) = \nabla f$，并得到以下Hessian-向量乘积的公式：

$$(\nabla f)' v = \frac{\partial}{\partial \alpha} \left(\nabla f|_{x+\alpha v} \right) \Big|_{\alpha=0} .$$

[6] In fact, extraordinarily difficult: “NP-complete” (Naumann, 2006).

计算上，在任意点 $x+\alpha v$ 处梯度 ∇f 的内部评估可以通过反向/伴随/反向传播算法高效完成。相比之下，关于单个输入 α 的外导数最好通过前向微分模式执行.[7] 由于Hessian矩阵是对称的（如第12节所述，在极大的普遍性下讨论），相同的算法适用于向量-Hessian乘积 $v^T(\nabla f)' = [(\nabla f)'v]^T$，这一点我们在下一个例子中加以利用。

标量值梯度函数：还有一种常见情况，人们经常结合正向和反向微分，但可能显得更加微妙，那就是对另一个标量值函数的梯度的标量值函数进行微分。考虑以下示例：

示例 41
让 $f(x) : \mathbb{R}^n \mapsto \mathbb{R}$ 是一个关于 $n \gg 1$ 个输入的标量值函数，其梯度为 $\nabla f|_x = f'(x)^T$，并且让 $g(z) : \mathbb{R}^n \mapsto \mathbb{R}$ 是另一个这样的函数，其梯度为 $\nabla g|_z = g'(z)^T$。现在，考虑标量值函数 $h(x) = g(\nabla f|_x) : \mathbb{R}^n \mapsto \mathbb{R}$ 并计算 $\nabla h|_x = h'(x)^T$。

表示 $z = \nabla f|_x$。根据链式法则，$h'(x) = g'(z)(\nabla f)'(x)$，但我们想避免显式计算大的 $n \times n$ 海森矩阵 $(\nabla f)'$。相反，如上所述，我们使用这样一个向量-海森积（由海森矩阵的对称性）等价于海森矩阵-向量积的转置乘以海森矩阵 $(\nabla f)'$ 与向量 $\nabla g = g'(z)^T$，这等价于一个方向导数：

$$\nabla h|_x = h'(x)^T = \frac{\partial}{\partial \alpha} \left(\nabla f|_{x+\alpha \nabla g|_z} \right) \Big|_{\alpha=0},$$

涉及对单个标量 $\alpha \in \mathbb{R}$ 的微分。因此，对于任何Hessian-向量积，我们可以通过以下方式评估 h 和 ∇h：

1. 评估 $h(x)$：通过反向模式评估 $z = \nabla f|_x$，并将其插入到 $g(z)$ 中。
2. 评估 ∇h：
   (a) 通过反向模式评估 $\nabla g|_z$。 (b) 通过反向模式实现 $\nabla f|_{x+\alpha \nabla g|_z}$，然后通过正向模式对 α 求导，在 $\alpha = 0$ 处评估。

这是一个“正向-反向”算法，其中正向模式被有效地用于对 $\alpha \in \mathbb{R}$ 的单输入导数，并结合反向模式对 $x, z \in \mathbb{R}^n$ 进行微分。

以下是一个实现上述“正向-反向”过程的Julia代码示例，用于仅针对此类$h(x) = g(\nabla f)$函数。在此，相对于α的前向微分是通过第8.1节中讨论的ForwardDiff.jl包实现的，而相对于x或z的反向微分是通过Zygote.jl包执行的。首先，让我们导入包并定义简单的示例函数$f(x) = 1/\|x\|$和$g(z) = (\sum_k z_k)^3$，以及通过Zygote计算h：

julia> 使用 ForwardDiff, Zygote, LinearAlgebra
julia> f(x) = 1/norm(x) julia> g(z) = sum(z)3 julia> h(
x) = g(Zygote.gradient(f, x)[1])

7自动微分食谱，JAX文档的一部分，在Hessian-向量积的部分讨论了此算法。它指出，也可以交换 $\partial/\partial\alpha$ 和 ∇_x 的导数并采用反向-正向模式，但建议在实践中这不太高效：“因为正向模式比反向模式开销更小，而且外层微分算子在这里需要微分比内层更大的计算，所以保持正向模式在外部效果最好。” 它还提出了另一种选择：使用恒等式 $(\nabla f)'v = \nabla(v^T \nabla f)$，可以应用反向-反向模式来求 $v^T \nabla f$ 的梯度，但这具有更多的计算开销。

现在，我们将通过正向-反向计算 ∇h：

julia>函数∇h(x)∇f(y)= Zygote.gradient(f, y)[1]∇g= Zygote.gradient(g, ∇f(x))[1]
return ForwardDiff.derivative(α -> ∇f(x + α*∇g), 0)

结束

我们现在可以插入一些随机数字并与有限差分检查进行比较：

```
julia> x = randn(5); δx = randn(5) * 1e-8;
```

julia> h(x)
-0.005284687528953334

julia> ∇h(x)
5-元素 Vector{Float64}： -0.006
779692698531759 0.007176439
898271982 -0.00661026419924
1697 -0.0012162087082746558
0.007663756720005014

julia> ∇h(x)' * δx # 方向导数 -3.0273434457397667e-10

julia> h(x+δx) - h(x) # 有限差分检查 -3.0273433933303284e
-10

有限差分检查匹配到大约7位有效数字，这是我们所能期望的最大值——正向反转代码工作！

问题 42
一个常见的上述过程的变体，在机器学习中经常出现，涉及一个将输入“数据”$x \in \mathbb{R}^n$和“参数”$p \in \mathbb{R}^N$映射到标量的函数$f(x,p) \in \mathbb{R}$。令$\nabla_x f$和$\nabla_p f$表示相对于x和p的梯度。

现在，假设我们有一个与之前相同的函数 $g(z) : \mathbb{R}^n \mapsto \mathbb{R}$，并定义 $h(x,p) = g(\nabla_x f|_{x,p})$。我们想要计算 $\nabla_p h = (\partial h/\partial p)^T$，这将涉及 f 对 x 和 p 的“混合”导数。

证明您可以通过以下方式计算 $\nabla_p h$：

$$\nabla_p h|_{x,p} = \frac{\partial}{\partial \alpha} \left(\nabla_p f|_{x+\alpha \nabla g|_z, p} \right) \Big|_{\alpha=0},$$

在 $z = \nabla_x f|_{x,p}$。 (关键的是，这避免了计算 $n \times N$ 的混合导数矩阵 $f_{\circ}$。)

尝试提出简单的示例函数 f 和 g，通过在 Julia 中类似上述方式实现上述公式（$\partial/\partial\alpha$ 的正向模式以及 ∇ 的反向模式），并检查你的结果与有限差分近似值是否一致。

9 对常微分方程解的微分

在这个讲座中，我们将考虑对出现在方程和/或初始条件中的参数进行微分常微分方程（ODEs）解的问题。这在许多实际应用中是一个重要的话题，例如评估不确定性的影响、拟合实验数据或机器学习（机器学习越来越多地结合ODE模型和神经网络）。正如之前的讲座一样，我们将发现，在计算这些导数时，“正向”和“反向”（“伴随”）技术之间存在关键的实际区别，这取决于参数的数量和所需的输出。

尽管对常微分方程（ODE）概念的基本了解对本次讲座的读者有所帮助，但我们仍将要求 在为了建立我们的符号和术语，进行简要回顾。

视频讲座由Frank Schäfer博士（麻省理工学院）在IAP 2023上提供。这些笔记遵循相同的基本方法，但在一些细微的符号细节上有所不同。

9.1 常微分方程（ODEs）

一个常微分方程（ODE）是关于“时间”[8] $t \in \mathbb{R}$的一个函数 $u(t)$ 的方程，以一个 rTranslated Tex 更多导数，最常见的是一阶形式

$$\frac{du}{dt} = f(u,t)$$

对于某些右侧函数 f。请注意，$u(t)$ 不必是标量函数——它可以是列向量 $u \in \mathbb{R}^n$、矩阵或其他可微对象。人们还可以用高阶导数 d^2u/dt^2 等来写 ODE，但结果是一个 ODE 可以仅通过使 u 成为具有更多分量的向量来用一阶导数来表示.[9] 要唯一确定一阶 ODE 的解，我们需要一些额外的信息，通常是初始值 $u(0) = u_0$ (在 u 处的值 $t = 0$)，在这种情况下，它被称为初值问题。这些事实以及 ODE 的许多其他性质，在许多微分方程的教科书中都有详细讨论，例如在 MIT 的 18.03 课程中。

常微分方程（ODEs）在众多应用中非常重要，因为许多现实系统的行为是用变化率（导数）来定义的。例如，你可能还记得牛顿力学定律，其中加速度（速度的导数）与力（可能随时间、位置和/或速度变化）相关，而相应ODE的解$u =$ [位置, 速度]告诉我们系统的轨迹。在化学中，u可能代表一个或多个反应分子的浓度，右侧f提供反应速率。在金融中，存在类似ODE的股票或期权价格模型。偏微分方程（PDEs）是相同想法的更复杂版本，例如其中$u(x,t)$是空间x以及时间t的函数，并且有一个$\frac{\partial u}{\partial t} = f(u,x,t)$，其中$f$可能涉及$u$的一些空间导数。

在线性代数（例如MIT的18.06课程）中，我们经常考虑线性常微分方程的初值问题，形式为$du/dt = Au$，其中u是一个列向量，A是一个方阵；如果A是一个常数矩阵（与t或u无关），那么解$u(t) = e^{At}u(0)$可以用矩阵指数e^{At}来描述。更一般地，有许多技巧可以找到各种类型常微分方程的显式解（各种函数f）。然而，正如不能找到大多数函数的积分的显式公式一样，大多数常微分方程的解也没有显式公式。

[8] Of course, the independent variable need not be time, it just needs to be a real scalar. But in a generic context it is convenient to imagine ODE solutions as evolving in time.

[9] For example, the second-order ODE $\frac{d^2v}{dt^2} + \frac{dv}{dt} = h(v,t)$ could be re-written in first-order form by defining $u = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} v \\ dv/dt \end{pmatrix}$, in which case $du/dt = f(u,t)$ where $f = \begin{pmatrix} u_2 \\ h(u_1,t) - u_2 \end{pmatrix}$.

在许多实际应用中，人们必须求助于近似数值解。幸运的是，如果您提供一个可以计算 $f(u,t)$ 的计算机程序，就有成熟的、复杂的软件库[10] 可以从 $u(0)$ 计算出 $u(t)$，对于任何所需的时刻集 t，达到任何所需的精度水平（例如，到 8 位有效数字）。

例如，最基本的数值常微分方程方法通过使用有限差分 $\frac{du}{dt} = f(u,t)$ 来近似，在一系列时间 $t_n = n\Delta t$ 上简单地计算 $n = 0, 1, 2, \ldots$ 的解，从而得到“显式”时间步算法：

$$u(t_{n+1}) \approx u(t_n) + \Delta t\, f(u(t_n), t_n).$$

使用这种技术，称为“欧拉法”，我们可以将解向前推进时间：从我们的初始条件 u_0 开始，我们计算 $u(t_1) = u(\Delta t)$，然后从 $u(\Delta t)$ 计算出 $u(t_2) = u(2\Delta t)$，以此类推。当然，除非我们将 Δt 设置得非常小，否则这可能会相当不准确，需要很多时间步才能达到给定的时间 t，还可能出现其他细微之处，如“不稳定性”，其中误差可能会随着每个时间步以指数速度累积。事实证明，欧拉法基本上已经过时：有更多更复杂的算法可以以更低的计算成本稳健地产生准确解。然而，从概念上讲，它们都与欧拉法相似：它们通过在几个附近时间 t 对 f 和 u 进行评估，以某种方式“外推” u，从而将解向前推进时间。

依赖计算机求解常微分方程（ODEs）在实际上几乎是必不可少的，但它也可以使处理ODEs变得更加有趣。如果您曾经上过ODEs的课程，您可能还记得为了手动获得解而进行的许多繁琐劳动（棘手的积分、多项式根、方程组、积分因子等）。相反，我们在这里可以专注于简单地设置正确的ODEs和积分，并相信计算机来完成剩余的工作。

9.2 ODE解的敏感性分析

通常，常微分方程依赖于一些额外的参数 $p \in \mathbb{R}^N$（或其他向量空间）。例如，这些可能是化学问题中的反应速率系数，力学问题中粒子的质量，线性常微分方程中矩阵 A 的元素，等等。因此，你实际上有一个如下形式的问题

$$\frac{\partial u}{\partial t} = f(u, p, t),$$

在解 $u(p,t)$ 同时依赖于时间 t 和参数 p 的地方，以及初始条件 $u(p,0) = u_0(p)$也可能取决于参数。

问题在于，我们如何计算解相对于常微分方程参数的导数 $\partial u/\partial p$? 按照惯例，这指的是对 u 的一阶变化，当 p 发生变化时，如图11所示：

$$u(p + dp, t) - u(p, t) = \frac{\partial u}{\partial p}[dp] \qquad (\text{an } n\text{-component infinitesimal vector}),$$

当然，$\partial u/\partial p$ (，可以将其视为一个 $n \times N$ 雅可比矩阵)，它依赖于 p 和 t。这类问题很常见。例如，它在以下方面很重要：

- 不确定性量化（UQ）：如果你在你的常微分方程（ODE）参数中存在一些不确定性（例如，你有一个化学反应，其中反应速率只知道实验值 ± 一些测量误差），导数 $\partial u/\partial p$ 告诉你（至少是一阶的）你的答案对每个这些参数的敏感性

[10] For a modern and full-featured example, see the DifferentialEquations.jl suite of ODE solvers in the Julia language.

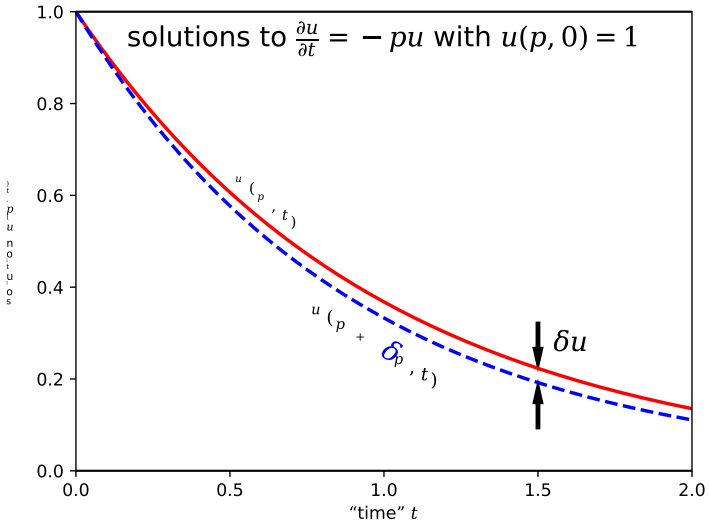


图11：如果我们有一个普通微分方程（ODE）$\frac{\partial u}{\partial t} = f(u,p,t)$，其解$u(p,t)$依赖于参数$p$，我们希望了解解$du = u(p+dp,t) - u(p,t)$因$p$的变化而产生的变化。在这里，我们展示一个简单示例$\frac{\partial u}{\partial t} = -pu$，其解$u(p,t) = e^{-pt}u(p,0)$是解析已知的，并展示了从改变$p = 1$到通过$\delta p = 0.1$的变化$\delta u$。

不确定性。

- 优化和拟合：通常，您想要选择参数 p 以最大化或最小化某些目标（或机器学习中的“损失”）。例如，如果您的常微分方程模型描述了某些具有未知反应速率或其他参数 p 的化学反应，您可能希望拟合参数 p 以最小化 $u(p,t)$ 与某些实验观察到的浓度之间的差异。

在优化后的后者情况下，您有一个解的标量目标函数，因为要最小化或最大化某物，您需要一个实数（而 u 可能是一个向量）。例如，这可以采取以下两种形式之一：

1. 一个依赖于特定时刻 T 的解 $u(p,T)$ 的实值函数 $g(u(p,T),T) \in \mathbb{R}$。例如，如果您有一个实验解 $u_*(t)$，您试图在 $t = T$ 处匹配，您可能需要最小化 $g(u(p,T),T) = \|u(p,T) - u_*(T)\|^2$。2. 一个依赖于许多时刻 $t \in (0,T)$ 的平均值（此处按 T 缩放）的实值函数 $G(p) = \int_0^T g(u(p,t),t)dt$。在拟合实验数据 $u_*(t)$ 的例子中，最小化 $G(p) = \int_0^T \|u(p,t) - u_*(t)\|^2 dt$ 相当于对时间 T（（例如，您实验的持续时间）的平均误差进行最小二乘拟合。

更普遍地，您可以通过在积分中包含一个非负权重函数 $w(t)$ 来给某些时间比其他时间赋予更大的权重：

$$G_w(p) = \int_0^\infty \underbrace{\|u(p,t) - u_*(t)\|^2}_{g(u(p,t),t)} w(t)\,dt,.$$

上述两种情况分别是选择 $w(t) = \delta(t-T)$ (a 狄拉克δ函数) 和 $w(t) = \begin{cases} 1\, t \leq T \text{否则（阶梯函数），分别。如问题} \end{cases}$ 43中所述，你可以让 $w(t)$ 成为δ函数的和来表示

数据在一系列离散时间点。

在两种情况下，由于这些是标量值函数，为了优化/拟合，人们希望知道梯度 $\nabla_p g$ 或 $\nabla_p G$，使得，像往常一样，

$$g(u(p+dp,t),t) - g(u(p,t),t) = (\nabla_p g)^T \, dp$$

因此，$\pm\nabla_p g$ 分别是 g 最大化/最小化的最陡上升/下降方向。值得强调的是，梯度（我们只对标量值函数定义）与其输入 p 具有相同的形状，因此 $\nabla_p g$ 是一个长度为 N (的向量，即参数的数量)，它依赖于 p 和 t。

这些只是“导数”，但可能你也能看到困难：如果我们没有$u(p,t)$的公式（显式解），只有一些可以给出任何参数p和t的$u(p,t)$数值软件，我们如何应用微分法则来找到$\partial u/\partial p$或$\nabla_p g$呢？当然，我们可以像第4节那样使用有限差分——只是对p和$p+\delta p$进行数值求解并相减——但如果我们要对许多参数($N \gg 1$)进行微分，这将相当慢，更不用说可能给出较差的精度了。实际上，人们通常在常微分方程（ODE）中有大量的参数想要进行微分。如今，我们的右手边函数$f(u,p,t)$甚至可以包含一个具有成千上万或数百万(N)个参数p的神经网络（这被称为“神经网络ODE”），我们需要所有N这些导数$\nabla_p g$或$\nabla_p G$来最小化“损失”函数g或G。因此，我们不仅需要找到一种方法来微分我们的ODE解（或其标量函数），而且这些导数必须高效地获得。结果证明，有两种方法可以做到这一点，而且两者都取决于导数是通过求解另一个ODE获得的：

- 前向模式：$\frac{\partial u}{\partial p}$实际上解决了另一个我们可以用相同的数值求解器对$u$进行积分的常微分方程。这为我们提供了我们想要的全部导数，但缺点是$\frac{\partial u}{\partial p}$的常微分方程比原始的$u$常微分方程大一个$N$因子，因此它只适用于小 N (参数)的情况。
- 反向（伴随）模式：对于标量目标，结果发现可以通过求解一个与u同样大小的“伴随”解$v(p,t)$的不同常微分方程来计算$\nabla_p g$或$\nabla_p G$，然后计算一些涉及u（“正向”解)和v的简单积分。这种方法的优势在于，只需大约是求解u成本的两倍，就可以得到所有N导数，无论参数数量N如何。缺点是，由于v必须“向后”积分时间（从$t=T$的“初始”条件开始，工作回$t=0$），并且依赖于u，因此必须存储所有$u(p,t)$对于所有$t \in [0,T]$（而不是在不再需要时向前推进时间并丢弃先前时间的值），这可能会为在长时间内集成的大的常微分方程系统要求大量的计算机内存。

我们现在将更详细地考虑这些方法中的每一个。

9.2.1 常微分方程的前向敏感性分析

让我们从我们的常微分方程 $\frac{\partial u}{\partial t} = f(u,p,t)$ 开始，考虑当 p 发生一个小的变化 dp 时 u 会发生什么：

$$\begin{aligned} d\underbrace{\left(\frac{\partial u}{\partial t}\right)}_{=f(u,p,t)} &= \frac{\partial}{\partial t}(du) = \frac{\partial}{\partial t}\left(\frac{\partial u}{\partial p}[dp]\right) = \frac{\partial}{\partial t}\left(\frac{\partial u}{\partial p}\right)[dp] \\ &= d(f(u,p,t)) = \left(\frac{\partial f}{\partial u}\frac{\partial u}{\partial p} + \frac{\partial f}{\partial p}\right)[dp], \end{aligned}$$

在本文中，我们使用了熟悉的规则（来自多元微积分）——交换偏导数的顺序——这是一个我们将在我们关于Hessian的讲座中明确重新推导的属性

并且二阶导数。将两行右侧相等，我们看到我们有一个常微分方程

$$\boxed{\frac{\partial}{\partial t}\left(\frac{\partial u}{\partial p}\right) = \frac{\partial f}{\partial u}\frac{\partial u}{\partial p} + \frac{\partial f}{\partial p}}$$

对于导数 $\frac{\partial u}{\partial p}$，其初始条件通过简单地对初始条件 $u(p,0) = u_0(p$ 求导得到 $)$
对于 u:

$$\left.\frac{\partial u}{\partial p}\right|_{t=0} = \frac{\partial u_0}{\partial p}.$$

我们可以因此将其插入任何常微分方程求解技术（通常是数值方法，除非我们非常幸运，能够针对特定的 f 对此常微分方程进行解析求解）以找到任何所需时间 t 的 $\frac{\partial u}{\partial p}$。简单，对吧?

唯一可能显得有点奇怪的是解的形状：$\frac{\partial u}{\partial p}$ 是一个线性算子，但常微分方程的解怎么可能是一个线性算子呢？实际上，这并没有什么问题，但思考几个例子会有所帮助:

- 如果 $u, p \in \mathbb{R}$ 是标量（即，我们有一个带有单个标量参数的单个标量常微分方程），那么 $\frac{\partial u}{\partial p}$ 只是一个（时间依赖的）数，并且我们的 $\frac{\partial u}{\partial p}$ 的常微分方程是一个具有普通标量初始条件的普通标量常微分方程。

- 如果 $u \in \mathbb{R}^n$ (a “系统” 的 n 常微分方程) 和 $p \in \mathbb{R}$ 是标量，那么 $\frac{\partial u}{\partial p} \in \mathbb{R}^n$ 是另一个列向量，我们的 $\frac{\partial u}{\partial p}$ 的常微分方程是另一个 n 常微分方程的系统。因此，我们解两个相同大小的 n 常微分方程，以获得 u 和 $\frac{\partial u}{\partial p}$。
- 如果 $u \in \mathbb{R}^n$ (a “系统” 的 n 常微分方程（ODE）) 和 $p \in \mathbb{R}^N$ 是 N 参数的向量，那么 $\frac{\partial u}{\partial p} \in \mathbb{R}^{n\times N}$ 是一个 $n \times N$ 雅可比矩阵。我们的 $\frac{\partial u}{\partial p}$ ODE 实际上是这个矩阵所有分量的 nN ODE 系统！用数值方法求解这个“矩阵 ODE”没有概念上的困难，但通常需要大约 N 倍于求解 u 的计算工作量，因为未知数有 N 倍之多！如果 N 很大，这可能会很昂贵!

这反映了我们对前向模式求导的一般观察：当需要求导的“输入”参数数量 N 较大时，它很昂贵。然而，前向模式很简单，特别是对于 $N \lesssim$ 大约100个或更多的情况，当求导常微分方程解时，它通常是第一个尝试的方法。给定 $\frac{\partial u}{\partial p}$，然后可以通过链式法则直接对标量目标进行求导:

$$\nabla_p g|_{t=T} = \left.\underbrace{\frac{\partial u}{\partial p}^T}_{\text{Jacobian}^T}\underbrace{\frac{\partial g}{\partial u}^T}_{\text{vector}}\right|_{t=T},$$

$$\nabla_p G = \int_0^T \nabla_p g \, dt.$$

左侧 $\nabla_p G$ 是 N 参数的标量函数的梯度，因此梯度是一个具有 N 个分量的向量。相应地，右侧也是一个 N-分量梯度的积分 $\nabla_p g$，而向量值函数的积分可以看作是逐元素积分（每个分量的积分组成的向量）。

9.2.2 常微分方程的逆/伴随敏感性分析

对于大的 $N \gg 1$ 和标量目标 g 或 G (等)，原则上我们可以通过应用“逆模式”或“伴随”方法，以与计算 u 相当的成本，更有效地计算导数。在其他课程中，我们通过从左到右（输出到输入）评估链式法则简单地获得了类似逆模式方法。

而不是从右到左。从概念上讲，对于常微分方程的过程是相似的，[11]但在代数上，导数计算相当复杂且不够直接。关键是要尽可能避免显式计算$\frac{\partial u}{\partial p}$，因为如果我们有很多参数（$p$很大），这可能会导致一个巨大的雅可比矩阵，尤其是如果我们有很多方程（u很大）。

特别地，让我们从我们的前向模式敏感性分析开始，并考虑导数 $G' = (\nabla_p G)^T$，其中 G 是一个时变目标函数 $g(u, p, t)$ (的积分，我们允许它显式地依赖于 p 以增加普遍性)。根据链式法则，

$$G' = \int_0^T \left(\frac{\partial g}{\partial p} + \frac{\partial g}{\partial u} \frac{\partial u}{\partial p} \right) dt,$$

涉及我们不需要的因子 $\frac{\partial u}{\partial p}$。为了摆脱它，我们将使用一个“奇怪的小技巧”（类似于拉格朗日乘数法）向这个方程添加零：

$$G' = \int_0^T \left[\left(\frac{\partial g}{\partial p} + \frac{\partial g}{\partial u} \frac{\partial u}{\partial p} \right) + v^T \underbrace{\left(\frac{\partial}{\partial t} \left(\frac{\partial u}{\partial p} \right) - \frac{\partial f}{\partial u} \frac{\partial u}{\partial p} - \frac{\partial f}{\partial p} \right)}_{=0} \right] dt$$

对于与 u 形状相同的某些函数 $v(t)$，它乘以我们的“前向模式”方程 $\partial u/\partial p$。如果 $u \in \mathbb{R}^n$ 则 $v \in \mathbb{R}^n$；更一般地，对于其他向量空间，将 v^T 读取为与 v 的内积。新项 $v^T(\cdots)$ 为零，因为括号内的表达式恰好是 $\frac{\partial u}{\partial p}$ 满足的常微分方程，正如我们在上面的前向模式分析中所获得的，无论 $v(t)$ 如何。这很重要，因为它允许我们自由选择 $v(t)$ 来消除不需要的 $\frac{\partial u}{\partial p}$ 项。特别是，如果我们首先对 $v^T \frac{\partial}{\partial t} \left(\frac{\partial u}{\partial p} \right)$ 项进行分部积分以将其变为 $-\left(\frac{\partial v}{\partial t} \right)^T \frac{\partial u}{\partial p}$ 加上一个边界项，然后重新组合项，我们发现：

$$G' = v^T \frac{\partial u}{\partial p}\bigg|_0^T + \int_0^T \left[\frac{\partial g}{\partial p} - v^T \frac{\partial f}{\partial p} + \underbrace{\left(\frac{\partial g}{\partial u} - v^T \frac{\partial f}{\partial u} - \left(\frac{\partial v}{\partial t} \right)^T \right)}_{\text{want to be zero!}} \frac{\partial u}{\partial p} \right] dt \, .$$

如果我们现在将$(\cdots)$项设为零，那么不需要的$\frac{\partial u}{\partial p}$将从$G$的积分计算中消失 $'$.
我们可以通过选择 $v(t)$ (，这可以是到目前为止的任何东西) 来满足“伴随”常微分方程：

$$\boxed{\frac{\partial v}{\partial t} = \left(\frac{\partial g}{\partial u} \right)^T - \left(\frac{\partial f}{\partial u} \right)^T v}.$$

我们应该选择什么初始条件来表示 $v(t)$? 嗯，我们可以使用这个选择来消除我们通过分部积分得到上面的边界项：

$$v^T \frac{\partial u}{\partial p}\bigg|_0^T = v(T)^T \underbrace{\frac{\partial u}{\partial p}\bigg|_T}_{\text{unknown}} - v(0)^T \underbrace{\frac{\partial u_0}{\partial p}}_{\text{known}} .$$

这里，未知项 $\frac{\partial u}{\partial p}\Big|_T$ 是一个问题——为了计算它，我们被迫回到从正向模式积分我们的大 $\frac{\partial u}{\partial p}$ 常微分方程。另一个项是好的：由于初始条件 u_0 总是给出的，我们应该明确知道它对 p 的依赖关系（并且在我们通常的初始条件 $\frac{\partial u_0}{\partial p} = 0$ 的情况下，我们将简单地有）。

[11] This “left-to-right” picture can be made very explicit if we imagine discretizing the ODE into a recurrence, e.g. via Euler's method for an arbitrarily small Δt, as described in the MIT course notes *Adjoint methods and sensitivity analysis for recurrence relations* by S. G. Johnson (2011).

不要依赖于p）。因此，为了消除$\frac{\partial u}{\partial p}\Big|_T$项，我们做出以下选择

$$\boxed{v(T) = 0}.$$

代替初始条件，我们的伴随常微分方程具有最终条件。这对数值求解器来说没有问题：这意味着伴随常微分方程是向后积分时间，从 $t = T$开始，向下到 $t = 0$。一旦我们解出了 $v(t)$ 的伴随常微分方程，我们就可以将其插入我们的 G' 方程中，通过简单的积分来获得我们的梯度：

$$\nabla_p G = (G')^T = -\left(\frac{\partial u_0}{\partial p}\right)^T v(0) + \int_0^T \left[\left(\frac{\partial g}{\partial p}\right)^T - \left(\frac{\partial f}{\partial p}\right)^T v\right] dt\,.$$

(如果您想显得更复杂，您可以通过增加一组表示 G' 积分函数的未知数和方程来同时计算这个 $\int_0^T$ 和 v 本身，通过增强伴随 ODE。但这主要只是计算上的便利，并没有改变关于过程的本质。)

仅剩的烦恼是伴随常微分方程依赖于所有 $t \in [0, T]$ 的 $u(p, t)$。通常，如果我们正在数值求解 $u(p, t)$ 的“正向”常微分方程，我们可以将解 u “向前”推进时间，并且只存储最近的一些时间步长的解。然而，由于伴随常微分方程从 $t = T$开始，我们只能在完成 u 的计算后才开始积分 v。这要求我们保存几乎所有之前计算出的 $u(p, t)$ 值，以便在积分 v (和 G') 期间在任意时间 $t \in [0, T]$ 评估 u。如果 u 很大（例如，它可能代表来自空间离散化偏微分方程（PDE）的数百万个网格点，如热扩散问题）并且需要许多时间步长 t，这可能会消耗大量的计算机内存。为了缓解这一挑战，已经采用了各种策略，通常集中在“检查点”技术，其中 u 只在时间子集 t 中保存，其他时间的值在 v 积分期间通过重新计算 u（从最近的“检查点”时间开始数值积分常微分方程）获得。然而，对这些技术的详细讨论超出了这些笔记的范围。

9.3 示例

让我们用一个简单的例子来说明上述技术。假设我们要对以下标量常微分方程进行积分{v*}

$$\frac{\partial u}{\partial t} = f(u, p, t) = p_1 + p_2 u + p_3 u^2 = p^T \begin{pmatrix} 1 \\ u \\ u^2 \end{pmatrix}$$

对于初始条件 $u(p, 0) = u_0 = 0$ 和三个参数 $p \in \mathbb{R}^3$。 (这可能是足够简单，可以用闭式解出，但我们这里不会去麻烦。）我们还将考虑标量函数

$$G(p) = \int_0^T \underbrace{[u(p, t) - u_*(t)]^2}_{g(u,p,t)} dt$$

我们可能希望最小化某些给定的 $u_*(t)$ (（例如，实验数据）或某些给定的公式如 $u_* = t^3$)，因此我们希望计算 $\nabla_p G$。

9.3.1 前向模式

雅可比矩阵 $\frac{\partial u}{\partial p} = \begin{pmatrix} \frac{\partial u}{\partial p_1} & \frac{\partial u}{\partial p_2} & \frac{\partial u}{\partial p_3} \end{pmatrix}$ 简单地是一个行向量，并满足我们的“前向模式”常微分方程：

$$\frac{\partial}{\partial t}\left(\frac{\partial u}{\partial p}\right) = \frac{\partial f}{\partial u}\frac{\partial u}{\partial p} + \frac{\partial f}{\partial p} = (p_2 + 2p_3 u)\frac{\partial u}{\partial p} + \begin{pmatrix} 1 & u & u^2 \end{pmatrix}$$

对于初始条件 $\left.\frac{\partial u}{\partial p}\right|_{t=0} = \frac{\partial u_0}{\partial p} = 0$。这是一个三阶耦合线性常微分方程组的非齐次系统，如果我们简单地转置等式两边，可能会看起来更传统：

$$\frac{\partial}{\partial t}\underbrace{\begin{pmatrix} \frac{\partial u}{\partial p_1} \\ \frac{\partial u}{\partial p_2} \\ \frac{\partial u}{\partial p_3} \end{pmatrix}}_{(\partial u/\partial p)^T} = (p_2 + 2p_3 u)\begin{pmatrix} \frac{\partial u}{\partial p_1} \\ \frac{\partial u}{\partial p_2} \\ \frac{\partial u}{\partial p_3} \end{pmatrix} + \begin{pmatrix} 1 \\ u \\ u^2 \end{pmatrix}.$$

这个事实表明它依赖于我们的“正向”解 $u(p,t)$，这使得手动解决它并不容易，但计算机可以毫无困难地进行数值求解。在计算机上，我们可能会通过将两个常微分方程合并成一个包含4个分量的单个常微分方程，同时求解 u 和 $\partial u/\partial p$：

$$\frac{\partial}{\partial t}\begin{pmatrix} u \\ (\partial u/\partial p)^T \end{pmatrix} = \begin{pmatrix} p_1 + p_2 u + p_3 u^2 \\ (p_2 + 2p_3 u)\,(\partial u/\partial p)^T + \begin{pmatrix} 1 \\ u \\ u^2 \end{pmatrix} \end{pmatrix}.$$

给定 $\partial u/\partial p$，我们然后可以将其代入 G 的链式法则：

$$\nabla_p G = 2\int_0^T [u(p,t) - u_*(t)]\frac{\partial u}{\partial p}^T dt$$

(再次，一个计算机可以数值评估的积分)。

9.3.2 反向模式

在反向模式下，我们有一个与 u) 形状相同的伴随解 $v(t) \in \mathbb{R}$ (，它解决了我们的伴随方程

$$\frac{\partial v}{dt} = \left(\frac{\partial g}{\partial u}\right)^T - \left(\frac{\partial f}{\partial u}\right)^T v = 2\left[u(p,t) - u_*(t)\right] - (p_2 + 2p_3 u)\, v$$

具有最终条件 $v(T) = 0$. 再次，计算机可以轻松数值求解（给定数值“正向”解 u），以找到 $v(t)$ 对于 $t \in [0, T]$。最后，我们的梯度是积分乘积：

$$\nabla_p G = -\int_0^T \begin{pmatrix} 1 \\ u \\ u^2 \end{pmatrix} v\, dt\,.$$

另一个有用的练习是考虑一个以求和形式出现的 G：

问题 43
假设 $G(p)$ 的形式为 K 项之和：

$$G(p) = \sum_{k=1}^{K} g_k(p, u(p, t_k))$$

对于时间 $t_k \in (0, T)$ 和函数 $g_k(p, u)$。例如，这可能在 $u_*(t_k)$ 在 K 离散时间处的实验数据的最小二乘拟合中出现，其中 $g_k(u(p, t_k)) = \|u_*(t_k) - u(p, t_k)\|^2$ 测量 $u(p, t_k)$ 与在时间 t_k 测量的数据之间的平方差。

1. 证明这样的 $G(p)$ 可以表示为本章公式的一个特例，通过定义我们的函数 $g(u, t)$ 为狄拉克δ函数之和 $\delta(t - t_k)$。

2. 解释这如何影响伴随解 $v(t)$：特别是，dv/dt 右侧引入的 delta 函数项如何导致 $v(t)$ 出现一系列不连续的跳跃。（在几个流行的数值常微分方程求解器中，可以通过离散时间的“回调”来引入这种不连续性。）

3. 解释这些delta函数如何也可能将求和引入到计算$\nabla_p G$中，但仅当g_k明确依赖于p (时，而不是仅仅通过u)。

9.4 进一步阅读

一个关于常微分方程（及其推广）的反向/伴随微分的经典参考文献，使用与今天类似的符号（除了伴随解 v 被表示为 $\lambda(t)$，以纪念拉格朗日乘数），是 Cao 等人（2003 年）（https://doi.org/10.1137/S1064827501380630），一篇更近期的综述文章是 Sapienza 等人（2024 年）（https://arxiv.org/abs/2406.09699）。还可以参考 SciMLSensitivity.jl 包（https://github.com/SciML/SciMLSensitivity.jl），用于 Chris Rackauckas 的令人惊叹的 DifferentialEquations.jl 软件套件在 Julia 中对常微分方程进行数值求解的敏感性分析。还有一个关于常微分方程伴随敏感性的 2021 年 YouTube 讲座（https://youtu.be/k6s2G5MZv-I），同样使用了类似的符号。对于递归关系，这个过程有一个离散版本，在这种情况下，可以得到一个反向顺序的“伴随”递归关系，如麻省理工学院 S. G. Johnson 的课程笔记中所述（https://math.mit.edu/~stevenj/18.336/recurrence2.pdf）。

该章节中的微分方法（例如，对于 $\partial u/\partial p$ 或 $\nabla_p G$）是在假设ODEs被精确求解的情况下推导的：对于 u 的精确ODE，我们推导出了导数的精确ODE。在计算机上，您将近似求解这些前向和伴随ODEs，因此得到的导数将仅是近似正确的（到由您的ODE求解器指定的容差）。这被称为先微分后离散化方法，它具有简单（与数值求解方案无关）的优点，但代价是略微不准确（您的近似导数不能精确预测近似解 u 的一阶变化）。另一种方法是先离散化后微分方法，其中您首先将ODE近似为离散时间递归公式（离散化），然后对递归进行精确微分。这的优点是精确微分您的近似解，但代价是复杂性（推导特定于您的离散化方案）。各种作者讨论了这些权衡及其影响，例如在M. D. Gunzburger的《流动控制与优化展望》（2002年）的第4章或Jensen等人（2014年）的论文中。

10 变分法

在这个讲座中，我们将把我们的微分工具应用于一种新的输入类型：既不是标量，也不是列向量，也不是矩阵，而是输入将是函数 $u(x)$，这些函数形成一个完美的向量空间（甚至可以有范数和内积）。[12] 结果表明，关于函数求导有很多惊人的应用，这些技术有时被称为“变分法”和/或“弗雷歇”导数。

10.1 功能：将函数映射到标量{v*}

示例 44
例如，考虑函数 $u(x)$，它将 $x \in [0,1] \to u(x) \in \mathbb{R}$ 映射。然后我们可以定义函数 f：

$$f(u) = \int_0^1 \sin(u(x))\,\mathrm{d}x.$$

这样的函数，将输入函数 u 映射到输出数字，有时被称为“函数”。在这种情况下，f' 或 ∇f 是什么?

回忆一下，对于任何函数 f，我们总是通过以下方程定义其导数为线性算子 $f'(u)$：

$$df = f(u+du) - f(u) = f'(u)[du]\,,$$

现在 du 表示一个任意“小值”函数 $du(x)$，它代表 $u(x)$ 的小变化，如图 12 所示的非无穷小 $\delta u(x)$ 的类似情况。在这里，我们可以通过被积函数的线性化来计算这个值：

$$\begin{aligned} df &= f(u+du) - f(u) \\ &= \int_0^1 \sin(u(x)+du(x)) - \sin(u(x))\,dx \\ &= \int_0^1 \cos(u(x))\,du(x)\,dx = f'(u)[du]\,, \end{aligned}$$

在最后一步中，我们将 $du(x)$ 设为任意小的 [13]，以便我们可以将 $\sin(u+du)$ 在 $du(x)$ 的一阶线性化。就是这样，我们得到了我们的导数 $f'(u)$，它是一个完美的线性操作，作用于 du!

10.2 函数的内积

为了在研究这样的“泛函”（从函数到 $\mathbb{R}$ 的映射）时定义一个梯度 ∇f，自然会问输入空间上是否存在内积。事实上，定义函数的内积有非常合适的方法！给定在 $x \in [0,1]$ 上定义的函数 $u(x), v(x)$，我们可以定义一个“欧几里得”内积：

$$\langle u, v\rangle = \int_0^1 u(x)v(x)\,\mathrm{d}x.$$

[12] Being fully mathematically rigorous with vector spaces of functions requires a lot of tedious care in specifying a well-behaved set of functions, inserting annoying caveats about functions that differ only at isolated points, and so forth. In this lecture, we will mostly ignore such technicalities—we will implicitly assume that our functions are integrable, differentiable, etcetera, as needed. The subject of *functional analysis* exists to treat such matters with more care.

[13] Technically, it only needs to be small “almost everywhere” since jumps that occur only at isolated points don’t affect the integral.

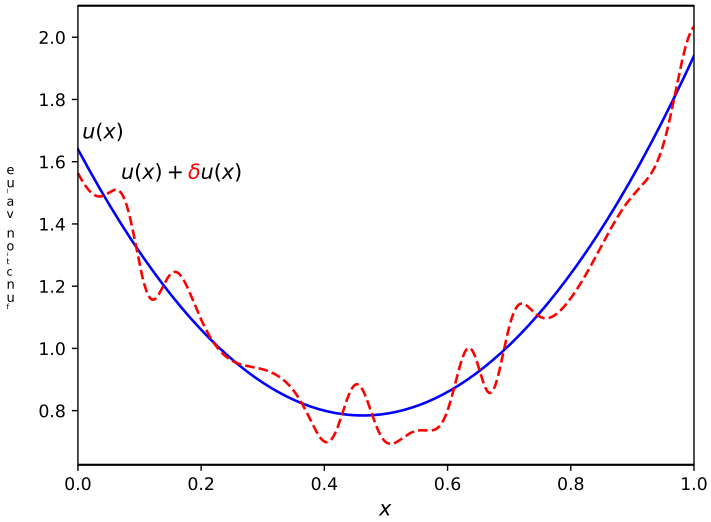


图12：如果我们$f(u)$的输入u是函数$u(x)$（，例如映射$[0,1]\mapsto\mathbb{R}$），那么微分的本质是在极限$\delta u(x)$变得任意小的情况下，对自身也是函数的小扰动$\delta u(x)$进行线性化f。在这里，我们展示了一个$u(x)$和一个扰动$u(x)+\delta u(x)$的例子。

请注意，这暗示了

$$\|u\| := \sqrt{\langle u,u\rangle} = \sqrt{\int_0^1 u(x)^2 dx}\,.$$

回忆一下，梯度 ∇f 被定义为我们将 du 与其内积以获得 df 的任何内容。因此，我们得到梯度如下：

$$df = f'(u)[du] = \int_0^1 \cos(u(x))\,du(x)\,dx = \langle \nabla f, du\rangle \implies \nabla f = \cos(u(x))\,.$$

两个无穷小 du 和 dx 可能会有些令人不安，但如果这让你感到困惑，你只需将其想成 e
$du(x)$ 作为一个小的非无穷小函数 $\delta u(x)$（，如图12)所示，其中我们正在省略高阶项。

梯度 ∇f 只是一个函数，cos($u(x)$)！像往常一样，∇f 与 u 具有相同的“形状”。

备注45。在这里比较上述积分的梯度与离散版本可能是有益的 n
在积分被替换为和的情况下。如果我们有

$$f(u) = \sum_{k=1}^{n} \sin(u_k)\Delta x$$

在 $\Delta x = 1/n$，对于与我们的先前 $u(x)$ 通过 $u_k = u(k\Delta x)$ 相关的向量 $u\in\mathbb{R}^n$，可以将其视为积分的“矩形法则”（或黎曼和，或欧拉）近似。然后，

$$\nabla_u f = \begin{pmatrix} \cos(u_1) \\ \cos(u_2) \\ \vdots \end{pmatrix} \Delta x\,.$$

为什么这个离散版本有一个 Δx 乘以梯度，而我们的连续版本没有？原因是，在连续版本中，我们有效地将 dx 包含在内积 $\langle u,v\rangle$ （的定义中，这是一个积分)。在离散情况下，通常的内积（用于定义传统梯度）是

只是一个没有 Δx 的和。然而，如果我们定义一个加权的离散内积 $\langle u, v\rangle = \sum_{k=1}^{n} u_k v_k \Delta x$，那么，根据第5节，这将改变梯度的定义，实际上将移除 Δx 项以对应连续版本。

10.3 示例：最小化弧长

我们现在考虑一个更具挑战性的例子，它具有直观的几何解释。

示例 46
设 u 是定义在 [0, 1] 上的可微函数，并考虑泛函

$$f(u) = \int_0^1 \sqrt{1 + u'(x)^2}\, dx.$$

求解当 $u(0) = u(1) = 0$. 时的 ∇f

几何上，你在大一微积分中学到，这仅仅是曲线 $u(x)$ 从 $x = 0$ 到 $x = 1$ 的长度。为了求导，首先注意到普通单变量微积分给我们提供了线性化

$$d\left(\sqrt{1+v^2}\right) = \sqrt{1+(v+dv)^2} - \sqrt{1+v^2} = \left(\sqrt{1+v^2}\right)' dv = \frac{v}{\sqrt{1+v^2}} dv\,.$$

因此，

$$\begin{aligned} df &= f(u+du) - f(u) \\ &= \int_0^1 \left(\sqrt{1+(u+du)'^2} - \sqrt{1+u'^2}\right) dx \\ &= \int_0^1 \frac{u'}{\sqrt{1+u'^2}}\, du' dx. \end{aligned}$$

然而，这是一个在 du' 上的线性算子，而不是（直接）在 du 上。抽象地说，这是可以的，因为 du' 本身是在 du 上的线性运算，所以我们有 $f'(u)[du]$ 作为两个线性运算的复合。然而，用 du 明确地重写它更有揭示性，例如，为了定义 ∇f。为了实现这一点，我们可以应用分部积分法来获得

$$f'(u)[du] = \int_0^1 \frac{u'}{\sqrt{1+u'^2}}\, du' dx = \left.\frac{u'}{\sqrt{1+u'^2}} du\right|_0^1 - \int_0^1 \left(\frac{u'}{\sqrt{1+u'^2}}\right)' du\, dx\,.$$

注意，到目前为止，我们不需要利用“边界条件”$u(0) = u(1) = 0$进行此计算。然而，如果我们想限制自己到这样的函数$u(x)$，那么我们的扰动du不能改变端点值，即我们必须有$du(0) = du(1) = 0$。（从几何上讲，假设我们想要找到（0, 0）和（1, 0）之间弧长最小的u，因此我们需要固定端点。）这意味着上述方程中的边界项为零。因此，我们有

$$df = -\int_0^1 \underbrace{\left(\frac{u'}{\sqrt{1+u'^2}}\right)'}_{\nabla f} du\, dx = \langle \nabla f, du\rangle\,.$$

此外，请注意，使泛函 f 最小的 u 具有性质 $\nabla f|_u = 0$。因此，对于

一个 u 最小化泛函 f（最短曲线），我们必须得到以下结果：

$$\begin{aligned} 0 = \nabla f &= -\left(\frac{u'}{\sqrt{1+u'^2}}\right)' \\ &= -\frac{u''\sqrt{1+u'^2} - u'\frac{u''u'}{\sqrt{1+u'^2}}}{1+u'^2} \\ &= -\frac{u''(1+u'^2) - u''u'^2}{(1+u'^2)^{3/2}} \\ &= -\frac{u''}{(1+u'^2)^{3/2}}. \end{aligned}$$

因此，$\nabla f = 0 \implies u''(x) = 0 \implies u(x) = ax + b$ 对于常数 a, b；对于这些边界条件 $a = b = 0$。换句话说，u 是水平直线段！

因此，我们恢复了熟悉的结果，即 $\mathbb{R}^2$ 中的直线段是最短的曲线之间的 n 两个点！

备注47。请注意，表达式 $\frac{u''}{(1+u'^2)^{3/2}}$ 是由 $y = u(x)$ 定义曲线的曲率的多元微积分公式。弧长梯度的（负）曲率并非巧合，最小弧长出现在零梯度 $=$ 零曲率处。

10.4 欧拉-拉格朗日方程

这种计算方式是被称为变分法的一个主题的一部分。当然，上面例子中的最终答案（一条直线）可能是显而易见的，但类似的方法可以应用于许多更有趣的问题。我们可以将这种方法概括如下：

示例 48
设 $f(u) = \int_a^b F(u, u', x)\, dx$，其中 u 是 $[a, b]$ 上的可微函数。假设 u 的端点是固定的（即其在 $x = a$ 和 $x = b$ 处的值是常数）。让我们计算 df 和 ∇f。

我们找到：

$$\begin{aligned} df &= f(u + du) - f(u) \\ &= \int_a^b \left(\frac{\partial F}{\partial u} du + \frac{\partial F}{\partial u'} du'\right) dx \\ &= \underbrace{\frac{\partial F}{\partial u'} du\Big|_a^b}_{=0} + \int_a^b \left(\frac{\partial F}{\partial u} - \left(\frac{\partial F}{\partial u'}\right)'\right) du\, dx\,, \end{aligned}$$

在 a 或 b 处，我们使用了 $du = 0$ 的事实，如果端点 $u(a)$ 和 $u(b)$ 是固定的。因此，

$$\nabla f = \frac{\partial F}{\partial u} - \left(\frac{\partial F}{\partial u'}\right)',$$

等于零在极值处。注意，$\nabla f = 0$ 得到一个关于 u 的二阶微分方程，称为欧拉-拉格朗日方程！

备注49。符号 $\partial F/\partial u'$ 是变分法中一个众所周知令人困惑的方面——在保持 u 固定的情况下，求“关于 u'”的导数意味着什么？一个更明确、尽管更冗长的方法是

将表达方式视为将 $F(u, v, x)$ 视为三个无关参数的函数，我们仅在关于第二个参数 v 求导后用 $v = u'$ 替换：

$$\frac{\partial F}{\partial u'} = \left.\frac{\partial F}{\partial v}\right|_{v=u'}.$$

这个想法有众多美妙的应用。例如，在网上搜索有关“最速降线问题”（此处有动画演示）和/或“最小作用量原理”的信息。另一个例子是悬链线，它最小化了悬挂电缆的势能。关于这个主题的经典教科书是Gelfand和Fomin的《变分法》。

随机函数的11个导数

未提供翻译文本 lecture notes from a guest lecture by Gaurav Arya in IAP 这篇文章是Gaurav Arya在IAP的客座讲座

11.1 简介

在这个课程中，我们学习了如何求各种疯狂函数的导数。回忆一下我们最早的例子 s:

$$f(A) = A^2, \tag{8}$$

在 A 是一个矩阵的情况下。为了区分这个函数，我们不得不回到起点，并询问：

问题50。如果我们稍微扰动输入，输出会如何变化?

为此，我们写下了一些类似的内容：

$$\delta f = (A + \delta A)^2 - A^2 = A(\delta A) + (\delta A)A + \underbrace{(\delta A)^2}_{\text{neglected}}. \tag{9}$$

我们称 δf 和 δA 为极限情况下 δA 变得任意小的差分。然后我们必须问：

问题51. 我们可以忽略微分中的哪些项?

我们决定忽略$(\delta A)^2$，通过$(\delta A)^2$是“高阶的”这一事实来证明这一点。我们剩下导数算子$\delta A \mapsto A(\delta A) + (\delta A)A$：在$A$邻域内对$f$的最佳可能线性近似。从高层次来看，这里的挑战主要在于处理复杂的输入和输出空间：f是矩阵值，同时也是矩阵接受者。我们必须问自己：在这种情况下，导数的概念甚至意味着什么?

在这个讲座中，我们将面临一个类似的挑战，但会遇到一种更奇怪类型的函数。这次，我们函数的输出将是随机的。现在，我们需要重新审视同样的问题。如果输出是随机的，我们如何描述它对输入变化的响应？以及我们如何形成一个有用的导数概念?

11.2 随机规划

更精确地说，我们将考虑随机或随机函数 X，具有实数输入 $p \in \mathbb{R}$ 和实值随机 -
变量输出。作为一个映射，我们可以将 X 写作

$$p \mapsto X(p), \tag{10}$$

在 $X(p)$ 是一个随机变量的情况下。（为了简化问题，我们将在本章中考虑 $p \in \mathbb{R}$ 和 $X(p) \in \mathbb{R}$，尽管当然它们可以像其他章节中那样推广到其他向量空间。目前，随机性已经足够复杂，可以处理了。）

想法是我们只能根据依赖于 p 的概率分布从 $X(p)$ 中采样。一个简单的例子是从区间 $[0, p]$ 中均匀（等概率）采样实数。更复杂的例子，假设 $X(p)$ 符合尺度为 p 的指数分布，对应于随机采样的实数 $x \geq 0$，其概率与 $e^{-x/p}$ 成正比。这可以表示为 $X(p) \sim \text{Exp}(p)$，并在 Julia 中实现：

julia> 使用 Distributions

julia> 样本_X(p) = rand(指数(p)) sample_X (具有1
个方法的泛型函数)

我们可以取几个样本：

julia> sample_X(10.0) 1.78
49785709142214

julia> sample_X(10.0) 4.43
5847397169775

julia> sample_X(10.0) 0.68
23343897949835

julia> mean(sample_X(10.0) for i = 1:10^9) # mean = p 9.9999303482
91866 中文翻译：julia> mean(sample_X(10.0) for i = 1:10^9) # mean
= p 9.999930348291866

如果我们程序每次都给出不同的输出，那么有用的导数概念又是什么呢？在我们尝试回答这个问题之前，让我们先问一下为什么我们可能想要求导。答案是，我们可能非常感兴趣于随机函数的统计特性，即可以用平均值表示的值。即使一个函数是随机的，其平均值（“期望值”），假设平均值存在，也可以是其参数的确定性函数，该函数具有传统的导数。

所以，为什么不先取平均值，然后再取这个平均值的普通导数呢？这种简单的方法适用于非常基本的随机函数（例如，上面的指数分布具有期望值 p，导数为 1），但对于更复杂的分布（如大型计算机程序在处理随机数时通常实现的那样）会遇到实际困难。

备注52。产生一个“无偏估计”$X(p)$比精确计算一个统计量通常要容易得多。（在这里，无偏估计意味着$X(p)$的平均值等于我们感兴趣的统计量。）

例如，在深度学习中，“变分自编码器”（VAE）是一种固有的随机性非常常见的架构。通过运行随机模拟 $X(p)$，很容易得到损失函数的随机无偏估计：此时，损失函数 $L(p)$ 是 $X(p)$ 的“平均”值，表示为期望值 $\mathbb{E}[X(p)]$。然而，要精确计算损失 $L(p)$，就需要对所有可能的结果进行积分，这通常是不切实际的。现在，为了训练VAE，我们还需要对 $L(p)$ 进行微分，即对 $\mathbb{E}[X(p)]$ 关于 p 进行微分！

可能在物理学中找到更直观的例子，在那里随机性可能被嵌入到你对物理过程的模型中。在这种情况下，很难回避你需要处理随机性的事实！例如，你可能有两个以平均速率 r 相互作用的粒子。但在现实中，这些相互作用实际发生的时间遵循随机过程。（事实上，第一次相互作用之前的时间可能是指数分布的，尺度为 $1/r$。）而且，如果你想（例如）将你的随机模型参数拟合到现实世界数据，那么拥有导数再次非常有用。

如果我们无法精确计算我们感兴趣的统计量，那么假设我们能够精确计算其导数似乎是不合理的。然而，我们可能希望随机估计其导数。也就是说，如果 $X(p)$ 代表产生我们统计量无偏估计的完整程序，那么我们希望我们的导数概念具有以下一个特性：我们应该能够从中构建一个无偏梯度估计器[14] $X'(p)$

[14]For more discussion of these concepts, see (e.g.) the review article “Monte Carlo gradient estimation in machine learning” (2020) by Mohamed *et al.* (https://arxiv.org/abs/1906.10652).

令人满意

$$\mathbb{E}[X'(p)] = \mathbb{E}[X(p)]' = \frac{\partial \mathbb{E}[X(p)]}{\partial p}. \tag{11}$$

当然，存在无限多个这样的估计量。例如，对于任何估计量 $X'(p)$，我们可以添加任何具有零平均的其他随机变量，而不会改变期望值。但在实践中，还有两个额外的考虑：(1) 我们希望 $X'(p)$ 容易计算/采样（与 $X(p)$ 一样容易），(2) 我们希望 $X'(p)$ 的方差（“分布”）足够小，以至于我们不需要太多样本来准确估计其平均值（希望不会比估计 $\mathbb{E}[X(p)]$ 差）。

11.3 随机微分和重参数化技巧

让我们从回答我们的第一个问题（问题50）开始：$X(p)$如何响应p的变化？让我们考虑一个特定的p并写下随机微分方程，取一个微小但非无穷小的δp，以避免现在考虑无穷小：

$$\delta X(p) = X(p + \delta p) - X(p), \tag{12}$$

δp代表p中的任意小变化。$\delta X(p)$是什么样的对象？

由于我们在减去两个随机变量，它本身也应该是随机变量。然而，$\delta X(p)$仍然没有完全指定！我们只指定了$X(p)$和$X(p+\delta p)$的边缘分布：要能够减去这两个变量，我们需要知道它们的联合分布。

一种可能性是将 $X(p)$ 和 $X(p + \delta p)$ 视为独立的。这意味着 $\delta X(p)$ 将被构建为独立样本的差值。让我们看看在这种情况下 $\delta X(p)$ 的样本会是什么样子！

julia>样本_X(p) = rand(指数分布(p)) sample_X (具
有1个方法的泛型函数)

julia>样本_δX(p, δp) =样本_X(p + δp) - 样本_X(p)样本_δX（具有1
个方法的泛型函数）

```
julia> p = 10; δp = 1e-5;
```

julia> sample_δX(p, δp) -26.0
00938718875904

julia>样本_δX(p, δp) -2.6157
162001718092

julia> sample_δX(p, δp) 6.35
2622554495474

julia>样本_δX(p, δp) -9.5321
5951927184

julia>样本_δX(p, δp) 1.2232
268930932104

我们可以观察到一些令人担忧的事情：即使对于一个非常小的 δp（我们选择了 $\delta p = 10^{-5}$），$\delta X(p)$ 仍然相当大 e:

基本上与原始随机变量一样大。如果我们想从 $\delta X(p)$ 构造一个导数，这可不是什么好消息：我们更希望它的幅度随着 δp 越来越小，就像在非随机情况下一样。在计算上，这将使得通过平均许多样本的 sample_δX(p, δp) / δp 来确定 $\mathbb{E}[X(p)]'$ 非常困难：我们需要大量的样本，因为对于小的 δp，方差，即随机值的“分布”，非常大。

让我们尝试一种不同的方法。对于所有 p 来说，将 $X(p)$ 视为一个随机变量族，所有这些变量都定义在同一个概率空间上是很自然的。概率空间，经过一些简化，是一个样本空间 Ω，并在样本空间上定义了一个概率分布 $\mathbb{P}$。从这个角度来看，每个 $X(p)$ 都可以表示为一个函数 $\Omega \to \mathbb{R}$。要从特定的 $X(p)$ 中采样，我们可以想象从 Ω 中根据 $\mathbb{P}$ 抽取一个随机 ω，然后将它插入到 $X(p)$ 中，即计算 $X(p)(\omega)$。（在计算上，这是大多数分布实际上是如何实现的：你从一个非常简单的分布开始，使用原始的伪随机数生成器，例如从 $\Omega = [0, 1)$ 中均匀抽取值 ω，然后通过某种方式转换 ω 来构建其他分布。）直观上，所有的“随机性”都存在于概率空间中，并且关键地 $\mathbb{P}$ 不依赖于 p：当 p 变化时，$X(p)$ 只是成为 Ω 上的一个不同的确定性映射。

这里的关键是，所有 $X(p)$ 函数现在都依赖于一个共享的随机源：ω 的随机抽取。这意味着 $X(p)$ 和 $X(p + \delta p)$ 具有一个非平凡的联合分布：它看起来是什么样子?

为了具体化，让我们研究上面的指数随机变量 $X(p) \sim \text{Exp}(p)$。使用“逆抽样”参数化，可以选择 Ω 为 $[0, 1)$，并且 $\mathbb{P}$ 为在 Ω 上的均匀分布；对于任何分布，我们都可以构造 $X(p)$ 为在 Ω 上的相应非递减函数（由 $X(p)$ 的累积概率分布的逆给出）。应用于 $X(p) \sim \text{Exp}(p)$，逆方法给出 $X(p)(\omega) = -p \log(1 - \omega)$。这已在下面实现，并且与上面使用的模糊 rand(Exponential(p)) 函数相比，是一种理论上等效的采样 $X(p)$ 的方法：

julia> sample_X2(p, ω) = -p * log(1 - ω) sample_X2 (
具有1个方法的泛型函数)

```
julia> # rand() samples a uniform random number in [0,1)
julia> sample_X2(p) = sample_X2(p, rand())
sample_X2 (generic function with 2 methods)
```

julia> sample_X2(10.0) 8.38
0816941818618

julia> sample_X2(10.0) 2.07
3939134369733

julia> sample_X2(10.0) 29.9
4586208847568

julia> sample_X2(10.0) 23.9
1658360124792

好的，那么我们的联合分布是什么样的？如图13所示，我们可以绘制 $X(p)$ 和 $X(p + \delta p)$ 如下

[15] Most computer hardware cannot generate numbers that are actually random, only numbers that *seem* random, called “pseudo-random” numbers. The design of these random-seeming numeric sequences is a subtle subject, steeped in number theory, with a long history of mistakes. A famous ironic quotation in this field is (Robert Coveyou, 1970): “Random number generation is too important to be left to chance.”

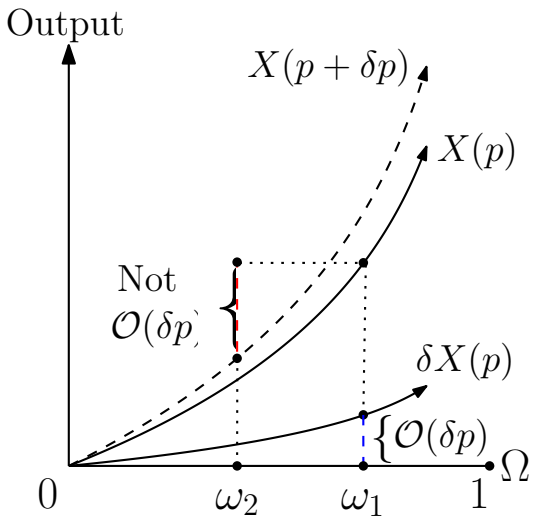


图13：对于通过反演法参数化的 $X(p) \sim \text{Exp}(p)$，我们可以将 $X(p)$、$X(p+\delta p)$ 和 $\delta X(p)$ 写作定义在具有 $\mathbb{P} = \text{Unif}(0,1)$ 的概率空间上的从 $\Omega = [0,1] \to \mathbb{R}$ 的函数。

函数在Ω上。为了同时采样这两个函数，我们使用相同的ω选择：因此，$\delta X(p)$可以通过在每个Ω上逐点减去两个函数来形成。最终，$\delta X(p)$本身是在同一概率空间上的一个随机变量，以相同的方式采样：我们根据$\mathbb{P}$随机选择ω，并使用上面描述的函数$\delta X(p)$评估$\delta X(p)(\omega)$。我们使用独立样本的第一个方法在图13中以红色表示，而第二个方法以蓝色表示。现在我们可以看到独立样本方法的缺陷：独立样本的$\mathcal{O}(1)$大小的“噪声”掩盖了$\mathcal{O}(\delta p)$大小的“信号”。

关于我们的第二个问题（问题51）：实际上如何取 $\delta p \to 0$ 的极限并计算导数？思路是在每个固定的样本 $\omega \in \Omega$ 上对 $\delta X(p)$ 进行微分。在概率论术语中，我们取随机变量 $\delta X(p)/\delta p$ 当 $\delta p \to 0$ 时的极限：

$$X'(p) = \lim_{\delta p \to 0} \frac{\delta X(p)}{\delta p}. \tag{13}$$

对于通过反演法参数化的 $X(p) \sim \text{Exp}(p)$，我们得到：

$$X'(p)(\omega) = \lim_{\delta p \to 0} \frac{-\delta p \log(1-\omega)}{\delta p} = -\log(1-\omega). \tag{14}$$

再次，$X'(p)$是在相同概率空间上的一个随机变量。声称$X'(p)$是我们一直在寻找的导数概念！确实，$X'(p)$本身实际上是一个有效的梯度估计器：

$$\mathbb{E}[X'(p)] = \mathbb{E}\left[\lim_{\delta p \to 0} \frac{\delta X(p)}{\delta p}\right] \overset{?}{=} \lim_{\delta p \to 0} \frac{\mathbb{E}[\delta X(p)]}{\delta p} = \frac{\partial \mathbb{E}[X(p)]}{\partial p}. \tag{15}$$

严格来说，需要证明上述极限与期望的交换是合理的。然而，在本章中，我们将满足于一种粗略的经验性证明：

```
julia> X′(p, ω) = -log(1 - ω)
X′ (具有1个方法的通用函数)

julia> X′(p) = X′(p, rand())
X′ (具有2个方法的通用函数)
```

julia> mean(X′(10.0) for i in 1:10000)
1.011689946421105

所以 $X'(p)$ 的确平均为 1，这是有道理的，因为 Exp(p) 的期望是 p，对于任何 p 的选择，其导数为 1。然而，关键在于这个导数的概念也适用于由简单随机变量（如指数随机变量）通过复合形成的更复杂的随机变量。事实上，它遵循与通常相同的链式法则！

让我们演示这一点。使用第8章中引入的共轭数，我们可以对指数分布样本平方的期望值进行微分，而无需对该数量有解析表达式。（我们推导出的 X' 的表达式已经作为 Julia 中的 ForwardDiff.jl 包的共轭数规则实现。）输出的共轭数的原值和共轭值是 $(X(p), X'(p))$ 联合分布的样本。

julia> 使用 Distributions, ForwardDiff: Dual

julia> 样本_X(p) = rand(Exponential(p))2 sample_X (
具有1个方法的泛型函数)

julia> sample_X(Dual(10.0, 1.0)) # sample a single dual number! Dual{Nothing}(
153.74964559529033,30.749929119058066)

julia> # 获取导数！julia> mean(sample_X(Dual(10.0, 1.0)).partials[1] for i in 1:10000
) 40.016569793650525

使用“重参数化技巧”来形成梯度估计器，就像我们在这里所做的那样，是一个相当古老的想法。它也被称为“路径”梯度估计器。最近，由于它在VAEs中的应用[例如Kingma & Welling (2013)：https://arxiv.org/abs/1312.6114]，它已经在机器学习中变得非常流行，并且可以在网上找到大量关于它的资源。由于组合只是通过通常的链式法则工作，它也可以在反向模式下工作，并且可以微分比上面更复杂的函数！

11.4 处理离散随机性

到目前为止，我们只考虑了连续随机变量。让我们看看离散随机变量会如何改变！让我们考虑一个简单的伯努利变量 $X(p) \sim$ Ber(p)，其以概率 p 为 1，以概率 $1-p$ 为 0。

julia> 样本_X(p) = rand(Bernoulli(p))
sample_X（具有1个方法的通用函数）

julia> p = 0.5 0.6

julia> sample_X(δp) # 生成false/true，相当于0/1
真

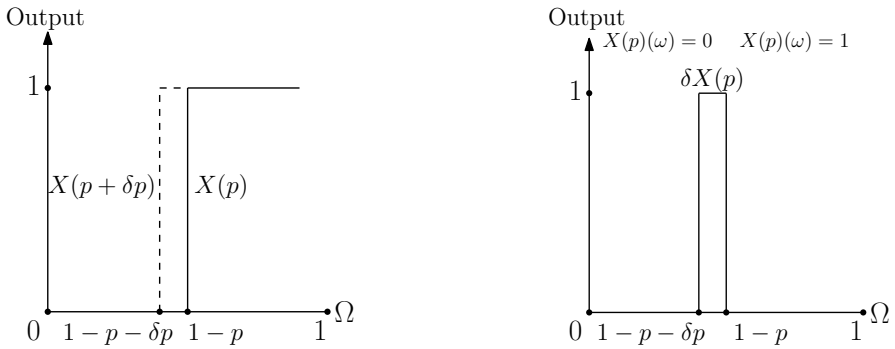


图14：对于$X(p) \sim$ Ber(p)通过反演法参数化，$X(p)$、$X(p+\delta p)$和$\delta X(p)$作为函数Ω：$[0,1] \to \mathbb{R}$的图。

julia> 样本_X(δp) 错误

julia> 样本_X(δp)
真

参数化伯努利变量如图2所示。再次使用反演法，伯努利变量的参数化看起来像一个阶梯函数：对于 $\omega < 1-p$，$X(p)(\omega) = 0$，而对于 $\omega \geq 1-p$，$X(p)(\omega) = 1$。

现在，当我们扰动 p 时会发生什么？让我们想象通过一个正数 δp 扰动 p。如图2所示，这里发生了非常不同的定性变化。在几乎每一个 ω 除了一个小的概率区域 δp，输出没有变化。因此，我们在上一小节中定义的量 $X'(p)$（严格来说，它是由一个“几乎确定”的极限定义的，该极限忽略了概率为0的区域）在每一个 ω 处都是0：毕竟，对于每一个 ω，都存在足够小的 δp 使得 $\delta X(p)(\omega) = 0$。

然而，这里确实有一个重要的导数贡献需要考虑。伯努利分布的期望是 p，所以我们预计导数应该是 1：但 $\mathbb{E}[X'(p)] = \mathbb{E}[0] = 0$。出问题的是，尽管 $\delta X(p)$ 以极小的概率为 0，但在这个极小概率区域上 $\delta X(p)$ 的值是 1，这是一个很大的值。特别是，当 δp 趋近于 0 时，它并不趋近于 0。因此，为了发展 $X(p)$ 的导数概念，我们需要以某种方式捕捉这些带有“无穷小”概率的大跳跃。

最近（2022年）由本章作者（Gaurav Arya）以及Frank Schäfer、Moritz Schauer和Chris Rackauckas共同发表的一篇论文（https://arxiv.org/abs/2210.08572），致力于将上述想法扩展到离散随机性，通过名为StochasticAD.jl的软件包实现此类随机过程的自动微分，从而发展出“随机导数”的概念。它将双数概念推广到随机三元组，其中包含第三个分量以精确捕捉这些大跳跃。例如，伯努利变量的随机三元组可能看起来像这样：

julia> 使用 StochasticAD, Distributions julia> f(p) = rand(Bernoulli(p)) # 1 以概率 p，
否则为 0 julia> stochastic_triple(f, 0.5) # 将 0.5 + δp 输入到 f StochasticTriple of Int64
: 0 + 0ε + （1 以概率 2.0ε）

这里，δp 表示为 ϵ，想象成“无穷小单位”，因此上述三元组表示从0到1的翻转，其概率的导数为2。

然而，这些问题的许多方面仍然很难，还有很多改进等待着未来的发展！如果您想了解更多，可能会对上面链接的论文和我们的包感兴趣，以及Mohamed等人于2020年发表的综述文章（https://arxiv.org/abs/1906.10652），这是一篇关于梯度估计领域的优秀调查。

课程结束时，我们考虑了一个使用StochasticAD.jl的可微随机游走示例。这里就是它！

julia> 使用 Distributions, StochasticAD

julia> 函数 X(p)
n = 0 对 i 在 1:100 中 n += rand(Bernoulli(p * (1 - (n+i)/2
00))) 结束 返回 n 结束

X (具有1个方法的通用函数)

julia> mean(X(0.5) for _ in 1:10000) # 在 p = 0.5 处计算 E[X(p)] 0.5 32.6956

julia> st = 随机三元组(X, 0.5) # 在 p = 0.5 下采样一个随机三元组 StochasticTriple of Int64:

32 + 0δp + （1 以概率 74.176358182210 52δp）

julia> 导数贡献(st) # 由这个三元组产生的导数估计 74.17635818221052

julia> # 通过多次采样计算 E[X(p)] 的 d/dp julia> mean(derivative_contribution(stochastic_triple(f, 0.5
)) for i in 1:10000) 56.65142976168479

12 二阶导数，双线性映射和Hessian矩阵

在这一章中，我们将本课程的原则应用于二阶导数，从概念上讲，它们只是导数的导数，但结果却有许多有趣的推论。我们从一个（可能）熟悉的来自多元微积分的标量值函数的例子开始，其中二阶导数就是一个称为Hessian的矩阵。然而，随后我们将展示，类似的原则可以应用于更复杂的输入和输出空间，推广到f''作为一个对称双线性映射的概念。

12.1 标量值函数的Hessian矩阵

回忆一下，对于一个将列向量 $x \in \mathbb{R}^n$ 映射到标量 ($f : \mathbb{R}^n \mapsto \mathbb{R}$) 的函数 $f(x) \in \mathbb{R}$，其第一导数 f' 可以用多元微积分中熟悉的梯度 $\nabla f = (f')^T$ 来表示：

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix} .$$

如果我们把 ∇f 视为一个映射 $x \in \mathbb{R}^n \mapsto \nabla f \in \mathbb{R}^n$ 的新（通常是非线性）函数，那么它的导数是一个 $n \times n$ 雅可比矩阵（一个将向量映射到向量的线性算子），我们可以用 f 的二阶导数来明确写出：

$$(\nabla f)' = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{pmatrix} = H .$$

这个矩阵，在此表示为 H，被称为 f 的 Hessian，其项为：

$$H_{i,j} = \frac{\partial^2 f}{\partial x_j \partial x_i} = \frac{\partial^2 f}{\partial x_i \partial x_j} = H_{j,i} .$$

你可以以任意顺序求偏导数是一个来自多元微积分的熟知事实（有时称为“混合偏导数的对称性”或“混合偏导数的相等性”），这意味着Hessian是一个对称矩阵 $H = H^T$。（我们稍后将会看到，这种对称性非常普遍地来自于二阶导数的构造。）

示例53
对于 $x \in \mathbb{R}^2$ 和函数 $f(x) = \sin(x_1) + x_1^2 x_2^3$，其梯度是

$$\nabla f = \begin{pmatrix} \cos(x_1) + 2x_1 x_2^3 \\ 3x_1^2 x_2^2 \end{pmatrix} ,$$

并且其Hessian是

$$H = (\nabla f)' = \begin{pmatrix} -\sin(x_1) + 2x_2^3 & 6x_1 x_2^2 \\ 6x_1 x_2^2 & 6x_1^2 x_2 \end{pmatrix} = H^T .$$

如果我们把Hessian视为∇f的雅可比矩阵，这告诉我们$H\,dx$预测∇f的一阶变化：

$$d(\nabla f) = \nabla f|_{x+dx} - \nabla f|_x = H\,dx\,.$$

注意，$\nabla f|_{x+dx}$ 表示在 $x+dx$ 上评估 ∇f，这与我们在 dx 上执行 $f'(x) = (\nabla f)^T$ 的 $df = (\nabla f)^T dx$ 非常不同。

相反，我们也可以将其视为预测 f 的二阶变化，一个二次近似（这可以看作是多维泰勒级数的前三项）：

$$f(x+\delta x) = f(x) + (\nabla f)^T\,\delta x + \frac{1}{2}\delta x^T\,H\,\delta x + o(\|\delta x\|^2)\,,$$

在 x 处同时评估 ∇f 和 H，我们已从无穷小 dx 转变为有限变化 δx，以强调在 $\|\delta x\|$ 中高于二阶的项被忽略的近似观点。您可以通过多种方式推导出这一点，例如通过对 δx 求导的两边来再现 $\nabla f|_{x+\delta x} = \nabla f|_x + H\,\delta x + o(\delta x)$：$f$ 的二次近似对应于 ∇f 的线性近似。与此方程相关，另一个有用（并且可以说是更基本的）关系，我们可以推导出（以下将更普遍地推导出）是：

$$dx^T H dx' = f(x+dx+dx') + f(x) - f(x+dx) - f(x+dx') = f''(x)[dx, dx']$$

在 dx 和 dx' 是两个独立的“无穷小”方向，并且我们已省略高于二阶的项的情况下。这个公式非常有启发性，因为它使用 H 将两个向量映射到一个标量，我们将在下面将其推广到双线性映射 $f''(x)[dx, dx']$ 的概念。此公式在 dx 和 dx' 交换时显然是对称的——$f''(x)[dx, dx'] = f''(x)[dx', dx]$——这又将使我们再次回到下面的对称 $H = H^T$。

备注54。考虑Hessian矩阵与其他Jacobian矩阵。Hessian矩阵表示标量值多元函数的二阶导数，总是方形且对称。一般来说，Jacobian矩阵表示向量值多元函数的一阶导数，可能不是方形，且很少对称。（然而，Hessian矩阵是∇f函数的Jacobian！）

12.2 一般二阶导数：双线性映射

回忆一下，正如我们在整个课程中所做的那样，我们通过输入的小（“无穷小”）变化 dx 对其变化 df 进行线性化来定义函数 f 的导数：

$$df = f(x+dx) - f(x) = f'(x)[dx]\,,$$

隐式地省略高阶项。如果我们类似地考虑二阶导数 f'' 作为仅将过程应用于 f' 而不是 f，我们得到以下公式，这个公式容易写出但需要一些思考来解释：

$$df' = f'(x+dx') - f'(x) = f''(x)[dx'].$$

(符号：dx'不是dx的某种导数；上标仅表示x中的不同任意小变化.) df'是什么“东西”？ 让我们考虑一个简单的具体例子：

示例 55
考虑以下函数 $f(x)$：$\mathbb{R}^2 \mapsto \mathbb{R}^2$ 将二维向量 $x \in \mathbb{R}^2$ 映射到二维向量 $f(x) \in \mathbb{R}^2$：

$$f(x) = \begin{pmatrix} x_1^2 \sin(x_2) \\ 5x_1 - x_2^3 \end{pmatrix}.$$

它的第一导数由一个2 × 2雅可比矩阵描述：

$$f'(x) = \begin{pmatrix} 2x_1 \sin(x_2) & x_1^2 \cos(x_2) \\ 5 & -3x_2^2 \end{pmatrix}$$

将输入向量 x 中的微小变化 dx 映射到输出向量 f 中相应的微小变化 $df = f'(x)dx$。

什么是 $df' = f''(x)[dx']$? 它必须在 x 中进行一个小变化 $dx' = (dx'_1, dx'_2)$ 并返回我们雅可比矩阵 f' 中的首次变化 $df' = f'(x + dx') - f'(x)$。如果我们简单地取我们雅可比矩阵（从向量 x 到矩阵 f' 的函数）的每一项的微分，我们发现：

$$df' = \begin{pmatrix} 2\,dx'_1 \sin(x_2) + 2x_1 \cos(x_2)\,dx'_2 & 2x_1\,dx'_1 \cos(x_2) - x_1^2 \sin(x_2)\,dx'_2 \\ 0 & -6x_2\,dx'_2 \end{pmatrix} = f''(x)[dx']$$

这是，df'是　一个2 × 2的“无穷小”项矩阵，形状相同　　e 作为 f'。

从这个观点来看，$f''(x)$ 是一个作用于向量 dx' 并输出 2×2 矩阵 $f''(x)[dx']$ 的线性算子，但这是许多情况下，将线性算子写成“规则”而不是“矩阵”这样的“事物”更容易的一个例子。“事物”必须是某种“三维矩阵”，或者我们必须将 f' “向量化为”一个包含 4 个元素的“列向量”，以便写出其 4 × 4 雅可比矩阵，如第 3 节（这可能会掩盖问题的潜在结构）。

此外，由于这个 df' 是一个线性算子（一个矩阵），我们可以将其作用于另一个向量 $dx = (dx_1, dx_2)$ 以获得：

$$df' \begin{pmatrix} dx_1 \\ dx_2 \end{pmatrix} = \begin{pmatrix} 2\sin(x_2)\,dx'_1\,dx_1 + 2x_1\cos(x_2)(\,dx'_2\,dx_1 + \,dx'_1\,dx_2) - x_1^2\sin(x_2)\,dx'_2\,dx_2 \\ -6x_2\,dx'_2\,dx_2 \end{pmatrix} = f''(x)[dx'][dx]\,.$$

注意，这个结果，我们将在下面称为 $f''(x)[dx', dx]$，与 $f(x)$ (a 2-分量向量)具有“相同形状”。此外，如果我们交换 dx 和 dx'，它也不会改变：$f''(x)[dx', dx] = f''(x)[dx, dx']$，这是我们将在下面进一步讨论的第二导数的关键对称性。

df' 是一个与$f'(x)$相同“形状”的（无穷小）对象，而不是$f(x)$。在这里，f'是一个线性算子，因此其变化df'也必须是一个（无穷小）线性算子（线性算子中的“小变化”），因此我们可以对任意的dx (或δx)进行操作，形式如下：

$$df'[dx] = f''(x)[dx'][dx] := f''(x)[dx', dx]\,,$$

我们在简洁性方面将两个括号合并。这个最终结果 $f''(x)[dx', dx]$ 与原始输出 $f(x)$ 具有相同类型的对象（向量）。这意味着 $f''(x)$ 是一个双线性映射：作用于两个向量，并且对任意一个向量单独考虑时都是线性的。（我们很快就会看到 dx 和 dx' 的顺序并不重要：$f''(x)[dx', dx] = f''(x)[dx, dx']$。）

更精确地说，我们有以下内容。

定义 56（双线性映射）

设 U, V, W 为一个向量空间，不一定相同。那么，一个双线性映射是一个函数 $B : U \times V \to W$，将一个 $u \in U$ 和 $v \in V$ 映射到 $B[u, v] \in W$，使得我们在两个自变量上都有线性关系：

$$\begin{cases} B[u, \alpha v_1 + \beta v_2] = \alpha B[u, v_1] + \beta B[u, v_2] \\ B[\alpha u_1 + \beta u_2, v] = \alpha B[u_1, v] + \beta B[u_2, v] \end{cases}$$

对于任何标量 α, β，

如果 {v*}，即输出是一个标量，那么它被称为双线性形式。

请注意，在一般情况下，即使两个输入 $U = V$（是“相同类型”的向量 u, v，我们可能也有 $B[u, v] \neq B[v, u]$，但在 f'' 的情况下，会发生一些非常特别的事情。特别是，我们可以证明 $f''(x)$ 是一个对称双线性映射，意味着

$$f''(x)[dx', dx] = f''(x)[dx, dx']$$

对于任意的 dx 和 dx'。为什么？因为，将 f'' 的定义应用于从 dx' 到 f' 的变化，然后应用 f' 的定义，从 dx 到 f 的变化，我们可以重新排列项以获得：

$$\begin{aligned} f''(x)[dx', dx] &= f'(x + dx')[dx] - f'(x)[dx] \\ &= \left(f(x + \underbrace{dx' + dx}_{=dx+dx'}) - f(x + dx') \right) - (f(x + dx) - f(x)) \\ &= \boxed{f(x + dx + dx') + f(x) - f(x + dx) - f(x + dx')} \\ &= (f(x + dx + dx') - f(x + dx)) - (f(x + dx') - f(x)) \\ &= f'(x + dx)[dx'] - f'(x)[dx'] \\ &= f''(x)[dx, dx'] \end{aligned}$$

我们在其中用方框标注了 f'' 的中间公式，以自然的方式强调其对称性。（这个方法之所以有效的基本原因是对于任何向量空间，“$+$” 操作总是交换的。几何解释如图 15 所示。）

示例 57

让我们回顾多元微积分中的熟悉例子，$f : \mathbb{R}^n \to \mathbb{R}$。也就是说，$f(x)$ 是一个关于列向量 $x \in \mathbb{R}^n$ 的标量值函数。f'' 是什么？

回忆起

$$f'(x) = (\nabla f)^T \implies f'(x)[dx] = \text{scalar } df = (\nabla f)^T dx.$$

同样地，

$$\begin{aligned} f''(x)[dx', dx] &= \text{scalar from two vectors, linear in both} \\ &= dx'^T H dx \,, \end{aligned}$$

在 H 必须恰好是 Sec. 12.1 中引入的 $n \times n$ 矩阵 Hessian 矩阵，因为像 $dx'^T H dx$ 这样的表达式是映射两个向量到标量的最一般可能的双线性形式。此外，既然我们现在知道

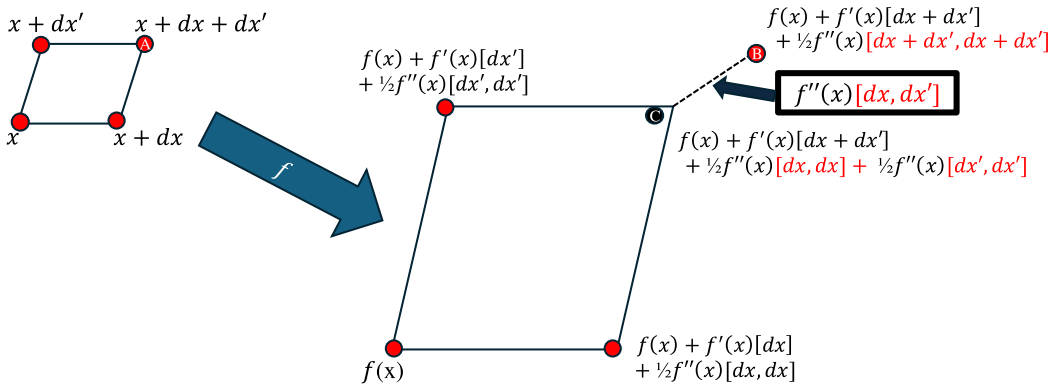


图15：$f''(x)[dx, dx']$的几何解释：一阶时，一个函数f将平行四边形映射到平行四边形。然而，二阶时它“展开”平行四边形：从点B (到点A)的像，从点C (到平行四边形)的完成，是二阶导数$f''(x)[dx, dx']$。f''作为双线性形式的对称性可以从输入平行四边形沿对角线从x到点A的镜像对称性几何上追溯。

那 f'' 总是对称双线性形式，我们必须有：

$$\begin{aligned} f''(x)[dx', dx] &= dx'^T H dx \\ &= f''(x)[dx, dx'] = dx^T H dx' = (dx^T H dx')^T \qquad (\text{scalar} = \text{scalar}^T) \\ &= dx'^T H^T dx \end{aligned}$$

对于所有 dx 和 dx'。这表明 $H = H^T$：Hessian 矩阵是对称的。如第 12.1 节所述，我们早已从多元微积分中知道这一点。然而，现在这个“混合偏导数的相等性”只是 f'' 是对称双线性映射的一个特例。

作为一个例子，让我们考虑上述一般公式的特殊情况：

> 示例 58
> 让 $f(x) = x^T A x$ 对于 $x \in \mathbb{R}^n$ 和 A 是一个 $n \times n$ 矩阵。如上所述，$f(x) \in \mathbb{R}$ (是标量输出)。计算 f''。

计算相当直接。首先，我们有

$$f' = (\nabla f)^T = x^T (A + A^T).$$

这表示 $\nabla f = (A + A^T)x$，是 x 的线性函数。因此，∇f 的雅可比矩阵是 Hessian $f'' = H = A + A^T$。此外，请注意，这表示

$$\begin{aligned} f(x) = x^T A x &= (x^T A x)^T \qquad (\text{scalar} = \text{scalar}^T) \\ &= x^T A^T x \\ &= \frac{1}{2}(x^T A x + x^T A^T x) = \frac{1}{2} x^T (A + A^T) x \\ &= \frac{1}{2} x^T H x = \frac{1}{2} f''[x, x]\,, \end{aligned}$$

这将是第12.3节二次逼近的一个特例（在这个例子中是精确的，因为 $f(x) = x^T A x$ 本身就是二次的）。

示例 59
让 $f(A) = \det A$ 对于 A 一个 $n \times n$ 矩阵。将 $f''(A)$ 表达为 $f''(A)[dA, dA']$ 的规则，以 dA 和 dA' 为基础。

从第3讲，我们有第一导数

$$f'(A)[dA] = df = \det(A)\,\mathrm{tr}(A^{-1}dA).$$

现在，我们想要计算这个公式中 $d'(df) = d'(f'(A)[dA]) = f'(A+dA')[dA] - f'(A)[dA]$ 的变化，即当我们把 A 替换为 dA' 时，同时将 dA 视为常数时的微分（表示为 d'）的变化：

$$\begin{aligned} f''(A)[dA, dA'] &= d'(\det A\,\mathrm{tr}(A^{-1}dA)) \\ &= \det A\,\mathrm{tr}(A^{-1}dA')\,\mathrm{tr}(A^{-1}dA) - \det A\,\mathrm{tr}(A^{-1}\,dA'A^{-1}\,dA) \\ &= f''(A)[dA', dA] \end{aligned}$$

在最后一行（对称性）可以通过迹的循环性质明确推导出来（尽管当然对于任何 f'' 都必须成立）。虽然这里的 f'' 是一个完美的双线性形式，作用于矩阵 dA, dA'，但将 f'' 表达为“Hessian 矩阵”并不很自然。

如果我们真的想用显式的Hessian矩阵来表示f''，我们可以使用第3节的“向量化”方法。让我们以使用Kronecker积（第3节）的tr($A^{-1}\,dA'A^{-1}\,dA$)项为例。一般来说，对于矩阵X, Y, B, C:

$$(\mathrm{vec}\,X)^T(B \otimes C)\,\mathrm{vec}\,Y = (\mathrm{vec}\,X)^T\,\mathrm{vec}\,(CYB^T) = \mathrm{tr}(X^TCYB^T) = \mathrm{tr}(B^TX^TCY)\,,$$

回忆起 (vec X)^T vec Y = tr(X^TY) 是Frobenius内积（第5节）。因此，

$$\mathrm{tr}(A^{-1}\,dA'A^{-1}\,dA) = \mathrm{vec}\,(dA'^T)^T(A^{-T} \otimes A^{-1})\,\mathrm{vec}\,(dA)\,.$$

这仍然不是我们想要的Hessian矩阵的形式，因为它涉及vec(dA'^T)而不是vec(dA')（这两个向量通过一个排列矩阵相关联，有时称为“交换”矩阵）。完成这个计算将是掌握克罗内克积的一个很好的练习，但得到一个明确的Hessian似乎是为了一个效用可疑的结果而进行的很多代数运算！

12.3 广义二次逼近

所以我们最终如何看待 f'' 呢？我们知道 f' 是 $f(x)$ 的线性化/线性近似，即 .

$$f(x + \delta x) = f(x) + f'(x)[\delta x] + o(\|\delta x\|).$$

现在，就像我们在12.1节上述简单情况的Hessian矩阵中做的那样，我们可以使用 f'' 来形成 $f(x)$ 的二次近似。特别是，可以证明

$$f(x + \delta x) = f(x) + f'(x)[\delta x] + \frac{1}{2}f''(x)[\delta x, \delta x] + o(\|\delta x\|^2).$$

注意，$\frac{1}{2}$因子与泰勒级数中相同。要推导这一点，只需将二次近似代入 o

$$f''(x)[dx, dx'] = f(x + dx + dx') + f(x) - f(x + dx) - f(x + dx').$$

并且检查右侧是否再现 $f''(x)$。（注意在这个公式中 dx 和 dx' 如何对称出现，这反映了 f'' 的对称性。）

12.4 赫斯矩阵与优化

许多二阶导数、Hessian矩阵和二次逼近在优化中的应用非常重要：函数 $f(x)$.[16] 的最小化（或最大化）

12.4.1 类牛顿法

当搜索复杂函数 $f(x)$ 的局部最小值（或最大值）时，一个常见的程序是通过对小 δx 的 $f(x+\delta x)$ 进行简化“模型”函数的近似，然后优化此模型以获得潜在的优化步骤。例如，对 $f(x+\delta x) \approx f(x)+f'(x)[\delta x]$（一个仿射模型，俗称“线性”）进行近似会导致梯度下降和相关算法。对于 $f(x+\delta x)$ 的更好近似通常会导致收敛速度更快的算法，因此一个自然的想法是利用二阶导数 f'' 来构建一个二次模型，如上所述，并加速优化。

对于无约束优化，最小化 $f(x)$ 相当于寻找导数 $f'=0$（即 $\nabla f=0$）的根，对 f 的二次近似得到导数 f' 的一阶（仿射）近似 $f'(x+\delta x) \approx f'(x)+f''(x)[\delta x]$。在 $\mathbb{R}^n$ 中，这是 $\delta(\nabla f) \approx H\delta x$。因此，最小化二次模型实际上是通过一阶近似找到 ∇f 根的牛顿步 $\delta x \approx -H^{-1}\nabla f$。因此，通过二次近似进行优化通常被视为牛顿算法的一种形式。如以下所述，在优化中也常见使用近似Hessian，从而产生“拟牛顿”算法。

更复杂的版本出现在有约束的优化中，例如在满足一个或多个非线性不等式约束 $c_k(x) \le 0$ 的条件下最小化目标函数 $f(x)$。在这种情况下，存在各种方法，这些方法同时考虑一阶和二阶导数，例如“顺序二次规划”[17] (SQP) 算法，该算法解决一系列涉及具有仿射约束的二次目标函数的“QP”近似（例如，参见 Nocedal 和 Wright 的《数值优化》一书，2006年）。

存在许多超出本课程范围的技术细节，必须解决这些细节才能将此类高级思想转化为实际算法。例如，二次模型仅对足够小的 δx 有效，因此必须有一些机制来限制步长。一种可能性是“回溯线搜索”：取牛顿步 $x+\delta x$，如果需要，逐步“回溯”到 $x+\delta x/10, x+\delta x/100, \ldots$，直到找到目标函数值足够降低。另一种常见想法是“信任域”：在约束 δx 足够小的条件下优化模型，例如 $\|\delta x\| \le s$ (球形信任域)，以及一些规则来根据模型预测 δf 的好坏自适应地增大或缩小信任域大小 (s)。根据对这些和其他细节的选择，存在许多类似牛顿/SQP算法的变体。

12.4.2 计算Hessian矩阵

通常，在更高维中找到 f'' 或 Hessian 矩阵往往计算成本高昂。如果 $f(x) : \mathbb{R}^n \to \mathbb{R}$，则 Hessian 矩阵 H 是一个 $n \times n$ 矩阵，如果 n 很大，它可能非常大——甚至存储 H 也可能具有威慑力，更

[16] Much of machine learning uses only variations on gradient descent, without incorporating Hessian information except implicitly via “momentum” terms. Partly this can be explained by the fact that optimization problems in ML are typically solved only to low accuracy, often have nonsmooth/stochastic aspects, rarely involve nonlinear constraints, and are often very high-dimensional. This is only a small corner of the wider universe of computational optimization!

[17] The term “programming” in optimization theory does not refer to software engineering, but is rather an anachronistic term for optimization problems. For example, “linear programming” (LP) refers to optimizing affine objectives and affine constraints, while “quadratic programming” (QP) refers to optimizing convex quadratic objectives with affine constraints.

计算较少。在使用自动微分（AD）时，Hessian 矩阵通常通过前向和反向模式的组合来计算（第 8.4.1 节），但 AD 并不能绕过大型 n 的基本缩放困难。

然而，人们可以以各种方式近似Hessian，而不是显式地计算H；在优化的背景下，近似Hessian出现在“拟牛顿”方法中，例如著名的“BFGS”算法及其变体。人们还可以推导出高效的方法来计算Hessian-向量乘积Hv，而无需显式地计算H，例如用于牛顿-克里洛夫方法。（这种乘积Hv相当于f'的方向导数，它可以通过第8.4.1节中所述的“正向-反向”AD高效地计算。）

12.4.3 极小值、极大值和鞍点

推广您可能从单变量和多变量微积分中回忆起的规则，我们可以使用二阶导数来确定极值是极小值、最大值还是鞍点。首先，标量函数 f 的极值是一个点 x_0，使得 $f'(x_0) = 0$。也就是说，

$$f'(x_0)[\delta x] = 0$$

对于任意的 δx。等价地，

$$\nabla f\big|_{x_0} = f'(x_0)^T = 0.$$

使用我们在 x_0 附近的二次近似，我们得到

$$f(x_0 + \delta x) = f(x_0) + \underbrace{f'(x_0)[\delta x]}_{=0} + \frac{1}{2} f''(x_0)[\delta x, \delta x] + o(\|\delta x\|^2).$$

局部最小值 x_0 的定义是对于任何 $\delta x \neq 0$ 且 $\|\delta x\|$ 足够小的 $f(x_0 + \delta x) > f(x_0)$。要在 $f' = 0$ 的点实现这一点，只需让 f'' 是一个正定的二次型即可：

$$f''(x_0)[\delta x, \delta x] > 0 \text{ for all } \delta x \neq 0 \iff \textbf{positive-definite } f''(x_0)\,.$$

例如，对于输入 $x \in \mathbb{R}^n$，使得 f'' 是一个实对称 $n \times n$ 海森矩阵 $f''(x_0) = H(x_0) = H(x_0)^T$，这对应于正定矩阵的通常标准：

$$f''(x_0)[\delta x, \delta x] = \delta x^T H(x_0) \delta x > 0 \text{ 对于所有 } \delta x \neq 0 \iff H(x_0) \text{ 正定} \iff H(x_0) > \text{的所有特征值 } 0\,.$$

在第一年微积分中，人们通常特别关注二维情况，其中 H 是一个 2×2 矩阵。在 2×2 情况下，有一种简单的方法来检查 H 的两个特征值的符号，以检查极值是极小值还是极大值：如果且仅当 $\det(H) > 0$ 和 $\operatorname{tr}(H) > 0$，特征值都是正的，因为 $\det(H) = \lambda_1 \lambda_2$ 和 $\operatorname{tr}(H) = \lambda_1 + \lambda_2$。然而，在更高维的情况下，需要更复杂的技巧来计算特征值和/或检查正定性，例如在麻省理工学院的 18.06（线性代数）和/或 18.335（数值方法导论）课程中讨论的那样。（在实践中，人们通常通过执行一种称为 Cholesky 分解的高斯消元法来检查正定性，并检查对角“主元”元素是否 > 0，而不是通过计算特征值，这要昂贵得多。）

类似地，当 f'' 是负定的，或者等价地，当 Hessian 的所有特征值都是负数时，点 x_0 是局部最大值。此外，当 f'' 是不定时，x_0 是鞍点，即特征值包括正数和负数。然而，某些特征值为零的情况更复杂，例如，当所有特征值都是 ≥ 0 但某些是 $= 0$ 时，该点是否为最小值取决于高阶导数。

12.5 进一步阅读

所有关于“双线性形式”等的形式主义可能看起来是为了抽象而进行的抽象探索。我们难道不能总是通过选择基（“向量化”我们的输入和输出）将事物简化为普通矩阵吗？然而，我们往往不想这样做，原因和为什么我们通常更喜欢将一阶导数表示为线性算子而不是显式的雅可比矩阵相同。将线性或双线性算子写成显式矩阵，例如在3节中vec($A\,dA + dA\,A$) = $(I \otimes A + A^T \otimes I)$ vec(dA)，往往掩盖了算子的底层结构，并引入了大量的代数复杂性，这毫无目的，同时也可能计算成本高昂（例如，用小矩阵A交换大矩阵$I \otimes A$）。

我们如本章所讨论的，将二次运算推广到任意向量空间的重要一般化形式是双线性映射和双线性形式，许多教科书和其他资料讨论了这些想法及其变体。例如，我们看到二阶导数可以看作是对称双线性形式。这与一个二次形式 $Q[x]$ 密切相关，这是通过将相同的向量两次代入对称双线性形式 $B[x, y] = B[y, x]$ 得到的，即 $Q[x] = B[x, x]$。(乍一看，可能觉得 Q 比 B “包含的信息更少”，但实际上并非如此。很容易看出，可以通过 $B[x, y] = (Q[x + y] - Q[x - y])/4$，称为“极化恒等式”，从 B 中恢复 Q。) 例如，出现在 $f(x + \delta x)$ 的二次近似中的 $f''(x)[\delta x, \delta x]/2$ 项是一个二次形式。$f''(x)$ 最熟悉的多变量版本是当 x 是列向量且 $f(x)$ 是标量时的 Hessian 矩阵，可汗学院有关于二次近似的初步介绍。

正定Hessian矩阵，或更一般地，确定二次型f''出现在标量值函数$f(x)$的极值（$f' = 0$）处，这些函数是局部极小值。关于这个想法有更多形式化的处理，反之，可汗学院有简单的二维版本，你可以通过观察行列式和单个条目（或迹）来检查2×2的特征值的符号。关于为什么病态的Hessian矩阵往往会使最速下降收敛缓慢，有一个很好的stackexchange讨论。关于这个主题的一些多伦多课程笔记也可能很有用。

最后，例如，请参阅斯坦福关于使用信任区域的序列二次优化的笔记（第2.2节），以及18.335关于BFGS拟牛顿方法的笔记。球面上的二次优化问题具有强对偶性，因此可以有效地求解，这一点在《凸优化》一书的第5.2.4节中进行了讨论。在自动Hessian计算方面已经做了大量工作，但对于大规模问题，你可能只能高效地计算Hessian-向量乘积，这等价于梯度的方向导数，可用于（例如）牛顿-克罗伊洛方法。

海森矩阵也被称为“曲率矩阵”，尤其是在优化中。如果我们有一个关于 n 变量的标量函数 $f(x)$，其“图形”是 R^{n+1} 中的点集 $(x, f(x))$；我们称最后一个维度为“垂直”维度。在“临界点” x（处，$\nabla f = 0$），那么 $v^T H v$ 是通常在第一年微积分中教授的曲线的普通曲率，该曲线是通过与从 x 和垂直方向 v 的平面相交得到的图形（“法线截面”）。H 的行列式，有时称为海森行列式，给出了高斯曲率。

一个与之密切相关的是单位法向量的导数。对于前一段中提到的图，我们可以假设 {v*} 到二阶。很容易看出，在任意点 {v*} 处，切线具有形式 ({v*}[{v*}]) {v*}，而法线则是 ({v*} 1)。在 {v*} 0 附近，这是一个二阶单位法向量，其导数是 ({v*} 0)。投影到水平方向，我们看到 Hessian 是单位法向量的导数。这在微分几何中被称为“形状算子”。

13 特征问题导数

13.1 在单位球上求导

几何上，我们知道球面上的速度向量（等价于切线）与半径垂直。我们的微分表示法以代数方式表达这一点，因为给定 $x \in \mathbb{S}^n$，我们得到 $x^T x = 1$，这表明

$$2x^T dx = d(x^T x) = d(1) = 0.$$

换句话说，在球体上的点 x（如果你愿意，可以称之为半径），dx，沿着球体移动的约束的线性化满足 $dx \perp x$。这是我们第一次看到无穷小扰动 dx 被约束的例子。见图16。

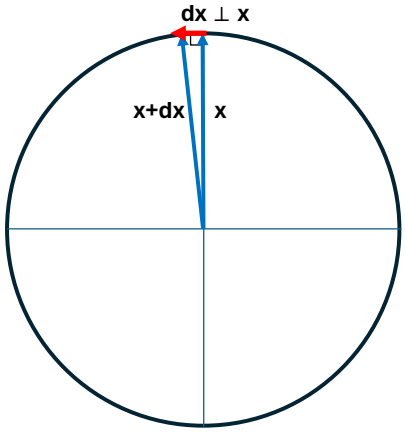


图16：球面上的微分($x^T x = 1$)：微分 dx 被限制为垂直于 x。

13.1.1 特殊情况：一个圆

让我们简单地考虑平面中的单位圆，其中 $x = (\cos\theta, \sin\theta)$ 对于某些 $\theta \in [0, 2\pi)$。那么，

$$x^T dx = (\cos\theta, \sin\theta) \cdot (-\sin\theta, \cos\theta) d\theta = 0.$$

这里，我们可以将 x 视为“外禀”坐标，因为它是在 $\mathbb{R}^2$ 中的一个向量。另一方面，θ 是一个“内禀”坐标，因为圆上的每个点都由一个 θ 指定。

第13.1.2节 在球体上

您可能记得，对于任意单位向量 $x^T x =$，秩为1的矩阵 xx^T 是一个投影矩阵（意味着它等于它的平方，并且它是对称的），它将向量投影到 x 方向上的分量。相应地，$I - xx^T$ 也是一个投影矩阵，但投影到 x 垂直的方向：从几何上看，该矩阵消除了 x 方向上的分量。特别是，如果 $x^T dx = 0$，那么 $(I - xx^T)dx = dx$. 因此

如果 $x^T dx = 0$ 且 A 是一个对称矩阵，我们有

$$\begin{aligned} d\left(\frac{1}{2}x^T Ax\right) &= (Ax)^T dx \\ &= x^T A(dx) \\ &= x^T A(I - xx^T)dx \\ &= ((I - xx^T)Ax)^T dx. \end{aligned}$$

换句话说，$(I - xx^T)Ax$ 是 $\frac{1}{2}x^T Ax$ 在球面上的梯度。

所以我们刚才做了什么？为了获得球面上的梯度，我们需要（i）一个在切线方向上正确的函数线性化，以及（ii）一个切向的方向（即满足线性化约束）。使用这个方法，我们得到球面上一般标量函数的梯度：

定理60
给定 $f : \mathbb{S}^n \to \mathbb{R}$，我们有

$$df = g(x)^T dx = ((I - xx^T)g(x))^T dx.$$

这个证明与我们之前对 $f(x) = \frac{1}{2}x^T Ax$ 做的完全相同。

13.2 在正交矩阵上求导

设 Q 为一个正交矩阵。那么，在计算上（如在 Julia 笔记本中执行的操作），可以看到 $Q^T d$ Q 这是一个反对称矩阵（有时称为斜对称矩阵）。

定义 61
一个矩阵 M 是反对称的，如果 $M = -M^T$。注意，因此所有反对称矩阵的对角线上都有零。

实际上，我们可以证明 $Q^T dQ$ 是反对称的。

定理62
给定 Q 是一个正交矩阵，我们有 $Q^T dQ$ 是反对称的。

证明。正交约束意味着 $Q^T Q = I$。对这方程求导，我们得到

$$Q^T dQ + dQ^T Q = 0 \implies Q^T dQ = -(Q^T dQ)^T.$$

这正是反对称的定义。 □

在继续之前，我们可能要问在 $\mathbb{R}^{n^2}$ 中正交矩阵的“表面”维度是多少。当 n =^2 时，所有正交矩阵都是旋转和反射，旋转的形式为

$$Q = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}.$$

因此，当 n =^2 时，我们有一个参数。

当 $n=3$ 时，飞机飞行员了解“滚转、俯仰和偏航”，这是当 $n=3$. 时正交矩阵的三个参数。一般来说，在 $\mathbb{R}^{n^2}$ 中，正交组的维度为 $n(n-1)/2$。

有几种方式可以看这个。

- 首先，正交性 $Q^TQ=I$ 强制 $n(n+1)/2$ 个约束，留下 $n(n-1)/2$ 个自由参数。
- 当我们进行 QR 分解时，R “吞噬”了 $n(n+1)/2$ 个参数，再次留下 $n(n-1)/2$ 个参数给 Q。
- 最后，如果我们考虑对称特征值问题，其中 $S=Q\Lambda Q^T$，S 有 $n(n+1)/2$ 个参数，Λ 有 n，那么 Q 有 $n(n-1)/2$。

13.2.1 对称特征分解的微分

设 S 为对称矩阵，Λ 为包含 S 特征值的对角矩阵，Q 为正交矩阵，其列向量为 S 的特征向量，使得 $S=Q\Lambda Q^T$。[为了简化，我们假设特征值是“简单的”（重数1）；重复的特征值由于特征向量基的模糊性，使得扰动分析变得非常复杂。] 然后，我们有

$$dS = dQ\,\Lambda Q^T + Q\,d\Lambda\,Q^T + Q\Lambda dQ^T,$$

这可以写成

$$Q^T dS\,Q = Q^T dQ\Lambda - \Lambda Q^T dQ + d\Lambda.$$

作为一个练习，可以检查上述等式的左右两边都是对称的。如果单独查看对角线项，这可能更容易，因为在这里 $(Q^T dS\,Q)_{ii} = q_i^T dS\,q_i$。由于 q_i 是 i 次特征向量，这表明 $q_i^T dS\,q_i = d\lambda_i$. (在物理学中，这有时被称为“Hellman–Feynman”定理，或非简并的一阶特征值扰动理论。)

有时我们考虑一个依赖于参数（如时间）的矩阵曲线 $S(t)$。如果我们要求 $\frac{d\lambda_i}{dt}$，这意味着它等于 $q_i^T\frac{dS(t)}{dt}q_i$.。那么我们如何得到一个特征值的梯度 $\nabla\lambda_i$呢？首先，请注意

$$\mathrm{tr}(q_iq_i^T)^T dS) = d\lambda_i \implies \nabla\lambda_i = q_iq_i^T.$$

关于特征向量？那些来自非对角元素，其中对于 $i\neq j$,

$$(Q^T dS\,Q)_{ij} = \left(Q^T\frac{dQ}{dt}\right)_{ij}(\lambda_j-\lambda_i).$$

因此，我们可以形成 $Q^T\frac{dQ}{dt}$ 的元素，并左乘以 Q 以获得 $\frac{dQ}{dt}$ (，因为 Q 与) 正交。

有趣的是，当在对称矩阵空间中沿一条线移动时，获取特征值的二阶导数。为了简化，假设 Λ 是对角线且 $S(t)=\Lambda+tE$. 因此，对以下进行微分

$$\frac{d\Lambda}{dt} = \mathrm{diag}\left(Q^T\frac{dS(t)}{dt}Q\right),$$

我们得到

$$\frac{d^2\Lambda}{dt^2} = \mathrm{diag}\left(Q^T\frac{d^2S(t)}{dt^2}Q\right) + 2\,\mathrm{diag}\left(Q^T\frac{dS(t)}{dt}\frac{dQ}{dt}\right).$$

评估此值在 $Q=I$，并认识到第一项为零，因为我们处于一条线上，所以我们有

$$\frac{d^2\Lambda}{dt^2} = 2\,\mathrm{diag}\left(E\cdot\frac{dQ}{dt}\right),$$

或者

$$\frac{d^2\Lambda}{dt^2} = 2\sum_{k\neq i} E_{ik}^2/(\lambda_i - \lambda_k).$$

使用这个，我们可以将特征值写成泰勒级数：

$$\lambda_i(\epsilon) = \lambda_i + \epsilon E_{ii} + \epsilon^2 \sum_{k\neq i} E_{ik}^2/(\lambda_i - \lambda_k) + \ldots.$$

(在物理学中这被称为二阶特征值扰动 理论.)

14 我们从这里走向何方

存在许多我们没有时间涵盖的主题，即使在16小时的讲座中也是如此。如果你进入这门课程时认为求导很简单，并且你已经在第一年微积分中学到了所有关于它的知识，希望我们已经说服你，这是一个极其丰富的主题，不可能在单一课程中穷尽。其中一些可能很不错的包括：

- 当自动微分（AD）遇到无法处理的情况时，您可能需要编写自定义雅可比-向量积（一个“Jvp”、“frule”或“pushforward”），以及/或反向模式中的自定义行向量-雅可比积（一个“vJp”、“rrule”、“pullback”或“Jacobian^T-向量积”）。在 Julia 的 Zygote AD 中，这通过 ChainRules 包来完成。在 Python 的 JAX 中，这分别通过 jax.custon_jvp 和/或 jax.custon_vjp 来完成。原则上，这很简单，但由于它们支持的通用性，API 可能需要一些时间来适应。

- 对于具有复数自变量 z (即复向量空间) 的函数 $f(z)$，当函数涉及共轭 z 时，例如，$|z|$, Re(z), 和 Im(z)，您不能随时取“普通”的复数导数。如果 $f(z)$ 是纯实值且非常数，例如涉及复数计算的优化问题，则必须发生这种情况。一个选项是将 $z = x + iy$ 写作并将 $f(z)$ 作为具有实数导数的双参数函数 $f(x, y)$ 处理，但如果您的问题“自然”地用复变量表示（例如，傅里叶频率域），则这可能很尴尬。一个常见的替代方案是“CR微积分”（或“Wirtinger微积分”），在其中您将

$$df = \left(\frac{\partial f}{\partial z}\right) dz + \left(\frac{\partial f}{\partial \overline{z}}\right) d\overline{z},$$

  好像 z 和 z是独立变量。这可以扩展到梯度、雅可比矩阵、最速下降法和牛顿迭代，例如。关于这个概念的一个很好的综述可以在K. Kreuz Delgado的这些UCSD课程笔记中找到。

- 许多关于矩阵函数和分解的衍生结果可以在文献中找到，其中一些推导相当棘手。例如，ChainRules包的一些参考文献已列在这个GitHub问题中。

- 另一个微分学的 重要推广是曲面上导数和微分几何，导致外导数。

- 当对矩阵 $A(x)$ 的特征值 λ 求导时，在特征值交叉点（多重性 $k >$ 为 1 的位置）会出现复杂性。在这里，特征值和特征向量通常不再可微。更普遍地，这个问题对于任何具有重复根的隐函数都会出现。在这种情况下，一个选项是使用一种称为广义梯度的扩展敏感性分析定义（一个 $k \times k$ 矩阵值线性算子 $G(x)[dx]$，其特征值是扰动 $d\lambda$）。例如，参见 Cox (1995)、Seyranian 等人 (1994) 和 Stechlinski (2022)。物理学家将相关思想称为“退化扰动理论”。类似想法的最近表述称为字典序方向导数。例如，参见 Nesterov (2005) 和 Barton 等人 (2017)。

  有时，涉及特征值的优化问题可以通过使用SDP约束来重新表述，以避免这种困难。例如，参见Men等人（2014）。

  对于一个有缺陷的矩阵，情况更糟：甚至广义导数也会爆炸，因为 $d\lambda$ 可以与（例如）扰动 $\|dA\|$ (的平方根成比例，对于具有代数重数 $= 2$ 和几何重数 $= 1$) 的特征值。

- Famous generalizations of differentation are the "distributional" and "weak" derivatives. For example, to obtain Dirac delta "functions" by differentiating discontinuities. This requires changing not only the definition of "derivative," but also changing the definition of *function*, as reviewed at an elementary level in these MIT course notes.