Ceres 非线性优化详细入门教程

- 时间：`20210217`
- 作者：`jqf`
- 目录

1. 基础知识

1.1 编程补遗

- **const 关键字与指针、函数**
    - const 关键字可以修饰指针，分为pointer to const 指向常量的指针、const pointer 常指针、const 修饰指针和指针指向的内容（双常指针，自己起的名），修饰函数，对于成员函数，不允许修改成员变量。

```
int b = 500;
const int* a = &b;  //修饰变量  pointer to const
int *const a = &b;  //修饰指针  const pointer
const int* const a = &b;  //修饰指针和变量 双常指针
void fun() const {};    //修饰函数 类内的成员变量不会被修改
```

- **模板函数**
    - c++模板函数简单来说就是实现了一个函数多种使用。

```
//模板函数定义的标准开头
```

```cpp
template <typename T>
T add(T a,T b)
{
  T sum = a+b;
  return sum;
}
int main()
{
  //注意模板函数调用的方式，需要在函数名后面用尖括号制定模板类型
  double sum1 = add<double>(3.5,1.1);
  double sum2 = add<int>(5,4);
  string sum3 = add<string>("a","b");
  return 0;
}
```

1.2 参考资料

- Ceres Solver 官方文档
- 一个很好的入门博客
- 几种损失函数比较

2. Ceres 简介

2.1 求解非线性最小二乘问题

- Ceres 能够求解如下形式的非线性最小二乘问题：

$$\min_x \frac{1}{2}\sum_i \rho_i(\|f_i(x_{i_1},x_{i_2},\dots,x_{i_k})\|^2) \quad (1)$$
$$s.t. \quad l_j \leqslant x_j \leqslant u_j$$

2.2 Parameter Blocks 参数块

- 顾名思义，**参数的块**。
  - **参数**，也就是需要求解的**优化变量**，可以是多维向量，例如公式中表示有 k 个待优化的变量。如式（2）所示。
  - **参数块**，需要优化的参数可能不止一组，比如在图像或点云匹配中，需要优化旋转和平移，优化参数可以用 **4x1** 的旋转四元数 q 和 **3x1** 的平移向量 t 作为独立的两组参数，此时参数块就是一个4维的向量和3维的向量，如果放在一起优化就是 7 维向量，如果是分开计算，就是1组4维+1组3维。注意区分参数的维度，和参数块的维度。

$$(x_{i_1},x_{i_2},\dots,x_{i_k}) \quad (2)$$

2.3 Cost Function 代价函数

- 简单理解代价就是预测值和实际值之间的差距，这与机器学习中的代价函数含义是相同的。我们求解最小二乘的目的就是找到**能让代价最小的一组优化变量**，也就是对应上面的**参数**。

$$f_i(x_{i_1},x_{i_2},\dots,x_{i_k}) \quad (3)$$

2.4 Residual Block 残差块

- 为什么叫残差**块**，字面意思，**残差的块**。
  - **残差**，虽然优化中的残差 r 肯定是标量，但是描述这个残差可能用到了不止一个标量。比如我们在优化位姿的过程中，残差为变换矩阵，即平移 t 和旋转 q。那么对于平移来说，有 $t=[x,y,z]^T$ **三个**标量，而对于旋转，如果用四元数来表示，则有 $q=[w,x,y,z]^T$ **四个**标量，残差 r 的最简单的形式就是一个 1x1 的标量，但实际上由于代价函数的区别，残差不一

定是 1x1 的标量，也可能是个向量，比如上面的形式。但是我们优化的是二乘项最小，也就是 $r^T r$ 最小，列向量的转置乘自己必然是个 1x1 标量。而实际优化我们是在利用二乘项 $r^T r$，也就是无论残差是什么维度，最后我们利用二乘都能计算出一个标量值。

- **残差块**，这里的块与数据是对应的，以典型的拟合曲线问题来说，优化曲线方程的参数，我们会有大量的数据，但是不知道曲线参数，我们把方程参数作为优化参数，对每组数据都会计算残差，比如我们有 100 个点，那么就会计算 100 个残差向量，这100个残差向量就叫残差块。通过给定初值，然后迭代优化，最终我们得到了能让这 100 个残差都收敛的一组方程参数，优化就完成了。注意区分残差的维度，和残差块的维度。

$$\rho_i(\|f_i(x_{i_1}, x_{i_2}, \ldots, x_{i_k})\|^2) \tag{4}$$

2.5 Loss Function 损失函数

- 该函数是一个标量值函数，用于**减少离群值对非线性最小二乘问题解的影响**，注意与代价函数做区分，代价函数是计算残差的，而损失函数是用于减少离群值使优化更稳定，例如，$\rho_i(x) = x$，上面的代价函数就变为如下形式，即标准的非线性最小二乘问题。

$$\min_x \frac{1}{2} \sum_i (\|f_i(x_{i_1}, x_{i_2}, \ldots, x_{i_k})\|^2) \tag{5}$$

- 实际使用中，损失函数有多种形式，这与机器学习中的概念也是一致的，如果熟悉鲁棒最小二乘，这个损失函数也常常称作**鲁棒核函数**，例如 huber 核，这些核函数没有直接利用残差的平方和来做为优化的目标，而是经过了一些计算，能让求解更稳定，简单参考。

2.6 lower and upper bounds 上下界限

- 是参数中每个优化变量对应的界限值。

$$l_j \leqslant x_j \leqslant u_j \tag{6}$$

3. 各个模块解析

3.1 Cost Function 代价函数

- 对于目标函数中的每个代价函数（例如 k 个）而言，它在 Ceres 中的功能是用于计算**残差向量**和**雅可比矩阵**。
- 举例说明，对于一组参数块 $[x_1, \ldots, x_k]$ 对应的代价函数 $f(x_1, \ldots, x_k)$ 来说，在参数块给定的情况下，`CostFunction` 就负责计算**残差向量** $f(x_1, \ldots, x_k)$ 和**雅可比矩阵** D_i：

$$J_i = D_i f(x_i, \ldots, x_k) \quad i \in P1, \ldots, k \tag{7}$$

- 看一下 `CostFunction` 虚基类的原型:

```
class CostFunction {
 public:
  virtual bool Evaluate(double const* const* parameters,
                        double* residuals,
                        double** jacobians) = 0; //纯虚函数
  const vector<int32>& parameter_block_sizes(); //设置参数块的维度
  int num_residuals() const; //设置残差的维度

 protected:
  vector<int32>* mutable_parameter_block_sizes();
  void set_num_residuals(int num_residuals);
};
```

- 注意在 **public** 下定义的两个成员函数 **parameter_block_sizes** 和 **num_residuals** 就是用于在子类中设置具体的参数块维度 **parameter_block_sizes_**和残差维度 **num_residuals_**。还有一个 **Evaluate** 纯虚函数需要在子类中实现，该函数就是用于计算残差向量和雅可比的。

3.2 SizedCostFunction 带维度的代价函数

- 更一般地来说，我们的**残差维度**和**参数维度**通常已知，那么 Ceres 提供了一种更便捷的虚基类 **SizedCostFunction** 如下所示，我们在继承的过程中只需要实现 **Evaluate** 函数即可。

```
template<int kNumResiduals, int... Ns>
class SizedCostFunction : public CostFunction {
 public:
  virtual bool Evaluate(double const* const* parameters,
                        double* residuals,
                        double** jacobians) const = 0;
};
```

3.3 CostFunction 中的 Evaluate 函数详解【关键点】

- **功能：**用于计算残差向量和雅可比矩阵
- **参数：**
  - `parameters`：**多维**向量，表示参数块，也就是多少组残差（即代价函数）需要计算。如式（8）中矩阵所示，对于每一组 **parameters[i].size()** 我们有 m 个参数（行的数量），整个参数块的 **parameters.size()** 是 n 也就是列的数量，最终形成了 $m \times n$ 大小的块，**但是特别注意的是，这里为了书写方便，用了矩阵的形式，实际上，每一组参数对应的维度 m 并不一定相当相等，还是拿旋转平移举例子，对应过来就是 2 列参数，一列是 4 维，1 列是 3 维。**

$$
\begin{aligned}
parameter_{[m\times 1]} &= \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \\
parameters_{[m\times n]} &= \begin{bmatrix} x_{11} & \dots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \dots & x_{mn} \end{bmatrix}
\end{aligned} \tag{8}
$$

    - `parameters[i]` **一维**向量，表示当前第 $i, (i \in [1, n])$ 组（列）参数块向量，里面包含了代价函数计算所需要的 m 个优化变量，这一组参数计算只能得到一个残差向量。
    - 相应的，`parameters[i][k]` 就表示，第 i 组参数块中的第 k 个参数具体的值，这在赋值的过程中要弄清楚，还是旋转平移举例，对于参数块中的两组参数 q, t，第一组参数是 `parameters[0][1,2,3,4]` 第二组参数是 `parameters[1][1,2,3]`。
  - `residuals` 表示具体的残差向量，注意维度要和 **num_residuals_** 一致。这个残差维度完全取决于残差的形式，比如是李代数还是其他什么的。
    - 如式（8）所示，残差的维度是 j（行的数量），**残差块**（**ResidualsBlock**）中一共存放有 k 组残差（列的数量），最终形成了 $j \times k$ 的残差矩阵。 residuals_block 为了区分块和单个残差。

$$
\begin{aligned}
residuals_{[j\times l]} &= \begin{bmatrix} r_1 \\ \vdots \\ r_j \end{bmatrix} \\
residuals\_block_{[k\times l]} &= \begin{bmatrix} r_{11} & \dots & r_{1k} \\ \vdots & \ddots & \vdots \\ r_{j1} & \dots & r_{jk} \end{bmatrix}
\end{aligned} \tag{8}
$$

- `jacobians` 表示雅可比矩阵，其维度与 `parameters` 一致，这个意思是说，参数块中有多少组参数，就对应有多少个雅可比，从英文来看，**jacobians** 后面的 **s** 表示这是一系列雅克比的集合。
  - 从通常意义来看，雅可比是一个矩阵，那么复数的雅可比矩阵是什么形式呢？实际上在 Ceres 中众多雅可比矩阵 **们** 构成了一个大矩阵，而每一个雅可比矩阵被转换成了一个**向量**，按照**行优先的顺序**排列。参考这里 。

    > `jacobians` is an array of arrays of size `CostFunction::parameter_block_sizes_.size()`.
    >
    > `jacobians[i]` is a row-major array of size `num_residuals x parameter_block_sizes_[i]`.

  - 如此，**jacobians** 维度与 **parameters** 的维度一致就解释得通了。由于 **parameters** 可能是几组不同维度的参数，每一组参数，就要对应计算一个雅可比（**向量**），最终，**jacobians** 的大小就和 **parameters** 的大小一模一样。
  - 正如文档中解释的那样，`jacobians[i]` 表示代价函数对于第 i 组参数 `parameters[i]` 的雅可比矩阵（向量）。实际上，这个雅可比 **jacobians[i]** 的维度应该是，**残差的维度 x 参数的维度，因为雅可比就是 残差向量 对 参数向量 求 偏导数**，按照分子形式来构建雅可比，雅可比的行数就是残差向量的维度。残差维度是 j，参数维度是 m，那么单独一个雅可比的维度就是 $j \times m$，转换成行优先的向量，什么是行优先，参考Row- and Column- major order（行优先和列优先顺序）。

$$
\begin{aligned}
jacobians[i]_{j\times m} &= \begin{bmatrix} J_{11} & \dots & J_{1m} \\ \vdots & \ddots & \vdots \\ J_{j1} & \dots & J_{jm} \end{bmatrix} \\
&= [(J_{11},\dots,J_{1m}),(J_{21},\dots,J_{2m}),\dots,(J_{j1},\dots,J_{jm})]^T
\end{aligned} \tag{9}
$$

  - 按照上面的解释，实际上的 **jacobians** 的列数就是 n 个，与参数块一致，行的数量就是 $row = j \times m$，最终就长这个样子，如式（10）所示。**这里仍然需要特别注意，我们利用了矩阵的形式来方便表达雅可比们，实际上程序中 jacobians 中的每个雅可比的维度对应每一组参数维度，由于参数块中每组参数的维度 m 不一定相等，下面式（10）中每一列的 row 的大小也是不一定相等的，与式（8）上面的解释保持一致。**

$$
jacobians_{[row\times n]} = \begin{bmatrix} J & \dots & J_{1n} \\ \vdots & \ddots & \vdots \\ J_{row} & \dots & J_{row\times n} \end{bmatrix}, row = j \times m \tag{10}
$$

- **大白话小结：**对于这些参数具体怎么在 Ceres 中存储，是向量还是矩阵，只要了解就好，关键还是明确在构建代码过程中，如果是自动求导，我们要传入的就是**残差和参数的维度**；如果手动求雅可比，每组雅可比的向量大小就是残差乘参数的大小，在程序中，每组雅可比一般是分开赋值的，互不影响。

3.4 AutoDiffCostFunction 自动求导代价函数

- 在 `CostFunction` 或 `SizedCostFunction` 中都需要实现求雅可比的方法，Ceres 还提供了一个自动求导数类，定义如下：

```
template <typename CostFunctor, //仿函数的结构体名称
       int kNumResiduals,  // Number of residuals, or ceres::DYNAMIC. 残差维度
       int... Ns>          // Size of each parameter block 参数维度
class AutoDiffCostFunction : public
SizedCostFunction<kNumResiduals, Ns> {
 public:
  AutoDiffCostFunction(CostFunctor* functor, ownership =
TAKE_OWNERSHIP);
  // Ignore the template parameter kNumResiduals and use
  // num_residuals instead.
  AutoDiffCostFunction(CostFunctor* functor,
                       int num_residuals,
                       ownership = TAKE_OWNERSHIP);
};
```

- 首先看这个自动求导的类是一个模板，需要传入一个 **CostFunctor** ，以及残差和参数维度来在定义该对象。那么这个 **CostFunctor** 是什么东西？有什么要求呢？因为这个知识点非常重要，我们单独开辟一个小节来解释。

3.5 仿函数 CostFunctor 详解【关键点】

- 参照官方文档的说明，为了获取一个自动求导类，我们必须定义一个模板化的运算符重载函数 **operator()** ，下文都叫 **functor** ，中文叫做**仿函数**或者**函数对象**。仿函数简单理解有三点：
  - 仿函数是使一个类的使用看上去像一个函数，从编程来说，这还是一个类。
  - 仿函数必须重载 **operator()** 运算符。
  - 调用仿函数，实际上就是通过类对象调用重载后的 **operator()** 运算符。
- 我们以一个常见的最小二乘误差为例，假设有一个标量误差 e

$$
\begin{aligned}
&e = k - x^T y \\
&x = [x_0, x_1]^T, y = [y_0, y_1]^T, k = 1 \times 1
\end{aligned} \tag{11}
$$

- 其中，各变量维度如式（11）所示，k 可能是测量值，而 $x^T y$ 可能是估计值，在最小二乘问题中，实际的代价应该是 $e^2 = (k - x^T y)^2$ ，但是**请特别注意**，但是，平方是由 Ceres 优化框架**隐式完成**的，所以提醒我们，在计算残差时不要自作聪明地加上平方了，原文如下：

> The actual cost added to the total problem is e^2, or $(k - x^T y)^2$; however, the squaring is implicitly done by the optimization framework.

- 上述残差模型的 **CostFunctor** 的实际例子如下边代码所示，这个例子说清了我们应该定义一个啥样的仿函数，并且如何去重载这个 **operator()** 运算符，官方说明解释了，自动求导框架会用合适的 **Jet** 对象来替换 T，但这一过程是隐藏的，所以在编程的时候，请把 **T** 当成一个普通的标量类型，例如 **double** 就可以了，也暂时不要去纠结什么是 **Jet**。

```
class MyScalarCostFunctor
{
    MyScalarCostFunctor(double k): k_(k) {} //构造函数，给成员 k_ 赋值
    // 核心函数 operator() 是一个模板
    // const T* const x 中指针 T 类型和变量 x 都是 const 不可修改
    // T* e 是可以修改的残差
    template <typename T>
    bool operator()(const T* const x , const T* const y, T* e) const
    {   //const 不修改成员变量，只用于计算
        //e是个标量，所以只有 [0]，而x,y是二维的，所以有 0,1
        e[0] = k_ - x[0] * y[0] - x[1] * y[1];
        //注意这里的残差没有 e[0] 平方，Ceres 内部会自动平方
```

```
        return true;
    }
private:
    double k_;
};
```

- 如何通过仿函数来构造一个代价函数对象，对于上述例子，每次计算残差都会事先有一个观测值 k 的实例传入，而我们的优化参数是x,y，那么就是两组参数，x,y的维度都是2，而计算出的残差是一个 1x1 标量，代价函数实例化时传入的维度就是 1，2，2 ，这个维度会进一步使用在仿函数 **MyScalarCostFunctor** 的 **operator()** 函数中，请看如下代码。

```
CostFunction* cost_function
    = new AutoDiffCostFunction<MyScalarCostFunctor, 1, 2, 2>(
        new MyScalarCostFunctor(1.0));              ^  ^  ^
                                                    |  |  |
                        Dimension of residual ------+  |  |
                        Dimension of x -----------------+  |
                        Dimension of y --------------------+
```

- **个人理解**：这个 `<MyScalarCostFunctor, 1, 2, 2>` 中的数字 **1，2，2** 即对应 `operator()(const T* const x , const T* const y, T* e)` 中的 **x,y,e** 的维度，**这就是仿函数的参数的意义**，即告诉残差计算代码中，残差的维度，以及参数块中每组参数的维度。因为在重载 **operator()** 时，形参全部都是指针，我们并不知道下标的界限，但是通过下面的构造过程，我们知道了各个变量的维度，相当于显式地说明了下标的维度，如此就可以方便的去索引具体的值，并计算残差和雅可比，那么残差计算代码中经常出现的 **[0]，[1]** 等下标的含义就不言自明了。

3.6 解析式代价函数的一个实例

- 上述介绍了利用自动求导类构造代价函数的例子，在实际的算法开发中，如果能够显式地指定雅可比矩阵，那么非线性优化地效率会更高，下面以一个非线性函数来展示解析求雅可比的实现过程：
- 函数形式如式（12）所示，也就是说，给定 n 组点对 $[x_i, y_i]$ ，来确定最符合这些数据的 $[b_1, b_2, b_3, b_4]$。

$$y = \frac{b_1}{(1 + e^{b_2 - b_3 x})^{1/b_4}} \tag{12}$$

- 要做拟合，目标函数的形式就应该是这样

$$\begin{aligned} E(b_1, b_2, b_3, b_4) &= \sum_i f^2(b_1, b_2, b_3, b_4, x_i, y_i) \\ &= \sum_i (y_i - \frac{b_1}{(1 + e^{b_2 - b_3 x_i})^{1/b_4}})^2 \end{aligned} \tag{13}$$

- 那么在 Ceres 中，我们需要定义的是 代价函数 f (**CostFunction**) 以及残差的计算方式，我们的残差也就是小 f 函数就是平方内的计算式：

$$residuals_i = f_i = y_i - \frac{b_1}{(1 + e^{b_2 - b_3 x_i})^{1/b_4}} \tag{14}$$

- 首先我们给出解析形式的雅可比矩阵，分析 **Jacobian** 的维度，**残差**是 $f_i = y_i - y_{est}$ 也就是实际数据减去估计值，是一个 **1 维的标量**，而优化变量是 $[b_1, b_2, b_3, b_4]$ 是一个四维的向量，雅可比就是 4x1 的矩阵，如下：

$$J_{4\times 1}=\begin{bmatrix}J_{11}\\J_{21}\\J_{31}\\J_{41}\end{bmatrix}=\begin{bmatrix}\frac{\partial f_i(b,x,y)}{\partial b_1}\\\frac{\partial f_i(b,x,y)}{\partial b_2}\\\frac{\partial f_i(b,x,y)}{\partial b_3}\\\frac{\partial f_i(b,x,y)}{\partial b_4}\end{bmatrix} \tag{15}$$

$$\begin{aligned}
D_1 f(b_1,b_2,b_3,b_4;x,y) &= \frac{1}{(1+e^{b_2-b_3x})^{1/b_4}}\\
D_2 f(b_1,b_2,b_3,b_4;x,y) &= \frac{-b_1 e^{b_2-b_3x}}{b_4(1+e^{b_2-b_3x})^{1/b_4+1}}\\
D_3 f(b_1,b_2,b_3,b_4;x,y) &= \frac{b_1 x e^{b_2-b_3x}}{b_4(1+e^{b_2-b_3x})^{1/b_4+1}}\\
D_4 f(b_1,b_2,b_3,b_4;x,y) &= \frac{b_1\log\left(1+e^{b_2-b_3x}\right)}{b_4^2(1+e^{b_2-b_3x})^{1/b_4}}
\end{aligned}$$

- 在得到这些雅可比形式后，我们就可以去定义 **CostFunction** 了，代码如下：

```
class Rat43Analytic : public SizedCostFunction<1,4> //1=残差维度，4=优化变量的维度
{
    public:
    //构造函数，传入的是每一组数据的已知量 x_i,y_i
    Rat43Analytic(const double x, const double y) : x_(x), y_(y) {}
    virtual ~Rat43Analytic() {}
    //Evaluate 函数是重载的，有固定形式的参数
    //该函数实际上是自动求导中的 仿函数（计算残差） + 手动计算雅可比 的组合
    virtual bool Evaluate(double const* const* parameters,
                          double* residuals,
                          double** jacobians) const
    {
        //parameters 表示参数块的维度，因为参数只有一组 1x4 的向量，所以参数块只有一级索引只有 0
        //参数块的2级索引就取决于上面传入的 4 ，也就是该参数有4个值。
        const double b1 = parameters[0][0];
        const double b2 = parameters[0][1];
        const double b3 = parameters[0][2];
        const double b4 = parameters[0][3];
        //定义一些中间变量
        const double t1 = exp(b2 -  b3 * x_);
        const double t2 = 1 + t1;
        const double t3 = pow(t2, -1.0 / b4);
        //残差向量只有一个值所以直接索引 [0]
        residuals[0] = b1 * t3 - y_;
        //手动定义雅可比，雅可比的维度是 1x4 的向量
        if (!jacobians) return true;
        double* jacobian = jacobians[0];//参数块只有1组，那么雅可比们也只有1组，该雅可比的大小=4x1
        if (!jacobian) return true;
        const double t4 = pow(t2, -1.0 / b4 - 1);
        //实际计算每一个偏导数的值并赋值 雅可比向量
        jacobian[0] = t3;
```

```
        jacobian[1] = -b1 * t1 * t4 / b4;
        jacobian[2] = -x_ * jacobian[1];
        jacobian[3] = b1 * log(t2) * t3 / (b4 * b4);
        return true;
    }
    private:
    const double x_;//外部传入的已知从参数块中的第1组参数 x
    const double y_;//外部传入的已知从参数块中的第2组参数 y
};
```

- **大白话小结：**
  - 我们一开始疑惑的是，仿函数跟 **Evaluate** 函数以及 **CostFunction** 之间的关系，实际上，仿函数主要是用来计算残差的，而形参就是参数向量和残差向量，你可以根据具体的残差维度任意地定义仿函数的形参，在实例化该仿函数时只要传正确的维度信息就可以，而仿函数内是没有计算雅可比的，也就是说仿函数多用于自动求导的情形，对应的就是 ceres 中提供的自动求导的代价函数。
  - 那么如果是手动求导，可以不用仿函数（当然也可以继续使用仿函数只计算残差，而在**Evaluate**函数内计算雅可比），而直接继承 **SizedCostFunction** 有维度代价函数，并实现 **Evaluate** 函数。 **Evaluate** 函数是个虚函数，形参的形式是固定的，在从形参中取参数和残差以及雅可比使用或赋值时，需要遵循一定的索引规律，**Evaluate** 的形参 **parameters** 和 **jacobians** 都是块的概念，需要二维索引，而 **residuals** 是向量，这些索引的界限都是对应在构造代价函数时就指定好的残差以及参数块的维度。

3.7 AddResidualBlock 添加残差块

- 顾名思义主要用于向优化问题中传递残差模块的信息，函数原型如下，传递的参数主要包括代价函数模块、损失函数模块和参数模块。

```
ResidualBlockId Problem::AddResidualBlock(CostFunction *cost_function,
                                          LossFunction *loss_function,
                                          const vector<double *>
parameter_blocks)
```

- 代价函数包含了参数模块的维度信息，`AddResidualBlock( )`函数会检测传入的参数模块是否和代价函数模块中定义的维数一致，维度不一致时程序会强制退出。
- 损失函数用于处理参数中含有野值的情况，避免错误量测对估计的影响，常用参数包括`HuberLoss`、`CauchyLoss`等（完整的参数列表参见Ceres API文档）；该参数可以取`NULL`或`nullptr`，此时损失函数为单位函数。
- 参数模块可一次性传入所有参数的**指针容器 vector<double*>** 或依次传入所有参数的指针 **double***。
- 这里简单给出定义，这个函数是要结合代码来更好地理解。

4. Ceres 求解优化问题的流程

- 本小节，我们利用一个ceres 官方 函数求极值的例子，来实际说明 ceres 的使用流程，以及解析和自动求导的实现方法。

4.1 Powell's function 函数求极值

- 我们利用 ceres 官方的例子来讲解，我们的不同点是对该例子中的自动求导提供了解析求导的对应实现方法。

- 先看函数原型

$$
\begin{aligned}
f_1(x) &= x_1 + 10x_2 \\
f_2(x) &= \sqrt{5}(x_3 - x_4) \\
f_3(x) &= (x_2 - 2x_3)^2 \\
f_4(x) &= \sqrt{10}(x_1 - x_4)^2 \\
F(x) &= [f_1(x),\ f_2(x),\ f_3(x),\ f_4(x)]
\end{aligned} \tag{16}
$$

- 对于这个函数求极值，我们的优化参数显而易见，就是函数的自变量 x_1, x_2, x_3, x_4 ，这里注意了，我们的函数有四个子函数，实际上对于整体大函数来看，我们有四个小的函数需要优化，也就是对应四个残差项，在这里残差直接为函数值，也就是个 1x1 标量，每一个小函数，又都有两组参数，比如 f_1 中有 x_1, x_2 这两组参数，也就是说每个小函数的参数块都是 2 ，残差维度是 1。

4.2 自动求导的实现

- 在明确了残差和参数的维度后，我们首先来看自动求导的实现，为了简化代码观看，省略了类似的仿函数定义 F2，F3和F4。

```
//step 0
struct F1
{
    template<typename T>
    bool operator()(const T *const x1, const T *const x2, T *residual) const
    {
        // f1 = x1 + 10 * x2;
        residual[0] = x1[0] + 10.0 * x2[0];
        return true;
    }
};
//struct F2 ...
//struct F3 ...
//struct F4 ...
int main(int argc, char **argv)
{
    //step 1 - 定义优化变量并赋初值
    double x1 = 3.0;
    double x2 = -1.0;
    double x3 = 0.0;
    double x4 = 1.0;
    //step 2 - 定义优化问题类对象&添加残差块
    Problem problem;
    //下面的AutoDiffCostFunction中的参数为仿函数，残差=1维，1组参数=1维，2组参数=1维
    problem.AddResidualBlock(new AutoDiffCostFunction<F1, 1, 1, 1>(new F1), NULL, &x1, &x2);
    problem.AddResidualBlock(new AutoDiffCostFunction<F2, 1, 1, 1>(new F2), NULL, &x3, &x4);
    problem.AddResidualBlock(new AutoDiffCostFunction<F3, 1, 1, 1>(new F3), NULL, &x2, &x3);
    problem.AddResidualBlock(new AutoDiffCostFunction<F4, 1, 1, 1>(new F4), NULL, &x1, &x4);
    //step 3 - 定义解决方法及选项
    Solver::Options options;
    options.max_num_iterations = 100;
    options.linear_solver_type = ceres::DENSE_QR;
    options.minimizer_progress_to_stdout = true;
    Solver::Summary summary;
    //step 4 - 求解问题
    Solve(options, &problem, &summary);
```

```
    return 0;
}
```

- 从上面的代码来看，使用自动求导代价函数 AutoDiffCostFunction 来求解一个非线性优化问题就是 **4步走**（仿-初-残-解）：
  - <1> **仿** - 定义每组残差的求法，即定义仿函数并实现 **operator()** 。
  - <2> **初** - 定义优化变量并赋初值，这些变量对象的值会被不断修改，最终会储存最优的值。
  - <3> **残** - 定义问题对象并添加残差块，这里需要注意我们在 3.7 小结提到了参数形式，第一个参数要代价函数的实例，代价函数中又要传入仿函数和残差以及参数的维度。
  - <4> **解** - 定义求解的方法并对迭代次数、求解类型等选项进行赋值。

4.3 解析求导的实现

- 在自动求导实现中，我们不需要手动计算雅可比，而是交给 ceres 自动推导，但是这样会牺牲一些计算效率，下面的代码就是解析求导的实现过程，与自动求导不同的是，这里我们继承了有维度的代价函数 SizedCostFunction ，而且我们没有使用仿函数来计算残差，而是直接在 Evaluate 函数中计算，这里的维度发生变化，下面来仔细说明：

- 注意 **SizedCostFunction** 后面的维度是 4 和 4 意味着我们的残差是 4 维对应一组参数是 4 维的。我们这里将 F1~F4 的子函数的参数合并在一起看成一组参数，那么 parameters 对应的1级索引只有1维，而2级索引就有4维。对应残差也是原来 F1~F4 的函数计算式，这里并没有改变。
- 重点是解析雅可比的计算，由于我们的参数是4维，残差也是4维，那么 4x4 = 16 ，我们的雅可比实际上是 4x4 的矩阵也就是 ceres 中的 16 维的向量。而参数块只有 1 个，jacobians 的1级索引就只有1维，而2级索引应该有16维，那么对于 **jacobians[0]** 中的 0~4 的值就是第一个残差值（f1）分别对4个参数的求偏导数，以此类推分别对这16个值进行赋值。
- 最后在求解问题的过程中步骤与自动求导一致，只是在添加残差块的时候，传入我们自定义的代价函数以及参数向量。

```
class PowellAnalyticAll : public SizedCostFunction<4, 4>
{
public:
    PowellAnalyticAll(){}
    virtual ~PowellAnalyticAll(){}
    virtual bool Evaluate(double const *const *parameters,
                          double *residuals,
                          double **jacobians) const
    {
        const double x1 = parameters[0][0];
        const double x2 = parameters[0][1];
        const double x3 = parameters[0][2];
        const double x4 = parameters[0][3];
        residuals[0] = x1 + 10.0 * x2;
        residuals[1] = sqrt(5.0) * (x3 - x4);
        residuals[2] = (x2 - 2.0 * x3) * (x2 - 2.0 * x3);
        residuals[3] = sqrt(10.0) * (x1 - x4) * (x1 - x4);
        if (jacobians != NULL && jacobians[0] != NULL)//J = 4x4 f only for jacobian[0]
        {
            //d(f1)/dx1234
            jacobians[0][0] = 1.0;
            jacobians[0][1] = 10.0;
            jacobians[0][2] = 0.0;
            jacobians[0][3] = 0.0;
            //d(f2)
            jacobians[0][4] = 0.0;
```

```
            jacobians[0][5] = 0.0;
            jacobians[0][6] = sqrt(5.0);
            jacobians[0][7] = -sqrt(5.0);
            //d(f3)
            jacobians[0][8] = 0.0;
            jacobians[0][9] = 2.0 * x2;
            jacobians[0][10] = -4.0 * x2;
            jacobians[0][11] = 0.0;
            //d(f4)
            jacobians[0][12] = 2.0 * sqrt(10.0);
            jacobians[0][13] = 0.0;
            jacobians[0][14] = 0.0;
            jacobians[0][15] = -2.0 * sqrt(10.0);
        }
        return true;
    }
};
int main(int argc, char **argv)
{
    Problem problemAnaly;
    std::vector<double *> para_blocks;
    double x_1234[4] = {3, -1, 0, 1};
    problemAnaly.AddResidualBlock(new PowellAnalyticAll, NULL, x_1234);
    Solver::Options options1;
    options1.max_num_iterations = 100;
    options1.linear_solver_type = ceres::DENSE_QR;
    options1.minimizer_progress_to_stdout = true;
    Solver::Summary summary1;
    Solve(options1, &problemAnaly, &summary1);
    return 0;
}
```

- 大白话小结，实际上利用 ceres 自动求导只要明确参数维度和残差维度，并给出残差的计算方法，自己实现解析求导就需要对雅可比矩阵有相当的了解，大部分场景中，如果不是追求高精度和高效率，都可以使用自动求导。

5. 小结

5.1 ceres 求解的一般流程 = 雅初残解

- 这里我们根据自动和解析求导来扩充之前总结的方法，还是 **4步走**：
    - <1> **雅** - 定义每组残差的求法，自动求导就是定义仿函数并实现 **operator()**，解析求导就是继承 **SizedCostFunction** 并实现 **Evaluate()**。
    - <2> **初** - 定义优化变量并赋初值，这些变量对象的值会被不断修改，最终会储存最优的值。
    - <3> **残** - 定义问题对象并添加残差块，根据参数、仿函数（自动求导）、代价函数（解析求导）进行传参。
    - <4> **解** - 定义求解的方法并对迭代次数、求解类型等选项进行赋值。

5.2 理解求函数极值和拟合曲线的区别

- 在构建代价函数 **CosfFunction** 的过程中有两个地方容易**产生混淆**，就是 **operator()** 中的参数和**类的构造函数中的参数**（即类的私有成员变量），明确对于不同的问题，这两个地方的参数含义不同。
- 例如，在**求函数的极值**的过程中，例如 $f(x, k_1, k_2) = y$，优化变量一般是函数的自变量 x，那么函数内部的已知常数参数 k_1, k_2 就需要储存在类的内部，并在类构造的时候传入，而 **operator()** 中传入的参数就是一组一组自变量 x 和因变量 y 的具体值以及残差，因为是求极值，让函数值最

小，一般来说函数的最小值出现在导数为零的地方，那么残差就应该是函数的导数 $[f(x,k_1,k_2)']^2$ ，**这样 Ceres 中的第一个例子就说得通了**，对于函数 $y=0.5*(10-x)^2$ ，它对于 x 的一阶导数是 $10-x$ ，那么我们的残差就是 $10-x$ ，而我们的目标函数应该就是 $(10-x)^2$。

- 反之，如果要求的**问题是曲线拟合**，还是这个函数 $f(x,k_1,k_2)=y$ ，其中 k_1,k_2 变为待优化的变量，而 x,y 是已知的，那么我们在类构造时就要传入 x,y 的值，他们也要定义为私有成员，而 k_1,k_2 和残差成为了 **operator()** 中的参数，而残差成为了自变量与因变量的差值 $y-f(x,k_1,k_2)$ ，我们要让差值最小化，目标函数就是 $(y-f(x,k_1,k_2))^2$。
- 简单地记忆，优化变量应该出现在 **operator()** 里，而 **“*函数参数*”** 应该出现在类成员中，这里函数参数加引号，求极值是函数参数是**常参数** a,b,c，求拟合时函数参数是**自/因变量** x,y 。

5.3 示例代码的使用说明

- 源码已经上传到 码云 ceres_tutorial 请直接下载或克隆。
- 编译前请先安装 ceres 和 glog。代码使用了 glog 作为日志输出，glog 是一款谷歌出品的日志工具，广泛用于 c++ 程序调试，使用简单，建议安装学习，参考 glog-github :

```
# 安装依赖
sudo apt-get install cmake libgoogle-glog-dev libgflags-dev libatlas-base-dev libeigen3-dev libsuitesparse-dev
# 安装 glog
git clone https://github.com/google/glog
sudo apt-get install autoconf automake libtool
cd glog
mkdir build && cd build
cmake ..
make
sudo make install
# 安装 ceres
git clone https://ceres-solver.googlesource.com/ceres-solver
cd ceres-solver
mkdir build && cd build
cmake ..
make
sudo make install
```

- 推荐的 IDE 为 Clion for linux clion-官网，版本不限，利用 clion 打开源码中的 cmakelists.txt 即可。不利用IDE直接编译源码，进入源码路径下打开命令行，依次执行即可，可利用文本编辑器来修改注释部分，以切换自动和手动求导。

```
git clone https://gitee.com/jqf64078/ceres_tutorial.git
cd ceres_tutorial
mkdir build && cd build
cmake ..
make
./Ceres_Tutorial
```