Analytic Derivatives

Consider the problem of fitting the following curve (Rat43) to data:

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

That is, given some data $\{x_i, y_i\},\ \forall i = 1, \ldots, n$, determine parameters b_1, b_2, b_3 and b_4 that best fit this data.

Which can be stated as the problem of finding the values of b_1, b_2, b_3 and b_4 are the ones that minimize the following objective function [1]:

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

To solve this problem using Ceres Solver, we need to define a `CostFunction` that computes the residual f for a given x and y and its derivatives with respect to b_1, b_2, b_3 and b_4.

Using elementary differential calculus, we can see that:

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

With these derivatives in hand, we can now implement the `CostFunction` as:

```
class Rat43Analytic : public SizedCostFunction<1,4> {
   public:
     Rat43Analytic(const double x, const double y) : x_(x), y_(y) {}
     virtual ~Rat43Analytic() {}
     virtual bool  Evaluate (double const* const* parameters,
                             double* residuals,
                             double** jacobians) const {
       const double b1 = parameters[0][0];
       const double b2 = parameters[0][1];
       const double b3 = parameters[0][2];
       const double b4 = parameters[0][3];

       residuals[0] = b1 *  pow(1 + exp(b2 -  b3 * x_), -1.0 / b4) - y_;

       if (!jacobians) return true;
       double* jacobian = jacobians[0];
       if (!jacobian) return true;

       jacobian[0] = pow(1 + exp(b2 - b3 * x_), -1.0 / b4);
       jacobian[1] = -b1 * exp(b2 - b3 * x_) *
                     pow(1 + exp(b2 - b3 * x_), -1.0 / b4 - 1) / b4;
       jacobian[2] = x_ * b1 * exp(b2 - b3 * x_) *
                     pow(1 + exp(b2 - b3 * x_), -1.0 / b4 - 1) / b4;
       jacobian[3] = b1 * log(1 + exp(b2 - b3 * x_)) *
                     pow(1 + exp(b2 - b3 * x_), -1.0 / b4) / (b4 * b4);
       return true;
     }

    private:
     const double x_;
     const double y_;
 };
```

This is tedious code, hard to read and with a lot of redundancy. So in practice we will cache some sub-expressions to improve its efficiency, which would give us something like:

```
class Rat43AnalyticOptimized : public SizedCostFunction<1,4> {
   public:
     Rat43AnalyticOptimized(const double x, const double y) : x_(x), y_(y) {}
     virtual ~Rat43AnalyticOptimized() {}
     virtual bool  Evaluate (double const* const* parameters,
                             double* residuals,
                             double** jacobians) const {
       const double b1 = parameters[0][0];
       const double b2 = parameters[0][1];
       const double b3 = parameters[0][2];
       const double b4 = parameters[0][3];

       const double t1 = exp(b2 -  b3 * x_);
       const double t2 = 1 + t1;
       const double t3 = pow(t2, -1.0 / b4);
       residuals[0] = b1 * t3 - y_;

       if (!jacobians) return true;
       double* jacobian = jacobians[0];
       if (!jacobian) return true;

       const double t4 = pow(t2, -1.0 / b4 - 1);
       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_;
     const double y_;
 };
```

What is the difference in performance of these two implementations?

CostFunction	Time (ns)
Rat43Analytic	255
Rat43AnalyticOptimized	92

`Rat43AnalyticOptimized` is 2.8 times faster than `Rat43Analytic`. This difference in run-time is not uncommon. To get the best performance out of analytically computed derivatives, one usually needs to optimize the code to account for common sub-expressions.

When should you use analytical derivatives?

1. The expressions are simple, e.g. mostly linear.
2. A computer algebra system like Maple , Mathematica, or SymPy can be used to symbolically differentiate the objective function and generate the C++ to **evaluate** them.

3. Performance is of utmost concern and there is algebraic structure in the terms that you can exploit to get better performance than automatic differentiation.

   That said, getting the best performance out of analytical derivatives requires a non-trivial amount of work. Before going down this path, it is useful to measure the amount of time being spent evaluating the Jacobian as a fraction of the total solve time and remember Amdahl's Law is your friend.
4. There is no other way to compute the derivatives, e.g. you wish to compute the derivative of the root of a polynomial:

$$a_3(x,y)z^3 + a_2(x,y)z^2 + a_1(x,y)z + a_0(x,y) = 0$$

   with respect to x and y. This requires the use of the Inverse Function Theorem
5. You love the chain rule and actually enjoy doing all the algebra by hand.

Footnotes

[1] The notion of best fit depends on the choice of the objective function used to measure the quality of fit, which in turn depends on the underlying noise process which generated the observations. Minimizing the sum of squared differences is the right thing to do when the noise is Gaussian. In that case the optimal value of the parameters is the Maximum Likelihood Estimate.