Contents

1 Least Squares Problem

The Least Squares Problem consists of find solution that minimizes the sum of squared residuals. The following two representations are mathematically equivalent:

$$f(x) = \sum_i r_i(x)^2$$

$$f(x) = \|F(x)\|_2^2, F(x) = [r_1(x), r_2(x), ..., r_n(x)]^T$$

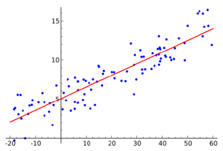
So our aim is to find:

$$x* = \text{argmin}_x\, f(x) = \text{argmin}_x\, \|F(x)\|_2^2$$

1.1 Linear Least Squares

Linear least squares is the least squares approximation of linear functions to data.

- Linear function: $y = m \cdot x + t$
  - Solve for m, t
- $r_i(m,t) = y_i - (m \cdot x_i + t)$



Linear least squares problems has only one global minimum and thus has analytical solution.

- $r_i(m,t) = y_i - (m \cdot x_i + t)$
- $x_i = \begin{bmatrix}1\\2\\3\\4\end{bmatrix}, y_i = \begin{bmatrix}6\\5\\7\\10\end{bmatrix}$
- $A = \begin{bmatrix}1 & 1\\2 & 1\\3 & 1\\4 & 1\end{bmatrix}, b = \begin{bmatrix}6\\5\\7\\10\end{bmatrix}$
- $Ax = b$ (over determined)
- Solve via normal equation $A^TAx = A^Tb$

- $A^TA = \begin{bmatrix}30 & 10\\28 & 4\end{bmatrix}$ $A^Tb = \begin{bmatrix}77\\28\end{bmatrix}$
- Solve: $\begin{bmatrix}30 & 10\\28 & 4\end{bmatrix}\begin{bmatrix}m\\t\end{bmatrix} = \begin{bmatrix}77\\28\end{bmatrix}$

-> $\begin{bmatrix}m\\t\end{bmatrix} = \begin{bmatrix}3.5\\1.4\end{bmatrix}$

Observe that here we take another form of the normal equation:

$$A^TAx = A^Tb$$

Instead of:

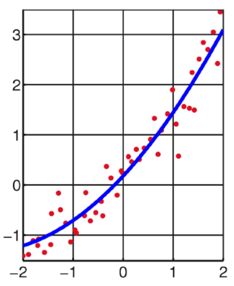
$$x = (A^TA)^{-1}A^Tb$$

The reason is the computation of the inverse of a matrix is in general not stable so we will try to solve the following equation using other techniques instead of computing the inverse:

$$A'x = b', \text{ where }, A' = A^TA \text{ and } b' = A^Tb$$

Any least squares problem that are linear in coefficients are considered as LLS and thus solvable (can achieve a reasonable good result) via normal equation.

- Quadratic function: $y = ax^2 + bx + c$
  - Solve for a, b, c
  - Linear with respect to a, b, c
- $r_i(a,b,c) = y_i - (ax^2 + bx + c)$
- $A = \begin{bmatrix}1 & 1 & 1\\4 & 2 & 1\\9 & 3 & 1\\16 & 4 & 1\end{bmatrix}, b = \begin{bmatrix}6\\5\\7\\10\end{bmatrix}$
- $Ax = b$ (over determined)
- Solve via normal equation $A^TAx = A^Tb$



1.1.1 Linear Least Squares: Solvers

In general is not a good idea to solve normal equation $A^T Ax = A^T b$ via:

- **Computing the Matrix Inverse**: inverting a matrix is very expensive and computing the matrix inverse for a large matrix is in general numerically unstable.
- **Gradient Descent**: we can solve it via GD but it would very inefficient.

There are two type of solvers: **Iterative** and **Direct** solvers.

Regarding iterative methods we have:

- **Jacobi iteration**: the idea behind the Jacobi iteration is that, if all variables are known but one, then its value is easy to find. Jacobi iteration consists of following steps:
  1. Guess initial values.
  2. Repeat until convergence
     - Compute the value of one variable assuming all others are known.
     - Repeat for all variables.

  This method has several advantages such as:

  - Simple to implement.
  - Low memory consumption.
  - Takes advantage of sparse structure.
  - Easy to parallelize.

  But it also has several cons such as:

  - Guaranteed convergence only for strictly diagonally dominant matrices.
  - Converges slowly.
- **Gauss-Seidel Iteration**. The idea behind Gauss-Seidel is very similar to Jacobi, the main difference between them are the way how do they split the coefficient matrix and how do they perform the update step.

  In each iteration of the Jacobi method every value is updated based on values obtained from the previous iteration. This makes it highly parallelizeble as there are no dependency between variables so we can update them all at the same time. In Gauss-Seidel, however, the update is performed sequentially. The first variable is updated based on the rest of variables, the second variable is updated based on the updated first variable and the rest of the variables, and so on. **So, Gauss-Seidel can not be parallelized but it converges faster than Jacobi.**
- **Conjugate Gradient Descent**. Conjugate gradient is a gradient approach to solve the linear system. **It's not performed on the raw problem directly but it first transforms the linear system into a quadratic system**. The raw problem of find a x such that $Ax = b$ is transformed to find the root of the derivative of $\frac{1}{2}x^t Ax - b^t x$. These two problems are equivalent because the derivative of the above function is exactly $Ax - b$.

  The CG is an improved version of the Steepest Descent by solving the Steepest Descent's zigzag problem. The idea behind CG is that we choose the new search directions that are conjugate to the previous search directions. In order to apply CG A must be positive-definite because to check whether two search directions p, q are conjugate $pAq = 0$ must be hold.

Regarding direct methods they are usually matrix factorization based. Direct methods factors the coefficient matrix A using "simple" matrices such as:

- **Diagonal/triangular matrices**. A linear system related to these type of matrices are very easy to solve. And in particular the inverse of a diagonal matrix is trivial to compute.
- **Orthogonal matrices**. These matrices are very easy to invert as the inverse it's just the transpose.

Once we have factorized the coefficient matrix then the linear system can be solved easily in the following way:

1. Decompose the coefficient matrix $A \Rightarrow Ax = A_1 A_2 A_3 ... A_k x = b$.
2. Solve $Ax = b$ using k easy systems:
$$A_1 x_1 = b \Rightarrow A_2 x_2 = x_1 \Rightarrow ... \Rightarrow A_k x = x_{k-1}$$
where $x_i = A_{i+1} A_{i+2} ... A_k x$. As each A_i is a "simple" matrix each of these $A_i x_i = x_{i-1}$ are easy to solve.

Some of the direct methods are:

- **QR-Decomposition**. QR-Decomposition decomposes the matrix A into QR where Q is an orthogonal matrix and R is an upper triangular matrix. This decomposition exists for any matrix and x can be solved easily by solving $Rx = Q^T b$.
- **LU-Decomposition**. LU-Decomposition decomposes the matrix A into LU where L is a lower triangular matrix and U is an upper triangular matrix. This decomposition only exists for non-singular square matrix and x can be solved following $Lx_1 = b \Rightarrow Ux = x_1$
- **Cholesky factorization**. Cholesky factorization decomposes the matrix A into LL^T where L is a lower triangular matrix. This decomposition only exists for square symmetric positive definite matrices and x can be solved following $Lx_1 = b \Rightarrow L^T x = x_1$.
- **SVD**. SVD decomposes the matrix A into $U \Sigma V^T$ where U, V are orthogonal matrices and Σ is a diagonal matrix. This decomposition exists for all matrices and x can be solved as $x = (V^T \Sigma^{-1} U)b$.

1.2 Non-Linear Least Squares

Non-Linear least square problems do not have a analytical solution and hence can only be solved using GD methods.

1.2.1 First Order Methods

The first order method like GD just uses the first order derivative of the objective function to perform the update step.

1.2.2 Second Order Methods

Second order methods uses also the second order derivative to compute each update step.

1.2.2.1 Newton's Method

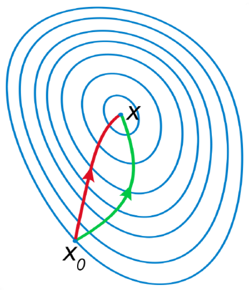
Newton's method makes a local quadratic approximation of the function at the current point using the 2nd order Taylor approximation and then jumps to the minimum of that approximation.

In this way it should converge faster than the simple GD because the learning rate $\alpha = H_f(x)^{-1}$ is not fixed and auto-adaptive. ("should" because Newton's method does not work for all kind of problems, at least not for optimizing complex DL model in a stochastic manner).

Newton's Method (2nd order):

– $x_{k+1} = x_k - H_f(x_k)^{-1} \nabla f(x_k)$

- In 1D
  – Root finding: $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$
  – Optimization (find root of derivative)
  $$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$



Newton (red) uses curvature information, and takes a more direct path than GD (green)

- Jacobian: $$\mathrm{J}_F(x) = \underbrace{\begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial x_1} & \cdots & \frac{\partial F_m}{\partial x_n} \end{bmatrix}}_{\#\text{variables}} \Big\} \#\text{residuals}$$ btw. $\nabla \mathrm{f}(\mathrm{x}) = 2 \cdot (\mathrm{J}_F(x))^T F(x)$

- Hessian: $$\mathrm{H}_f(x) = \underbrace{\begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}}_{\#\text{variables}} \Big\} \#\text{variables}$$ $\mathrm{H}_f(\mathrm{x}) = \mathrm{J}_{\nabla f}(x)^{\mathrm{T}}$

1.2.2.2 Gauss-Newton Method

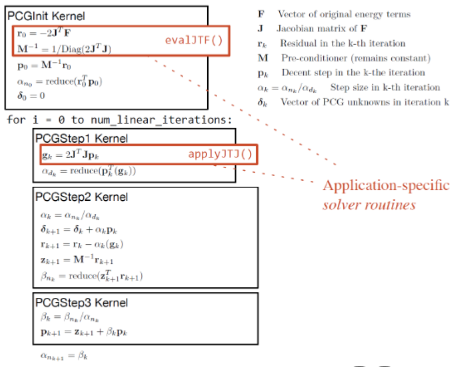
Second order derivatives are often hard to obtain. Instead, we can approximate the Hessian matrix using the Jacobian, this is the purpose of the **Gauss-Newton method**.

- $x_{k+1} = x_k - H_f(x_k)^{-1} \nabla f(x_k)$
  – 'true' 2nd derivatives are often hard to obtain (e.g., numerics)
  – $H_f \approx 2 J_F^T J_F$
- Gauss-Newton (GN):
  $$x_{k+1} = x_k - [2 J_F(x_k)^T J_F(x_k)]^{-1} \nabla f(x_k)$$

- Solve linear system (again, inverting a matrix is unstable):
  $$2(J_F(x_k)^T J_F(x_k)) \underbrace{(x_k - x_{k+1})}_{\text{Solve for delta vector}} = \nabla f(x_k)$$

- $$\underbrace{2(J_F(x_k)^T J_F(x_k))}_{\boldsymbol{A}} \cdot \underbrace{(x_k - x_{k+1})}_{\boldsymbol{X}} = \underbrace{\nabla f(x_k)}_{\boldsymbol{b}}$$
- Solve $Ax = b$
  – Could do matrix-free: applyJTJ, evalJTF

- Solve $Ax = b$
- Common in our research:
  - Use Pre-conditioned Conjugate Gradient Descent (PCG)
  - Easy to parallelize; e.g., on GPUs


PCGInit Kernel
$\mathbf{r}_0 = -2\mathbf{J}^T\mathbf{F}$
$\mathbf{M}^{-1} = 1/\mathrm{Diag}(2\mathbf{J}^T\mathbf{J})$
evalJTF()
$\mathbf{p}_0 = \mathbf{M}^{-1}\mathbf{r}_0$
$\alpha_{n_0} = \mathrm{reduce}(\mathbf{r}_0^T\mathbf{p}_0)$
$\boldsymbol{\delta}_0 = 0$
for i = 0 to num_linear_iterations:
PCGStep1 Kernel
$\mathbf{g}_k = 2\mathbf{J}^T\mathbf{J}\mathbf{p}_k$
applyJTJ()
$\alpha_{d_k} = \mathrm{reduce}(\mathbf{p}_k^T(\mathbf{g}_k))$
PCGStep2 Kernel
$\alpha_k = \alpha_{n_k}/\alpha_{d_k}$
$\boldsymbol{\delta}_{k+1} = \boldsymbol{\delta}_k + \alpha_k\mathbf{p}_k$
$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k(\mathbf{g}_k)$
$\mathbf{z}_{k+1} = \mathbf{M}^{-1}\mathbf{r}_{k+1}$
$\beta_{n_k} = \mathrm{reduce}(\mathbf{z}_{k+1}^T\mathbf{r}_{k+1})$
PCGStep3 Kernel
$\beta_k = \beta_{n_k}/\alpha_{n_k}$
$\mathbf{p}_{k+1} = \mathbf{z}_{k+1} + \beta_k\mathbf{p}_k$
$\alpha_{n_{k+1}} = \beta_k$
$\mathbf{F}$ Vector of original energy terms
$\mathbf{J}$ Jacobian matrix of $\mathbf{F}$
$\mathbf{r}_k$ Residual in the k-th iteration
$\mathbf{M}$ Pre-conditioner (remains constant)
$\mathbf{p}_k$ Decent step in the k-the iteration
$\alpha_k = \alpha_{n_k}/\alpha_{d_k}$ Step size in k-th iteration
$\boldsymbol{\delta}_k$ Vector of PCG unknowns in iteration k
Application-specific *solver routines*

1.2.2.3 Levenberg Method

Gauss-Newton method highly depends on the initialization as it converges faster when the parameters are close to their optimal value and slower when they are far. So the **Levenberg** method is just a "damped" version of the Gauss-Newton method which acts like Gradient Descent method when the parameters are far from their optimal and acts more like the Gauss-Newton method when the parameters are close to their optimal value.

The level of "interpolation" between Gauss-Newton and Gradient Descent is controlled by the damping factor λ. If λ is small then it acts more like Gauss-Newton and if λ is large then it acts more like Gradient Descent.

The damping factor λ is adjusted in each iteration ensuring that the cost function is decreasing:

$$f(x_k) > f(x_{k+1})$$

and if not fulfilled then we increase λ.

- Levenberg
  - "damped" version of Gauss-Newton:

$$(2J_F(x_k)^T J_F(x_k) + \lambda \cdot I) \cdot (x_k - x_{k+1}) = \nabla f(x_k)$$

Tikhonov regularization

  - The damping factor λ is adjusted in each iteration ensuring:

$$f(x_k) > f(x_{k+1})$$

    - if not fulfilled increase λ

    $\rightarrow$Trust region

$\rightarrow$"Interpolation" between Gauss-Newton (small λ) and Gradient Descent (large λ)

1.2.2.4 Levenberg-Marquat Method

The **Levenberg-Marquat** method is an extension of the Levenberg method. Instead of a plain Gradient Descent for large λ, it scale each component of the gradient according to the curvature. In this way it avoids slow convergence in components with a small gradient.

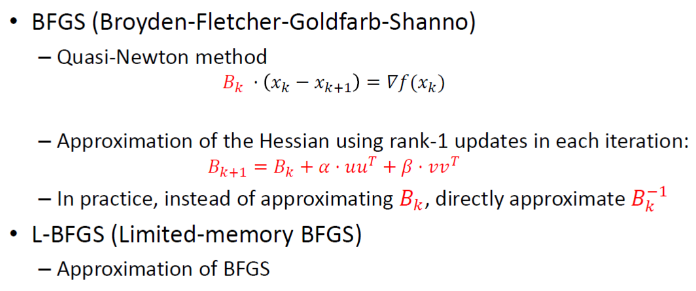
- Levenberg-Marquardt (LM)
  - Extension of Levenberg:

$$(2J_F(x_k)^T J_F(x_k) + \lambda \cdot diag(J_F(x_k)^T J_F(x_k))) \cdot (x_k - x_{k+1}) = \nabla f(x_k)$$

  - Idea: Instead of a plain Gradient Descent for large λ, scale each component of the gradient according to the curvature.
    - Avoids slow convergence in components with a small gradient

1.2.2.5 BFGS and L-BFGS methods

BFGS and L-BFGS are Quasi-Newton methods. It approximates the Hessian using rank-1 updates in each iteration. And in practice, instead of approximating H we directly approximates H^{-1}. L-BFGS is just an approximation of BFGS with limited-memory.



- BFGS (Broyden-Fletcher-Goldfarb-Shanno)
  - Quasi-Newton method
    $$B_k \cdot (x_k - x_{k+1}) = \nabla f(x_k)$$
  - Approximation of the Hessian using rank-1 updates in each iteration:
    $$B_{k+1} = B_k + \alpha \cdot uu^T + \beta \cdot vv^T$$
  - In practice, instead of approximating B_k, directly approximate B_k^{-1}
- L-BFGS (Limited-memory BFGS)
  - Approximation of BFGS

1.3 Handling outliers

L2 norm is not robust to outliers. There are several options that we can use to handle outliers:

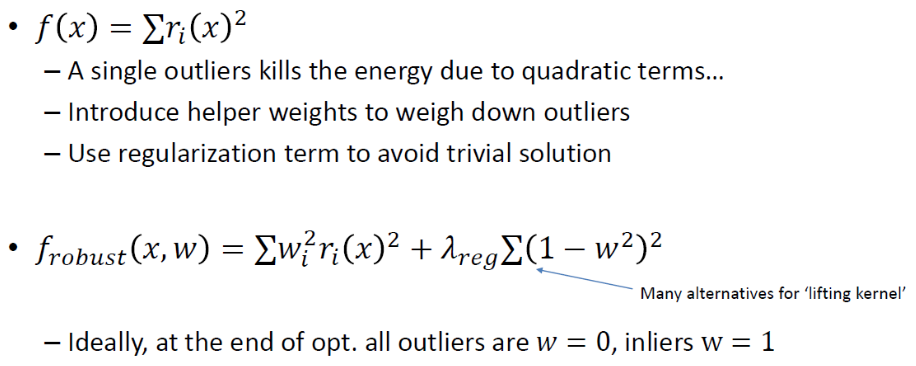
- **RANSAC**: RANSAC is a resampling technique that generates candidate solution by using the minimum number of observations required to estimate the underlying model parameters. It's essentially a trial-and-error based approach and the basic algorithm is summarized as follows:

  1. Select randomly the mimum number of points required to determine the model parameters.
  2. Solve for the parameters of the model.
  3. Determine how many points from the set of all points fit with a predefined tolerance ϵ.
  4. If the fraction of the number of inliers over the total number points in the set exceeds a predefined threshold, then re-estimate the model parameters using all the identified inliers and terminate.
  5. Otherwise, repeat steps 1 thorough 4 for a maximum of N times.

  N is chosen high enough to ensure that at least one of the sets of random samples does not include any outlier.

- **Lifting Schemes**: Lifting Schemes is a technique which introduces helper weights to weigh down outliers. The cost function is reformulated in a way that, at the end of the optimization all outliers will have weight 0 and inliers will have weight 1. Also, a regularization term is introduced in order to avoid trivial solution such as assigning 0 to all the terms to make the final cost 0.
  $$f_{robust}(x, w) = \sum_i w_i^2 r_i(x)^2 + \lambda_{reg} \sum_i (1 - w_i^2)^2$$

  There should not be a linear growth relation between the lifting kernel and the cost function so the optimizer can find a way to improve it.

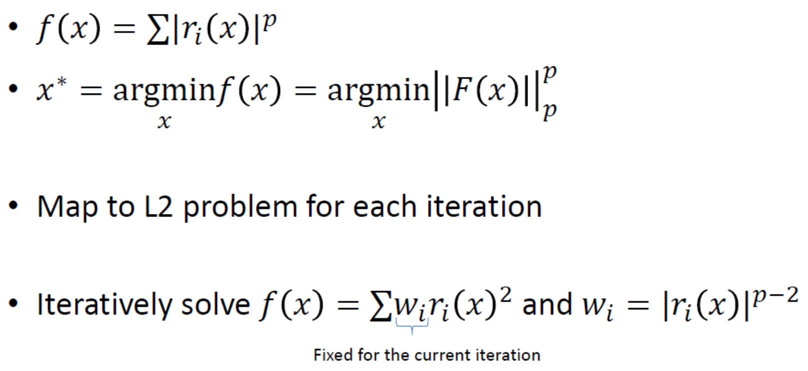


- $f(x) = \sum r_i(x)^2$
  - A single outliers kills the energy due to quadratic terms…
  - Introduce helper weights to weigh down outliers
  - Use regularization term to avoid trivial solution
- $f_{robust}(x, w) = \sum w_i^2 r_i(x)^2 + \lambda_{reg} \sum (1 - w^2)^2$

  Many alternatives for 'lifting kernel'
  - Ideally, at the end of opt. all outliers are $w = 0$, inliers $w = 1$

- **Robust norms**: such as **L1-norm**, **Huber Norm** or **p-norms**. If we use **p-norm** and we'd like to use methods like Gauss-Newton to solve it, then we can make use of the **IRLS** method. The key idea behind IRLS is to map the least p-norm problem to a weighted L2-norm problem for each iteration. In this way, we can treat it like a standard weighted least square problem and use 2nd order solvers to solve it.

  The weight of each residual is set to be $\|r_i(x)\|^{p-2}$ and it changes in every iteration as x is getting updated, that's why it's called reweighted least square problem.



- $f(x) = \sum |r_i(x)|^p$
- $x^* = \underset{x}{\operatorname{argmin}} f(x) = \underset{x}{\operatorname{argmin}} \left|\left|F(x)\right|\right|_p^p$
- Map to L2 problem for each iteration
- Iteratively solve $f(x) = \sum \underbrace{w_i}_{\text{Fixed for the current iteration}} r_i(x)^2$ and $w_i = |r_i(x)|^{p-2}$

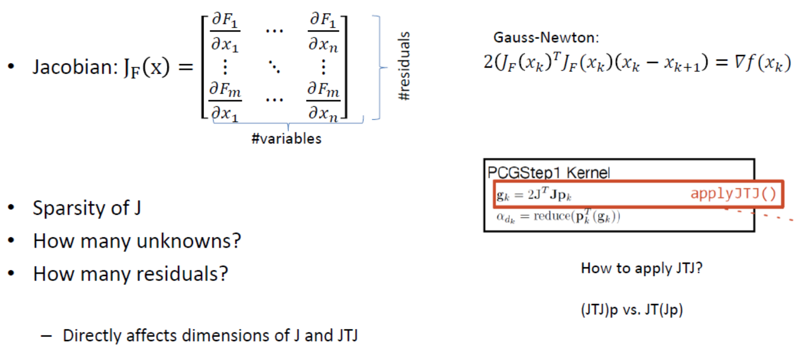
1.4 Convexification

In general the coarse-to-fine strategy is applied in order to make the energy landscape smoother and convex and thus make the optimization faster and achieve a better result. For example, for the RGB-alignment the coarse-to-fine strategy would work in the following way:

1. We blur the original image and rescale it to different sizes.
2. We then perform optimization on the coarest level, and use the estimated parameters as the initialization for the next level. This process is repeated until we get to the finnest levels.

- Convexification
- Make energy landscape smoother and convex!
  - Smoothing
  - Coarse-to-fine strategies over unknowns
  - Coarse-to-fine strategies over residuals

1.5 Performance/Efficiency Considerations



- Jacobian: $J_F(x) = \begin{array}{cc} \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial x_1} & \cdots & \frac{\partial F_m}{\partial x_n} \end{bmatrix} & \text{#residuals} \\ \text{#variables} & \end{array}$
- Sparsity of J
- How many unknowns?
- How many residuals?
  - Directly affects dimensions of J and JTJ

Gauss-Newton:

$2(J_F(x_k)^T J_F(x_k)(x_k - x_{k+1}) = \nabla f(x_k)$

PCGStep1 Kernel

$\mathbf{g}_k = 2\mathrm{J}^T\mathbf{J}\mathbf{p}_k$ applyJTJ()

$\alpha_{d_k} = \mathrm{reduce}(\mathbf{p}_k^T(\mathbf{g}_k))$

How to apply JTJ?

(JTJ)p vs. JT(Jp)

1.6 Computing Derivatives

1.6.1 Numeric Derivatives

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

- Forward Differences
  $\frac{df(x)}{dx} \approx \frac{f(x+h)-f(x)}{h}$
- Central Differences
  $\frac{df(x)}{dx} \approx \frac{f(x+h)-f(x-h)}{2h}$

- Easy to implement -> good for debugging
- Slow and numerically unstable

1.6.2 Automatic Differentiation

- $f(x) = x^2$
- Choose infinitesimal unit e, such that $e \neq 0$ but $e^2 = 0$
  – Dual number (similar to complex numbers)
- $f(10+e) = (10+e)^2 =$
  $100 + 2 \cdot 10 \cdot e + e^2 =$
  $100 + 20 \cdot e$

Try it out!

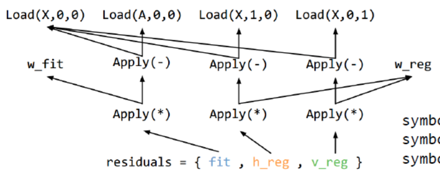
This is zero

This is $\frac{df}{dx}$

```
template <typename T, int N>
struct Jet {
        ...
        template<typename T, int N> inline
                Jet<T, N> operator*(const Jet<T, N>& f, const Jet<T, N>& g) {
                Jet<T, N> h;
                h.a = f.a * g.a;
                h.v = f.a * g.v + f.v * g.a;
                return h;
        }
        T a;  // The scalar part.
        Eigen::Matrix<T, N, 1> v;  // The infinitesimal part.
};
```

1.6.3 Symbolic Differentiation

- For instance, D* [Guenter 07]
- Analyze compute graph at compile time!
  - Can simplify / fuse terms efficiently
  - Optimal solution is NP-Complete (but many heuristics)



Could of course also do on whiteboard ☺

```
symbolic_derivative(h_reg, X(1,0)) --> -w_reg
symbolic_derivative(h_reg, X(0,0)) -->  w_reg
symbolic_derivative(v_reg, X(0,1)) --> -w_reg
```

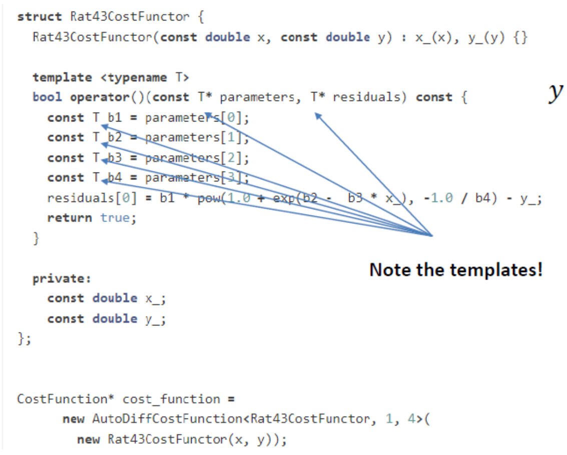
1.7 Non-linear Solvers

- **Ceres**
  - Uses Eigen as backend for linear solves (has also its own PCG)
  - Automatic differentiation using dual numbers ("jet.h")

- Alglib
  - Numerical differentiation or hand-provided

- Symbolic solvers
  - Maple
    - Good for derivations
    - Not so great simplification / code conversion

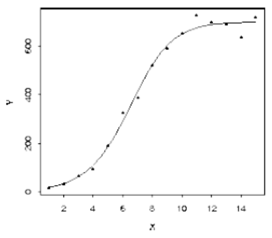
1.7.1 Ceres



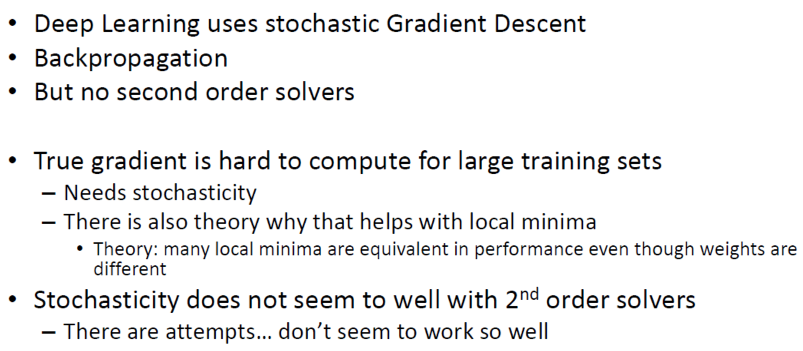
```
struct Rat43CostFunctor {
  Rat43CostFunctor(const double x, const double y) : x_(x), y_(y) {}

  template <typename T>
  bool operator()(const T* parameters, T* residuals) const {
    const T b1 = parameters[0];
    const T b2 = parameters[1];
    const T b3 = parameters[2];
    const T b4 = parameters[3];
    residuals[0] = b1 * pow(1.0 + exp(b2 -  b3 * x_), -1.0 / b4) - y_;
    return true;
  }

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


CostFunction* cost_function =
      new AutoDiffCostFunction<Rat43CostFunctor, 1, 4>(
        new Rat43CostFunctor(x, y));
```

Note the templates!

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

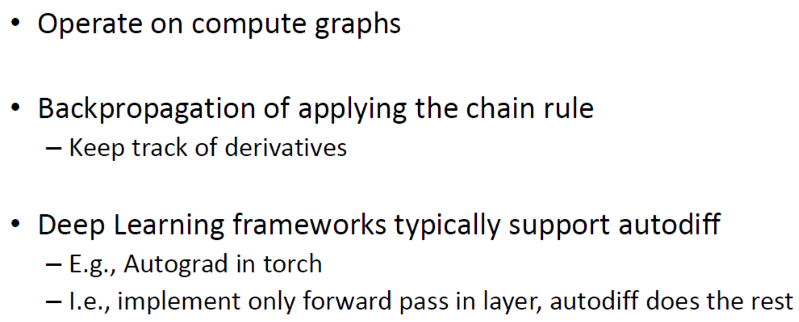


1.8 Connection to DL



- Deep Learning uses stochastic Gradient Descent
- Backpropagation
- But no second order solvers

- True gradient is hard to compute for large training sets
  - Needs stochasticity
  - There is also theory why that helps with local minima
    - Theory: many local minima are equivalent in performance even though weights are different
- Stochasticity does not seem to well with 2nd order solvers
  - There are attempts… don't seem to work so well

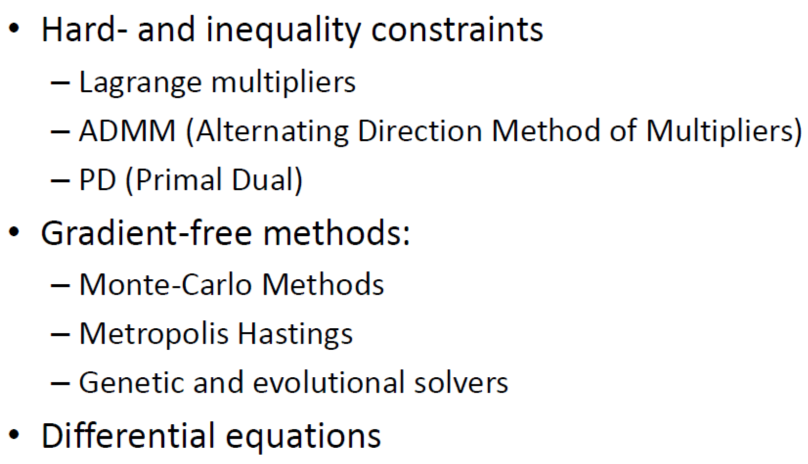


- Operate on compute graphs

- Backpropagation of applying the chain rule
  - Keep track of derivatives

- Deep Learning frameworks typically support autodiff
  - E.g., Autograd in torch
  - I.e., implement only forward pass in layer, autodiff does the rest

1.9 Connection to Other Optimizations



- Hard- and inequality constraints
  - Lagrange multipliers
  - ADMM (Alternating Direction Method of Multipliers)
  - PD (Primal Dual)
- Gradient-free methods:
  - Monte-Carlo Methods
  - Metropolis Hastings
  - Genetic and evolutional solvers
- Differential equations