第 6 讲
非线性优化

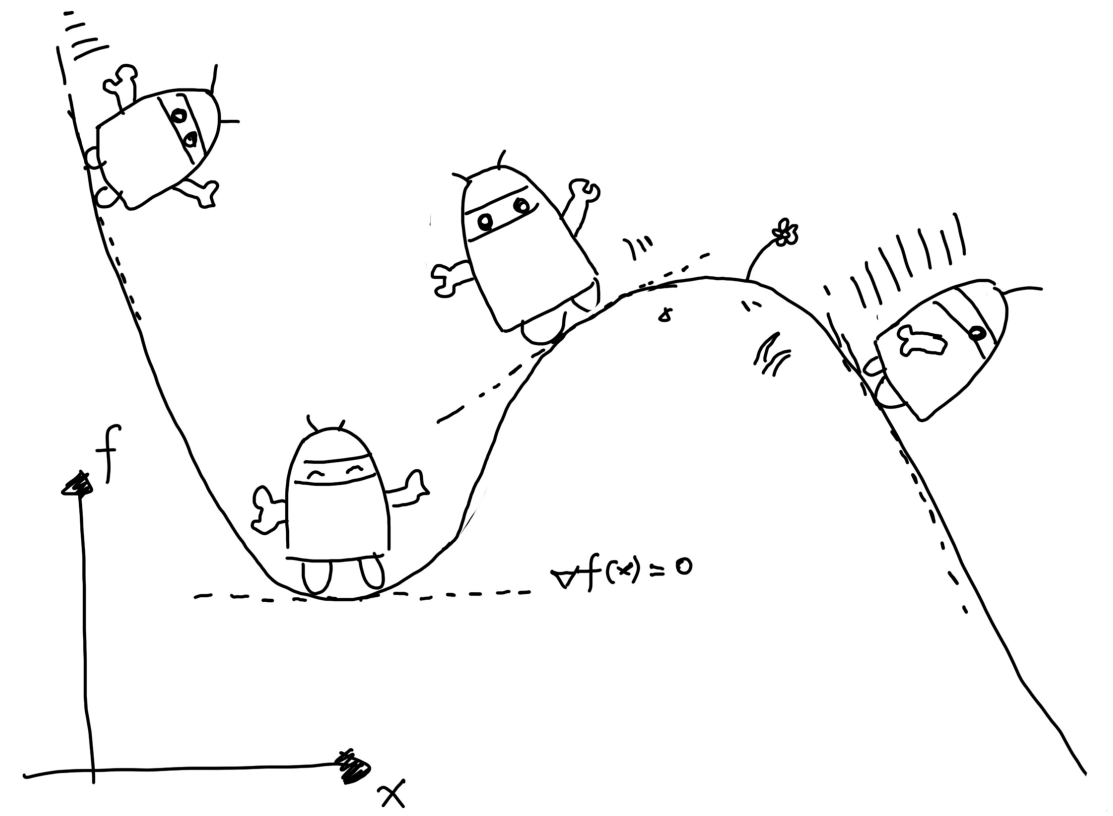
本节目标

1. 理解最小二乘法的含义和处理方式。
2. 理解 Gauss-Newton, Levenburg-Marquadt 等下降策略。
3. 学习 Ceres 库和 g2o 库的基本使用方法。

在前面几章，我们介绍了经典 SLAM 模型的运动方程和观测方程。现在我们已经知道，方程中的位姿可以由变换矩阵来描述，然后用李代数进行优化。观测方程由相机成像模型给出，其中内参是随相机固定的，而外参则是相机的位姿。于是，我们已经弄清了经典 SLAM 模型在视觉情况下的具体表达。

然而，由于噪声的存在，运动方程和观测方程的等式必定不是精确成立的。尽管相机可以非常好地符合针孔模型，但遗憾的是，我们得到的数据通常是受各种未知噪声影响的。即使我们有着高精度的相机，运动方程和观测方程也只能近似的成立。所以，与其假设数据必须符合方程，不如来讨论，如何在有噪声的数据中进行准确的状态估计。

大多现代视觉 SLAM 算法都不需要那么高成本的传感器，甚至也不需要那么昂贵的处理器来计算这些数据，这全是算法的功劳。由于在 SLAM 问题中，同一个点往往会被一个相机在不同的时间内多次观测，同一个相机在每个时刻观测到的点也不止一个。这些因素交织在一起，使我们拥有了更多的约束，最终能够较好地从噪声数据中恢复出我们需要的东西。本节就将介绍如何通过优化处理噪声数据，并且由这些表层逐渐深入到图优化本质，提供图优化的解决算法初步介绍并且提供训练实例。



$$f(x+\Delta x) \approx f(x) + \nabla f(x)\Delta x + \frac{1}{2}\Delta x^T H(x)\Delta x + \cdots$$

6.1 状态估计问题

6.1.1 最大后验与最大似然

接着前面几章的内容，我们回顾一下第二讲讨论的经典 SLAM 模型。它由一个状态方程和一个运动方程构成，如式（2.5）所示：

$$\begin{cases} \boldsymbol{x}_k = f\left(\boldsymbol{x}_{k-1}, \boldsymbol{u}_k\right) + \boldsymbol{w}_k \\ \boldsymbol{z}_{k,j} = h\left(\boldsymbol{y}_j, \boldsymbol{x}_k\right) + \boldsymbol{v}_{k,j} \end{cases}. \tag{6.1}$$

通过第四讲的知识，我们了解到这里的 $\boldsymbol{x}_k$ 乃是相机的位姿。我们可以使用变换矩阵或李代数表示它。至于观测方程，第五讲已经说明了它的内容，即针孔相机模型。为了让读者对它们有更深的印象，我们不妨讨论一下它们的具体参数化形式。首先，位姿变量 $\boldsymbol{x}_k$ 可以由 $\boldsymbol{T}_k$ 或 $\exp(\boldsymbol{\xi}_k^\wedge)$ 表达，二者是等价的。由于运动方程在视觉 SLAM 中没有特殊性，我们暂且不讨论它，主要讨论观测方程。假设在 $\boldsymbol{x}_k$ 处对路标 $\boldsymbol{y}_j$ 进行了一次观测，对应到图像上的像素位置 $\boldsymbol{z}_{k,j}$，那么，观测方程可以表示成：

$$s\boldsymbol{z}_{k,j} = \boldsymbol{K}\exp\left(\boldsymbol{\xi}^\wedge\right)\boldsymbol{y}_j. \tag{6.2}$$

根据上一讲的内容，读者应该知道这里 $\boldsymbol{K}$ 为相机内参，s 为像素点的距离。同时这里的 $\boldsymbol{z}_{k,j}$ 和 $\boldsymbol{y}_j$ 都必须以齐次坐标来描述，且中间有一次齐次到非齐次的转换。如果你还不熟悉这个过程，请回到上一讲再仔细看一看。

现在，考虑数据受噪声的影响后，会发生什么改变。在运动和观测方程中，我们**通常**假设两个噪声项 $\boldsymbol{w}_k, \boldsymbol{v}_{k,j}$ 满足零均值的高斯分布：

$$\boldsymbol{w}_k \sim N\left(0, \boldsymbol{R}_k\right), \boldsymbol{v}_k \sim N\left(0, \boldsymbol{Q}_{k,j}\right). \tag{6.3}$$

在这些噪声的影响下，我们希望通过带噪声的数据 $\boldsymbol{z}$ 和 $\boldsymbol{u}$，推断位姿 $\boldsymbol{x}$ 和地图 $\boldsymbol{y}$（以及它们的概率分布），这构成了一个状态估计问题。由于在 SLAM 过程中，这些数据是随着时间逐渐到来的，所以在历史上很长一段时间内，研究者们使用滤波器，尤其是扩展卡尔曼滤波器（EKF）求解它。卡尔曼滤波器关心当前时刻的状态估计 $\boldsymbol{x}_k$，而对之前的状态则不多考虑；相对的，近年来普遍使用的非线性优化方法，使用所有时刻采集到的数据进行状态估计，并被认为优于传统的滤波器 [13]，成为当前视觉 SLAM 的主流方法。因此，本书重点介绍以非线性优化为主的优化方法，对卡尔曼滤波器则留到第十讲再进行讨论。本讲将介绍非线性优化的基本知识，然后在第十、十一讲中对它们进行更深入的分析。

首先，我们从概率学角度来看一下我们正在讨论什么问题。在非线性优化中，我们把

所有待估计的变量放在一个“状态变量”中：

$$\boldsymbol{x} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_M\}.$$

现在，我们说，对机器人状态的估计，就是求已知输入数据 $\boldsymbol{u}$ 和观测数据 $\boldsymbol{z}$ 的条件下，计算状态 $\boldsymbol{x}$ 的条件概率分布：

$$P(\boldsymbol{x}|\boldsymbol{z}, \boldsymbol{u}). \tag{6.4}$$

类似于 $\boldsymbol{x}$，这里 $\boldsymbol{u}$ 和 $\boldsymbol{z}$ 也是对所有数据的统称。特别地，当我们没有测量运动的传感器，只有一张张的图像时，即只考虑观测方程带来的数据时，相当于估计 $P(\boldsymbol{x}|\boldsymbol{z})$ 的条件概率分布。如果忽略图像在时间上的联系，把它们看作一堆彼此没有关系的图片，该问题也称为 Structure from Motion（SfM），即如何从许多图像中重建三维空间结构 [22]。在这种情况下，SLAM 可以看作是图像具有时间先后顺序的，需要实时求解一个 SfM 问题。为了估计状态变量的条件分布，利用贝叶斯法则，有：

$$P(\boldsymbol{x}|\boldsymbol{z}) = \frac{P(\boldsymbol{z}|\boldsymbol{x})\,P(\boldsymbol{x})}{P(\boldsymbol{z})} \propto P(\boldsymbol{z}|\boldsymbol{x})\,P(\boldsymbol{x}). \tag{6.5}$$

贝叶斯法则左侧通常称为**后验概率**。它右侧的 $P(\boldsymbol{z}|\boldsymbol{x})$ 称为**似然**，另一部分 $P(\boldsymbol{x})$ 称为**先验**。**直接求后验分布是困难的，但是求一个状态最优估计，使得在该状态下，后验概率最大化（Maximize a Posterior，MAP），则是可行的：**

$$\boldsymbol{x}^*{}_{MAP} = \arg\max P(\boldsymbol{x}|\boldsymbol{z}) = \arg\max P(\boldsymbol{z}|\boldsymbol{x})P(\boldsymbol{x}). \tag{6.6}$$

请注意贝叶斯法则的分母部分与待估计的状态 $\boldsymbol{x}$ 无关，因而可以忽略。贝叶斯法则告诉我们，求解最大后验概率，**相当于最大化似然和先验的乘积**。进一步，我们当然也可以说，对不起，我不知道机器人位姿大概在什么地方，此时就没有了**先验**。那么，可以求解 $\boldsymbol{x}$ 的**最大似然估计（Maximize Likelihood Estimation, MLE）：**

$$\boldsymbol{x}^*{}_{MLE} = \arg\max P(\boldsymbol{z}|\boldsymbol{x}). \tag{6.7}$$

直观地说，似然是指“在现在的位姿下，可能产生怎样的观测数据”。由于我们知道观测数据，所以最大似然估计，可以理解成：**“在什么样的状态下，最可能产生现在观测到的数据”**。这就是最大似然估计的直观意义。

6.1.2 最小二乘的引出

那么如何求最大似然估计呢？我们说，在高斯分布的假设下，最大似然能够有较简单的形式。回顾观测模型，对于某一次观测：

$$\boldsymbol{z}_{k,j} = h\left(\boldsymbol{y}_j, \boldsymbol{x}_k\right) + \boldsymbol{v}_{k,j},$$

由于我们假设了噪声项 $\boldsymbol{v}_k \sim N\left(0, \boldsymbol{Q}_{k,j}\right)$，所以观测数据的条件概率为：

$$P(\boldsymbol{z}_{j,k}|\boldsymbol{x}_k, \boldsymbol{y}_j) = N\left(h(\boldsymbol{y}_j, \boldsymbol{x}_k), \boldsymbol{Q}_{k,j}\right).$$

它依然是一个高斯分布。为了计算使它最大化的 $\boldsymbol{x_k}, \boldsymbol{y_j}$，我们往往使用**最小化负对数**的方式，来求一个高斯分布的最大似然。

高斯分布在负对数下有较好的数学形式。考虑一个任意的高维高斯分布 $\boldsymbol{x} \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$，它的概率密度函数展开形式为:

$$P\left(\boldsymbol{x}\right) = \frac{1}{\sqrt{(2\pi)^N \det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}\left(\boldsymbol{x}-\boldsymbol{\mu}\right)\right). \tag{6.8}$$

取它的负对数，则变为：

$$-\ln\left(P\left(\boldsymbol{x}\right)\right) = \frac{1}{2}\ln\left((2\pi)^N \det\left(\boldsymbol{\Sigma}\right)\right) + \frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}\left(\boldsymbol{x}-\boldsymbol{\mu}\right). \tag{6.9}$$

对原分布求最大化相当于对负对数求最小化。在最小化上式的 $\boldsymbol{x}$ 时，第一项与 $\boldsymbol{x}$ 无关，可以略去。于是，只要最小化右侧的二次型项，就得到了对状态的最大似然估计。代入 SLAM 的观测模型，相当于我们在求：

$$\boldsymbol{x}^* = \arg\min\left(\left(\boldsymbol{z}_{k,j} - h\left(\boldsymbol{x}_k, \boldsymbol{y}_j\right)\right)^T \boldsymbol{Q}_{k,j}^{-1}\left(\boldsymbol{z}_{k,j} - h\left(\boldsymbol{x}_k, \boldsymbol{y}_j\right)\right)\right). \tag{6.10}$$

我们发现，该式等价于最小化噪声项（即误差）的平方（$\boldsymbol{\Sigma}$ 范数意义下）。因此，对于所有的运动和任意的观测，我们定义数据与估计值之间的误差：

$$\begin{aligned} \boldsymbol{e}_{v,k} &= \boldsymbol{x}_k - f\left(\boldsymbol{x}_{k-1}, \boldsymbol{u}_k\right) \\ \boldsymbol{e}_{y,j,k} &= \boldsymbol{z}_{k,j} - h\left(\boldsymbol{x}_k, \boldsymbol{y}_j\right), \end{aligned} \tag{6.11}$$

并求该误差的平方之和：

$$J(\boldsymbol{x}) = \sum_k \boldsymbol{e}_{v,k}^T \boldsymbol{R}_k^{-1} \boldsymbol{e}_{v,k} + \sum_k \sum_j \boldsymbol{e}_{y,k,j}^T \boldsymbol{Q}_{k,j}^{-1} \boldsymbol{e}_{y,k,j}. \tag{6.12}$$

这就得到了一个总体意义下的最小二乘问题（Least Square Problem）。我们明白它的最优解等价于状态的最大似然估计。直观来讲，由于噪声的存在，当我们把估计的轨迹与地图代入 SLAM 的运动、观测方程中时，它们并不会完美的成立。这时候怎么办呢？我们把状态的估计值进行**微调**，使得整体的误差下降一些。当然这个下降也有限度，它一般会到达一个**极小值**。这就是一个典型非线性优化的过程。

仔细观察式（6.12），我们发现 SLAM 中的最小二乘问题具有一些特定的结构：

- 首先，整个问题的目标函数由许多个误差的（加权的）平方和组成。虽然总体的状态变量维数很高，但每个误差项都是简单的，仅与一两个状态变量有关。例如运动误差只与 $\boldsymbol{x}_{k-1}, \boldsymbol{x}_k$ 有关，观测误差只与 $\boldsymbol{x}_k, \boldsymbol{y}_j$ 有关。每个误差项是一个小规模的约束，我们之后会谈论如何对它们进行线性近似，最后再把这个误差项的小雅可比矩阵块放到整体的雅可比矩阵中。由于这种做法，我们称每个误差项对应的优化变量为**参数块（Parameter Block）**。

- 整体误差由很多小型误差项之和组成的问题，其增量方程的求解会具有一定的稀疏性（会在第十讲详细讲解），使得它们在大规模时亦可求解。

- 其次，如果使用李代数表示，则该问题是**无约束**的最小二乘问题。但如果用旋转矩阵（变换矩阵）描述位姿，则会引入旋转矩阵自身的约束（旋转矩阵必须是正交阵且行列式为 1）。额外的约束会使优化变得更困难。这体现了李代数的优势。

- 最后，我们使用了平方形式（二范数）度量误差，它是直观的，相当于欧氏空间中距离的平方。但它也存在着一些问题，并且不是唯一的度量方式。我们亦可使用其他的范数构建优化问题。

现在，我们要介绍如何求解这个最小二乘问题。本章将介绍**非线性优化的基本知识**，特别地，针对这样一个通用的无约束非线性最小二乘问题，探讨它是如何求解的。在后续几章，我们会大量使用本章的结果，详细讨论它在 SLAM 前端、后端中的应用。

6.2 非线性最小二乘

我们先来考虑一个简单的最小二乘问题：

$$\min_x \frac{1}{2} \| f(\boldsymbol{x}) \|_2^2. \tag{6.13}$$

这里自变量 $\boldsymbol{x} \in \mathbb{R}^n$，$f$ 是任意一个非线性函数，我们设它有 m 维：$f(\boldsymbol{x}) \in \mathbb{R}^m$。下面讨论如何求解这样一个优化问题。

如果 f 是个数学形式上很简单的函数，那问题也许可以用解析形式来求。令目标函数的导数为零，然后求解 $\boldsymbol{x}$ 的最优值，就和一个求二元函数的极值一样：

$$\frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{x}} = \boldsymbol{0}. \tag{6.14}$$

解此方程，就得到了导数为零处的极值。它们可能是极大、极小或鞍点处的值，只要挨个儿比较它们的函数值大小即可。但是，这个方程是否容易求解呢？这取决于 f 导函数的形式。在 SLAM 中，我们使用李代数来表示机器人的旋转和位移。尽管我们在李代数章节讨论了它的导数形式，但这不代表我们就能够顺利求解上式这样一个复杂的非线性方程。

对于不方便直接求解的最小二乘问题，我们可以用**迭代**的方式，从一个初始值出发，不断地更新当前的优化变量，使目标函数下降。具体步骤可列写如下：

1. 给定某个初始值 $\boldsymbol{x}_0$。
2. 对于第 k 次迭代，寻找一个增量 $\Delta \boldsymbol{x}_k$，使得 $\|f(\boldsymbol{x}_k + \Delta \boldsymbol{x}_k)\|_2^2$ 达到极小值。
3. 若 $\Delta \boldsymbol{x}_k$ 足够小，则停止。
4. 否则，令 $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta \boldsymbol{x}_k$，返回 2.

这让求解导函数为零的问题，变成了一个不断寻找梯度并下降的过程。直到某个时刻增量非常小，无法再使函数下降。此时算法收敛，目标达到了一个极小，我们完成了寻找极小值的过程。在这个过程中，我们只要找到迭代点的梯度方向即可，而无需寻找全局导函数为零的情况。

接下来的问题是，增量 $\Delta \boldsymbol{x}_k$ 如何确定?——实际上，研究者们已经花费了大量精力探索增量的求解方式。我们将介绍两类办法，它们用不同的手段来寻找这个增量。目前这两种方法在视觉 SLAM 的优化问题上也被广泛采用，大多数优化库都可以使用它们。

6.2.1 一阶和二阶梯度法

求解增量最直观的方式是将目标函数在 $\boldsymbol{x}$ 附近进行泰勒展开：

$$\|f(\boldsymbol{x} + \Delta \boldsymbol{x})\|_2^2 \approx \|f(\boldsymbol{x})\|_2^2 + \boldsymbol{J}(\boldsymbol{x})\Delta \boldsymbol{x} + \frac{1}{2}\Delta \boldsymbol{x}^T \boldsymbol{H} \Delta \boldsymbol{x}. \tag{6.15}$$

这里 $\boldsymbol{J}$ 是 $\|f(\boldsymbol{x})\|^2$ 关于 $\boldsymbol{x}$ 的导数（雅可比矩阵），而 $\boldsymbol{H}$ 则是二阶导数（海塞（Hessian）矩阵）。我们可以选择保留泰勒展开的一阶或二阶项，对应的求解方法则为一阶梯度或二阶梯度法。如果保留一阶梯度，那么增量的方向为：

$$\Delta \boldsymbol{x}^* = -\boldsymbol{J}^T(\boldsymbol{x}). \tag{6.16}$$

它的直观意义非常简单，只要我们沿着反向梯度方向前进即可。当然，我们还需要该方向上取一个步长 λ，求得最快的下降方式。这种方法被称为**最速下降法**。

另一方面，如果保留二阶梯度信息，那么增量方程为：

$$\Delta \boldsymbol{x}^* = \arg\min \|f(\boldsymbol{x})\|_2^2 + \boldsymbol{J}(\boldsymbol{x})\Delta \boldsymbol{x} + \frac{1}{2}\Delta \boldsymbol{x}^T \boldsymbol{H} \Delta \boldsymbol{x}. \tag{6.17}$$

求右侧等式关于 $\Delta \boldsymbol{x}$ 的导数并令它为零，就得到了增量的解：

$$\boldsymbol{H}\Delta \boldsymbol{x} = -\boldsymbol{J}^T. \tag{6.18}$$

该方法称又为牛顿法。我们看到，一阶和二阶梯度法都十分直观，只要把函数在迭代点附近进行泰勒展开，并针对更新量作最小化即可。由于泰勒展开之后函数变成了多项式，所以求解增量时只需解线性方程即可，避免了直接求导函数为零这样的非线性方程的困难。不过，这两种方法也存在它们自身的问题。最速下降法过于贪心，容易走出锯齿路线，反而增加了迭代次数。而牛顿法则需要计算目标函数的 $\boldsymbol{H}$ 矩阵，这在问题规模较大时非常困难，我们通常倾向于避免 $\boldsymbol{H}$ 的计算。所以，接下来我们详细地介绍两类更加实用的方法：高斯牛顿法和列文伯格——马夸尔特方法。

6.2.2 Gauss-Newton

Gauss Newton 是最优化算法里面最简单的方法之一。它的思想是将 $f(\boldsymbol{x})$ 进行一阶的泰勒展开（请注意不是目标函数 $f(\boldsymbol{x})^2$）：

$$f(\boldsymbol{x} + \Delta \boldsymbol{x}) \approx f(\boldsymbol{x}) + \boldsymbol{J}(\boldsymbol{x})\Delta \boldsymbol{x}. \tag{6.19}$$

这里 $\boldsymbol{J}(\boldsymbol{x})$ 为 $f(\boldsymbol{x})$ 关于 $\boldsymbol{x}$ 的导数，实际上是一个 $m \times n$ 的矩阵，也是一个雅可比矩阵。根据前面的框架，当前的目标是为了寻找下降矢量 $\Delta \boldsymbol{x}$，使得 $\|f(\boldsymbol{x} + \Delta \boldsymbol{x})\|^2$ 达到最小。为了求 $\Delta \boldsymbol{x}$，我们需要解一个线性的最小二乘问题：

$$\Delta \boldsymbol{x}^* = \arg\min_{\Delta \boldsymbol{x}} \frac{1}{2}\|f(\boldsymbol{x}) + \boldsymbol{J}(\boldsymbol{x})\Delta \boldsymbol{x}\|^2. \tag{6.20}$$

这个方程与之前有什么不一样呢？根据极值条件，将上述目标函数对 $\Delta \boldsymbol{x}$ 求导，并令导数为零。由于这里考虑的是 $\Delta \boldsymbol{x}$ 的导数（而不是 $\boldsymbol{x}$），我们最后将得到一个线性的方程。为此，先展开目标函数的平方项：

$$\begin{aligned}\frac{1}{2}\|f(\boldsymbol{x})+\boldsymbol{J}(\boldsymbol{x})\Delta \boldsymbol{x}\|^2 &= \frac{1}{2}(f(\boldsymbol{x})+\boldsymbol{J}(\boldsymbol{x})\Delta \boldsymbol{x})^T(f(\boldsymbol{x})+\boldsymbol{J}(\boldsymbol{x})\Delta \boldsymbol{x}) \\ &= \frac{1}{2}\left(\|f(\boldsymbol{x})\|_2^2+2f(\boldsymbol{x})^T\boldsymbol{J}(\boldsymbol{x})\Delta \boldsymbol{x}+\Delta \boldsymbol{x}^T\boldsymbol{J}(\boldsymbol{x})^T\boldsymbol{J}(\boldsymbol{x})\Delta \boldsymbol{x}\right).\end{aligned}$$

求上式关于 $\Delta \boldsymbol{x}$ 的导数，并令其为零：

$$2\boldsymbol{J}(\boldsymbol{x})^T f(\boldsymbol{x})+2\boldsymbol{J}(\boldsymbol{x})^T\boldsymbol{J}(\boldsymbol{x})\Delta \boldsymbol{x}=\boldsymbol{0}.$$

可以得到如下方程组：

$$\boldsymbol{J}(\boldsymbol{x})^T\boldsymbol{J}(\boldsymbol{x})\Delta \boldsymbol{x}=-\boldsymbol{J}(\boldsymbol{x})^T f(\boldsymbol{x}). \tag{6.21}$$

注意，我们要求解的变量是 $\Delta \boldsymbol{x}$，因此这是一个**线性方程组**，我们称它为**增量方程**，也可以称为**高斯牛顿方程** (Gauss Newton equations) 或者**正规方程** (Normal equations)。我们把左边的系数定义为 $\boldsymbol{H}$，右边定义为 $\boldsymbol{g}$，那么上式变为：

$$\boldsymbol{H}\Delta \boldsymbol{x}=\boldsymbol{g}. \tag{6.22}$$

这里把左侧记作 $\boldsymbol{H}$ 是有意义的。对比牛顿法可见，Gauss-Newton 用 $\boldsymbol{J}^T\boldsymbol{J}$ 作为牛顿法中二阶 Hessian 矩阵的近似，从而省略了计算 $\boldsymbol{H}$ 的过程。**求解增量方程是整个优化问题的核心所在**。如果我们能够顺利解出该方程，那么 Gauss-Newton 的算法步骤可以写成：

1. 给定初始值 $\boldsymbol{x}_0$。
2. 对于第 k 次迭代，求出当前的雅可比矩阵 $\boldsymbol{J}(\boldsymbol{x}_k)$ 和误差 $f(\boldsymbol{x}_k)$。
3. 求解增量方程：$\boldsymbol{H}\Delta \boldsymbol{x}_k=\boldsymbol{g}$.
4. 若 $\Delta \boldsymbol{x}_k$ 足够小，则停止。否则，令 $\boldsymbol{x}_{k+1}=\boldsymbol{x}_k+\Delta \boldsymbol{x}_k$，返回 2.

从算法步骤中可以看到，增量方程的求解占据着主要地位。原则上，它要求我们所用的近似 $\boldsymbol{H}$ 矩阵是可逆的（而且是正定的），但实际数据中计算得到的 $\boldsymbol{J}^T\boldsymbol{J}$ 却只有半正定

性。也就是说，在使用 Gauss Newton 方法时，可能出现 $\boldsymbol{J}^T\boldsymbol{J}$ 为奇异矩阵或者病态 (ill-condition) 的情况，此时增量的稳定性较差，导致算法不收敛。更严重的是，就算我们假设 $\boldsymbol{H}$ 非奇异也非病态，如果我们求出来的步长 $\Delta\boldsymbol{x}$ 太大，也会导致我们采用的局部近似 (6.19) 不够准确，这样一来我们甚至都无法保证它的迭代收敛，哪怕是让目标函数变得更大都是有可能的。

尽管 Gauss Newton 法有这些缺点，但是它依然值得我们去学习，因为在非线性优化里，相当多的算法都可以归结为 Gauss Newton 法的变种。这些算法都借助了 Gauss Newton 法的思想并且通过自己的改进修正 Gauss Newton 法的缺点。例如一些**线搜索方法** (line search method)，这类改进就是加入了一个标量 α，在确定了 $\Delta\boldsymbol{x}$ 进一步找到 α 使得 $\|f(\boldsymbol{x}+\alpha\Delta\boldsymbol{x})\|^2$ 达到最小，而不是像 Gauss Newton 法那样简单地令 $\alpha=1$。

Levenberg-Marquadt 方法在一定程度上修正了这些问题，一般认为它比 Gauss Newton 更为鲁棒。尽管它的收敛速度可能会比 Gauss Newton 更慢，被称之为**阻尼牛顿法** (Damped Newton Method)，但是在 SLAM 里面却被大量应用。

6.2.3 Levenberg-Marquadt

由于 Gauss-Newton 方法中采用的近似二阶泰勒展开只能在展开点附近有较好的近似效果，所以我们很自然地想到应该给 $\Delta\boldsymbol{x}$ 添加一个信赖区域（Trust Region），不能让它太大而使得近似不准确。非线性优化种有一系列这类方法，这类方法也被称之为**信赖区域方法** (Trust Region Method)。在信赖区域里边，我们认为近似是有效的；出了这个区域，近似可能会出问题。

那么如何确定这个信赖区域的范围呢？一个比较好的方法是根据我们的近似模型跟实际函数之间的差异来确定这个范围：如果差异小，我们就让范围尽可能大；如果差异大，我们就缩小这个近似范围。因此，考虑使用

$$\rho=\frac{f\left(\boldsymbol{x}+\Delta\boldsymbol{x}\right)-f\left(\boldsymbol{x}\right)}{\boldsymbol{J}\left(\boldsymbol{x}\right)\Delta\boldsymbol{x}}. \tag{6.23}$$

来判断泰勒近似是否够好。ρ 的分子是实际函数下降的值，分母是近似模型下降的值。如果 ρ 接近于 1，则近似是好的。如果 ρ 太小，说明实际减小的值远少于近似减小的值，则认为近似比较差，需要缩小近似范围。反之，如果 ρ 比较大，则说明实际下降的比预计的更大，我们可以放大近似范围。

于是，我们构建一个改良版的非线性优化框架，该框架会比 Gauss Newton 有更好的效果：

1. 给定初始值 $\boldsymbol{x}_0$，以及初始优化半径 μ。
2. 对于第 k 次迭代，求解：

$$\min_{\Delta \boldsymbol{x}_k} \frac{1}{2} \| f(\boldsymbol{x}_k) + \boldsymbol{J}(\boldsymbol{x}_k) \Delta \boldsymbol{x}_k \|^2, \quad s.t. \| \boldsymbol{D} \Delta \boldsymbol{x}_k \|^2 \leq \mu, \tag{6.24}$$

   这里 μ 是信赖区域的半径，$\boldsymbol{D}$ 将在后文说明。
3. 计算 ρ。
4. 若 $\rho > \frac{3}{4}$，则 $\mu = 2\mu$；
5. 若 $\rho < \frac{1}{4}$，则 $\mu = 0.5\mu$；
6. 如果 ρ 大于某阈值，认为近似可行。令 $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta \boldsymbol{x}_k$。
7. 判断算法是否收敛。如不收敛则返回 2，否则结束。

这里近似范围扩大的倍数和阈值都是经验值，可以替换成别的数值。在式（6.24）中，我们把增量限定于一个半径为 μ 的球中，认为只在这个球内才是有效的。带上 $\boldsymbol{D}$ 之后，这个球可以看成一个椭球。在 Levenberg 提出的优化方法中，把 $\boldsymbol{D}$ 取成单位阵 $\boldsymbol{I}$，相当于直接把 $\Delta \boldsymbol{x}$ 约束在一个球中。随后，Marqaurdt 提出将 $\boldsymbol{D}$ 取成非负数对角阵——实际中通常用 $\boldsymbol{J}^T \boldsymbol{J}$ 的对角元素平方根，使得在梯度小的维度上约束范围更大一些。

不论如何，在 L-M 优化中，我们都需要解式（6.24）那样一个子问题来获得梯度。这个子问题是带不等式约束的优化问题，我们用 Lagrange 乘子将它转化为一个无约束优化问题：

$$\min_{\Delta \boldsymbol{x}_k} \frac{1}{2} \| f(\boldsymbol{x}_k) + \boldsymbol{J}(\boldsymbol{x}_k) \Delta \boldsymbol{x}_k \|^2 + \frac{\lambda}{2} \| \boldsymbol{D} \Delta \boldsymbol{x} \|^2. \tag{6.25}$$

这里 λ 为 Lagrange 乘子。类似于 Gauss-Newton 中的做法，把它展开后，我们发现该问题的核心仍是计算增量的线性方程：

$$\left(\boldsymbol{H} + \lambda \boldsymbol{D}^T \boldsymbol{D} \right) \Delta \boldsymbol{x} = \boldsymbol{g}. \tag{6.26}$$

可以看到，增量方程相比于 Gauss-Newton，多了一项 $\lambda \boldsymbol{D}^T \boldsymbol{D}$。如果考虑它的简化形式，即 $\boldsymbol{D} = \boldsymbol{I}$，那么相当于求解：

$$(\boldsymbol{H} + \lambda \boldsymbol{I})\, \Delta \boldsymbol{x} = \boldsymbol{g}.$$

我们看到，当参数 λ 比较小时，$\boldsymbol{H}$ 占主要地位，这说明二次近似模型在该范围内是比较好的，L-M 方法更接近于 G-N 法。另一方面，当 λ 比较大时，$\lambda \boldsymbol{I}$ 占据主要地位，L-M 更接近于一阶梯度下降法（即最速下降），这说明附近的二次近似不够好。L-M 的求解方式，可在一定程度上避免线性方程组的系数矩阵的非奇异和病态问题，提供更稳定更准确的增量 $\Delta \boldsymbol{x}$。

在实际中，还存在许多其它的方式来求解函数的增量，例如 Dog-Leg 等方法。我们在这里所介绍的，只是最常见而且最基本的方式，也是视觉 SLAM 中用的最多的方式。总而言之，非线性优化问题的框架，分为 Line Search 和 Trust Region 两类。Line Search 先固定搜索方向，然后在该方向寻找步长，以最速下降法和 Gauss-Newton 法为代表。而 Trust Region 则先固定搜索区域，再考虑找该区域内的最优点。此类方法以 L-M 为代表。实际问题中，我们通常选择 G-N 或 L-M 之一作为梯度下降策略。

6.2.4 小结

由于我不希望这本书变成一本让人觉得头疼的数学书，所以这里只罗列了最常见的两种非线性优化方案，Gauss Newton 和 Levernberg-Marquardt。我们避开了许多数学性质上的讨论。如果读者对优化感兴趣，可以进一步阅读专门介绍数值优化的书籍（这是一个很大的课题），例如 [23]。以 G-N 和 L-M 为代表的优化方法，在很多开源的优化库都已经实现并提供给用户，我们会在下文进行实验。最优化是处理许多实际问题的基本数学工具，不光在视觉 SLAM 起着核心作用，在类似于深度学习等其它领域，它也是求解问题的核心方法之一。我们希望读者能够根据自身能力，去了解更多的最优化算法。

也许你发现了，无论是 G-N 还是 L-M，在做最优化计算的时候，都需要提供变量的初始值。你也许会问到，这个初始值能否随意设置？当然不是。实际上非线性优化的所有迭代求解方案，都需要用户来提供一个良好的初始值。由于目标函数太复杂，导致在求解空间上的变化难以琢磨，对问题提供不同的初始值往往会导致不同的计算结果。这种情况是非线性优化的通病：大多数算法都容易陷入局部极小值。因此，无论是哪类科学问题，我们提供初始值都应该有科学依据，例如视觉 SLAM 问题中，我们会用 ICP，PnP 之类的算法提供优化初始值。总之，一个良好的初始值对最优化问题非常重要！

也许读者还会对上面提到的最优化产生疑问：如何求解线性增量方程组呢？我们只讲到了增量方程是一个线性方程，但是直接对系数矩阵进行求逆岂不是要进行大量的计算？当然不是。在视觉 SLAM 算法里，经常遇到 $\Delta \boldsymbol{x}$ 的维度大到好几百或者上千，如果你是要做大规模的视觉三维重建，就会经常发现这个维度可以轻易达到几十万甚至更高的级别。

要对那么大个矩阵进行求逆是大多数处理器无法负担的，因此存在着许多针对线性方程组的数值求解方法。在不同的领域有不同的求解方式，但几乎没有一种方式是直接求系数矩阵的逆，我们会采用矩阵分解的方法来解线性方程，例如 QR、Cholesky 等分解方法。这些方法通常在矩阵论等教科书中可以找到，我们不多加介绍。

幸运的是，视觉 SLAM 里，这个矩阵往往有特定的稀疏形式，这为实时求解优化问题提供了可能性。我们在第十章中详细介绍它的原理。利用稀疏形式的消元，分解，最后再进行求解增量，会让求解的效率大大提高。在很多开源的优化库上，维度为一万多的变量在一般的 PC 上就可以在几秒甚至更短的时间内就被求解出来，其原因也是因为用了更加高级的数学工具。视觉 SLAM 算法现在能够实时地实现，也是多亏了这系数矩阵是稀疏的，如果是矩阵是稠密的，恐怕优化这类视觉 SLAM 算法就不会被学界广泛采纳了 [24, 25, 26]。

6.3 实践：Ceres

我们前面说了很多理论，现在来实践一下前面提到的优化算法。在本章的实践部分中，我们主要向大家介绍两个 C++ 的优化库：来自谷歌的 Ceres 库 [27] 以及基于图优化的 g2o 库 [28]。由于 g2o 的使用还需要讲一点图优化的相关知识，所以我们先来介绍 Ceres，然后介绍一些图优化理论，最后来讲 g2o。由于优化算法在之后的视觉里程计和后端中都会出现，所以请读者务必掌握优化算法的意义，理解程序的内容。

6.3.1 Ceres 简介

Ceres 库面向通用的最小二乘问题的求解，作为用户，我们需要做的就是定义优化问题，然后设置一些选项，输入进 Ceres 求解即可。Ceres 求解的最小二乘问题最一般的形式如下（带边界的核函数最小二乘）：

$$\begin{aligned} \min_{x} \quad & \frac{1}{2}\sum_{i}\rho_i\left(\left\|f_i\left(x_{i_1},\dots x_{i_n}\right)\right\|^2\right) \\ s.t. \quad & l_j \le x_j \le u_j. \end{aligned} \tag{6.27}$$

可以看到，目标函数由许多平方项，经过一个**核函数** $\rho(\cdot)$ 之后，求和组成[①]。在最简单的情况下，取 ρ 为恒等函数，则目标函数即为许多项的平方和。在这个问题中，优化变量为 $x_1,\dots,x_n$，f_i 称为**代价函数**（Cost function），在 SLAM 中亦可理解为误差项。l_j 和 u_j 为第 j 个优化变量的上限和下限。在最简单的情况下，取 $l_j=-\infty, u_j=\infty$（不限制优化变量的边界），并且取 ρ 为恒等函数时，就得到了无约束的最小二乘问题，和我们先前说的是一致的。

在 Ceres 中，我们将定义优化变量 $\boldsymbol{x}$ 和每个代价函数 f_i，再调用 Ceres 进行求解。我

[①] 核函数的详细讨论见第十讲。

们可以选择使用 G-N 或者 L-M 进行梯度下降，并设定梯度下降的条件，Ceres 会在优化之后，将最优估计值返回给我们。下面，我们通过一个曲线拟合的实验，来实际操作一下 Ceres，理解优化的过程。

6.3.2 安装 Ceres

为了使用 Ceres，首先要做的就是编译安装它啦！由于某些原因，目前国内下载谷歌资源并不方便，因此我们建议去 github 上下载 Ceres：`https://github.com/ceres-solver/ceres-solver`。本书的 3rdparty 下也附带了 Ceres 库。

与之前碰到的库一样，Ceres 是一个 cmake 工程。先来安装它的依赖项，在 Ubuntu 中都可以用 apt-get 安装，主要是谷歌自己使用的一些日志和测试工具：

```
sudo apt-get install liblapack-dev libsuitesparse-dev libcxsparse3.1.2 libgflags-dev libgoogle-glog-dev
 libgtest-dev
```

然后，进入 Ceres 库，使用 cmake 编译并安装它。这个过程我们已经做过很多遍了，此处就不再赘述。安装完成后，在/usr/local/include/ceres 下找到 Ceres 的头文件，并在/usr/local/lib/下找到名为 libceres.a 的库文件。有了头文件和库文件，就可以使用 Ceres 进行优化计算了。

6.3.3 使用 Ceres 拟合曲线

我们的演示实验包括使用 Ceres 和接下来的 g2o 进行曲线拟合。假设有一条满足以下方程的曲线：

$$y = \exp(ax^2 + bx + c) + w,$$

其中 a, b, c 为曲线的参数，w 为高斯噪声。我们故意选择了这样一个非线性模型，以使问题不至于太简单。现在，假设我们有 N 个关于 x, y 的观测数据点，想根据这些数据点求出曲线的参数。那么，可以求解下面的最小二乘问题以估计曲线参数：

$$\min_{a,b,c} \frac{1}{2} \sum_{i=1}^{N} \left\| y_i - \exp\left(ax_i^2 + bx_i + c\right) \right\|^2. \tag{6.28}$$

请注意，在这个问题中，待估计的变量是 a, b, c，而不是 x。我们写一个程序，先根据模型生成 x, y 的真值，然后在真值中添加高斯分布的噪声。随后，使用 Ceres 从带噪声的数据中拟合参数模型。

slambook/ch6/ceres_curve_fitting/main.cpp

```cpp
#include <iostream>
#include <opencv2/core/core.hpp>
#include <ceres/ceres.h>
#include <chrono>

using namespace std;

// 代价函数的计算模型
struct CURVE_FITTING_COST
{
    CURVE_FITTING_COST ( double x, double y ) : _x ( x ), _y ( y ) {}
    // 残差的计算
    template <typename T>
    bool operator() (
        const T* const abc, // 模型参数，有 3 维
        T* residual ) const // 残差
    {
        // y-exp(ax^2+bx+c)
        residual[0] = T ( _y ) - ceres::exp ( abc[0]*T ( _x ) *T ( _x ) + abc[1]*T ( _x ) + abc[2] );
        return true;
    }
    const double _x, _y; // x,y 数据
};

int main ( int argc, char** argv )
{
    double a=1.0, b=2.0, c=1.0; //   真实参数值
    int N=100; //                    数据点
    double w_sigma=1.0; //           噪声 Sigma 值
    cv::RNG rng; //                  OpenCV 随机数产生器
    double abc[3] = {0,0,0}; //      abc 参数的估计值

    vector<double> x_data, y_data; // 数据

    cout<<"generating data: "<<endl;
    for ( int i=0; i<N; i++ )
    {
        double x = i/100.0;
        x_data.push_back ( x );
        y_data.push_back (
            exp ( a*x*x + b*x + c ) + rng.gaussian ( w_sigma )
        );
        cout<<x_data[i]<<" "<<y_data[i]<<endl;
    }

    // 构建最小二乘问题
    ceres::Problem problem;
    for ( int i=0; i<N; i++ )
    {
        problem.AddResidualBlock ( // 向问题中添加误差项
```

```
51          // 使用自动求导，模板参数：误差类型，输出维度，输入维度，数值参照前面 struct 中写法
52          new ceres::AutoDiffCostFunction<CURVE_FITTING_COST, 1, 3> (
53              new CURVE_FITTING_COST ( x_data[i], y_data[i] )
54          ),
55          nullptr, //         核函数，这里不使用，为空
56          abc //              待估计参数
57      );
58  }
59
60  // 配置求解器
61  ceres::Solver::Options options; // 这里有很多配置项可以填
62  options.linear_solver_type = ceres::DENSE_QR; // 增量方程如何求解
63  options.minimizer_progress_to_stdout = true; // 输出到cout
64
65  ceres::Solver::Summary summary; //         优化信息
66  chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
67  ceres::Solve ( options, &problem, &summary ); // 开始优化
68  chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
69  chrono::duration<double> time_used = chrono::duration_cast<chrono::duration<double>>( t2-t1 );
70  cout<<"solve time cost = "<<time_used.count()<<" seconds. "<<endl;
71
72  // 输出结果
73  cout<<summary.BriefReport() <<endl;
74  cout<<"estimated a,b,c = ";
75  for ( auto a:abc ) cout<<a<<" ";
76  cout<<endl;
77
78  return 0;
79  }
```

程序需要说明的地方均已加注释。可以看到，我们利用 OpenCV 的噪声生成器，生成了 100 个带高斯噪声的数据。随后利用 Ceres 进行拟合。Ceres 的用法是这样的：

1. 定义 Cost Function 模型。方法是书写一个类，并在类中定义带模板参数的 () 运算符，这样该类成为了一个拟函数（Functor，C++ 术语）。这种定义方式使得 Ceres 可以像调用函数一样，对该类的某个对象（比如说 a）调用 a<double>() 方法——这使对象具有像函数那样的行为。

2. 调用 AddResidualBlock 将误差项添加到目标函数中。由于优化需要梯度，我们有若干种选择：（1）使用 Ceres 的自动求导（Auto Diff）；（2）使用数值求导（Numeric Diff）；（3）自行推导解析的导数形式，提供给 Ceres。其中自动求导在编码上是最方便的，于是我们就使用自动求导啦！

3. 自动求导需要指定误差项和优化变量的维度。这里的误差则是标量，维度为 1；优化的是 a, b, c 三个量，维度为 3。于是，在自动求导类的模板参数中设定变量维度为 1,3。

4. 设定好问题后，调用 solve 函数进行求解。你可以在 option 里配置（非常详细的）优化选项。例如，我们可以选择使用 Line Search 还是 Trust Region，迭代次数，步长等等。读者可以查看 Options 的定义，看看有哪些优化方法可选，当然默认的配置已经可以用在很广泛的问题上了。

最后，我们来看看实验结果。调用 build/curve_fitting 以查看优化结果：

```
% build/curve_fitting
generating data:
0 2.71828
0.01 2.93161
0.02 2.12942
0.03 2.46037
......
iter cost    cost_change   |gradient| |step| tr_ratio  tr_radius ls_iter iter_time total_time
0 1.824887e+04 0.00e+00 1.38e+03  0.00e+00 0.00e+00 1.00e+04 0      4.09e-05  1.48e-04
1 2.748700e+39 -2.75e+39 0.00e+00 7.67e+01 -1.52e+35 5.00e+03 1      1.09e-04  3.13e-04
2 2.429783e+39 -2.43e+39 0.00e+00 7.62e+01 -1.35e+35 1.25e+03 1      3.57e-05  3.75e-04
......
18 5.310764e+01 3.42e+00 8.50e+00  2.81e-01 9.89e-01 2.53e+03 1      3.09e-05  1.15e-03
19 5.125939e+01 1.85e+00 2.84e+00  2.98e-01 9.90e-01 7.60e+03 1      2.85e-05  1.19e-03
20 5.097693e+01 2.82e-01 4.34e-01  1.48e-01 9.95e-01 2.28e+04 1      2.82e-05  1.23e-03
21 5.096854e+01 8.39e-03 3.24e-02  2.87e-02 9.96e-01 6.84e+04 1      3.04e-05  1.27e-03
solve time cost = 0.00133349 seconds.
Ceres Solver Report: Iterations: 22, Initial cost: 1.824887e+04, Final cost: 5.096854e+01, Termination:
 CONVERGENCE
estimated a,b,c = 0.891943 2.17039 0.944142
```

从 Ceres 给出的优化过程中可以看到，整体误差从 18248 左右下降到了 50.9，并且梯度也是越来越小。在迭代 22 次后算法收敛，最后的估计值为：

$$a = 0.891943, b = 2.17039, c = 0.944142.$$

而我们设定的真值为

$$a = 1, b = 2, c = 1.$$

它们相差不多。

为了更直观地显示数据，我们可以把它画出来，如图 6-1 所示。这个图显示了带噪声的数据、真实模型和估计模型，可以看到估计模型和真实模型非常接近，几乎重合。我们同时记录了 Ceres 的运行时间，对这样一个 100 个点的优化问题，计算时间约在 1.3 毫秒左右（虚拟机上）。

希望读者通过这个简单的例子，对 Ceres 的使用方法有一个大致的了解。它的优点是提供了自动求导工具，使得我们不必去计算很麻烦的雅可比矩阵。Ceres 的自动求导是通

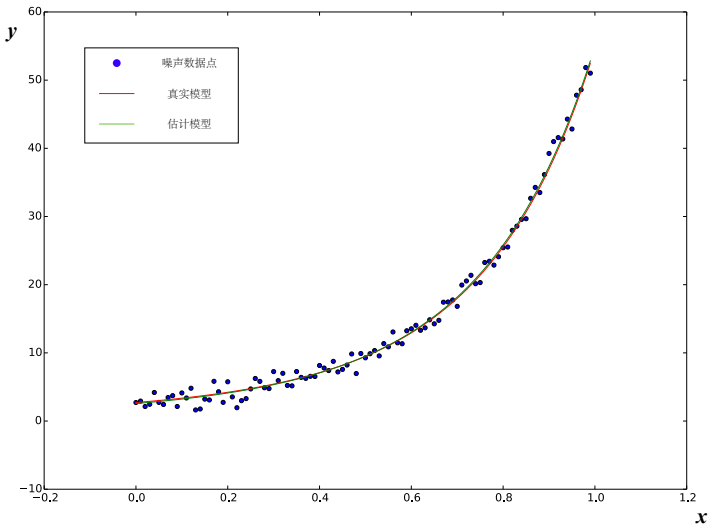


图 6-1 使用 Ceres 进行曲线拟合。红线为真实模型，绿线为估计模型，它们非常接近。

过模板元实现的，在编译时期就可以完成自动求导工作，不过仍然是数值导数。本书大部分时候仍然会介绍雅可比矩阵的计算，因为那样对理解问题更有帮助，而且在优化中更少出现问题。此外，Ceres 的优化过程配置也很丰富，使得它适合很广泛的最小二乘优化问题，包括 SLAM 中的各种问题。

6.4 实践：g2o

本章的第二个实践部分将介绍另一个（主要在 SLAM 领域）广为使用的优化库：g2o（General Graphic Optimization，G^2O）。它是一个基于**图优化**的库。图优化是一种将非线性优化与图论结合起来的理论，因此在使用它之前，我们花一点篇幅介绍一个图优化理论。

6.4.1 图优化理论简介

我们已经介绍了非线性最小二乘的求解方式。它们是由很多个误差项之和组成的。然而，仅有一组优化变量和许多个误差项，我们并不清楚它们之间的**关联**。比方说，某一个优化变量 x_j 存在于多少个误差项里呢？我们能保证对它的优化是有意义的吗？进一步，我们希望能够直观地看到该优化问题**长什么样**。于是，就说到了图优化。

图优化，是把优化问题表现成**图（Graph）**的一种方式。这里的**图**是图论意义上的图。一个图由若干个**顶点（Vertex）**，以及连接着这些节点的**边（Edge）**组成。进而，用**顶点**表示**优化变量**，用**边**表示**误差项**。于是，对任意一个上述形式的非线性最小二乘问题，我

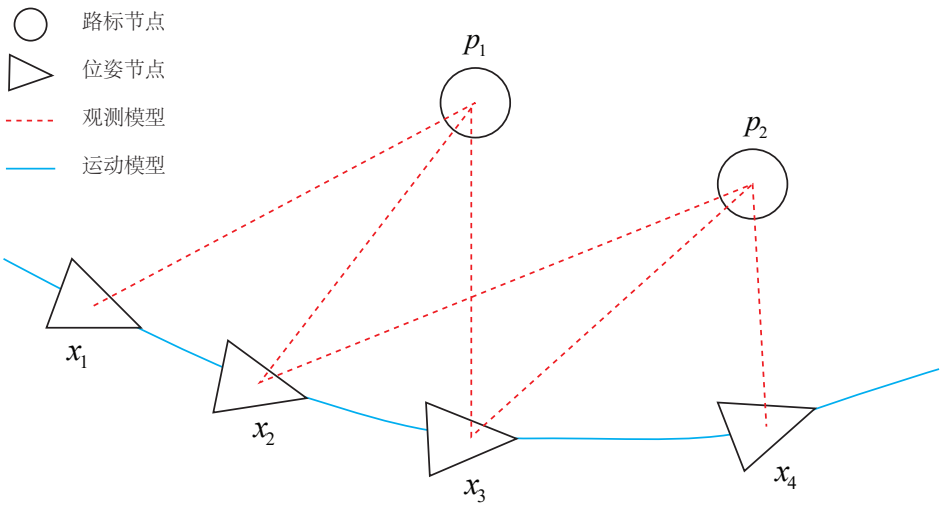


图 6-2 图优化的例子。

们可以构建与之对应的一个**图**。

图 6-2 是一个简单的图优化例子。我们用三角形表示相机位姿节点，用圆形表示路标点，它们构成了图优化的顶点；同时，蓝色线表示相机的运动模型，红色虚线表示观测模型，它们构成了图优化的边。此时，虽然整个问题的数学形式仍是式（6.12）那样，但现在我们可以直观地看到问题的**结构**了。如果我们希望，也可以做**去掉孤立顶点**或**优先优化边数较多（或按图论的术语，度数较大）的顶点**这样的改进。但是最基本的图优化，是用图模型来表达一个非线性最小二乘的优化问题。而我们可以利用图模型的某些性质，做更好的优化。

g2o 为 SLAM 提供了图优化所需的内容。下面我们来演示一下 g2o 的使用方法。

6.4.2 g2o 的编译与安装

在使用一个库之前，我们需要对它进行编译和安装。读者应该已经体验很多次这个过程了，它们基本都是大同小异的。关于 g2o，读者可以从 github 下载它：`https://github.com/RainerKuemmerle/g2o`，或从本书提供的第三方代码库中获得。

解压代码包后，你会看到 g2o 库的所有源码，它也是一个 CMake 工程。我们先来安装它的依赖项（部分依赖项与 Ceres 有重合）：

```
sudo apt-get install libqt4-dev qt4-qmake libqglviewer-dev libsuitesparse-dev libcxsparse3.1.2
libcholmod-dev
```

然后，按照 cmake 的方式对 g2o 进行编译安装即可，我们略去该过程的说明。安装完成后，g2o 的头文件将在/usr/local/g2o 下，库文件在/usr/local/lib/下。现在，我们重新考虑 Ceres 例程中的曲线拟合实验，在 g2o 中实验一遍。

6.4.3 使用 g2o 拟合曲线

为了使用 g2o，首先要做的是将曲线拟合问题抽象成图优化。这个过程中，只要记住**节点为优化变量，边为误差项**即可。因此，曲线拟合的图优化问题可以画成图 6-3 的形式。

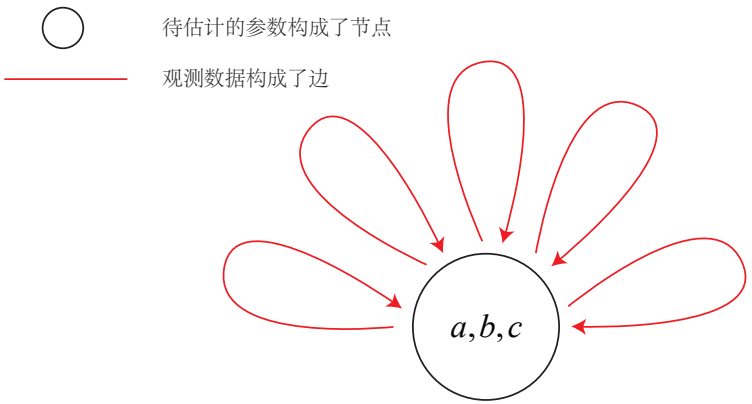


图 6-3 曲线拟合对应的图优化模型。（莫明其妙地有些像华为的标志）

在曲线拟合问题中，整个问题只有一个顶点：曲线模型的参数 a, b, c；而每个带噪声的数据点，构成了一个个误差项，也就是图优化的边。但这里的边与我们平时想的边不太一样，它们是**一元边**（Unary Edge），即**只连接一个顶点**——因为我们整个图只有一个顶点。所以在图 6-3 中，我们就只能把它画成自己连到自己的样子了。事实上，图优化中一条边可以连接一个、两个或多个顶点，这主要反映在每个误差与多少个优化变量有关。在稍微有些玄妙的说法中，我们把它叫做**超边**（Hyper Edge），整个图叫做**超图**（Hyper Graph）[①]。

弄清了这个图模型之后，接下来就是在 g2o 中建立该模型，进行优化了。作为 g2o 的用户，我们要做的事主要有以下几个步骤：

[①] 虽然我个人并不太喜欢有些故作玄虚的说法。

1. 定义顶点和边的类型；
2. 构建图；
3. 选择优化算法；
4. 调用 g2o 进行优化，返回结果。

下面来演示一下程序。

slambook/ch6/g2o_curve_fitting/main.cpp

```
#include <iostream>
#include <g2o/core/base_vertex.h>
#include <g2o/core/base_unary_edge.h>
#include <g2o/core/block_solver.h>
#include <g2o/core/optimization_algorithm_levenberg.h>
#include <g2o/core/optimization_algorithm_gauss_newton.h>
#include <g2o/core/optimization_algorithm_dogleg.h>
#include <g2o/solvers/dense/linear_solver_dense.h>
#include <Eigen/Core>
#include <opencv2/core/core.hpp>
#include <cmath>
#include <chrono>
using namespace std;

// 曲线模型的顶点，模板参数：优化变量维度和数据类型
class CurveFittingVertex: public g2o::BaseVertex<3, Eigen::Vector3d>
{
public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW
    virtual void setToOriginImpl() // 重置
    {
        _estimate << 0,0,0;
    }

    virtual void oplusImpl( const double* update ) // 更新
    {
        _estimate += Eigen::Vector3d(update);
    }
    // 存盘和读盘：留空
    virtual bool read( istream& in ) {}
    virtual bool write( ostream& out ) const {}
};

// 误差模型 模板参数：观测值维度，类型，连接顶点类型
class CurveFittingEdge: public g2o::BaseUnaryEdge<1,double,CurveFittingVertex>
{
```

```
public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW
    CurveFittingEdge( double x ): BaseUnaryEdge(), _x(x) {}
    // 计算曲线模型误差
    void computeError()
    {
        const CurveFittingVertex* v = static_cast<const CurveFittingVertex*> (_vertices[0]);
        const Eigen::Vector3d abc = v->estimate();
        _error(0,0) = _measurement - std::exp( abc(0,0)*_x*_x + abc(1,0)*_x + abc(2,0) ) ;
    }
    virtual bool read( istream& in ) {}
    virtual bool write( ostream& out ) const {}
public:
    double _x; // x 值， y 值为 _measurement
};

int main( int argc, char** argv )
{
    double a=1.0, b=2.0, c=1.0; //   真实参数值
    int N=100; //                    数据点
    double w_sigma=1.0; //           噪声Sigma值
    cv::RNG rng; //                  OpenCV随机数产生器
    double abc[3] = {0,0,0}; //      abc参数的估计值

    vector<double> x_data, y_data; // 数据

    cout<<"generating data: "<<endl;
    for ( int i=0; i<N; i++ )
    {
        double x = i/100.0;
        x_data.push_back ( x );
        y_data.push_back (
            exp ( a*x*x + b*x + c ) + rng.gaussian ( w_sigma )
        );
        cout<<x_data[i]<<" "<<y_data[i]<<endl;
    }

    // 构建图优化，先设定g2o
    // 矩阵块：每个误差项优化变量维度为 3 ，误差值维度为 1
    typedef g2o::BlockSolver< g2o::BlockSolverTraits<3,1> > Block;
    // 线性方程求解器：稠密的增量方程
    Block::LinearSolverType* linearSolver = new g2o::LinearSolverDense<Block::PoseMatrixType>();
    Block* solver_ptr = new Block( linearSolver ); // 矩阵块求解器
    // 梯度下降方法，从GN, LM, DogLeg 中选
    g2o::OptimizationAlgorithmLevenberg* solver = new g2o::OptimizationAlgorithmLevenberg( solver_ptr )
    ;
    // 取消下面的注释以使用 GN 或 DogLeg
    // g2o::OptimizationAlgorithmGaussNewton* solver = new g2o::OptimizationAlgorithmGaussNewton(
    solver_ptr );
    // g2o::OptimizationAlgorithmDogleg* solver = new g2o::OptimizationAlgorithmDogleg( solver_ptr );
```

```
    g2o::SparseOptimizer optimizer; // 图模型
    optimizer.setAlgorithm( solver ); // 设置求解器
    optimizer.setVerbose( true ); // 打开调试输出

    // 往图中增加顶点
    CurveFittingVertex* v = new CurveFittingVertex();
    v->setEstimate( Eigen::Vector3d(0,0,0) );
    v->setId(0);
    optimizer.addVertex( v );

    // 往图中增加边
    for ( int i=0; i<N; i++ )
    {
        CurveFittingEdge* edge = new CurveFittingEdge( x_data[i] );
        edge->setId(i);
        edge->setVertex( 0, v ); //          设置连接的顶点
        edge->setMeasurement( y_data[i] ); // 观测数值
        // 信息矩阵：协方差矩阵之逆
        edge->setInformation( Eigen::Matrix<double,1,1>::Identity()*1/(w_sigma*w_sigma) );
        optimizer.addEdge( edge );
    }

    // 执行优化
    cout<<"start optimization"<<endl;
    chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
    optimizer.initializeOptimization();
    optimizer.optimize(100);
    chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
    chrono::duration<double> time_used = chrono::duration_cast<chrono::duration<double>>( t2-t1 );
    cout<<"solve time cost = "<<time_used.count()<<" seconds. "<<endl;

    // 输出优化值
    Eigen::Vector3d abc_estimate = v->estimate();
    cout<<"estimated model: "<<abc_estimate.transpose()<<endl;

    return 0;
}
```

在这个程序中，我们从 g2o 派生出了用于曲线拟合的图优化顶点和边：CurveFittingVertex 和 CurveFittingEdge，这实质上是扩展了 g2o 的使用方式。在这两个派生类中，我们重写了重要的虚函数：

1. 顶点的更新函数：oplusImpl。我们知道优化过程最重要的是增量 $\Delta\boldsymbol{x}$ 的计算，而该函数处理的是 $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta\boldsymbol{x}$ 的过程。

   读者会觉得这并不是什么值得一提的事情，因为仅仅是个简单的加法而已，为什么 g2o 不帮我们完成呢？在曲线拟合过程中，由于优化变量（曲线参数）本身位于**向量空间**中，这个更新计算确实就是简单的加法。但是，当优化变量不处于向量空间中时，

比方说 $\boldsymbol{x}$ 是相机位姿，它本身不一定有加法运算。这时，就需要重新定义**增量如何加到现有的估计**上的行为了。按照第四讲的解释，我们可能使用左乘更新或右乘更新，而不是直接的加法。

2. 顶点的重置函数：setToOriginImpl。这是平凡的，我们把估计值置零即可。

3. 边的误差计算函数：computeError。该函数需要取出边所连接的顶点的当前估计值，根据曲线模型，与它的观测值进行比较。这和最小二乘问题中的误差模型是一致的。

4. 存盘和读盘函数：read, write。由于我们并不想进行读写操作，就留空了。

定义了顶点和边之后，我们在 main 函数里声明了一个图模型，然后按照生成的噪声数据，往图模型中添加顶点和边，最后调用优化函数进行优化。g2o 会给出优化的结果：

```
% build/curve_fitting
generating data:
0 2.71828
0.01 2.93161
0.02 2.12942
......
iteration= 13 chi2= 101.937020 time= 4.06e-05 cumTime= 0.00048135 edges= 100 schur= 0 lambda= 3678.088107 levenbergIter= 6
iteration= 14 chi2= 101.937020 time= 3.2215e-05 cumTime= 0.000513565 edges= 100 schur= 0 lambda= 19616.469906 levenbergIter= 3
iteration= 15 chi2= 101.937020 time= 0.000108524 cumTime= 0.000622089 edges= 100 schur= 0 lambda= 836969.382664 levenbergIter= 4
iteration= 16 chi2= 101.937020 time= 0.000159817 cumTime= 0.000781906 edges= 100 schur= 0 lambda= 224672257893341.656250 levenbergIter= 7
solve time cost = 0.00173976 seconds.
estimated model: 0.890911 2.1719 0.943629
```

我们使用 L-M 方法进行梯度下降，在迭代了 16 次后，最后优化结果与 Ceres 实验中相差无几。我们亦在程序中提供了使用 G-N 和 DogLeg 下降方式，请读者去掉它们前面的注释符号，自行对比一下各种梯度下降方法的差异。

6.5 小结

本节介绍了 SLAM 中经常碰到的一种非线性优化问题：由许多个误差项平方和组成的最小二乘问题。我们介绍了它的定义和求解，并且讨论了两种主要的梯度下降方式：Gauss-Newton 和 Levenberg-Marquardt。在实践部分中，我们分别使用了 Ceres 和 g2o 两种优化库求解同一个曲线拟合问题，发现它们给出了相似的结果。

由于我们还没有详细谈 Bundle Adjustment，所以实践部分选择了曲线拟合这样一个简单但有代表性的例子，以演示一般的非线性最小二乘求解方式。特别地，如果用 g2o 来拟合曲线，我们必须先把问题转换为图优化，定义新的顶点和边，这种做法是有一些迂回

的——g2o 的主要目的并不在此。相比之下，Ceres 定义误差项，求曲线拟合问题则自然了很多，因为它本身即是一个优化库。然而，在 SLAM 中，更多的问题是，一个带有许多个相机位姿和许多个空间点的优化问题如何求解。特别地，当相机位姿以李代数表示时，误差项关于相机位姿的导数如何计算，将是一件值得详细讨论的事。我们将在后续的章节中发现，g2o 提供了大量的顶点和边的类型，使得它在相机位姿估计问题中非常方便。而在 Ceres 中，我们不得不自己实现每一个 Cost Function，带来了一些不便。

在实践部分的两个程序中，我们没有去计算曲线模型关于三个参数的导数，而是利用了优化库的数值求导，这使得理论和代码都会简洁一些。Ceres 库提供了基于模板元的自动求导和运行时的数值求导，而 g2o 只提供了运行时数值求导这一种方式。但是，对于大多数问题，如果我们能够推导出雅可比矩阵的解析形式并告诉优化库，就可以避免数值求导中的诸多问题。

最后，希望读者能够适应 Ceres 和 g2o 这些大量使用模板编程的方式。也许一开始会看上去比较吓人（特别是 Ceres 设置 Problem 和 g2o 初始化部分的代码），但是一旦熟悉之后，就会觉得这样的方式是自然的，而且容易扩展。我们将在 SLAM 后端章节中，继续讨论稀疏性、核函数、位姿图（Pose Graph）等问题。

习题

1. 证明线性方程 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ 当系数矩阵 $\boldsymbol{A}$ 超定时，最小二乘解为 $\boldsymbol{x} = (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T\boldsymbol{b}$.

2. 调研最速下降法、牛顿法、GN 和 LM 各有什么优缺点。除了我们举的 Ceres 库和 g2o 库，还有哪些常用的优化库？你可能会找到一些 MATLAB 上的库。

3. 为什么 GN 的增量方程系数矩阵可能不正定？不正定有什么几何含义？为什么在这种情况下解就不稳定了？

4. DogLeg 是什么？它与 GN 和 LM 有何异同？请搜索相关的材料，例如[①]。

5. 阅读 Ceres 的教学材料以更好地掌握它的用法：`http://ceres-solver.org/tutorial.html`.

6. 阅读 g2o 自带的文档，你能看懂它吗？如果还不能完全看懂，请在第十、十一两讲之后回来再看。

7. * 请更改曲线拟合实验中的曲线模型，并用 Ceres 和 g2o 进行优化实验。例如，你可以使用更多的参数和更复杂的模型。

[①] `http://www.numerical.rl.ac.uk/people/nimg/course/lectures/raphael/lectures/lec7slides.pdf`