Mathematic methods, Curve fitting

RAIL Undergraduate Student

Jaewon Lee

Index

- Jacobian
- Least Square Method
- Analytic Derivatives
- Numeric Derivatives
- Gradient Descent Algorithm
- Newton's Method
- Gauss-Newton method
- Levenberg-Marquardt Method
- Curve fitting – Ceres Solver

Jacobian - Property

- 다변수 벡터 함수의 도함수 행렬

- n차원 벡터 $x \in \mathbb{R}^n$를 입력으로 받고 m차원 벡터$f(x) \in \mathbb{R}^m$를 출력으로 생성하는 벡터 함수

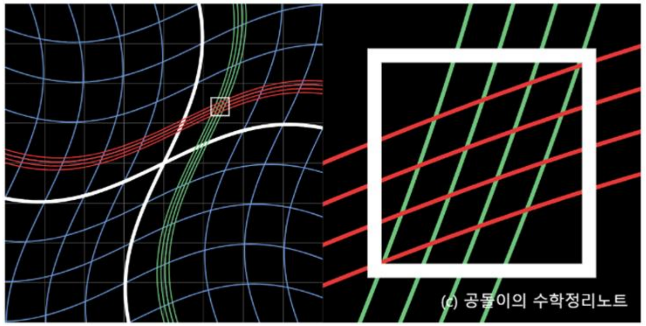
  $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$이 있다고 할 때, Jacobian은 다음과 같은 m x n행렬로 정의할 수 있다.

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

- 모두 1차 편미분 계수로 구성되어 있다.

- 자코비안에서 하고자 하는 것의 기하학적 의미는 미소 영역에서 비선형 변환을 선형 변환으로

  근사 시키는 것.



Jacobian - Example

- f: $\mathbb{R}^2 \rightarrow \mathbb{R}^2$인 함수 $f(x, y) = (xy, sinxy)$

- 모든 편도함수는 $\frac{\partial f_1}{\partial x} = y$, $\frac{\partial f_1}{\partial y} = \text{x}$, $\frac{\partial f_2}{\partial x} = ycosxy$, $\frac{\partial f_2}{\partial y} = xcosxy$

- 이 때, 자코비안 행렬은 다음과 같다.

$$J(\mathbf{f})(\mathbf{x}) = \begin{pmatrix} y & x \\ y\cos xy & x\cos xy \end{pmatrix} \qquad \forall x, y \in \mathbb{R}$$

- 자코비안 행렬식(Jacobian determinant)는 다음과 같다.

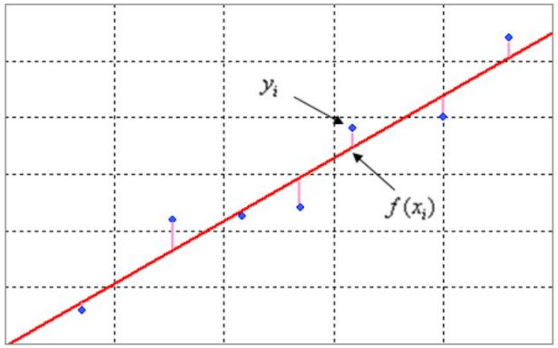
$$\det J(\mathbf{f})(\mathbf{x}) = 0 \qquad \forall x, y \in \mathbb{R}$$

- 결론, 행렬은 선형 변환을 나타내고

- 다변수 벡터 함수의 도함수 행렬인 Jacobian Matrix는 미소 영역내 비선형변환에서 선형 변환으로의 근사를 나타낸다.

Least Square Method

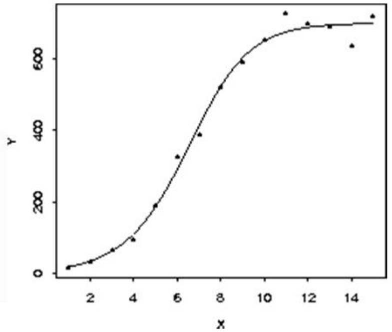
- 최소자승법 (최소제곱법)
- 실험적으로 데이터를 얻은 경우, 데이터의 경향을 알아내거나 데이터를 대변하기 위한 함수를 알아내기 위해 직선 혹은 곡선으로 근사해야한다.
- 이 때 사용되는 것이 least square method이다.
- $\sum_{i=1}^{n}(y_i - f(x_i))^2$ 식과 같이 데이터와 함수의 차를 제곱하여 더한 값이 최소가 되도록 해야한다.
- 실제 데이터와 직선 혹은 곡선의 차이를 residual이라고 표현한다.
- Residual 표현 : $\sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n}(y_i - f(x_i))^2$



Analytic Derivatives

- 해석적(분석적) 도함수 : 수식을 논리에 따라 전개하여 미분하는 것. (오차가 없는 미분 값 도출)
- 다음과 같은 Curve의 fitting문제를 고려해보자(Rat43).

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



- 주어진 데이터는 $\{x_i, y_i\}$이고, i는 정수다. 이 데이터를 가장 잘 대변할 수 있는 함수의 parameter b1,b2,b3,b4를 결정할 수 있다.
- 목적함수(E)를 최소화하는 것은 아래와 같으며, least square method를 아래의 식에서 확인할 수 있다.(residual)

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

- Ceres Solver를 이용하여 문제를 풀기 위해서는 하나의 residual을 계산할 수 있는 CostFunction을 정의 해야한다.

Analytic Derivatives

- For Jacobian,

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

- Rat43Analytic Class를 정의하고, 그 내부구조를 확인해보면, 다음과 같다.
- [Residual](), [Jacobian](), Parameter들을 확인할 수 있다.

```
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);
```

Analytic Derivatives

- Class Rat43AnalyticOptimized

- 공통으로 계산하는 부분을 변수로 지정

- 앞 선 페이지지의 Rat43Analytic보다 2배 이상 빠른 계산 속도를 보임

```
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_;
};
```

Analytic Derivatives – When we use?

- 표현식이 간단하거나, 선형에 가까울 때
- Automatic differentiation보다 더 나은 성능을 얻을 수 있는 대수적 구조가 있을 때
  - Jacobian을 계산할 때, 미리 계산한 공통 부분을 통해 성능을 높일 수 있을 때

Numeric Derivatives - Forward Differences

- 수치 도함수 : 수치적 미분은 해석적 미분 방식으로는 풀 수 없는 문제가 있을 때 수치적 접근을 통해 근사값을 찾는 방식
- 수치 미분의 근간은 차분의 근사를 통해 미분하는 것.

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

한 순간의 변화량, 미분

- 하지만 컴퓨터에서 수치적으로 limit연산을 수행할 수 없으므로 h에 작은 값을 선택하고 도함수를 구해야 한다.(Forward Differences)

$$Df(x) \approx \frac{f(x+h) - f(x)}{h}$$

- 이 도함수를 Taylor Series를 통해 전개하면 결국 O(h)라는 오차를 얻을 수 있다.

$$f(x+h) = f(x) + hDf(x) + \frac{h^2}{2!}D^2f(x) + \frac{h^3}{3!}D^3f(x) + \cdots$$

$$Df(x) = \frac{f(x+h) - f(x)}{h} - \left[\frac{h}{2!}D^2f(x) + \frac{h^2}{3!}D^3f(x) + \cdots\right]$$

$$Df(x) = \frac{f(x+h) - f(x)}{h} + O(h)$$

Numeric Derivatives – Central Differences

- 수치 미분에서는 x지점에서의 진정한 함수의 기울기를 구할 수 없고, 앞선 페이지에서의 식들은 모두 x+h와 x지점 사이의 기울기였고, 그것의 근사였다.
- 컴퓨터 상에서 h를 무한히 0으로 근사시키는 limit연산은 불가능해서 차분의 근사를 통한 식은 오차가 발생하는 한계를 지닌다.
- 이 오차를 개선하기 위해 x와 x+h의 차분이 아닌 x-h와 x+h사이의 차분을 계산하는 방법이 있고, 이를 중심차분법(Central Differences)이라 한다.

$$Df(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Numeric Derivatives – Central Differences

- 하지만 전방차분법 수식에 비해 중심차분법에서는 한 번의 계산이 더 필요하다.

$$Df(x) \approx \frac{f(x+h)-f(x)}{h} \qquad Df(x) \approx \frac{f(x+h)-f(x-h)}{2h}$$

- 그럴만한 가치가 있다면, 같은 방법으로 Taylor Series전개를 통해 오차를 살펴보면 O(h^2)으로 h 값에 대해 오차가 감소하는 것을 확인할 수 있다. (Taylor급수 또한 한 점 근처에서의 도함수 값을 무한 급수로 표현한 것이기에 이용)

$$\begin{aligned}
f(x+h) &= f(x) + hDf(x) + \frac{h^2}{2!}D^2f(x) + \frac{h^3}{3!}D^3f(x) + \frac{h^4}{4!}D^4f(x) + \cdots \\
f(x-h) &= f(x) - hDf(x) + \frac{h^2}{2!}D^2f(x) - \frac{h^3}{3!}D^3f(c_2) + \frac{h^4}{4!}D^4f(x) + \cdots \\
Df(x) &= \frac{f(x+h)-f(x-h)}{2h} + \frac{h^2}{3!}D^3f(x) + \frac{h^4}{5!}D^5f(x) + \cdots \\
Df(x) &= \frac{f(x+h)-f(x-h)}{2h} + O(h^2)
\end{aligned}$$

Numeric Derivatives – Ridders' Method

- 부동소수점으로 인한 오차를 겪는 아주 작은 값인 h를 요구하지 않고 함수의 미분 값을 추정하는 방법.
- O(h^2)보다 오차가 더 빨리 감소하도록 하는 방법을 찾는다.
- Richardson Extrapolation을 미분 문제에 적용함으로써 수행한다.
- 1. K라는 term을 통해 h와 독립되도록 함. 중심차분식을 다음과 같이 정의

$$\begin{aligned} Df(x) &= \frac{f(x+h)-f(x-h)}{2h} + \frac{h^2}{3!}D^3f(x) + \frac{h^4}{5!}D^5f(x) + \cdots \\ &= \frac{f(x+h)-f(x-h)}{2h} + K_2h^2 + K_4h^4 + \cdots \end{aligned}$$

$$A(1,m) = \frac{f(x+h/2^{m-1}) - f(x-h/2^{m-1})}{2h/2^{m-1}}.$$

- 2. 정의에 따라 아래의 두 식을 정리, 오차가 O(h^4)이 되는 것을 확인할 수 있다.

$$Df(x) = A(1,1) + K_2h^2 + K_4h^4 + \cdots$$

$$Df(x) = A(1,2) + K_2(h/2)^2 + K_4(h/2)^4 + \cdots$$

$$Df(x) = \frac{4A(1,2) - A(1,1)}{4-1} + O(h^4)$$

Numeric Derivatives – Ridders' Method

- A(n, m)에 대한 식으로 귀결하면, 다음과 같다.

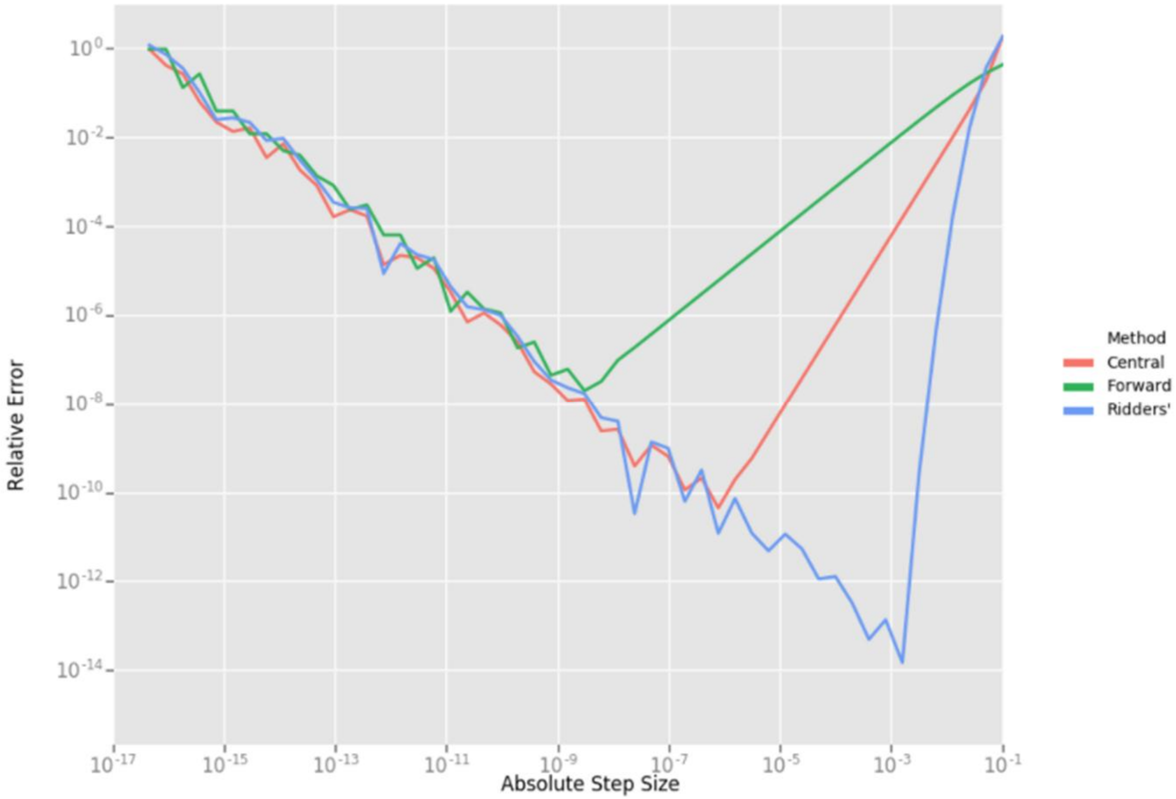
$$A(n,m)=\begin{cases}\dfrac{f(x+h/2^{m-1})-f(x-h/2^{m-1})}{2h/2^{m-1}} & n=1\\[2ex] \dfrac{4^{n-1}A(n-1,m+1)-A(n-1,m)}{4^{n-1}-1} & n>1\end{cases}$$

- A(n, 1)은 O(h^2n)의 오차를 가지게 된다.

$$\begin{array}{ccccc} A(1,1) & A(1,2) & A(1,3) & A(1,4) & \cdots \\ & A(2,1) & A(2,2) & A(2,3) & \cdots \\ & & A(3,1) & A(3,2) & \cdots \\ & & & A(4,1) & \cdots \\ & & & & \ddots \end{array}$$

- n이 증가할수록 계산은 한 번 혹은 몇 번의 시도내에 수행이 가능하다. (at a time.)
- 그러므로, 컴퓨터구조에 따른 부동소수점의 한계에 상관없이 h값이 매우 작지 않아도 오차가 훨씬 더 적은 값으로 근사가 가능하다.

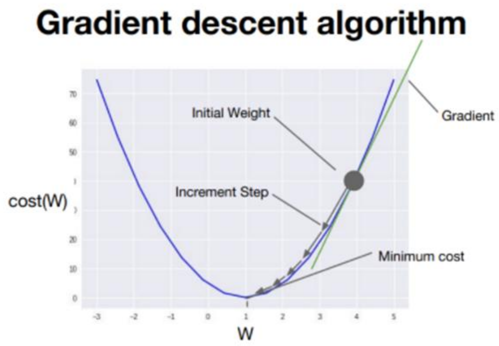
Numeric Derivatives



- 하지만 Ridders' Method가 높은 정확도를 위해 훨씬 더 많은 런타임을 소모한다.
- Rat43NumericDiffForward : 262ns < Rat43NumericDiffRidders : 3760ns

Gradient Descent Algorithm

- 경사하강법
- 목표로 하는 함수를 찾기 위해 Cost가 최소가 되게 하는 파라미터를 결정하는 최적화 알고리즘
- 반복적인 과정을 통해(iteration) 선형의 근삿값을 찾게 된다.



- 함수에서 local minimum을 찾기 위한 과정이 된다.
- Cost function을 파라미터(W)에 대해 편미분해주고 기울기가 감소하는 방향(그레디언트의 반대 방향)으로 W가 진행할 수 있도록, W에서 빼주는 과정을 거친다.

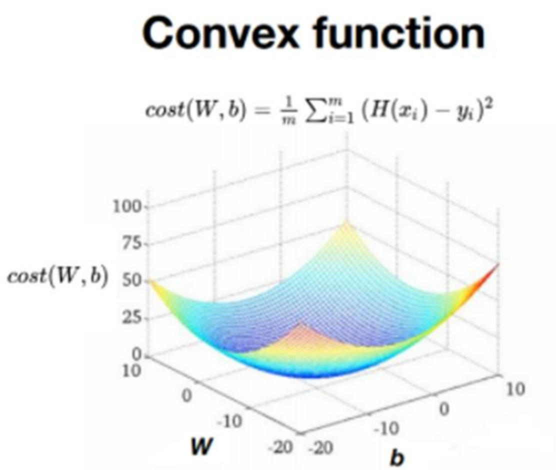
$$W := W - \alpha \frac{\partial}{\partial W} \frac{1}{2m} \sum_{i=1}^{m} (W(x_i) - y_i)^2$$

$$W := W - \alpha \frac{1}{2m} \sum_{i=1}^{m} 2(W(x_i) - y_i)x_i$$

$$W := W - \alpha \frac{1}{m} \sum_{i=1}^{m} (W(x_i) - y_i)x_i$$

Gradient Descent Algorithm

- Local minimum과 Global minimum이 일치하지 않는 복잡한 함수인 경우, 가장 낮은 지점으로 도달할 수 있을 것이라는 보장을 할 수 없다.
- 이러한 경우 GD는 사용할 수 없다.
- Cost function이 Convex function(볼록 함수)이라면 항상 같은 최저점에 도착한다는 것을 보장할 수 있기에 Gradient Descent를 적절히 사용할 수 있다.



Gradient Descent Algorithm

- More General,
- 다변수 함수 E의 gradient는

$$\nabla E = \left(\frac{\partial E}{\partial x_1}, \frac{\partial E}{\partial x_2}, \ldots, \frac{\partial E}{\partial x_n} \right)$$

- 함수의 최솟값을 찾을 수 있는 x1, x2, x3, …을 결정하는 과정 상기.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \lambda_k \nabla E(\mathbf{x}_k), \quad k \geq 0$$

- 이를 최소자승법(nonlinear)에 적용하여 Jacobian으로 표현할 수 있다. 함수 E를 residual로 표현하면, (p는 모델 파라미터)

$$E(\mathbf{p}) = \sum_{i=1}^{n} r_i(\mathbf{p})^2 = [\, r_1(\mathbf{p}) \cdots r_n(\mathbf{p}) \,] \begin{bmatrix} r_1(\mathbf{p}) \\ \vdots \\ r_n(\mathbf{p}) \end{bmatrix} = \mathbf{r}^T \mathbf{r}$$

- E(p)에 Gradient를 취해주면 다음과 같은 Jacobian 식을 얻게 된다.

$$\nabla E(\mathbf{p}) = 2 \begin{bmatrix} \frac{\partial r_1(\mathbf{p})}{\partial p_1} & \cdots & \frac{\partial r_1(\mathbf{p})}{\partial p_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial r_n(\mathbf{p})}{\partial p_1} & \cdots & \frac{\partial r_n(\mathbf{p})}{\partial p_m} \end{bmatrix}^T \begin{bmatrix} r_1(\mathbf{p}) \\ \vdots \\ r_n(\mathbf{p}) \end{bmatrix}$$

$$\mathbf{p}_{k+1} = \mathbf{p}_k - \lambda_k \nabla E(\mathbf{p}_k) = \mathbf{p}_k - 2\lambda_k J_{\mathbf{r}}^T(\mathbf{p}_k)\mathbf{r}(\mathbf{p}_k), \quad k \geq 0$$

Newton's method

- 뉴턴 기법(뉴턴 랩슨법)은 함수 E(x) = 0이 되는 해를 찾기 위해 임의의 초기값 x0로 부터 시작하여 아래 식에 따라 x_{k+1}을 반복적으로 갱신함으로써 해를 찾는 과정.

$$x_{k+1} = x_k - \frac{E(x_k)}{E'(x_k)}, \quad k \geq 0$$

- 미분이 함수에서 접선의 기울기임을 이용하여 E와 E'의 부호에 따라 x를 감소시킬지 증가시킬지를 결정하고 E'(기울기)의 크기에 따라 x를 얼마나 많이 증가 혹은 감소할지 결정하는 방식.
- E(x)가 최소 혹은 최대가 되는 값을 찾는 것은 차수를 한 개씩 증가시켜 식을 변형하여 적용한다.

$$x_{k+1} = x_k - \frac{E'(x_k)}{E''(x_k)}, \quad k \geq 0$$

Newton's method

- 뉴턴법을 또한 다변수 함수의 최적화 문제로 확장하면 다음과 같이 수정할 수 있다.

$$x_{k+1}=x_k-\frac{E'(x_k)}{E''(x_k)},\ \ k\geq 0 \qquad \mathbf{x}_{k+1}=\mathbf{x}_k-\gamma H_E(\mathbf{x}_k)^{-1}\nabla E(\mathbf{x}_k),\ \ k\geq 0$$

- E'은 gradient로 E''는 Hessian Matrix로 표현하여 값을 나타낸다.

$$\nabla E(\mathbf{x}_k)=\begin{bmatrix} \frac{\partial E(\mathbf{x}_k)}{\partial x_1} \\ \vdots \\ \frac{\partial E(\mathbf{x}_k)}{\partial x_n} \end{bmatrix},\ \ H_E(\mathbf{x}_k)=\begin{bmatrix} \frac{\partial^2 E(\mathbf{x}_k)}{\partial^2 x_1} & \cdots & \frac{\partial^2 E(\mathbf{x}_k)}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 E(\mathbf{x}_k)}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 E(\mathbf{x}_k)}{\partial^2 x_n} \end{bmatrix}$$

- 위 식을 이용하면 뉴턴법을 nonlinear least square 문제에 직접 적용도 가능하다.
- 하지만 E함수가 두 번 미분 가능해야하며 2차 미분까지 계산해야하는 부담이 존재한다.
- 그래서 비선형 최소 자승 문제의 경우 일차 미분만으로 계산이 가능한 가우스-뉴턴법이 사용된다.

Gauss-Newton method

- 뉴턴법을 연립방정식 형태로 확장시킨 형태, 비선형 최소 자승 문제에 대한 대표적인 최적화 방법 중 하나.
- 1차 미분만으로 목적 함수에 대한 해를 찾을 수 있다.
- 앞 서 살펴본 residual로 표현된 식을 다시 살펴보면

$$E(\mathbf{p})=\sum_{i=1}^{n} r_i(\mathbf{p})^2=\begin{bmatrix} r_1(\mathbf{p}) & \cdots & r_n(\mathbf{p}) \end{bmatrix}\begin{bmatrix} r_1(\mathbf{p}) \\ \vdots \\ r_n(\mathbf{p}) \end{bmatrix}=\mathbf{r}^T\mathbf{r}$$

- E(p)를 최소화시키기 위한, E'(p) = 0으로 만드는 가우스-뉴턴 해는 초기 모델 파라미터(p)에서 부터 p를 아래 수식에 의해 반복적으로 assign하며 얻어진다.

$$\mathbf{p}_{k+1}=\mathbf{p}_k-(J_\mathbf{r}^T J_\mathbf{r})^{-1}J_\mathbf{r}^T\mathbf{r}(\mathbf{p}_k),\ \ k\geq 0$$

Gauss-Newton method

$$\mathbf{p}_{k+1}=\mathbf{p}_k-(J_\mathbf{r}^TJ_\mathbf{r})^{-1}J_\mathbf{r}^T\mathbf{r}(\mathbf{p}_k),\ \ k\geq 0$$

- 변수 n개인 m개의 방정식 (m >= n)

$$\begin{aligned} &f_1(x_1,x_2,\ldots,x_n)=0 \\ &f_2(x_1,x_2,\ldots,x_n)=0 \\ &\cdots \\ &f_m(x_1,x_2,\ldots,x_n)=0 \end{aligned}$$

- 1. 임의의 초기 값 x를 정한 후 $\boldsymbol{r}(\boldsymbol{p}^{\boldsymbol{k}})$의 값이 0인지 확인
- 2. 0에 근사한 값이면 종료, 아닌 경우에
- 3. $\boldsymbol{r}(\boldsymbol{p}^{\boldsymbol{k}})$의 접선을 그리는데, 이에 대한 미분은 Jacobian으로 표현된다.
- 4. $\boldsymbol{p}^{k+1}=\boldsymbol{p}^{k}-\frac{\boldsymbol{r}(p^k)}{\mathcal{J}(p^k)}$ 와 같이 접선과 x축이 만나는 지점을 다음 x로 지정한다.
- 5. Jacobian을 분모로 계산하지 못하기 때문에(행렬로 나눗셈 x) 이를 변환하는 과정이 필요하다.
- Jacobian의 Pseudo Inverse사용

$$X^{k+1}=X^k-P \text{ ------------(6)}$$

$$J(X^k)P=F(X^k) \text{ ------------(7)}$$

$$J(x^i)p=F(x^i) \rightarrow p=(J(x^i)^TJ(x^i))^{-1}J(x^i)^TF(x^i)$$

$$\mathbf{p}_{k+1}=\mathbf{p}_k-(J_\mathbf{r}^TJ_\mathbf{r})^{-1}J_\mathbf{r}^T\mathbf{r}(\mathbf{p}_k),\ \ k\geq 0$$

Levenberg-Marquardt Method

- 앞 선 두 기법이 결합된 형태이다.(GD, Gauss-Newton)
- 해로부터 멀리 떨어져 있을 때에는 Gradient Descent 방식으로 동작하고, 해 근처에서는 가우스-뉴턴 방식으로 해를 찾는다.
- 가우스-뉴턴법보다 안정적으로 해를 구할 수 있고 비교적 빠르기에 비선형 최소 자승문제에 있어서는 대부분 이 방법이 사용된다.

$$\mathbf{p}_{k+1}=\mathbf{p}_k-(J_\mathbf{r}^T J_\mathbf{r})^{-1} J_\mathbf{r}^T \mathbf{r}(\mathbf{p}_k),\ \ k\geq 0$$

- 가우스-뉴턴법에서 이차미분인 Hessian의 의미를 가지는 $J_r^T J_r$에 대한 역행렬 계산을 요하는데, 역행렬이 존재하지 않는 특이 행렬이라면(비가역) 해가 발산할 수 있는 문제점을 가지고 있다.
- Levenberg는 항등행렬(I)의 상수배를 추가로 더해주며, 발산의 위험성을 낮춘 방법이다.

$$\mathbf{p}_{k+1}=\mathbf{p}_k-(J_\mathbf{r}^T J_\mathbf{r}+\mu_k I)^{-1} J_\mathbf{r}^T \mathbf{r}(\mathbf{p}_k),\ \ k\geq 0$$

Levenberg-Marquardt Method

- Levenberg-Marquardt 방법은 항등행렬 대신에 diag($J_r^T J_r$)을 더해주는 방식이다.
- diag(A)는 행렬의 대각 원소들만 제외하고 나머지 원소를 0으로 만드는 대각행렬
- 이유는 $J_r^T J_r$의 대각 원소들이 각 파라미터 성분에 대한 곡률을 나타냄.
- 즉, Levenberg-Marquardt 방법은 가우스-뉴턴법의 singular 문제를 피하면서도 곡률을 반영하여 효과적으로 해를 찾을 수 있는 방법이다.

$$\mathbf{p}_{k+1}=\mathbf{p}_k-(J_\mathbf{r}^T J_\mathbf{r}+\mu_k diag(J_\mathbf{r}^T J_\mathbf{r}))^{-1} J_\mathbf{r}^T \mathbf{r}(\mathbf{p}_k),\ \ k\geq 0$$

Curve fitting

- 최소자승법과 비선형 최소자승법의 본래 목표는 주어진 data에 대해 Curve를 fitting하는 것이다.
- 이제 주어진 함수를 주어진 데이터에 대해서 fitting하는 과정을 본 예제로 확인.
- 주어진 데이터 샘플은 $y = e^{0.3x+0.1}$ 곡선과 가우시안 노이즈($\sigma = 0.2$)를 더한 데이터들이다.
- 주어진 데이터를 통하여 역으로 $y = e^{mx+c}$를 fitting할 것이다.

- -Residual : $y = e^{mx+c}$

```
struct ExponentialResidual {
  ExponentialResidual(double x, double y) : x_(x), y_(y) {}

  template <typename T>
  bool operator()(const T* const m, const T* const c, T* residual) const {
    residual[0] = y_ - exp(m[0] * x_ + c[0]);
    return true;
  }

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

Curve fitting

- \- Cost Function (Auto Differential)

```
double m = 0.0;
double c = 0.0;

Problem problem;
for (int i = 0; i < kNumObservations; ++i) {
  problem.AddResidualBlock(
      new AutoDiffCostFunction<ExponentialResidual, 1, 1, 1>(
          new ExponentialResidual(data[2 * i], data[2 * i + 1])),
      NULL,
      &m,
      &c);
}
```

```
leejaewon@Ubuntu-18-04-6-LTS: ~/RAIL/ceres-solver/Example_curve/build
File Edit View Search Terminal Help
leejaewon@Ubuntu-18-04-6-LTS:~/RAIL/ceres-solver/Example_curve/build$ .
bash: .: filename argument required
.: usage: . filename [arguments]
leejaewon@Ubuntu-18-04-6-LTS:~/RAIL/ceres-solver/Example_curve/build$ ./curve_fitting
iter      cost      cost_change  |gradient|   |step|    tr_ratio  tr_radius  ls_iter  iter_time  total_time
   0  1.211734e+02    0.00e+00    3.61e+02   0.00e+00   0.00e+00  1.00e+04        0    1.83e-04    2.09e-04
   1  2.334822e+03   -2.21e+03    3.61e+02   7.52e-01  -1.87e+01  5.00e+03        1    3.70e-05    2.75e-04
   2  2.331438e+03   -2.21e+03    3.61e+02   7.51e-01  -1.86e+01  1.25e+03        1    6.91e-06    2.87e-04
   3  2.311313e+03   -2.19e+03    3.61e+02   7.48e-01  -1.85e+01  1.56e+02        1    5.01e-06    2.94e-04
   4  2.137268e+03   -2.02e+03    3.61e+02   7.22e-01  -1.70e+01  9.77e+00        1    5.01e-06    3.02e-04
   5  8.553131e+02   -7.34e+02    3.61e+02   5.78e-01  -6.32e+00  3.05e-01        1    4.05e-06    3.09e-04
   6  3.306595e+01    8.81e+01    4.10e+02   3.18e-01   1.37e+00  9.16e-01        1    1.84e-04    4.97e-04
   7  6.426770e+00    2.66e+01    1.81e+02   1.29e-01   1.10e+00  2.75e+00        1    1.81e-04    6.81e-04
   8  3.344546e+00    3.08e+00    5.51e+01   3.05e-02   1.03e+00  8.24e+00        1    1.79e-04    8.63e-04
   9  1.987485e+00    1.36e+00    2.33e+01   8.87e-02   9.94e-01  2.47e+01        1    1.82e-04    1.05e-03
  10  1.211585e+00    7.76e-01    8.22e+00   1.05e-01   9.89e-01  7.42e+01        1    1.80e-04    1.23e-03
  11  1.063265e+00    1.48e-01    1.44e+00   6.06e-02   9.97e-01  2.22e+02        1    1.78e-04    1.41e-03
  12  1.056795e+00    6.47e-03    1.18e-01   1.47e-02   1.00e+00  6.67e+02        1    1.72e-04    1.59e-03
  13  1.056751e+00    4.39e-05    3.79e-03   1.28e-03   1.00e+00  2.00e+03        1    1.76e-04    1.77e-03
Ceres Solver Report: Iterations: 14, Initial cost: 1.211734e+02, Final cost: 1.056751e+00, Termination: CONVERGENCE
Initial m: 0 c: 0
Final   m: 0.291861 c: 0.131439
leejaewon@Ubuntu-18-04-6-LTS:~/RAIL/ceres-solver/Example_curve/build$
```

- 구성한 residual에 사전에 정의된 데이터들을 넣어주며, 연산
- 14번의 반복을 통해 m = 0.29.., C = 0.13에 도달
- (kNumObservations 변수는 데이터의 개수)

Curve fitting : RIDDERS' Method

- \- Cost Function (NumericDiffCostFunction : RIDDERS)

```cpp
Problem problem;
for (int i = 0; i < kNumObservations; ++i) {
  problem.AddResidualBlock(
      new NumericDiffCostFunction<ExponentialResidual, RIDDERS, 1, 1, 4>
        (new ExponentialResidual(data[2 * i], data[2 * i + 1])),
      NULL,
      &m,
      &c);
}
```

- 구성한 residual에 사전에 정의된 데이터들을 넣어주며, 연산
- 14번의 반복을 통해 m = 0.29.., C = 0.13에 도달, 런타임 증가

```
leejaewon@Ubuntu-18-04-6-LTS: ~/RAIL/ceres-solver/Example_curve_ridder/build
File  Edit  View  Search  Terminal  Help
leejaewon@Ubuntu-18-04-6-LTS:~/RAIL/ceres-solver/Example_curve_ridder$ cd build
leejaewon@Ubuntu-18-04-6-LTS:~/RAIL/ceres-solver/Example_curve_ridder/build$ ./curve_fitting
iter      cost      cost_change  |gradient|   |step|    tr_ratio  tr_radius  ls_iter  iter_time  total_time
   0  1.211734e+02    0.00e+00    3.61e+02   0.00e+00   0.00e+00  1.00e+04        0    3.81e-03    3.83e-03
   1  2.334822e+03   -2.21e+03    3.61e+02   7.52e-01  -1.87e+01  5.00e+03        1    1.91e-05    3.88e-03
   2  2.331438e+03   -2.21e+03    3.61e+02   7.51e-01  -1.86e+01  1.25e+03        1    5.96e-06    3.89e-03
   3  2.311313e+03   -2.19e+03    3.61e+02   7.48e-01  -1.85e+01  1.56e+02        1    5.96e-06    3.90e-03
   4  2.137268e+03   -2.02e+03    3.61e+02   7.22e-01  -1.70e+01  9.77e+00        1    5.01e-06    3.91e-03
   5  8.553131e+02   -7.34e+02    3.61e+02   5.78e-01  -6.32e+00  3.05e-01        1    5.96e-06    3.92e-03
   6  3.306595e+01    8.81e+01    4.10e+02   3.18e-01   1.37e+00  9.16e-01        1    3.77e-03    7.69e-03
   7  6.426770e+00    2.66e+01    1.81e+02   1.29e-01   1.10e+00  2.75e+00        1    3.79e-03    1.15e-02
   8  3.344546e+00    3.08e+00    5.51e+01   3.05e-02   1.03e+00  8.24e+00        1    3.81e-03    1.53e-02
   9  1.987485e+00    1.36e+00    2.33e+01   8.87e-02   9.94e-01  2.47e+01        1    3.82e-03    1.91e-02
  10  1.211585e+00    7.76e-01    8.22e+00   1.05e-01   9.89e-01  7.42e+01        1    3.79e-03    2.29e-02
  11  1.063265e+00    1.48e-01    1.44e+00   6.06e-02   9.97e-01  2.22e+02        1    3.77e-03    2.67e-02
  12  1.056795e+00    6.47e-03    1.18e-01   1.47e-02   1.00e+00  6.67e+02        1    3.82e-03    3.05e-02
  13  1.056751e+00    4.39e-05    3.79e-03   1.28e-03   1.00e+00  2.00e+03        1    3.82e-03    3.44e-02
Ceres Solver Report: Iterations: 14, Initial cost: 1.211734e+02, Final cost: 1.056751e+00, Termination: CONVERGENCE
Initial m: 0 c: 0
Final   m: 0.291861 c: 0.131439
```