Bundle Adjustment Math Reading Seminar Thing

LukášP

Brno University of Technology
Faculty of Information Technology
Božetěchova 2, 612 66 Brno
ipolok@fit.vutbr.cz

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Talk structure

- 3D geometry
- Projective geometry
- Problem Statement
- Minimum Geometric Problems
- Least squares, Moore-Penrose pseudoinverse, SVD
- Nonlinear Least Squares
- Intermezzo I – computing derivatives
  - Symbolic / Numerical differentiation
  - Dual numbers
  - Lie groups basics
- Handling Severe Nonlinearity
  - Levenberg Marquardt
  - Dogleg
- Handling outliers – robust estimation
- Intermezzo II – solving linear systems
  - Direct methods
  - Schur complement trickery
  - ~~Iterative methods~~
  - ~~Inexact step NLS~~
- Calculating covariances

- Position representations
  - Euclidean $[x, y, z]$ — 3D
  - Inverse depth $[{}^{x}/_{z}, {}^{y}/_{z}, {}^{1}/_{z}]$ — 3D
  - Inverse distance $[x, y, z, {}^{1}/_{d}]$, where $\|[x, y, z]\| = 1$ — 1D
- Rotation representations
  - Rotation matrix $[\boldsymbol{r}, \boldsymbol{u}, \boldsymbol{f}]$ — 9D
    - Hard to constrain orthogonality in numerical manipulation
  - (Unit) Quaternion $\left[[x, y, z] \cdot \sin\left({}^{\theta}/_{2}\right), \cos\left({}^{\theta}/_{2}\right)\right]$ — 4D
    - Double cover of SO(3)!
  - Axis-angle $\left[[x, y, z] \cdot \theta\right]$ — 3D
  - Exponential map of $\mathfrak{so}(3)$ $\left[[x, y, z] \cdot \theta\right]_{\times}$ — 3D

- Use homogenous coordinates
  - [x/w, y/w, z/w] → [x, y, z, w]
- Projection matrix
  - Vision

$$\begin{pmatrix} f & s & c_x & \\ & f & c_y & \\ & & 1 & \\ & & 1 & 0 \end{pmatrix}$$

We're not interested in the w coordinate (it is „always" 1).

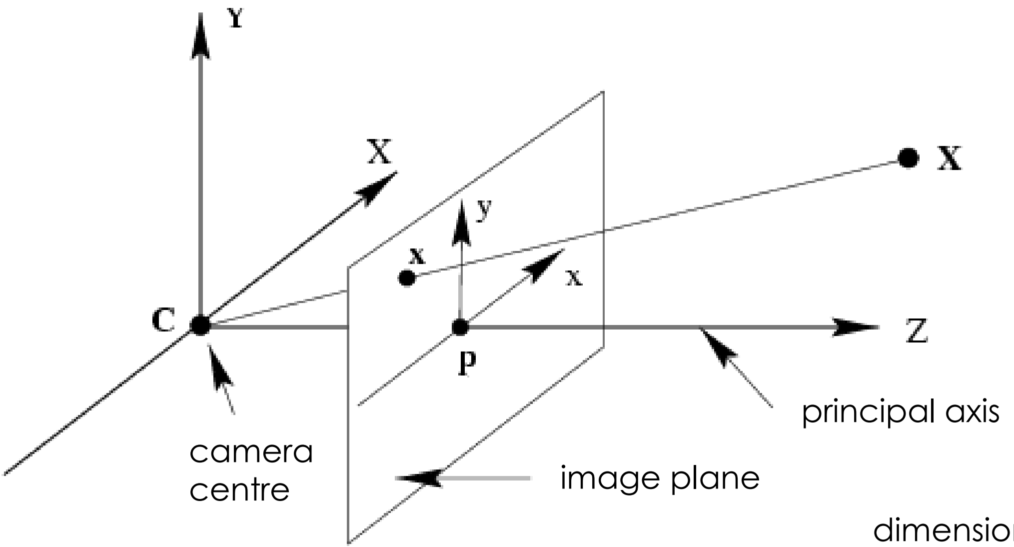
Leads to division by depth.

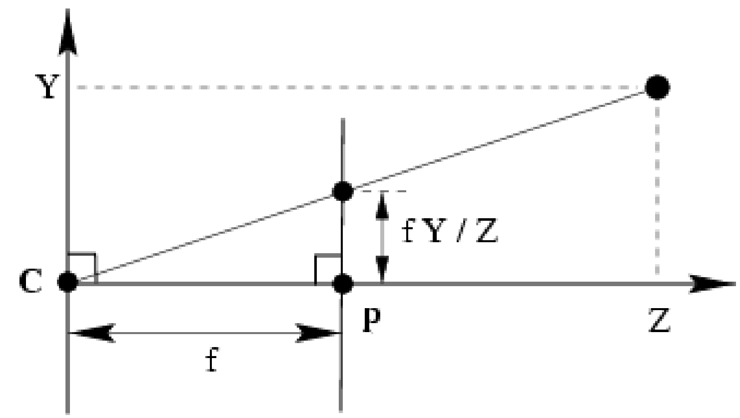
  - OpenGL

$$\begin{pmatrix} \frac{2n}{r-l} & & \frac{r+l}{r-l} & \\ & \frac{2n}{t-b} & \frac{t+b}{t-b} & \\ & & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ & & -1 & \end{pmatrix}$$

BRNO UNIVERSITY OF TECHNOLOGY

- Stolen from Bronek Pribyl's VGE lecture (I think)





intrinsics K

dimension wrangling magic

pose

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} f & & p_x \\ & f & p_y \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \end{bmatrix} \begin{bmatrix} \mathbf{R} & t \\ 0_3^T & 1 \end{bmatrix}}_{\mathbf{P} = K\,[R\ t]} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \qquad \lambda x = \mathbf{P}.X$$

x

X

Other Camera Models

- Spherical
- Curvilinear
- Catadioptric
- …

- Brown's radial distortion model

$$x_d = x\left(1 + f\left(\|x - [c_x \quad c_y]\|\right)\right)$$

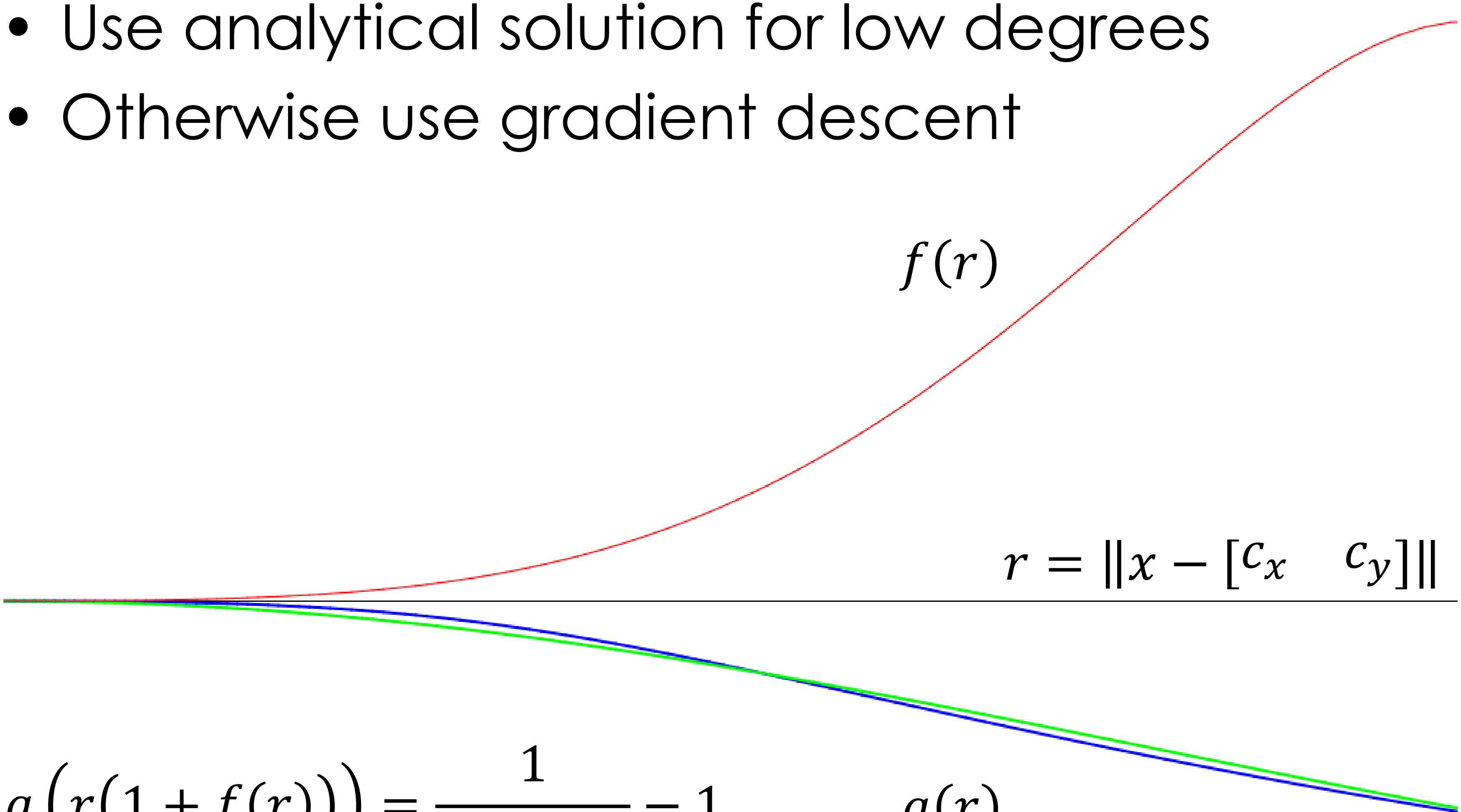
  - Here, $f(\cdot)$ is typically a polynomial function, e.g.:

$$f(a) = [a^3 \quad a^5 \quad a^7]^T \boldsymbol{f}\ ,$$

where $\boldsymbol{f}$ is a vector of polynomial coefficients

Lens Distortion Models

- Problem with invertibility of polynomial $f(\cdot)$
- Use analytical solution for low degrees
- Otherwise use gradient descent

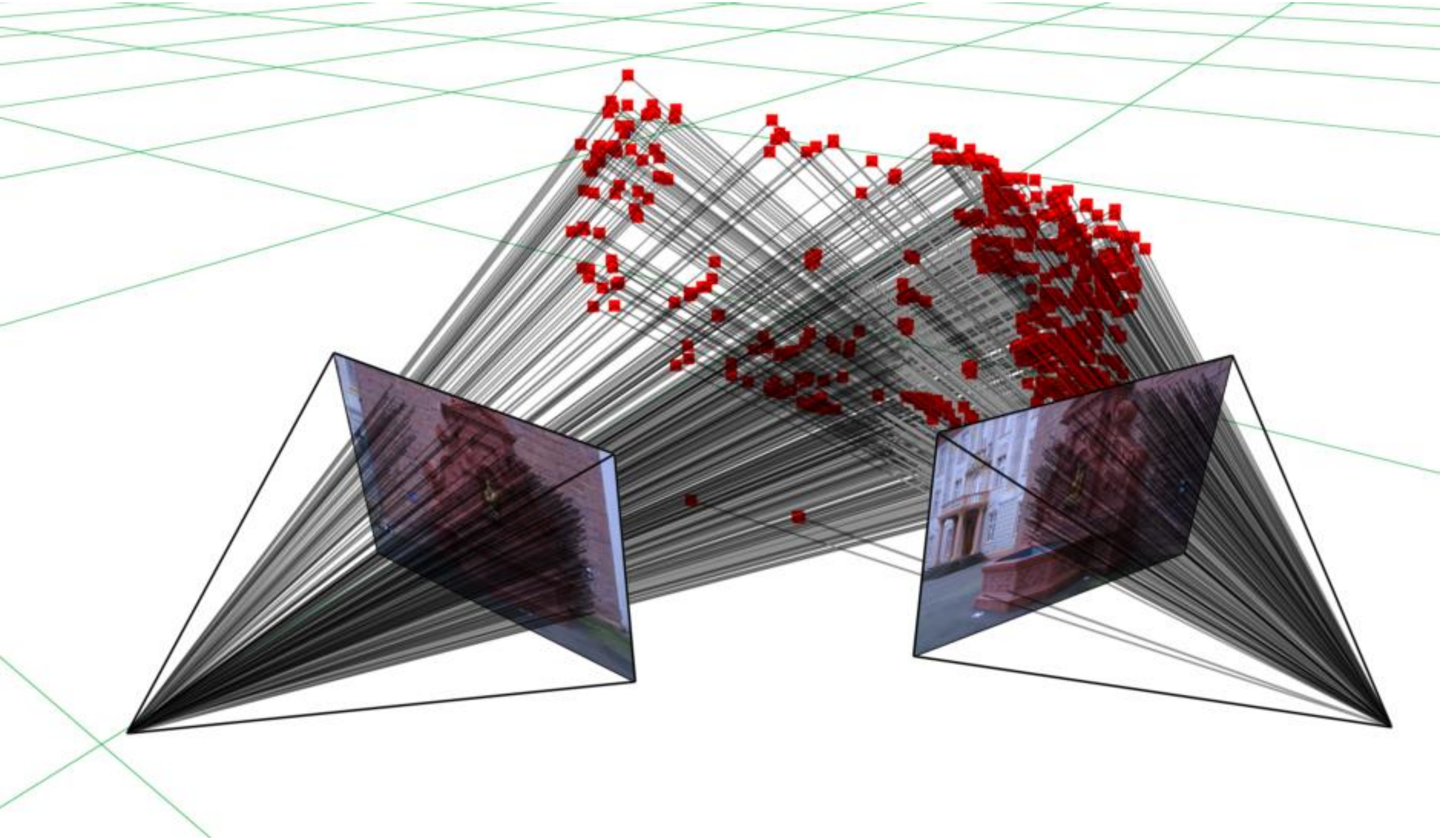


$$g\Big(r\big(1 + f(r)\big)\Big) = \frac{1}{1 + f(r)} - 1$$

- Bundle Adjustment
  - „Having a lot of images of a scene, let‘s build a 3D model“
  - Sparse approach (bundler)
    - Find interest point in the images
    - Match the interest points
    - Calculate spatial rotation / translation between images
    - Triangulate points
  - Semi-dense approach (LSD)
    - Somehow triangulate points in the first camera pair
    - Color the points (from image pixels)
    - For following cameras, reproject the 3D points, optimize rotation / translation to minimize color difference
  - Dense approach (PMVS / CMVS)

- Bundle Adjustment
  - „Having a lot of images of a scene, let‘s build a 3D model“
  - Sparse approach (bundler)
  - Semi-dense approach (LSD)
  - Dense approach (PMVS / CMVS)
    - Solve dense pixel correspondences
    - Early methods modifications of dynamic programming
    - Gives per-pixel dense depth
      - Requires known relative camera poses (ok for stereo)
    - Can solve alignment by e.g. Iterative closest point (ICP)

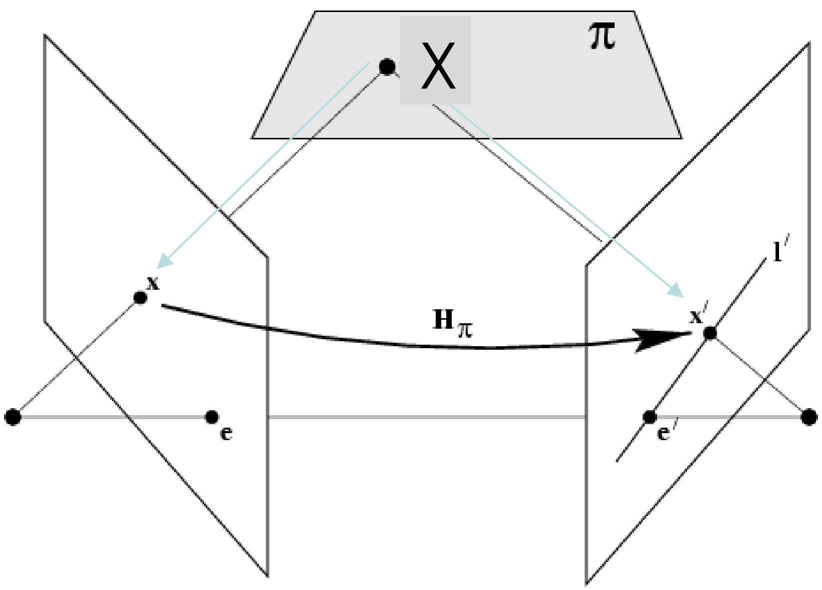
Bundler / Sparse Approach

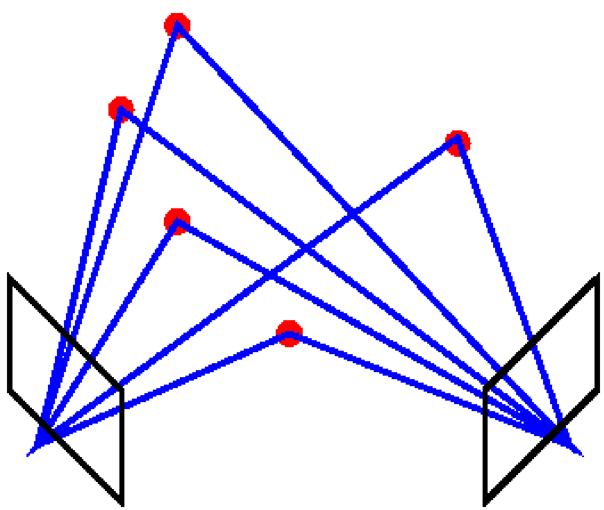


- How do we calculate positions from images?
- MGPs! http://cmp.felk.cvut.cz/mini/
(also recent VGS-IT by Tom Pajdla)

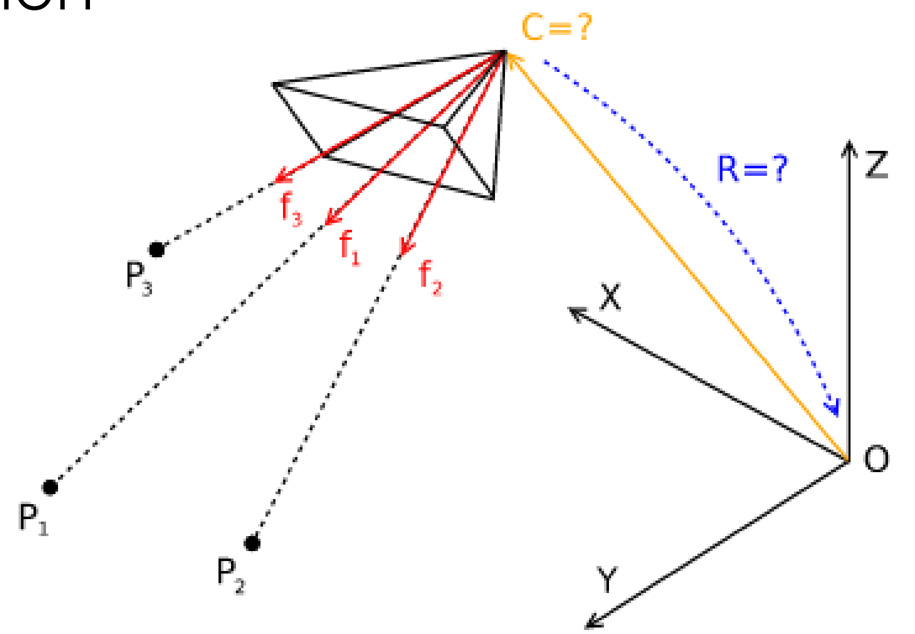
- Given a bare minimum of data, get solution by applying some geometric constraints

- Five-point algorithm [Nister et al. 2004]
  - Given five 2D points, find [R t] between the cameras
  - Employs Epileptic geometry
    - Fundamental matrix — $F = K_2^{-T}([t]_\times R)K_1^{-1} = K_2^{-T}EK_1^{-1}$
    - Epipolar constraint — $x'Fx = 0$
    - Solve 10^{th} order polynomial — meh
    - Decompose E to R and t — multiple (im)possible solutions





- Perspective three Point (P3P) [Kneip et al. 2013]
  - Given three 3D points and their 2D observations, find [R t] of the observing camera
  - From knowledge of K, convert 2D observations to 3D directions
  - Leads to quartic equation
  - Up to four solutions



Minimal Geometric Problems

- Kabsch‘s algorithm [Kabsch 1976]
  - Many variants (Coustias, Horn, Umeyama, …)
  - Given two sets of 3D points, calculate [R t] that aligns them
  - Calculate centroids $c_1 = \frac{1}{N}\sum_{i=1}^{N} x_i$ and $c_2 = \frac{1}{N}\sum_{i=1}^{N} x'_i$
  - Translation t = $c_2 - c_1$
  - Rotation derived from covariance

$$A = \sum_{i=1}^{N}(x_i - c_1)^T(x'_i - c_2)$$

as

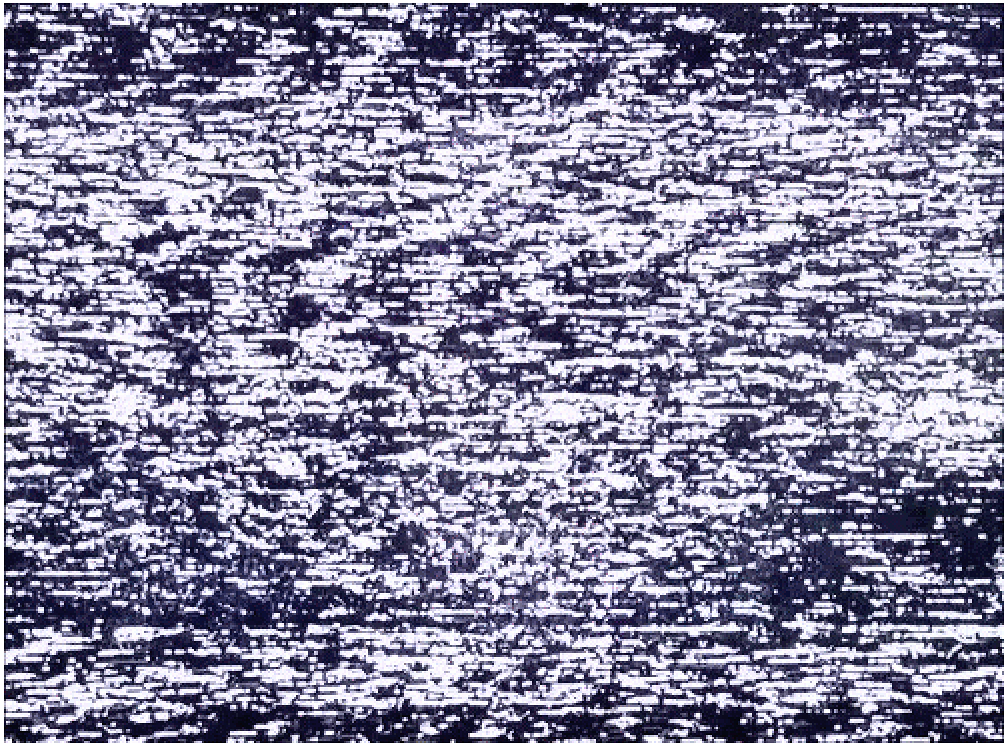
$$R = V\begin{bmatrix} 1 & & \\ & 1 & \\ & & \mathrm{sign}\left(\det(VU^T)\right) \end{bmatrix}U^T$$

using A = USV^T so actually not a *minimal* problem (but certainly a *geometric* one)

- Use 5-point algorithm to get [R t] of the first two cameras
- Keep using P3P for each consecutive camera
- Keep aligning and concatenating

BA Using Minimal Geometric Problems

- Use 5-point algorithm to get [R t] of the first two cameras
- Keep using P3P for each consecutive camera
- Keep aligning and concatenating
- But …



BA Using Minimal Geometric Problems

- Use 5-point algorithm to get [R t] of the first two cameras
- Keep using P3P for each consecutive camera
- Keep aligning and concatenating

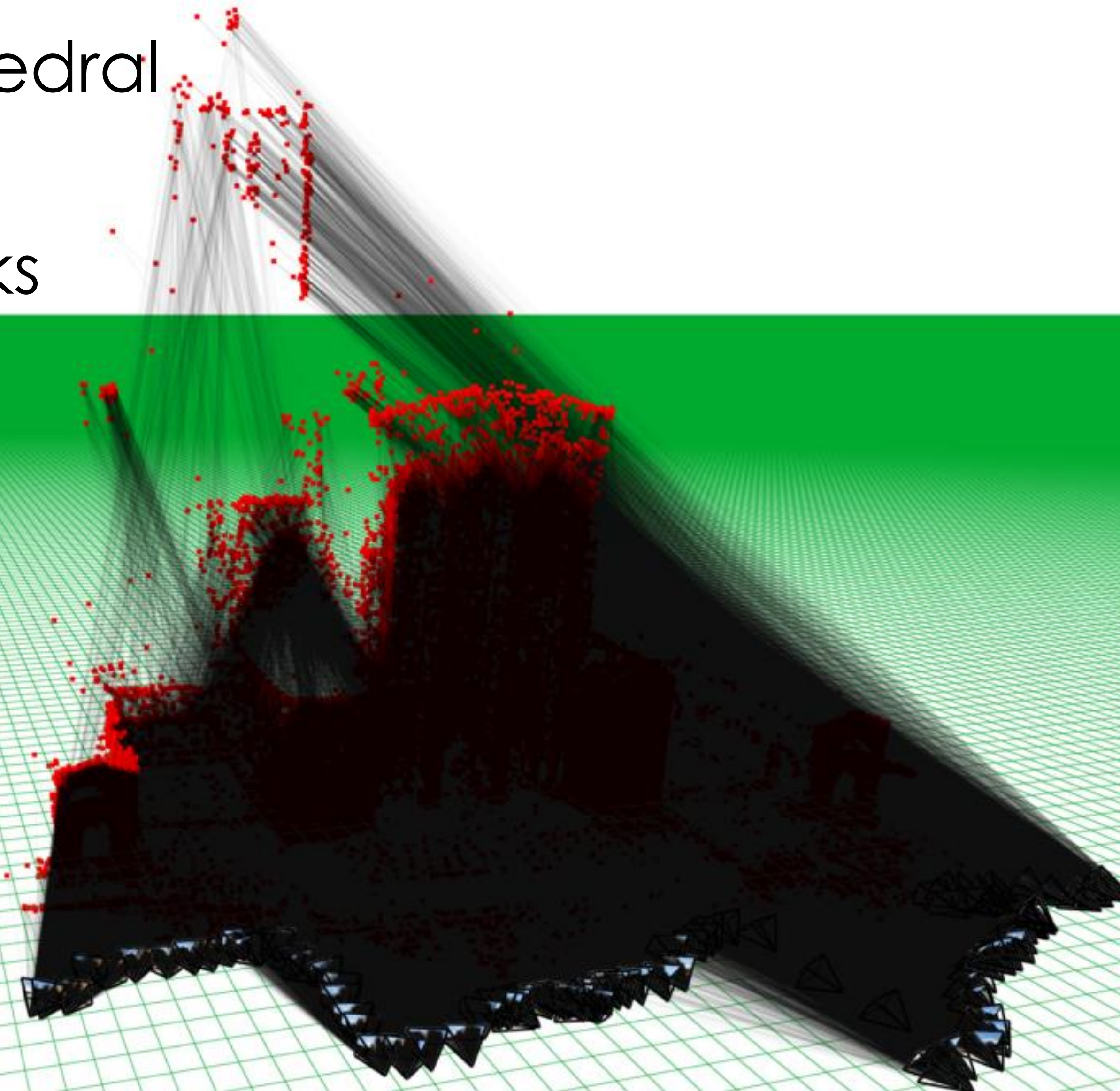
- Use RANSAC to select the points for the MGPs
- Not enough! Need to do *maximum likelihood estimation* (MLE)

- Bundle Adjustment
  - We have a set of „cameras“
    - Each described by a pose [R t], projection K
    - Possibly also lens distortion paramaters l and function
  - We have a set of observed points
    - Each described by its position X
  - We have a set of observations
    - Link between point X, camera C
    - Position of point in the image x
    - Residual to minimize

$$r = \mathrm{err}\left(x,\ \mathrm{distort}\left(\mathrm{l}, \mathrm{dehomog}\left(P\begin{bmatrix} R & t \\ & 1 \end{bmatrix}X\right)\right)\right)$$

| Problem Statement (II) – BA Graph Example

- Guildford Cathedral
- 92 poses
- 57957 landmarks



- Simultaneous Localization and Mapping–SLAM
  - We have a set of „robot poses“
    - Each described by a [R t] matrix
  - We *may* have a set of landmarks
    - Each described by its position l
  - We have a set of observations
    - Odometry observations (links between two poses)
      - Estimated distance travelled D (also a pose itself)
      - Residual to minimize

$$r = \mathrm{err}\left(\begin{bmatrix} R & t \\ & 1 \end{bmatrix}, \begin{bmatrix} R & t \\ & 1 \end{bmatrix} \oplus D\right)$$

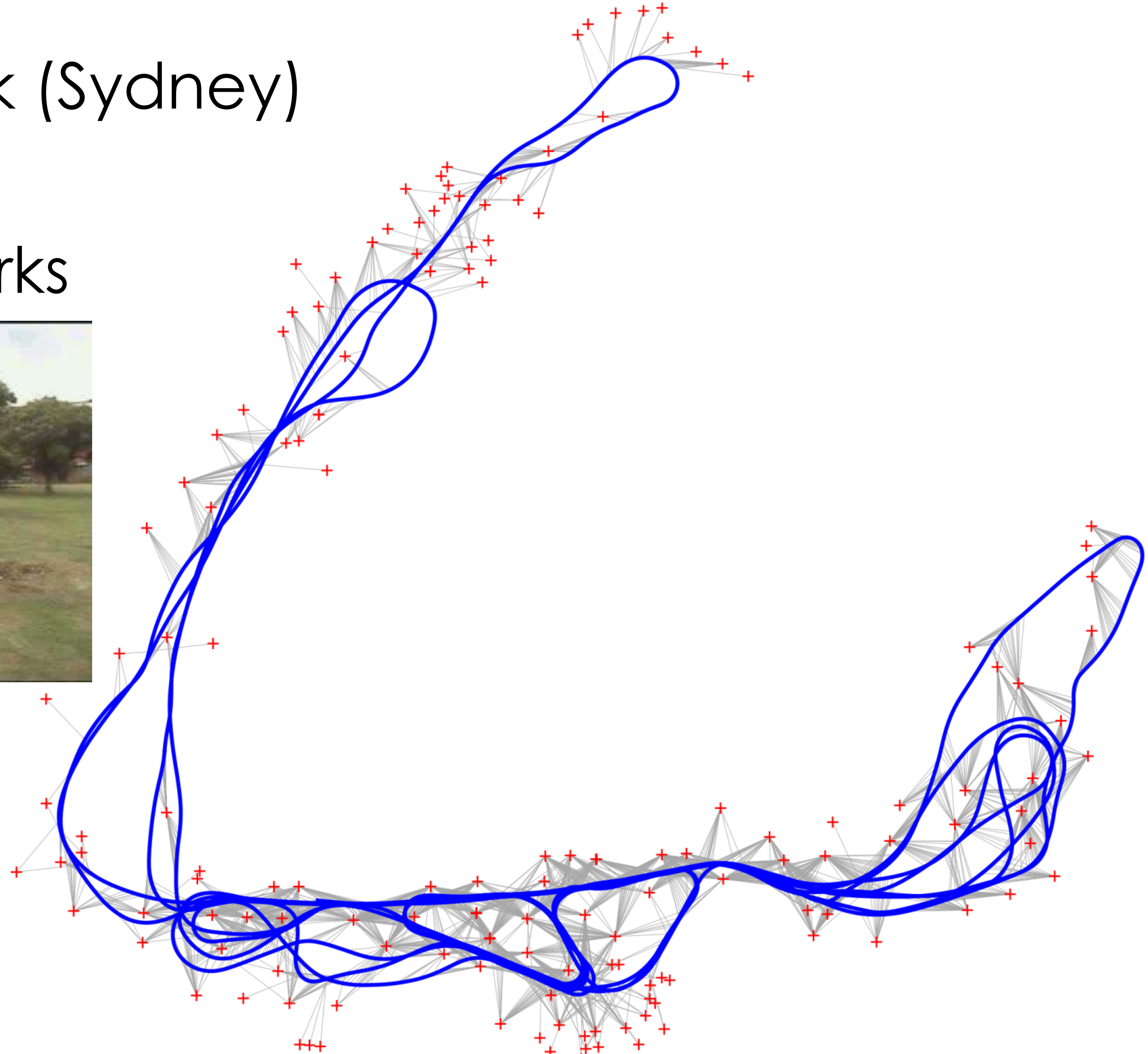
    - Landmark observations (pose-landmark links)
      - „Measurement“ of the landmark (e.g. Range-bearing vector)
      - Residual to minimize

$$r = \mathrm{err}\left(l, \begin{bmatrix} R & t \\ & 1 \end{bmatrix} \oplus [r \quad b]\right)$$

Problem Statement (II) – SLAM Graph Example

BRNO UNIVERSITY OF TECHNOLOGY

- Victoria Park (Sydney)
- 6969 poses
- 151 landmarks



- In general, inference on *graphical* models
  - Set of variables V, each contains state vector $\mathbf{v}_l$
  - Set of edges E, each a triplet $\{\mathbf{i}_k, \mathbf{j}_k, \mathbf{o}_k\}$
    - Where **i** and **j** are vectors containing vertex indices
  - Minimize

$$\sum_{k=0}^{|E|} \mathrm{err}\left(\boldsymbol{v}_{\boldsymbol{i}_k} \ominus \boldsymbol{v}_{\boldsymbol{j}_k} \oplus \boldsymbol{o}_k\right),$$

where $\ominus$ and $\oplus$ are vectorial difference and composition operators (e.g. (inverse) matrix multiplication), subject to their precedence and commutativity

- In general, inference on *graphical* models
  - Set of variables V, each contains state vector $\mathbf{v}_l$
  - Set of edges E, each a triplet $\{\mathbf{i}_k, \mathbf{j}_k, \mathbf{o}_k\}$
    - Where **i** and **j** are vectors containing vertex indices
  - Minimize

$$\sum_{k=0}^{|E|} \operatorname{err}\left(\boldsymbol{v}_{\boldsymbol{i}_k} \ominus \boldsymbol{v}_{\boldsymbol{j}_k} \oplus \boldsymbol{o}_k\right)$$

  - Problem: if there are different kinds of observations, how to make different $\operatorname{err}(\cdot)$ comparable?
  - Solution: $\operatorname{err}(v) = \|v\|_{\Sigma_k} = v\Sigma_k v^T$, where Σ_k is information matrix (inverse covariance) for observation k

- Given a set of constraints (observations), find a solution that minimizes L2 error
- For a linear over-determined problem $A\boldsymbol{x} = \boldsymbol{b}$ (where the unknown is $\boldsymbol{x}$)
- The error is $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$, squared error is* $\boldsymbol{r}^2 = (\boldsymbol{b} - A\boldsymbol{x})^T(\boldsymbol{b} - A\boldsymbol{x}) = \boldsymbol{b}^T\boldsymbol{b} - 2\boldsymbol{x}^T A^T\boldsymbol{b} + \boldsymbol{x}^T A^T A\boldsymbol{x}$
- This error minimizes where the first derivative cancels $\boldsymbol{r}^{2\prime} = 0$
- Differentiating by $\boldsymbol{x}$ gives $-A^T\boldsymbol{b} + (A^T A)\boldsymbol{x} = 0$
- Hence $\boldsymbol{x} = (A^T A)^{-1}A^T\boldsymbol{b} = A^+\boldsymbol{b}$
- A^+ is the Moore-Penrose pseudo-inverse

*Note that $\boldsymbol{x}^T A^T\boldsymbol{b}$ is a scalar, thus allowing summation $\boldsymbol{x}^T A^T\boldsymbol{b} + \boldsymbol{b}^T A\boldsymbol{x} = \boldsymbol{2x}^T A^T\boldsymbol{b}$.

- Sometimes we want to weight the observations
- Treated easily, as

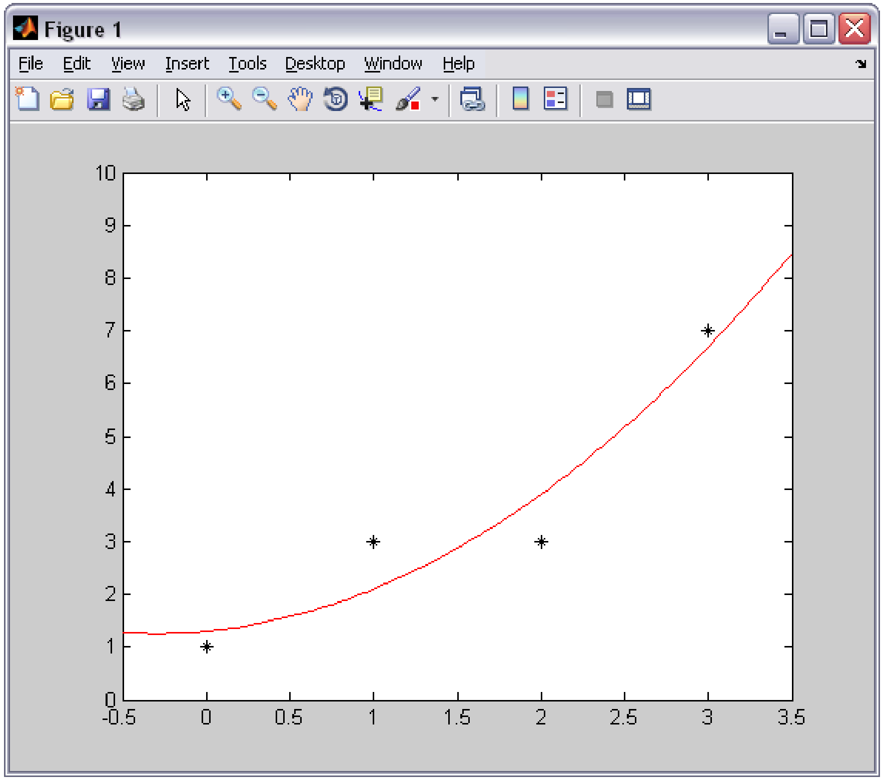
  $$\boldsymbol{x} = (A^T W A)^{-1} A^T W \boldsymbol{b} \ ,$$

  where W is a diagonal matrix containing the weights
- The weights should be reciprocal variances of the estimated variables (inverse covariances if estimating vectorial quantities)
- Can „bake" weights into A by using $\hat{A} = A\sqrt{W}$ since $W = W^T$ (diagonal matrix)

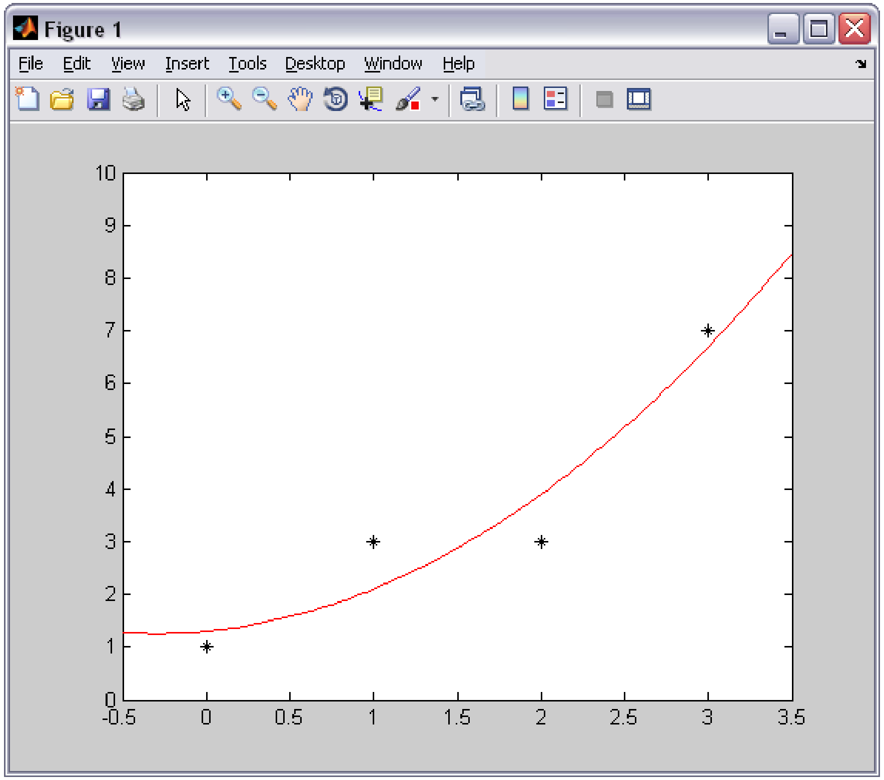
- By solving the *normal equation* $A^T A\boldsymbol{x} = A^T\boldsymbol{b}$
  - The condition number of $A^T A$ is greater than of A
  - If A was sparse but any of its rows is full, $A^T A$ is dense
- By using orthogonal decompositions
  - Notably SVD of A, $A = USV^T$ and $A^+ = VS^+U^*$ where U^* is conjugate transpose (equals U^T if real)
  - Obtaining S^+ as easy as inverting diagonal entries while skipping zeros
  - So solve $\boldsymbol{x} = VS^+U^T\boldsymbol{b}$
    - Slow to compute (expensive Householder reduction to bidiagonal form, followed by iterative diagonalization)
    - May be more precise, can threshold S to reduce noise
  - Sometimes, there is an additional constraint $|\boldsymbol{x}| = 1$, then only use the smallest singular value, zero the rest

Least Squares Demo (Matlab)

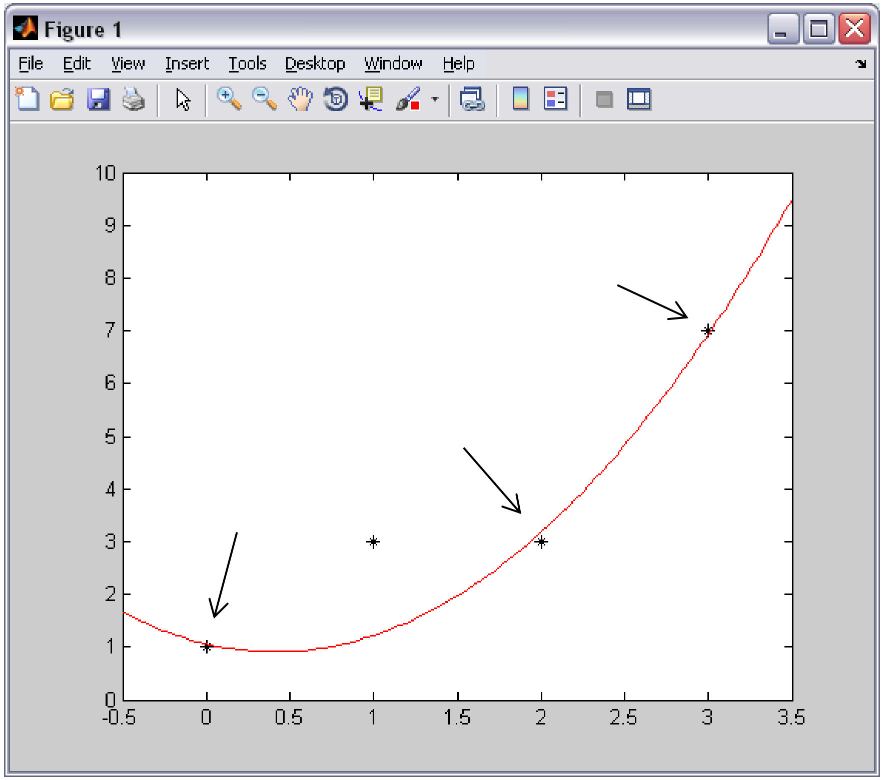

```
O = [0 1; 1 3; 2 3; 3 7];
% observations of some 1D function
% O(:,1) are the arguments, a
% O(:,2) are the desired fit values, b

% we are trying to estimate b = p + qa + ra^2
% where x = [p q r] is our unknown

m = length(O);
A = [ones(m, 1), O(:, 1), O(:, 1).^2];
% stack the equations (1p + aq + a^2r)

x = (A' * A) \ A' * O(:,2)
% solve normal equation (backslash = solve)

xx = linspace(min(O(:, 1)) - .5, max(O(:, 1)) + .5);
yy = x(1) + x(2) * xx + x(3) * (xx.^2);
% evaluate the estimated model

plot(O(:, 1), O(:, 2), '*k') % plot a-s, b-s
hold on
plot(xx, yy, '-r') % plot the fitted curve
hold off
```



```
O = [0 1; 1 3; 2 3; 3 7];
% observations of some 1D function
% O(:,1) are the arguments, a
% O(:,2) are the desired fit values, b

% we are trying to estimate b = p + qa + ra^2
% where x = [p q r] is our unknown

m = length(O);
A = [ones(m, 1), O(:, 1), O(:, 1).^2];
% stack the equations (1p + aq + a^2r)

[U,S,V] = svd(A); % take SVD of A
Splus = zeros(size(S')); % S' may be rectangular
n = size(A, 2); % length of the diagonal
Splus(1:n, 1:n) = diag(1 ./ diag(S))' % recip. diag.
x = V * Splus * U' * O(:,2)
% solve using SVD

xx = linspace(min(O(:, 1)) - .5, max(O(:, 1)) + .5);
yy = x(1) + x(2) * xx + x(3) * (xx.^2);
% evaluate the estimated model

plot(O(:, 1), O(:, 2), '*k') % plot a-s, b-s
hold on
plot(xx, yy, '-r') % plot the fitted curve
hold off
```



```
O = [0 1; 1 3; 2 3; 3 7];
% observations of some 1D function
% O(:,1) are the arguments, a
% O(:,2) are the desired fit values, b

W = diag([10 1 10 10])
% weights for the observations

m = length(O);
A = [ones(m, 1), O(:, 1), O(:, 1).^2];
% stack the equations (1p + aq + a^2r)

x = (A' * W * A) \ A' * W * O(:, 2)
% solve normal equation (backslash = solve)

xx = linspace(min(O(:, 1)) - .5, max(O(:, 1)) + .5);
yy = x(1) + x(2) * xx + x(3) * (xx.^2);
% evaluate the estimated model

plot(O(:, 1), O(:, 2), '*k') % plot a-s, b-s
hold on
plot(xx, yy, '-r') % plot the fitted curve
hold off
```



BRNO UNIVERSITY OF TECHNOLOGY

- The estimation for bundle adjustment is highly *nonlinear* (rotations, projections, …)
- Formally, instead of linear $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$ we now have $\boldsymbol{r} = \boldsymbol{b} - h(\boldsymbol{x})$ with nonlinear function $h(\cdot)$
- To minimize $s = \boldsymbol{r}^T\boldsymbol{r}$, we again set $\frac{\partial s}{\partial \boldsymbol{x}} = 2\sum \boldsymbol{r}_i \frac{\partial \boldsymbol{r}_i}{\partial \boldsymbol{x}} = 0$
- Instead of directly getting $\boldsymbol{x}$, at iteration k, we improve ${}^{k+1}\boldsymbol{x} = {}^{k}\boldsymbol{x} + \Delta\boldsymbol{x}$
- To solve this, we shall
  - Linearize the problem using Taylor expansion
  - Assume Gaussian noise to be able to do that

- We call ${}^{k}\boldsymbol{x}$ our *linearization point*
- Approximate
  $$h(\boldsymbol{x}) \approx h\left({}^{k}\boldsymbol{x}\right) + \frac{\partial h\left({}^{k}\boldsymbol{x}\right)}{\partial \boldsymbol{x}}\left(\boldsymbol{x} - {}^{k}\boldsymbol{x}\right) = h\left({}^{k}\boldsymbol{x}\right) + \boldsymbol{J}\Delta\boldsymbol{x}$$
- The Jacobian $\boldsymbol{J}$ changes with the linearization
- At step k, we solve linearized problem
  $\Delta\boldsymbol{b} = \boldsymbol{b} - h\left({}^{k}\boldsymbol{x}\right)$,
  $\boldsymbol{r} = \boldsymbol{b} - h(\boldsymbol{x}) = \left(\boldsymbol{b} - h\left({}^{k}\boldsymbol{x}\right)\right) + \left(h\left({}^{k}\boldsymbol{x}\right) - h(\boldsymbol{x})\right)$,
  $\boldsymbol{r} \approx \Delta\boldsymbol{b} - \boldsymbol{J}\Delta\boldsymbol{x}$ and that is a linear model!
- Solve familiar $\boldsymbol{J}^T\boldsymbol{J}\Delta\boldsymbol{x} = \boldsymbol{J}^T\Delta\boldsymbol{b}$
- Note that also using Hessian $\boldsymbol{H} = \frac{\partial h\left({}^{k}\boldsymbol{x}\right)}{\partial \boldsymbol{x}\partial \boldsymbol{x}^T}$ would converge faster (this is 1^{st} order method)

- Gauss-Newton algorithm

  Take initial guess of ${}^{0}\boldsymbol{x}$, set $k = 0$

  Repeat until end of time

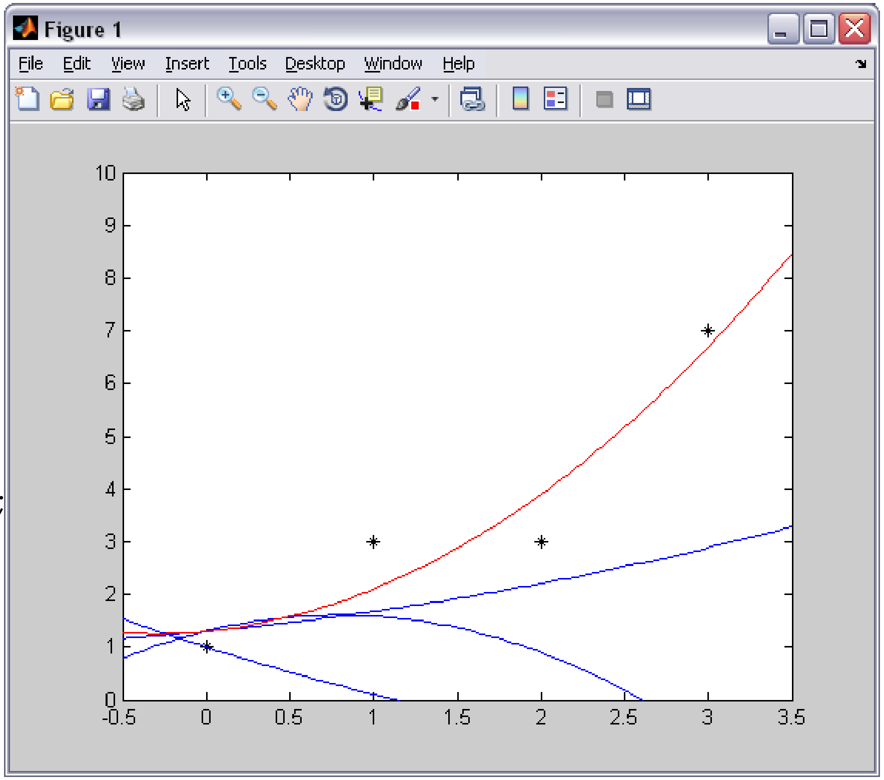
  > Linearize the system $\boldsymbol{J} = \frac{\partial h({}^{k}\boldsymbol{x})}{\partial \boldsymbol{x}}$
  >
  > Calculate residual $\Delta\boldsymbol{b} = \boldsymbol{b} \ominus h({}^{k}\boldsymbol{x})$ note the vectorial op
  >
  > Solve $\boldsymbol{J}^T\boldsymbol{J}\Delta\boldsymbol{x} = \boldsymbol{J}^T\Delta\boldsymbol{b}$
  >
  > If $||\Delta\boldsymbol{x}|| < t$ then
  >
  > stop
  >
  > ${}^{k+1}\boldsymbol{x} = {}^{k}\boldsymbol{x} \oplus \Delta\boldsymbol{x}$ note the vectorial op
  >
  > $k = k + 1$

- Unlike linear LS, we need initial guess ${}^{0}\boldsymbol{x}$
- Sometimes, LLS can be used to estimate it
- For BA, there are MGPs to take care of that

Nonlinear Least Squares Demo (Matlab)

```
O = [0 1; 1 3; 2 3; 3 7];
% observations of some 1D function
% O is vector of pairs arg. a and desired fit b

x = [1 -1 -.1]';
% guess 0x (deliberately a bad guess, to take a few steps)

plot(O(:, 1), O(:, 2), '*k') % plot a-s, b-s
hold on
xx = linspace(min(O(:, 1)) - .5, max(O(:, 1)) + .5);
for i = 1:10
   yy = x(1) + x(1) * x(2) * xx + x(1) * x(2) * x(3) * (xx.^2);
   plot(xx, yy, '-b') % plot the initial guess
   % evaluate the initial guess

   J = [1 + x(2) * O(:, 1) + x(2) * x(3) * O(:, 1).^2, ...
      x(1) * O(:, 1) + x(1) * x(3) * O(:, 1).^2, x(1) * x(2) * O(:, 1).^2];
   % calculate the Jacobian

   db = O(:,2) - x(1) - O(:,1) * x(1) * x(2) - O(:,1).^2 * x(1) * x(2) * x(3);
   % calculate current error vector

   dx = (J' * J) \ J' * db; norm_dx = norm(dx)
   % solve

   if(norm_dx < 1e-6) % see if we optimize
      break
   end
   x = x + dx % increment
end

yy = x(1) + x(1) * x(2) * xx + x(1) * x(2) * x(3) * (xx.^2);
plot(xx, yy, '-r') % plot the final in red
hold off
```

```
% we are trying to estimate b = p + pqa + pqra^2
% where x = [p q r] is our unknown
% h(x) = p + pqa + pqra^2, J = dh(x) / dx = [1+qa+qra^2, pa+pra^2, qa^2]
% we use this parameterization to make it nonlinear (would work with linear
% too but would be able to optimize in a single step, which would be boring)
```



And now for something completely different ...

INTERMEZZO I

- Can readily use Matlab's symbolic toolbox
- Can use a cookbook (e.g. [J. Blanco, 2010, A tutorial on se(3) transformation parameterizations and on-manifold optimization, (TR)].)
- Derivatives know nothing about vectorial, so

$$\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} := \frac{\partial f(\boldsymbol{x} \oplus \boldsymbol{\varepsilon})}{\partial \boldsymbol{\varepsilon}}$$

- When dealing with rotations, commutability becomes an issue, ${}^{k+1}\boldsymbol{x} = {}^{k}\boldsymbol{x} \oplus \Delta\boldsymbol{x}$ must match $\boldsymbol{J}$

„If rotations and translations commuted, we could simply do all our rotations in the morning, before leaving home."

- Worked example – pose SLAM
  - We have two poses, $[R_1\ \boldsymbol{t}_1]$ and $[R_2\ \boldsymbol{t}_2]$
  - We want to avoid optimizing 4×4 matrices
  - Internally, they are optimized as $\mathbb{R}^6$ [axis-angle $\boldsymbol{t}$]
  - That gives us $\mathrm{v}([R\ \boldsymbol{t}]) = [\mathrm{aa}(R)\ \boldsymbol{t}]$ and $\mathrm{m}([\boldsymbol{r}\ \boldsymbol{t}]) = [R\ \boldsymbol{t}]$
  - We have $\oplus([\boldsymbol{r}_1\ \boldsymbol{t}_1], [\boldsymbol{r}_2\ \boldsymbol{t}_2]) = \mathrm{v}\big(\mathrm{m}([\boldsymbol{r}_1\ \boldsymbol{t}_1]) \cdot \mathrm{m}([\boldsymbol{r}_2\ \boldsymbol{t}_2])\big)$ and $\ominus([\boldsymbol{r}_1\ \boldsymbol{t}_1], [\boldsymbol{r}_2\ \boldsymbol{t}_2]) = \mathrm{v}\left(\big(\mathrm{m}([\boldsymbol{r}_1\ \boldsymbol{t}_1])\big)^{-1} \cdot \mathrm{m}([\boldsymbol{r}_2\ \boldsymbol{t}_2])\right)$
  - For NLS, we need $\boldsymbol{J} = \frac{\partial h({}^{k}\boldsymbol{x})}{\partial \boldsymbol{x}} = \sum_{k=1}^{|E|} \frac{\partial\left(\boldsymbol{x}_{i_k} \ominus \boldsymbol{x}_{j_k}\right)}{\partial\left[\boldsymbol{x}_{i_k}\ \boldsymbol{x}_{j_k}\right]}$
  - Also, error $\Delta\boldsymbol{b} = \sum_{k=1}^{|E|} \boldsymbol{o}_k \ominus \left(\boldsymbol{x}_{i_k} \ominus \boldsymbol{x}_{j_k}\right)$
  - Then update $\hat{\boldsymbol{x}}_i = \Delta\boldsymbol{x}_i \oplus \boldsymbol{x}_i$ for each variable $\boldsymbol{x}_i \in V$
  - Recall that this is on graph $\{V, E\}, \{\boldsymbol{i}_k, \boldsymbol{j}_k, \boldsymbol{o}_k\} \in E$

- A simple rule that allows decomposition of derivatives

$$\left(f\left(g(x)\right)\right)' = f'\left(g(x)\right) \cdot g'(x)$$

so instead of calculating long $\left(f\left(g(x)\right)\right)'$, we calculate much shorter $f'(y)$ and $g'(x)$ and multiply them numerically when evaluating $\boldsymbol{J}$

- We typically calculate derivatives of $\oplus$, $\ominus$, $\mathrm{v}(\cdot)$ and $\mathrm{m}(\cdot)$ and chain rule the rest

- Lets try to get a Jacobian of $\mathrm{v}([R\ \boldsymbol{t}]) = [\mathrm{aa}(R)\ \boldsymbol{t}]$

```
syms r00 r01 r02 r10 r11 r12 r20 r21 r22 t0 t1 t2 real
R = [r00 r01 r02; r10 r11 r12; r20 r21 r22]
t = [t0 t1 t2]'
% we have a rotation matrix and a translation vector

qw = sqrt(1 + r00 + r11 + r22) / 2;
qx = (r21 - r12) / (4 * qw);
qy = (r02 - r20) / (4 * qw);
qz = (r10 - r01) / (4 * qw);
% simplified matrix to quaternion (would need several branches for numerical stability)

qnorm = simplify(sqrt([qx qy qz] * [qx qy qz]'));
halfangle = asin(qnorm);
% get half of the rotation angle

ax = simplify(qx / qnorm * 2 * halfangle);
ay = simplify(qy / qnorm * 2 * halfangle);
az = simplify(qz / qnorm * 2 * halfangle);
% get the rotation as axis-angle

vec = [ax ay az t0 t1 t2]'
Rtvec = [r00 r01 r02 r10 r11 r12 r20 r21 r22 t0 t1 t2]';
J = simplify(jacobian(vec, Rtvec));
% let matlab calculate the derivatives

ccode(J)
% export to C
```

The output of ccode() I

55 kB

SLAM Derivatives Continued

- Some more tricks

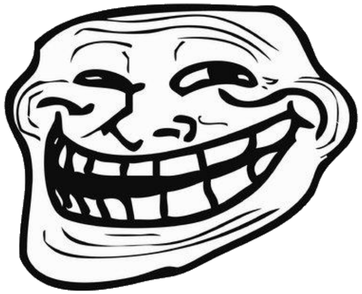
```
syms qnorm_ qx_ qy_ qz_ qw_ ax_ ay_ az_ real
J = subs(J, ax, ax_)
J = subs(J, ay, ay_)
J = subs(J, az, az_)
J = subs(J, qx, qx_)
J = subs(J, qy, qy_)
J = subs(J, qz, qz_)
J = subs(J, qw, qw_)
J = subs(J, qnorm, qnorm_)
% try substituting common subexpressions to tame the beast (don't simplify J in the previous slide!)

ccode(J)
% export to C
```

- Still generates 34 kB of C code
- Gotcha – simple() and simplify() are different
- Lesson learned – don't use [R t], use Quaternions instead!

The output of ccode() II

34 kB

- Better yet



```
quat_t = [qw qx qy qz t0 t1 t2]
J0 = jacobian(quat_t, Rtvec);
%
syms ax_ ay_ az_ qx_ qy_ qz_ qw_ real
%qnorm = (sqrt([qx qy qz] * [qx qy qz]')); % MATLAB trolls you if U no careful …
qnorm = (sqrt(qx^2 + qy^2 + qz^2));
halfangle_ = asin(qnorm_);
ax_ = (qx_ / qnorm_ * 2 * halfangle_);
ay_ = (qy_ / qnorm_ * 2 * halfangle_);
az_ = (qz_ / qnorm_ * 2 * halfangle_);
vec_ = [ax_ ay_ az_ t0 t1 t2]'
quat_t_ = [qw_ qx_ qy_ qz_ t0 t1 t2]
J1 = jacobian(vec_, quat_t_);
% calculate the jacobians separately (note how J1 is on separate variables)

% chain rule as:
%  f(    g(R))' =  f'(    g(R)) *     g'(R)
% aa(quat(R))' = aa'(quat(R)) * quat'(R)
%             J =             J1 *        J0

J1 = subs(J1, qx_, qx); J1 = subs(J1, qy_, qy); J1 = subs(J1, qz_, qz); J1 = subs(J1, qw_, qw)
% substitute the quaternion to J1 (in practice, we would substitute *values* rather than *formulas*)

error = simplify(J - J1 * J0) % verify, this prints a matrix of zeros
```

- Generates 1.94 and 2.29 kB of C for J0 and J1

The output of ccode() III

```
T[0][0] = -(r21-r12)/sqrt(pow(1.0+r00+r11+r22,3.0))/4.0;      T[0][1] = 0.0;      T[0][2] = 0.0;      T[0][3] = 0.0;      T[0][4] = -(r21-
r12)/sqrt(pow(1.0+r00+r11+r22,3.0))/4.0;      T[0][5] = -1/(sqrt(1.0+r00+r11+r22))/2.0;      T[0][6] = 0.0;      T[0][7] =
1/(sqrt(1.0+r00+r11+r22))/2.0;      T[0][8] = -(r21-r12)/sqrt(pow(1.0+r00+r11+r22,3.0))/4.0;      T[0][9] = 0.0;      T[0][10] = 0.0;      T[0][11]
= 0.0;      T[1][0] = -(r02-r20)/sqrt(pow(1.0+r00+r11+r22,3.0))/4.0;      T[1][1] = 0.0;      T[1][2] = 1/(sqrt(1.0+r00+r11+r22))/2.0;      T[1][3]
= 0.0;      T[1][4] = -(r02-r20)/sqrt(pow(1.0+r00+r11+r22,3.0))/4.0;      T[1][5] = 0.0;      T[1][6] = -1/(sqrt(1.0+r00+r11+r22))/2.0;
T[1][7] = 0.0;      T[1][8] = -(r02-r20)/sqrt(pow(1.0+r00+r11+r22,3.0))/4.0;      T[1][9] = 0.0;      T[1][10] = 0.0;      T[1][11] = 0.0;      T[2][0]
= -(r10-r01)/sqrt(pow(1.0+r00+r11+r22,3.0))/4.0;      T[2][1] = -1/(sqrt(1.0+r00+r11+r22))/2.0;      T[2][2] = 0.0;      T[2][3] =
1/(sqrt(1.0+r00+r11+r22))/2.0;      T[2][4] = -(r10-r01)/sqrt(pow(1.0+r00+r11+r22,3.0))/4.0;      T[2][5] = 0.0;      T[2][6] = 0.0;      T[2][7] =
0.0;      T[2][8] = -(r10-r01)/sqrt(pow(1.0+r00+r11+r22,3.0))/4.0;      T[2][9] = 0.0;      T[2][10] = 0.0;      T[2][11] = 0.0;      T[3][0] = 0.0;
T[3][1] = 0.0;      T[3][2] = 0.0;      T[3][3] = 0.0;      T[3][4] = 0.0;      T[3][5] = 0.0;      T[3][6] = 0.0;      T[3][7] = 0.0;      T[3][8] = 0.0;
T[3][9] = 1.0;      T[3][10] = 0.0;      T[3][11] = 0.0;      T[4][0] = 0.0;      T[4][1] = 0.0;      T[4][2] = 0.0;      T[4][3] = 0.0;      T[4][4] = 0.0;
T[4][5] = 0.0;      T[4][6] = 0.0;      T[4][7] = 0.0;      T[4][8] = 0.0;      T[4][9] = 0.0;      T[4][10] = 1.0;      T[4][11] = 0.0;      T[5][0] = 0.0;
T[5][1] = 0.0;      T[5][2] = 0.0;      T[5][3] = 0.0;      T[5][4] = 0.0;      T[5][5] = 0.0;      T[5][6] = 0.0;      T[5][7] = 0.0;      T[5][8] = 0.0;
T[5][9] = 0.0;      T[5][10] = 0.0;      T[5][11] = 1.0;
```

```
T[0][0] = 2.0/sqrt(qx_*qx_+qy_*qy_+qz_*qz_)*asin(sqrt(qx_*qx_+qy_*qy_+qz_*qz_))-
2.0*qx_*qx_/sqrt(pow(qx_*qx_+qy_*qy_+qz_*qz_,3.0))*asin(sqrt(qx_*qx_+qy_*qy_+qz_*qz_))+2.0*qx_*qx_/(qx_*qx_+qy_*qy_+qz_*
qz_)/sqrt(1.0-qx_*qx_-qy_*qy_-qz_~*qz_);      T[0][1] = -
2.0*qx_/sqrt(pow(qx_*qx_+qy_*qy_+qz_*qz_,3.0))*asin(sqrt(qx_*qx_~+qy_*qy_+qz_*qz_))*qy_+2.0*qx_/(qx_*qx_+qy_*qy_+qz_*qz
_)*qy_/sqrt(1.0-qx_*qx_~-qy_*qy_-qz_*qz_);      T[0][2] = -
2.0*qx_/sqrt(pow(qx_*qx_+qy_*qy_+qz_*qz_,3.0))*asin(sqrt(qx_*qx_~+qy_*qy_+qz_*qz_))*qz_+2.0*qx_/(qx_*qx_+qy_*qy_+qz_*qz
_)*qz_/sqrt(1.0-qx_*qx_~-qy_*qy_-qz_*qz_);      T[0][3] = 0.0;      T[0][4] = 0.0;      T[0][5] = 0.0;      T[1][0] = -
2.0*qx_/sqrt(pow(qx_*qx_+qy_*qy_+qz_*qz_,3.0))*asin(sqrt(qx_*qx_~+qy_*qy_+qz_*qz_))*qy_+2.0*qx_/(qx_*qx_+qy_*qy_+qz_*qz
_)*qy_/sqrt(1.0-qx_*qx_~-qy_*qy_-qz_*qz_);      T[1][1] = 2.0/sqrt(qx_*qx_+qy_*qy_+qz_*qz_)*asin(sqrt(qx_*qx_+qy_*qy_+qz_*qz_))-
2.0*qy_*qy_/sqrt(pow(qx_*qx_+qy_*qy_+qz_*qz_,3.0))*asin(sqrt(qx_*qx_+qy_*qy_+qz_*qz_))+2.0*qy_*qy_/(qx_*qx_+qy_*qy_+qz_
*qz_)/sqrt(1.0-qx_*qx_-qy_*qy_-qz_~*qz_);      T[1][2] = -
2.0*qy_/sqrt(pow(qx_*qx_+qy_*qy_+qz_*qz_,3.0))*asin(sqrt(qx_*qx_~+qy_*qy_+qz_*qz_))*qz_+2.0*qy_/(qx_*qx_+qy_*qy_+qz_*qz
_)*qz_/sqrt(1.0-qx_*qx_~-qy_*qy_-qz_*qz_);      T[1][3] = 0.0;      T[1][4] = 0.0;      T[1][5] = 0.0;      T[2][0] = -
2.0*qx_/sqrt(pow(qx_*qx_+qy_*qy_+qz_*qz_,3.0))*asin(sqrt(qx_*qx_~+qy_*qy_+qz_*qz_))*qz_+2.0*qx_/(qx_*qx_+qy_*qy_+qz_*qz
_)*qz_/sqrt(1.0-qx_*qx_~-qy_*qy_-qz_*qz_);      T[2][1] = -
2.0*qy_/sqrt(pow(qx_*qx_+qy_*qy_+qz_*qz_,3.0))*asin(sqrt(qx_*qx_~+qy_*qy_+qz_*qz_))*qz_+2.0*qy_/(qx_*qx_+qy_*qy_+qz_*qz
_)*qz_/sqrt(1.0-qx_*qx_~-qy_*qy_-qz_*qz_);      T[2][2] = 2.0/sqrt(qx_*qx_+qy_*qy_+qz_*qz_)*asin(sqrt(qx_*qx_+qy_*qy_+qz_*qz_))-
2.0*qz_*qz_/sqrt(pow(qx_*qx_+qy_*qy_+qz_*qz_,3.0))*asin(sqrt(qx_*qx_+qy_*qy_+qz_*qz_))+2.0*qz_*qz_/(qx_*qx_+qy_*qy_+qz_*
qz_)/sqrt(1.0-qx_*qx_-qy_*qy_-qz_~*qz_);      T[2][3] = 0.0;      T[2][4] = 0.0;      T[2][5] = 0.0;      T[3][0] = 0.0;      T[3][1] = 0.0;      T[3][2]
= 0.0;      T[3][3] = 1.0;      T[3][4] = 0.0;      T[3][5] = 0.0;      T[4][0] = 0.0;      T[4][1] = 0.0;      T[4][2] = 0.0;      T[4][3] = 0.0;      T[4][4] =
1.0;      T[4][5] = 0.0;      T[5][0] = 0.0;      T[5][1] = 0.0;      T[5][2] = 0.0;      T[5][3] = 0.0;      T[5][4] = 0.0;      T[5][5] = 1.0;
```

4 kB

```
syms temp0 temp1 temp2 temp3 real
subexpr(J0, 'temp0')
>> temp0 =
>>      1+r00+r11+r22
>> J0 =
>>      [ -1/4*(r21-r12)/temp0^(3/2),                    0, ...
subexpr(ans, 'temp1')
subexpr(ans, 'temp2')
...
J0_ = ans;
ccode(temp0)
ccode(temp1)
ccode(temp2)
ccode(J0_)
```

The output of ccode() IV

```
T.setZero();    temp5 = 1.0+r00+r11+r22;    temp6 = sqrt(temp5);   temp7 = pow(temp5, 3.0/2);
T[0][0] = -(r21-r12)/temp7/4.0;  T[0][4] = -(r21-r12)/temp7/4.0;    T[0][5] = -0.5/temp6;
T[0][7] = 0.5/temp6;    T[0][8] = -(r21-r12)/temp7/4.0;  T[1][0] = -(r02-r20)/temp7/4.0;
T[1][2] = 0.5/temp6;     T[1][4] = -(r02-r20)/temp7/4.0;    T[1][6] = -0.5/temp6;
T[1][8] = -(r02-r20)/temp7/4.0;    T[2][0] = -(r10-r01)/temp7/4.0;     T[2][1] = -0.5/temp6;
T[2][3] = 0.5/temp6;      T[2][4] = -(r10-r01)/temp7/4.0;    T[2][8] = -(r10-r01)/temp7/4.0;
T[3][9] = 1.0;      T[4][10] = 1.0;      T[5][11] = 1.0;

T.setZero();    temp0 = qx_*qx_+qy_*qy_+qz_*qz;    temp1 = sqrt(temp0);
temp2 = asin(temp1);    temp3 = sqrt(1.0-temp0);    temp4 = sqrt(pow(temp0,3.0))*temp2;
T[0][0] = 2.0/temp1*temp2-2.0*qx_*qx_/temp4+2.0*qx_*qx_/(temp0)/temp3;
T[0][1] = -2.0*qx_/temp4*qy_+2.0*qx_/(temp0)*qy_/temp3;
T[0][2] = -2.0*qx_/temp4*qz_+2.0*qx_/(temp0)*qz_/temp3;
T[1][0] = -2.0*qx_/temp4*qy_+2.0*qx_/(temp0)*qy_/temp3;
T[1][1] = 2.0/temp1*temp2-2.0*qy_*qy_/temp4+2.0*qy_*qy_/(temp0)/temp3;
T[1][2] = -2.0*qy_/temp4*qz_+2.0*qy_/(temp0)*qz_/temp3;
T[2][0] = -2.0*qx_/temp4*qz_+2.0*qx_/(temp0)*qz_/temp3;
T[2][1] = -2.0*qy_/temp4*qz_+2.0*qy_/(temp0)*qz_/temp3;
T[2][2] = 2.0/temp1*temp2-2.0*qz_*qz_/temp4+2.0*qz_*qz_/(temp0)/temp3;
T[3][3] = 1.0;      T[4][4] = 1.0;      T[5][5] = 1.0;
```

1 kB

- Much easier for fast prototyping
- Typically not horribly imprecise but can be slow

```
Vector6d x1, x2; // the two variables

Vector6d expectation = x1 ⊖ x2; // the function we're differentiating

Matrix6d J1; // derivative w.r.t. x1
for(int i = 0; i < 6; ++ i) {
        Vector6d eps;
        eps.setZero(); eps(i) = 1e-9;
        Vector6d shift1 = eps ⊕ x1; // apply infinitesimal shift (ordering!)
        Vector6d value = shift1 ⊖ x2; // see how that changes the output
        J1.col(i) = (value - expectation) * 1e+9; // note cwise op
}
// do the same for x2
```

- Also good for prototyping but quite slow
- A bit like complex numbers, with special semantics (while $i^2 = -1$, we use $e^2 = 0$)
- Consider $f(x) = x^2$, inject $y = x + e$
- Evaluate using our "complex" arithmetics
  $f(y) = (x + e)^2 = x^2 + 2xe + e^2 = x^2 + 2xe$

  value derivative
- For functions of multiple arguments, we need to put e in each argument separately and re-evaluate several times
- Rules for multiplication, division, transcendentals (can derive from Taylor series)

Lie Group Basics

- Groups on differentiable manifolds
- Lie group SE(3) and associated algebra $\mathfrak{se}(3)$
- SE(3) maps to Euclidean space $\mathbb{E}^3$ ($[\mathrm{R}\ \mathrm{t}]$ are in it)
- $\mathfrak{se}(3)$ is the *tangent space* of real skew-symmetric 4×4 matrices
- *Exponential map* goes from $\mathfrak{se}(3)$ to SE(3)
- *Logarithmic map* goes from SE(3) to $\mathfrak{se}(3)$
- *Vectorial operator** $[\mathbf{x}]_\vee$ packs skew-sym to vec
- *Cross operator* $[\mathbf{x}]_\times$ goes back to skew-sym
- *Lie bracket* $[\mathbf{x}, \mathbf{y}] = \mathbf{xy} - \mathbf{yx}$ (where $x, y \in \mathfrak{se}(3)$)

*several different notations exist

- Let's have skew symmetric $\mathfrak{so}(3)$ matrix A

$$\boldsymbol{v} = [a\ b\ c],\ [\boldsymbol{v}]_\times = A = \begin{bmatrix} 0 & -c & b \\ c & 0 & -a \\ -b & a & 0 \end{bmatrix}$$

- Now, thinking about matrix exponent

$$R = e^A = I + \sum_{i=1} \frac{A^i}{i!} \text{ with } A^0 = I$$

- So exponential map of $\mathfrak{so}(3)$ yields R, which is *orthogonal* (i.e. rotation matrix, in SO(3))
- There's also a logarithm of matrix

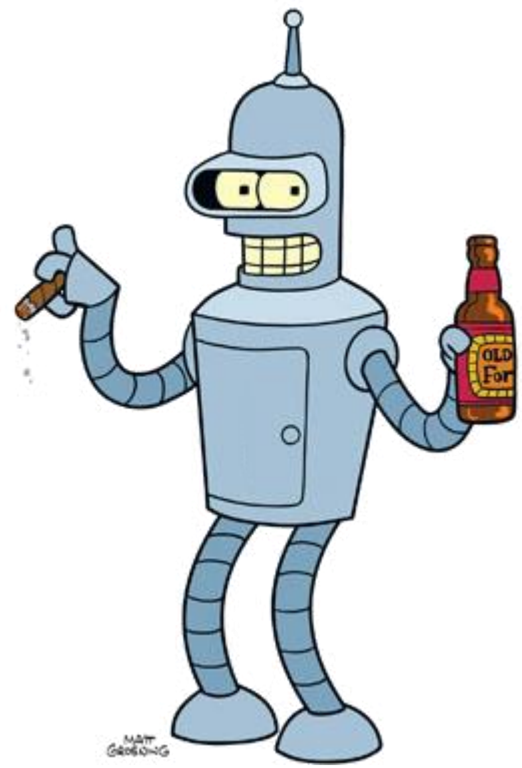
- Following

$$\boldsymbol{v} = [a\ b\ c],\ [\boldsymbol{v}]_\times = A = \begin{bmatrix} 0 & -c & b \\ c & 0 & -a \\ -b & a & 0 \end{bmatrix}$$

$$B = \boldsymbol{v}\boldsymbol{v}^T = A^2 = \begin{bmatrix} a^2 & ab & ac \\ ab & b^2 & bc \\ ac & bc & c^2 \end{bmatrix}$$

$$e^A = \cos(\theta)\, I + \frac{\sin(\theta)}{\theta} A + \frac{1-\cos(\theta)}{\theta^2} B$$

- Seems familiar? :)

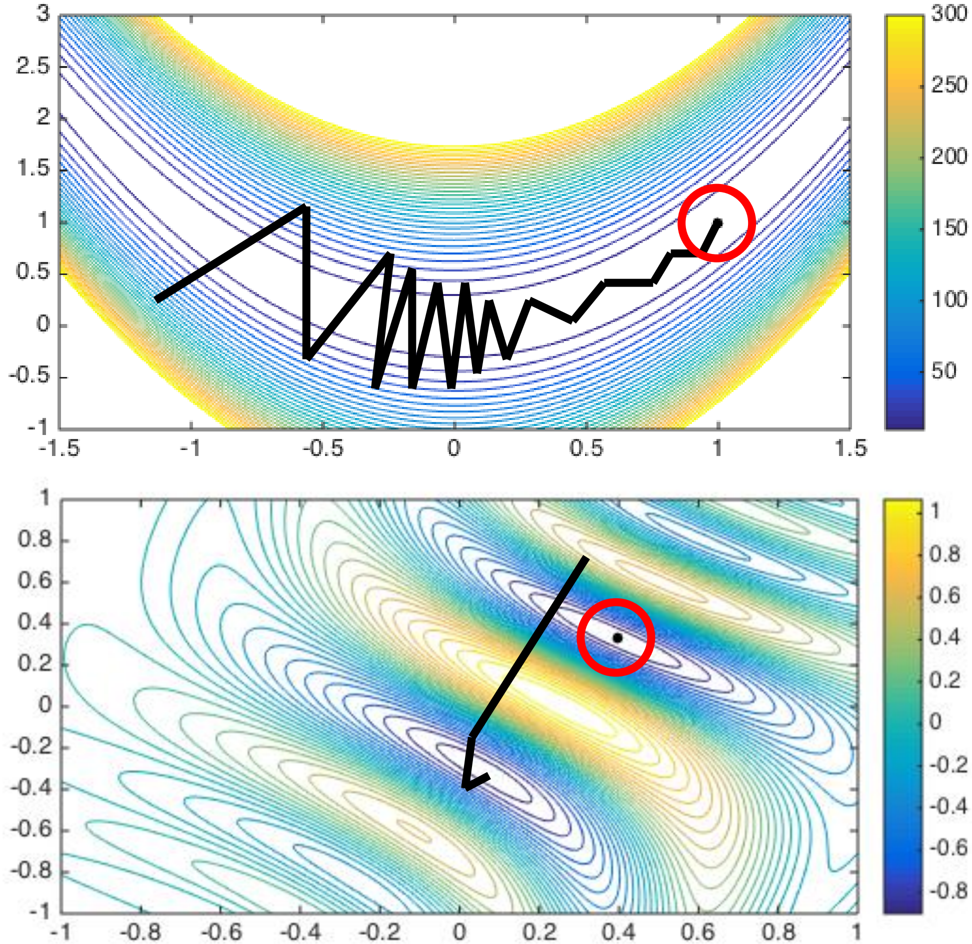


- Going back to derivatives, we can calculate derivatives of exp, log, (inverse) compose, just like we did for $\oplus$, $\ominus$, $\mathbf{v}(\cdot)$ and $\mathbf{m}(\cdot)$
- Much of that has nice closed form
- As for least squares, we choose (vectorial) $\mathfrak{se}(3)$ or $\mathfrak{sim}(3)$ as the internal representation
- See Tom Drummond's TooN library for gritty details and code (https://www.edwardrosten.com/cvd/toon/html-user/)

And now …

BACK TO BUNDLING

- Gauss-Newton only computes step from *local* gradient, never looks back



Rosenbrock

Chebyshev

This would happen more with gradient descent. This is illustrative.

- Several algorithms that address this
- Levenberg-Marquardt
- Recall ordinary NLS

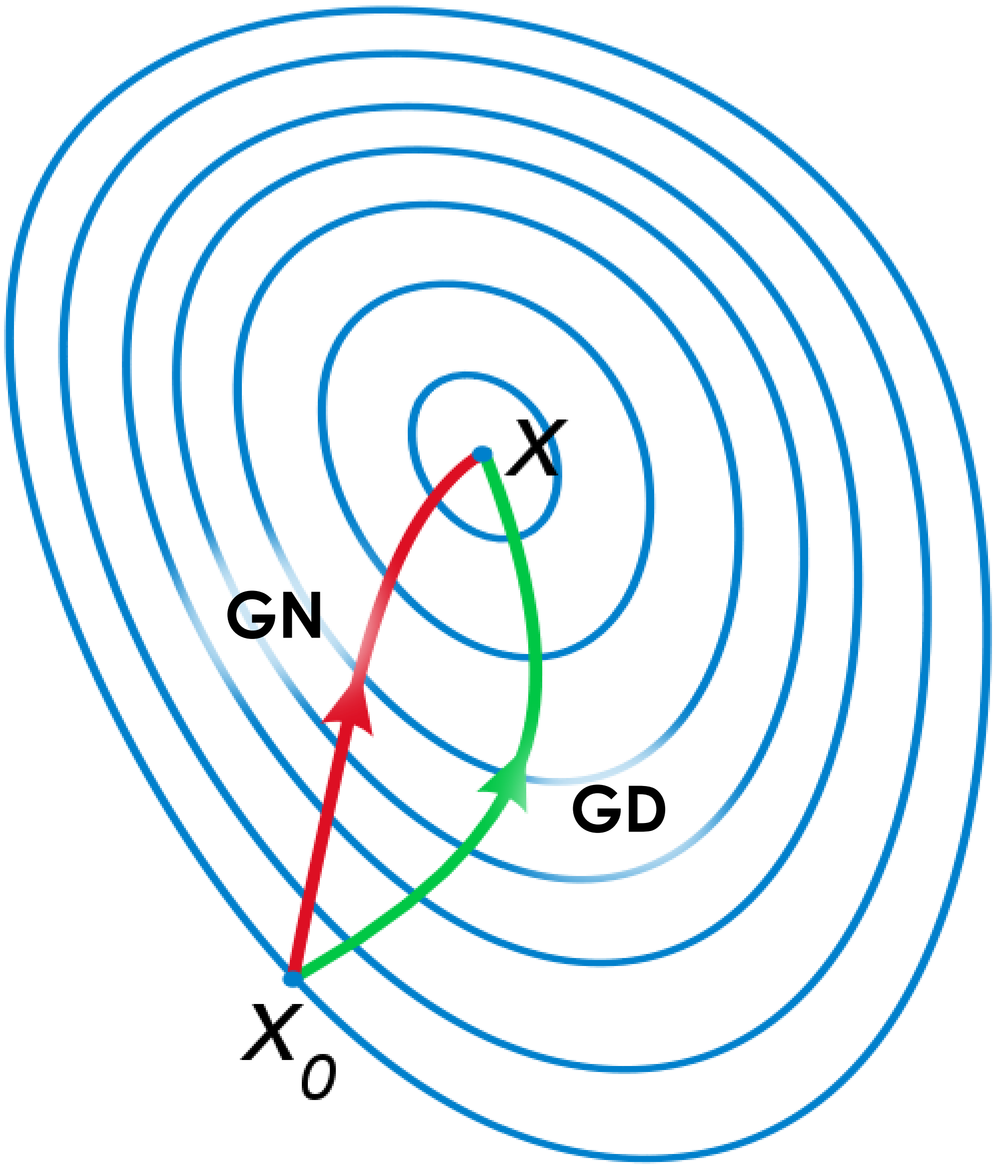
$$\boldsymbol{J}^T\boldsymbol{J}\Delta\boldsymbol{x} = \boldsymbol{J}^T\Delta\boldsymbol{b}$$

  - We're more or less doing $j^2x = jb \sim jx = b$
  - We can control step size by magnitude of j
- Levenberg-Marquardt NLS

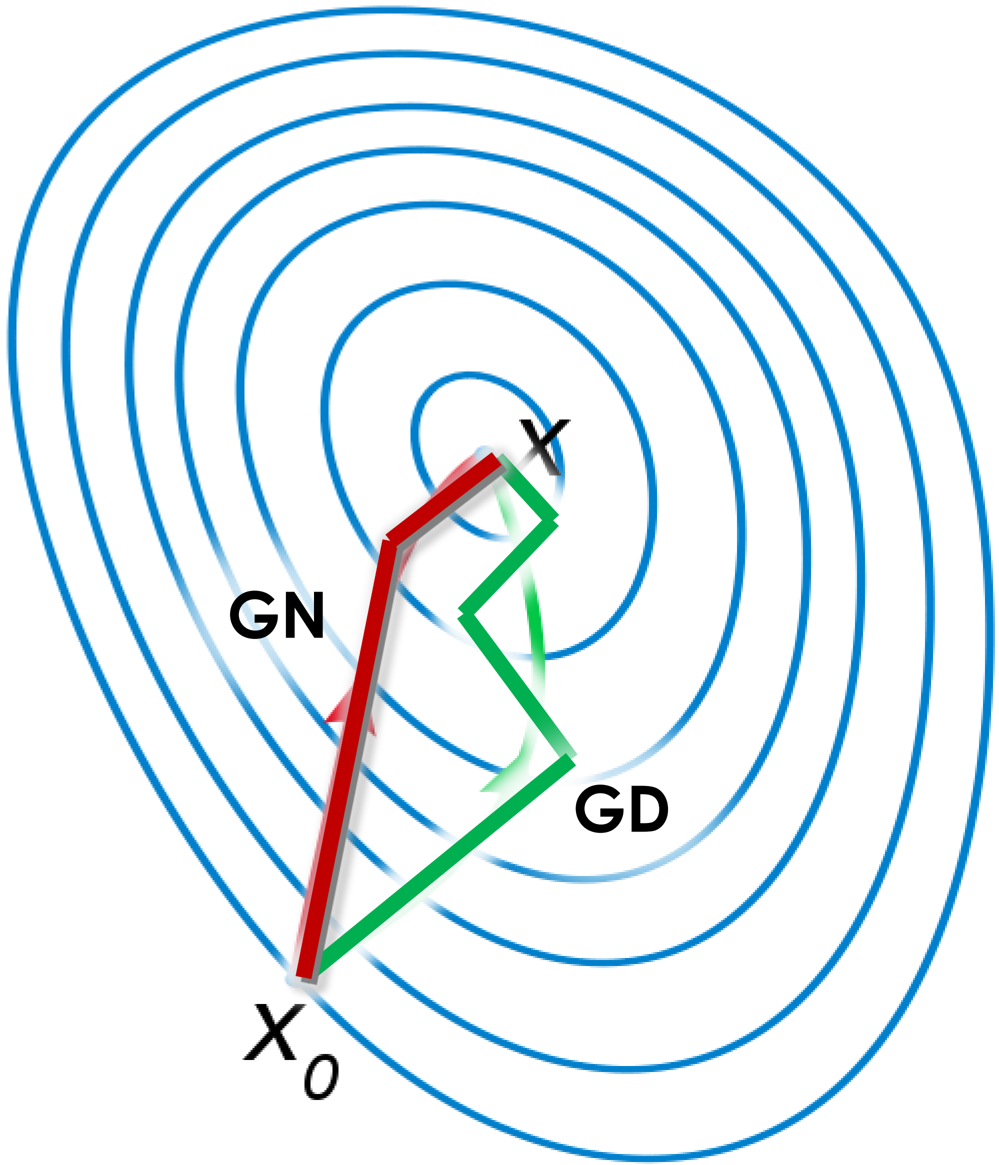
$$\left(\boldsymbol{J}^T\boldsymbol{J} + \alpha\boldsymbol{K}\right)\Delta\boldsymbol{x} = \boldsymbol{J}^T\Delta\boldsymbol{b}$$

  with common choices $\boldsymbol{K} = \boldsymbol{I}$ or $\boldsymbol{K} = \mathrm{diag}\left(\boldsymbol{J}^T\boldsymbol{J}\right)$
- Changing the value of α inversely proportionally controls step size and *direction* choosing between GN and steepest descent

- GN typically converges faster
- GD less prone to get stuck but tends to zigzag a lot

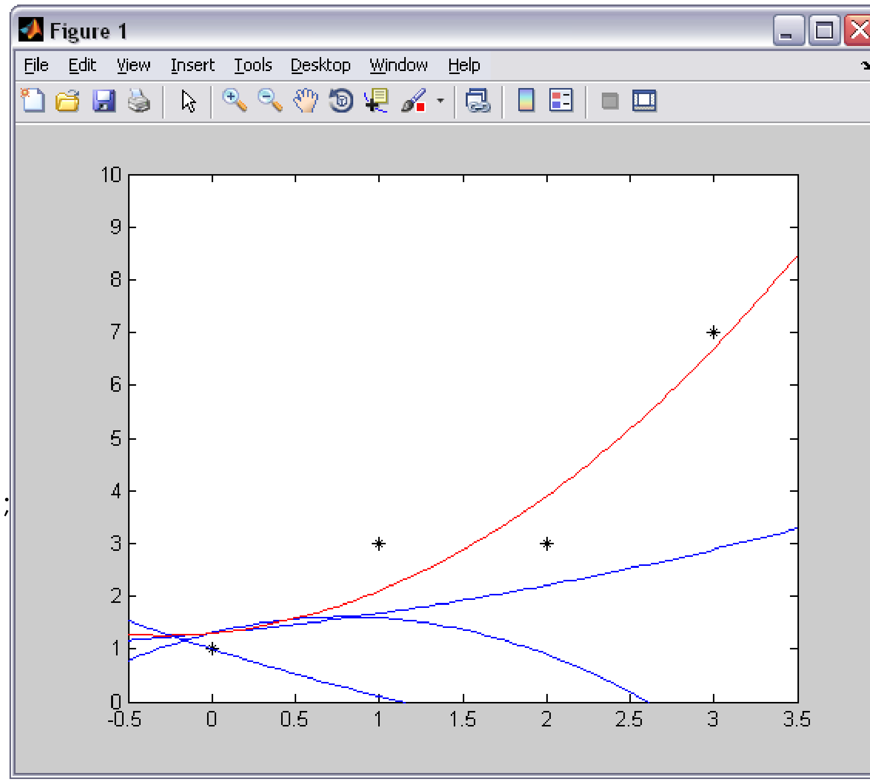


- GN typically converges faster
- GD less prone to get stuck but tends to zigzag a lot



Effects of Levenberg-Marquardt Damping

```
O = [0 1; 1 3; 2 3; 3 7];                         % we are trying to estimate b = p + pqa + pqra^2
% observations of some 1D function                % where x = [p q r] is our unknown
% O is vector of pairs arg. a and desired fit b   % h(x) = p + pqa + pqra^2, J = dh(x) / dx = [1+qa+qra^2, pa+pra^2, qa^2]
                                                  % we use this parameterization to make it nonlinear (would work with linear
x = [1 -1 -.1]';                                  % too but would be able to optimize in a single step, which would be boring)
% guess 0x (deliberately a bad guess, to take a few steps)

plot(O(:, 1), O(:, 2), '*k') % plot a-s, b-s
hold on
xx = linspace(min(O(:, 1)) - .5, max(O(:, 1)) + .5);
for i = 1:100
   yy = x(1) + x(1) * x(2) * xx + x(1) * x(2) * x(3) * (xx.^2);
   plot(xx, yy, '-b') % plot the initial guess
   % evaluate the initial guess

   J = [1 + x(2) * O(:, 1) + x(2) * x(3) * O(:, 1).^2, ...
      x(1) * O(:, 1) + x(1) * x(3) * O(:, 1).^2, x(1) * x(2) * O(:, 1).^2];
   % calculate the Jacobian

   db = O(:,2) - x(1) - O(:,1) * x(1) * x(2) - O(:,1).^2 * x(1) * x(2) * x(3);
   % calculate current error vector

   dx = (J' * J + 0 * eye(size(J, 2))) \ J' * db; norm_dx = norm(dx)
   % solve

   if(norm_dx < 1e-6) % see if we optimize
      break
   end
   x = x + dx % increment
end

yy = x(1) + x(1) * x(2) * xx + x(1) * x(2) * x(3) * (xx.^2);
plot(xx, yy, '-r') % plot the final in red
hold off
```

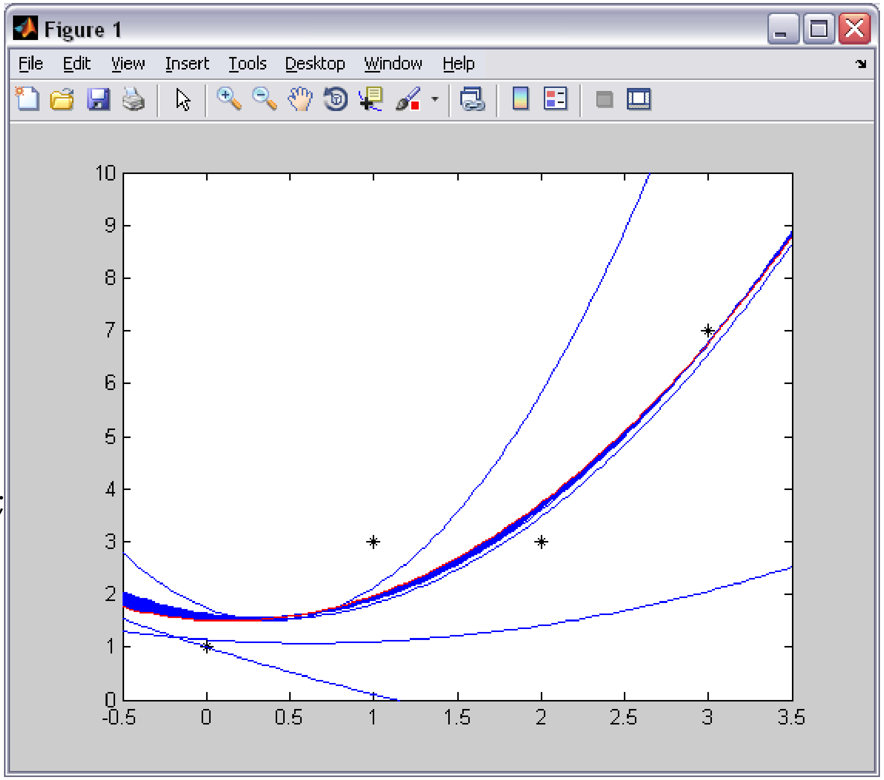


Effects of Levenberg-Marquardt Damping

```
O = [0 1; 1 3; 2 3; 3 7];
% observations of some 1D function
% O is vector of pairs arg. a and desired fit b
```

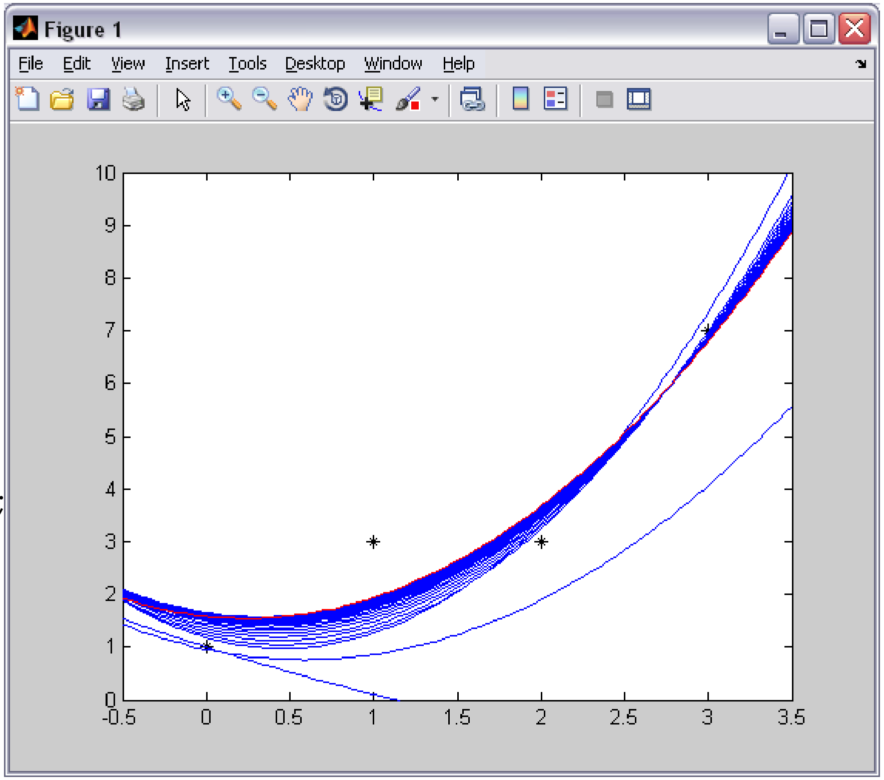
```
% we are trying to estimate b = p + pqa + pqra^2
% where x = [p q r] is our unknown
% h(x) = p + pqa + pqra^2, J = dh(x) / dx = [1+qa+qra^2, pa+pra^2, qa^2]
% we use this parameterization to make it nonlinear (would work with linear
% too but would be able to optimize in a single step, which would be boring)
```

```
x = [1 -1 -.1]';
% guess ^0x (deliberately a bad guess, to take a few steps)

plot(O(:, 1), O(:, 2), '*k') % plot a-s, b-s
hold on
xx = linspace(min(O(:, 1)) - .5, max(O(:, 1)) + .5);
for i = 1:100
   yy = x(1) + x(1) * x(2) * xx + x(1) * x(2) * x(3) * (xx.^2);
   plot(xx, yy, '-b') % plot the initial guess
   % evaluate the initial guess

   J = [1 + x(2) * O(:, 1) + x(2) * x(3) * O(:, 1).^2, ...
      x(1) * O(:, 1) + x(1) * x(3) * O(:, 1).^2, x(1) * x(2) * O(:, 1).^2];
   % calculate the Jacobian

   db = O(:,2) - x(1) - O(:,1) * x(1) * x(2) - O(:,1).^2 * x(1) * x(2) * x(3);
   % calculate current error vector

   dx = (J' * J + 1 * eye(size(J, 2))) \ J' * db; norm_dx = norm(dx)
   % solve

   if(norm_dx < 1e-6) % see if we optimize
      break
   end
   x = x + dx % increment
end

yy = x(1) + x(1) * x(2) * xx + x(1) * x(2) * x(3) * (xx.^2);
plot(xx, yy, '-r') % plot the final in red
hold off
```



BRNO UNIVERSITY OF TECHNOLOGY

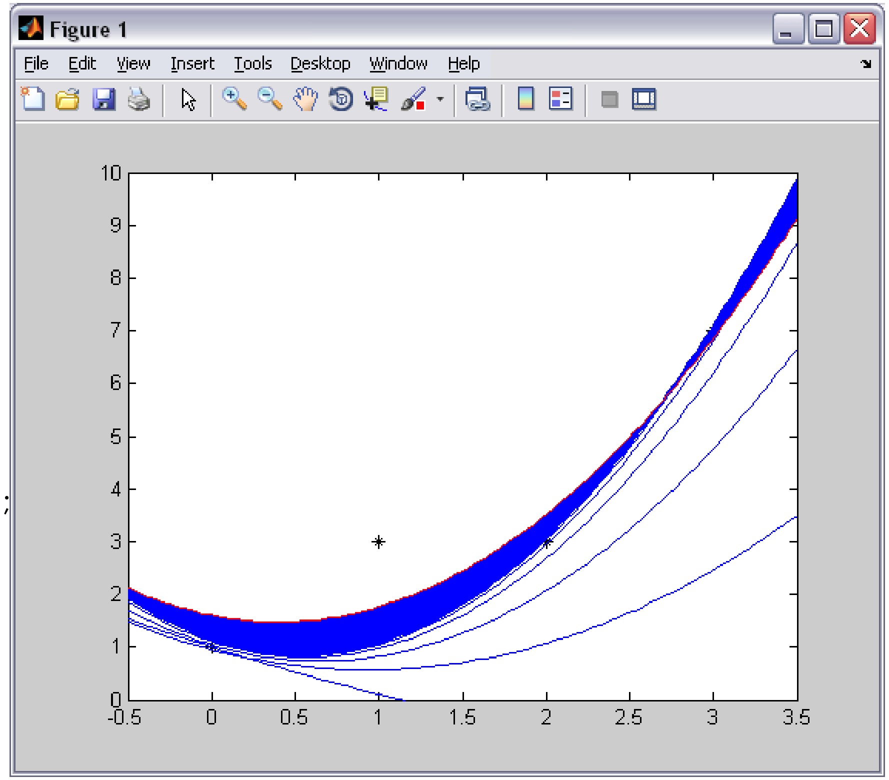
```
O = [0 1; 1 3; 2 3; 3 7];
% observations of some 1D function
% O is vector of pairs arg. a and desired fit b

% we are trying to estimate b = p + pqa + pqra^2
% where x = [p q r] is our unknown
% h(x) = p + pqa + pqra^2, J = dh(x) / dx = [1+qa+qra^2, pa+pra^2, qa^2]
% we use this parameterization to make it nonlinear (would work with linear
% too but would be able to optimize in a single step, which would be boring)

x = [1 -1 -.1]';
% guess 0x (deliberately a bad guess, to take a few steps)

plot(O(:, 1), O(:, 2), '*k') % plot a-s, b-s
hold on
xx = linspace(min(O(:, 1)) - .5, max(O(:, 1)) + .5);
for i = 1:100
   yy = x(1) + x(1) * x(2) * xx + x(1) * x(2) * x(3) * (xx.^2);
   plot(xx, yy, '-b') % plot the initial guess
   % evaluate the initial guess

   J = [1 + x(2) * O(:, 1) + x(2) * x(3) * O(:, 1).^2, ...
      x(1) * O(:, 1) + x(1) * x(3) * O(:, 1).^2, x(1) * x(2) * O(:, 1).^2];
   % calculate the Jacobian

   db = O(:,2) - x(1) - O(:,1) * x(1) * x(2) - O(:,1).^2 * x(1) * x(2) * x(3);
   % calculate current error vector

   dx = (J' * J + 10 * eye(size(J, 2))) \ J' * db; norm_dx = norm(dx)
   % solve

   if(norm_dx < 1e-6) % see if we optimize
      break
   end
   x = x + dx % increment
end

yy = x(1) + x(1) * x(2) * xx + x(1) * x(2) * x(3) * (xx.^2);
plot(xx, yy, '-r') % plot the final in red
hold off
```



Effects of Levenberg-Marquardt Damping

```
O = [0 1; 1 3; 2 3; 3 7];
% observations of some 1D function
% O is vector of pairs arg. a and desired fit b

x = [1 -1 -.1]';
% guess ^0x (deliberately a bad guess, to take a few steps)

plot(O(:, 1), O(:, 2), '*k') % plot a-s, b-s
hold on
xx = linspace(min(O(:, 1)) - .5, max(O(:, 1)) + .5);
for i = 1:100
   yy = x(1) + x(1) * x(2) * xx + x(1) * x(2) * x(3) * (xx.^2);
   plot(xx, yy, '-b') % plot the initial guess
   % evaluate the initial guess

   J = [1 + x(2) * O(:, 1) + x(2) * x(3) * O(:, 1).^2, ...
      x(1) * O(:, 1) + x(1) * x(3) * O(:, 1).^2, x(1) * x(2) * O(:, 1).^2];
   % calculate the Jacobian

   db = O(:,2) - x(1) - O(:,1) * x(1) * x(2) - O(:,1).^2 * x(1) * x(2) * x(3);
   % calculate current error vector

   dx = (J' * J + 100 * eye(size(J, 2))) \ J' * db; norm_dx = norm(dx)
   % solve

   if(norm_dx < 1e-6) % see if we optimize
      break
   end
   x = x + dx % increment
end

yy = x(1) + x(1) * x(2) * xx + x(1) * x(2) * x(3) * (xx.^2);
plot(xx, yy, '-r') % plot the final in red
hold off
```

```
% we are trying to estimate b = p + pqa + pqra^2
% where x = [p q r] is our unknown
% h(x) = p + pqa + pqra^2, J = dh(x) / dx = [1+qa+qra^2, pa+pra^2, qa^2]
% we use this parameterization to make it nonlinear (would work with linear
% too but would be able to optimize in a single step, which would be boring)
```

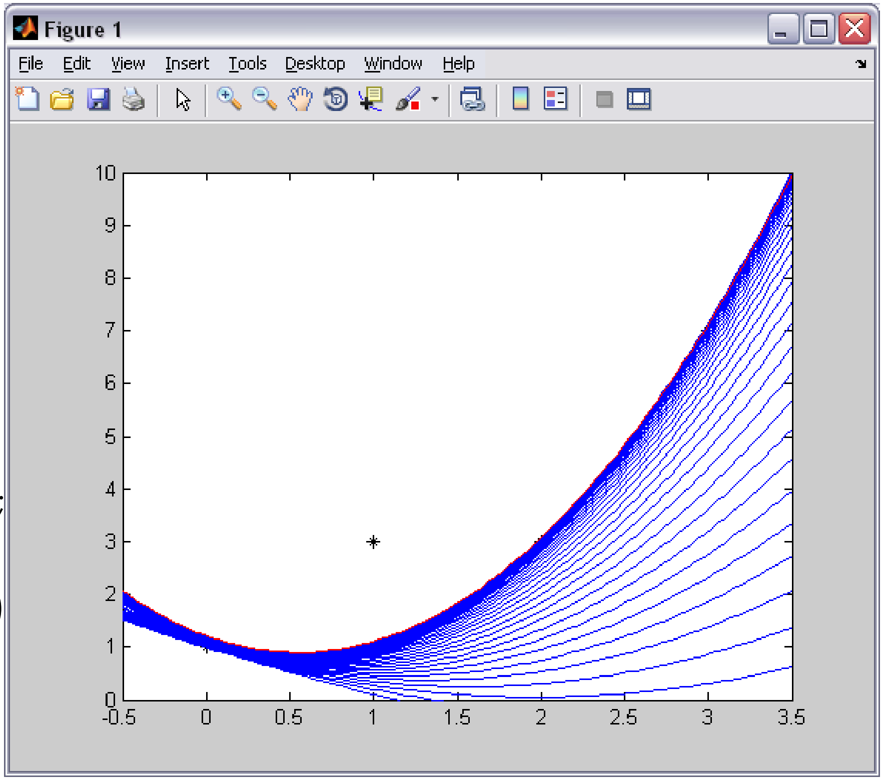


Effects of Levenberg-Marquardt Damping

```
O = [0 1; 1 3; 2 3; 3 7];
% observations of some 1D function
% O is vector of pairs arg. a and desired fit b

x = [1 -1 -.1]';
% guess ^0x (deliberately a bad guess, to take a few steps)
```

```
% we are trying to estimate b = p + pqa + pqra^2
% where x = [p q r] is our unknown
% h(x) = p + pqa + pqra^2, J = dh(x) / dx = [1+qa+qra^2, pa+pra^2, qa^2]
% we use this parameterization to make it nonlinear (would work with linear
% too but would be able to optimize in a single step, which would be boring)
```

```
plot(O(:, 1), O(:, 2), '*k') % plot a-s, b-s
hold on
xx = linspace(min(O(:, 1)) - .5, max(O(:, 1)) + .5);
for i = 1:100
   yy = x(1) + x(1) * x(2) * xx + x(1) * x(2) * x(3) * (xx.^2);
   plot(xx, yy, '-b') % plot the initial guess
   % evaluate the initial guess

   J = [1 + x(2) * O(:, 1) + x(2) * x(3) * O(:, 1).^2, ...
      x(1) * O(:, 1) + x(1) * x(3) * O(:, 1).^2, x(1) * x(2) * O(:, 1).^2];
   % calculate the Jacobian

   db = O(:,2) - x(1) - O(:,1) * x(1) * x(2) - O(:,1).^2 * x(1) * x(2) * x(3);
   % calculate current error vector

   dx = (J' * J + 1000 * eye(size(J, 2))) \ J' * db; norm_dx = norm(dx)
   % solve

   if(norm_dx < 1e-6) % see if we optimize
      break
   end
   x = x + dx % increment
end

yy = x(1) + x(1) * x(2) * xx + x(1) * x(2) * x(3) * (xx.^2);
plot(xx, yy, '-r') % plot the final in red
hold off
```

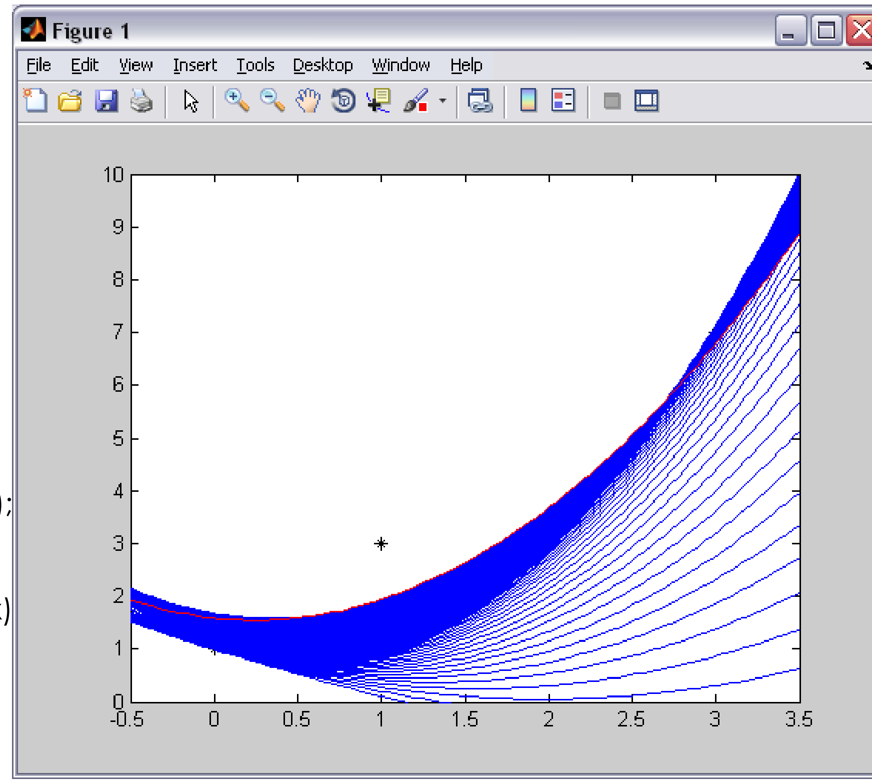


Effects of Levenberg-Marquardt Damping

```
O = [0 1; 1 3; 2 3; 3 7];
% observations of some 1D function
% O is vector of pairs arg. a and desired fit b

x = [1 -1 -.1]';
% guess 0x (deliberately a bad guess, to take a few steps)
```

```
% we are trying to estimate b = p + pqa + pqra^2
% where x = [p q r] is our unknown
% h(x) = p + pqa + pqra^2, J = dh(x) / dx = [1+qa+qra^2, pa+pra^2, qa^2]
% we use this parameterization to make it nonlinear (would work with linear
% too but would be able to optimize in a single step, which would be boring)
```

```
plot(O(:, 1), O(:, 2), '*k') % plot a-s, b-s
hold on
xx = linspace(min(O(:, 1)) - .5, max(O(:, 1)) + .5);
for i = 1:1000
   yy = x(1) + x(1) * x(2) * xx + x(1) * x(2) * x(3) * (xx.^2);
   plot(xx, yy, '-b') % plot the initial guess
   % evaluate the initial guess

   J = [1 + x(2) * O(:, 1) + x(2) * x(3) * O(:, 1).^2, ...
      x(1) * O(:, 1) + x(1) * x(3) * O(:, 1).^2, x(1) * x(2) * O(:, 1).^2];
   % calculate the Jacobian

   db = O(:,2) - x(1) - O(:,1) * x(1) * x(2) - O(:,1).^2 * x(1) * x(2) * x(3);
   % calculate current error vector

   dx = (J' * J + 1000 * eye(size(J, 2))) \ J' * db; norm_dx = norm(dx);
   % solve

   if(norm_dx < 1e-6) % see if we optimize
      break
   end
   x = x + dx % increment
end

yy = x(1) + x(1) * x(2) * xx + x(1) * x(2) * x(3) * (xx.^2);
plot(xx, yy, '-r') % plot the final in red
hold off
```



Levenberg-Marquardt

- To control step size, use *optimization gain*
$$g(\Delta\boldsymbol{x}) = \frac{S(\boldsymbol{x}) - S(\boldsymbol{x} + \Delta\boldsymbol{x})}{L(0) - L(\Delta\boldsymbol{x})}$$
with linear model error $L(\Delta\boldsymbol{x}) = \frac{1}{2}||h(\boldsymbol{x}) + \boldsymbol{J}\Delta\boldsymbol{x}||^2$
and squared error $S(\widehat{\boldsymbol{x}}) = \sum h(\widehat{\boldsymbol{x}})^{\boldsymbol{2}}$
- If the gain is positive, take the step, reduce α
- If the gain is negative, keep the current linearization and increase α at exponential rate with each failed step

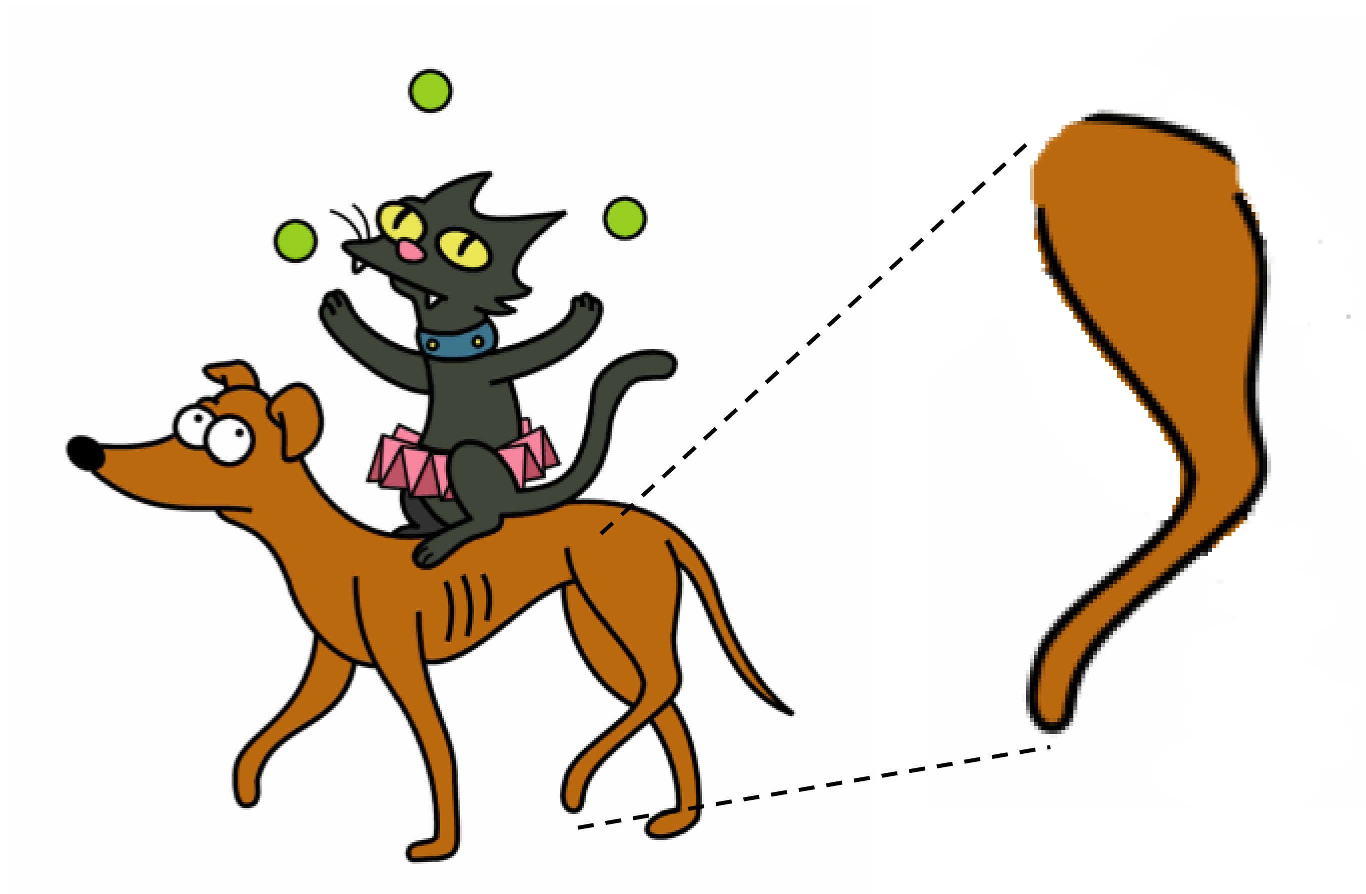
- See [Madsen, 1999, Methods for NLS Problems] for further details

- Quick, **shout** answer to **WIN SAUSAGE**!

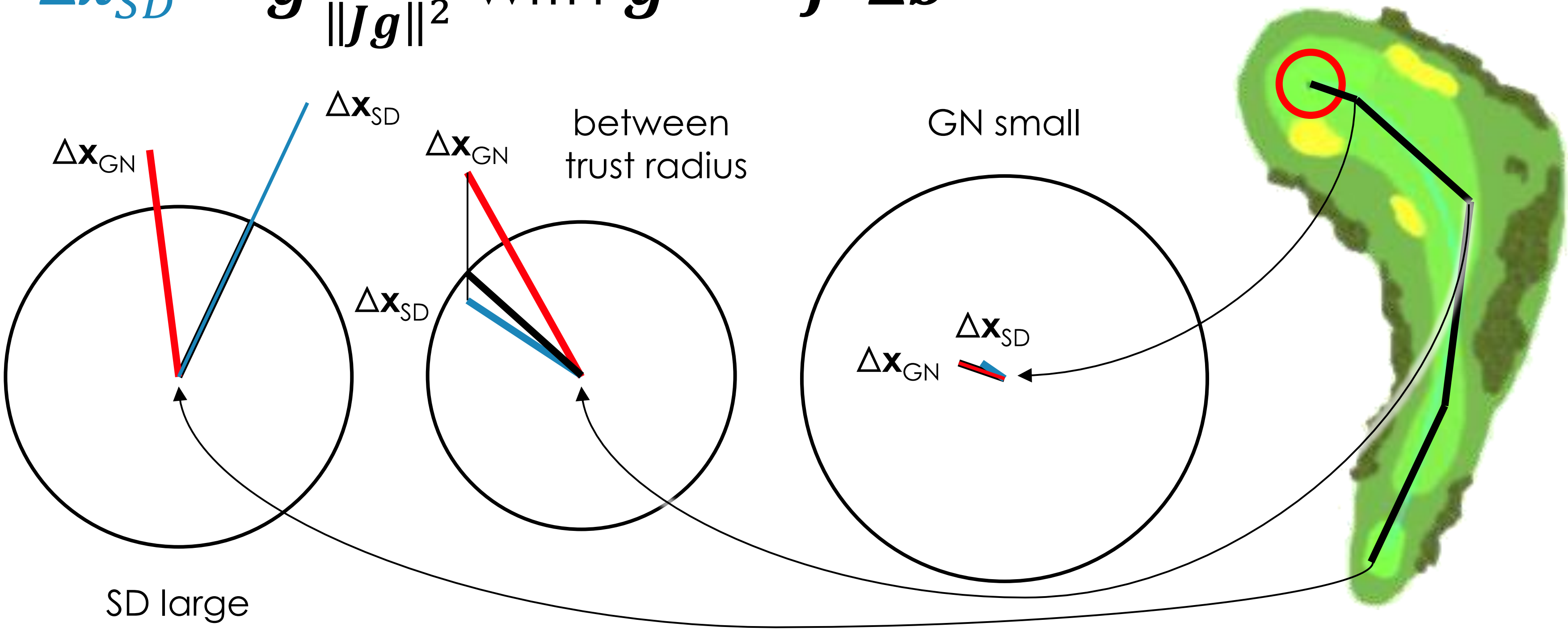
- Mike Powell was:
  A. Cat lady
  B. Hotdog eating champion
  C. Golfer
  D. Computer Scienceman

Dogleg optimizer

- Powell’s Dogleg (there’s also subspace Dogleg)



- Change gradient direction based on magnitude of $\Delta\boldsymbol{x}$, generate step of specified size
- In solving the ordinary normal equation $\boldsymbol{J}^T\boldsymbol{J}\Delta\boldsymbol{x}_{GN} = \boldsymbol{J}^T\Delta\boldsymbol{b}$ we can recover $\Delta\boldsymbol{x}_{GN}$ and also $\Delta\boldsymbol{x}_{SD} = \boldsymbol{g}\frac{\|\boldsymbol{g}\|^2}{\|\boldsymbol{J}\boldsymbol{g}\|^2}$ with $\boldsymbol{g} = -\boldsymbol{J}^T\Delta\boldsymbol{b}$



- Formally, for trust radius $\underline{\Delta}$, the step is

$$\Delta\boldsymbol{x}_{DL} = \begin{cases} \Delta\boldsymbol{x}_{GN} & if\ \|\Delta\boldsymbol{x}_{GN}\| \leq \underline{\Delta} \\ \underline{\Delta} \cdot \Delta\boldsymbol{x}_{SD} \big/ \|\Delta\boldsymbol{x}_{SD}\| & if\ \|\Delta\boldsymbol{x}_{SD}\| \geq \underline{\Delta} \\ \Delta\boldsymbol{x}_{SD} + \beta(\Delta\boldsymbol{x}_{GN} - \Delta\boldsymbol{x}_{SD}) & otherwise \end{cases}$$

where $\beta = \frac{-b+\sqrt{b^2-4ac}}{2a}$ with $a = \|\Delta\boldsymbol{x}_{GN} - \Delta\boldsymbol{x}_{SD}\|^2$, $b = \Delta\boldsymbol{x}_{SD}{}^T(\Delta\boldsymbol{x}_{GN} - \Delta\boldsymbol{x}_{SD})$ and $c = \|\Delta\boldsymbol{x}_{SD}\|^2 - \underline{\Delta}^2$

- $\underline{\Delta}$ is changed based on optimization gain
- The initial trust radius can be large (unlike LM, DL does not resolve the lin. system on bad step)

Handling Outliers – Robust Estimation

- Outliers are a problem in computer vision
  - Bad feature matching (mismatched feature)
  - Reflections (matched to a good feature in a mirror)
  - Moving objects (cars, pedestrians, wind)
- Can try to reject suspicious observations
- Can use robust estimation

- Try to identify outliers in the LS framework
- LS minimize *squared* error

$$\boldsymbol{x} = \operatorname{argmin}_{\boldsymbol{x}} \frac{1}{2} \sum \left(b - h(x)\right)^{2}$$

big outliers have huge (squared) influence

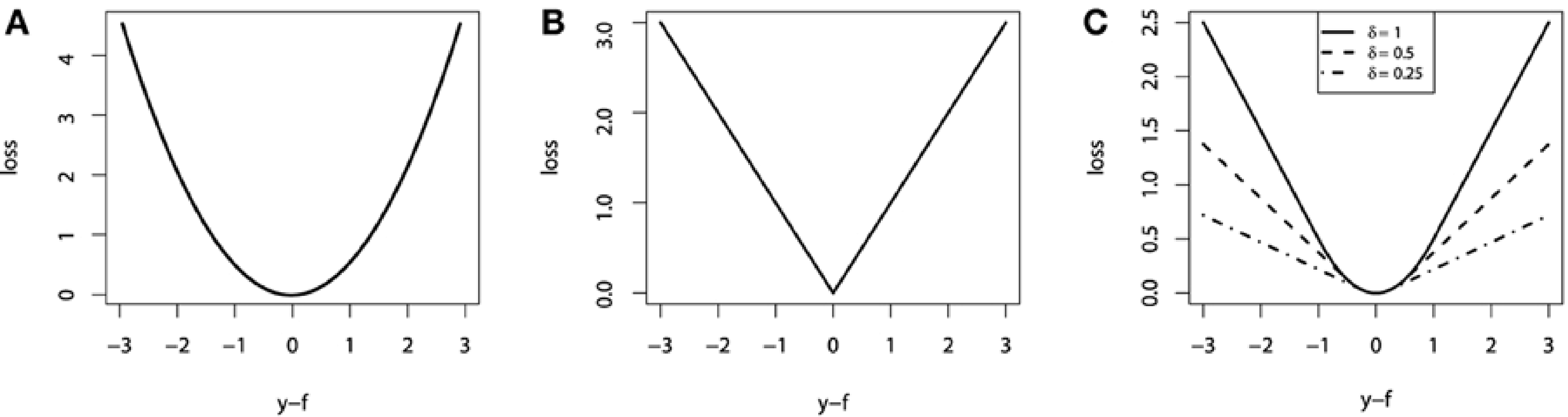
- Try to generalize the error

$$\boldsymbol{x} = \operatorname{argmin}_{\boldsymbol{x}} \sum \rho\left(b - h(x)\right)$$

where $\rho(u) = \frac{1}{2} u^2$ is ordinary LS *loss function*

- Try to minimize pseudo-L1 error

$$\rho(u) = \begin{cases} \frac{1}{2}u^2 & if\ |u| \le a \\ \frac{1}{2}a(2|u| - a) & otherwise \end{cases}$$



Huber kernel

Normal Equations for Robust Estimation

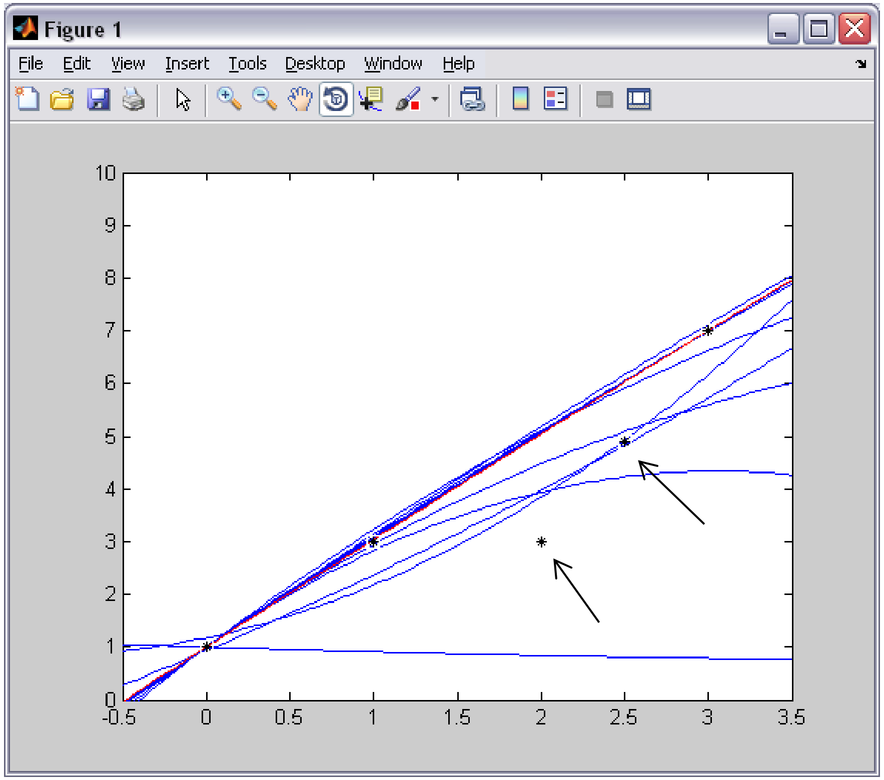
- So how to plug it in? Notice that for LS, $\rho(u) = \frac{1}{2}u^2$ and $\rho'(u) = 1$
- The derivative of loss is *score function* $\psi(u) = \rho'(u)$
- This can be used for weights,

$$\boldsymbol{J}^T\boldsymbol{W}\boldsymbol{J}\Delta\boldsymbol{x}_{GN} = \boldsymbol{J}^T\boldsymbol{W}\Delta\boldsymbol{b}$$

where $\boldsymbol{W} = \mathrm{diag}\left(\frac{\psi(\boldsymbol{u})}{\boldsymbol{u}}\right)$

- Let's try setting $\boldsymbol{u} = \Delta\boldsymbol{b}$

```
O = [0 1; 1 3; 2 3; 2.5 4.9; 3 7];
% observations of some 1D function
% O(:,1) are the arguments, a
% O(:,2) are the desired fit values, b

a = 1.345;
% guess some parameter for outlier rejection

for i = 1:10
   J = ... % calculate the Jacobian

   db = ... % calculate current error vector

   w = zeros(length(db), 1);
   for j = 1:length(db)
      u = db(j);
      if(abs(u) <= a)
         w(j) = 1 / u; % resolve division by zero
      else
         w(j) = a * sign(u) / u;
      end
   end
   W = diag(w); % calculate the robust weights

   dx = (J' * W * J) \ J' * W * db;
   % solve normal equation (backslash = solve)
end

% [plot stuff]
```

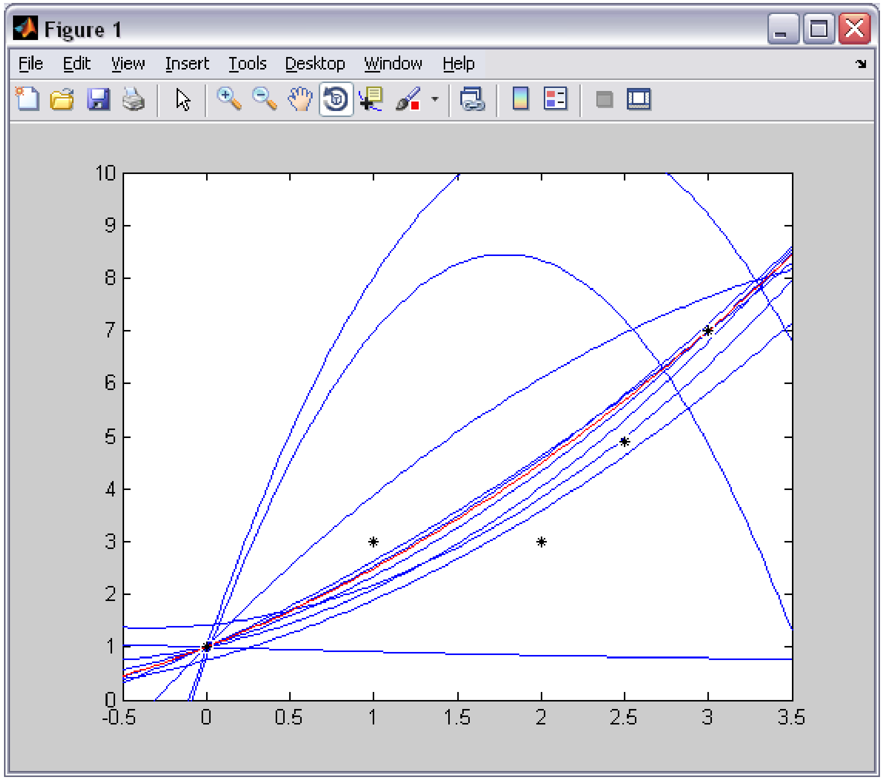


- Now we‘re overfitting, the points at the bottom are all *treated as* outliers
- Changing the value of a is a temporary solution
- We need to approximate *scale of the problem*
- MAD (median absolute deviation)
  $s = 1.4826 \operatorname{med}\left(\operatorname{abs}(\Delta\boldsymbol{b})\right)$
- Now set $\boldsymbol{u} = \frac{\Delta\boldsymbol{b}}{s}$

- Alternative to MAD - Huber‘s second proposal

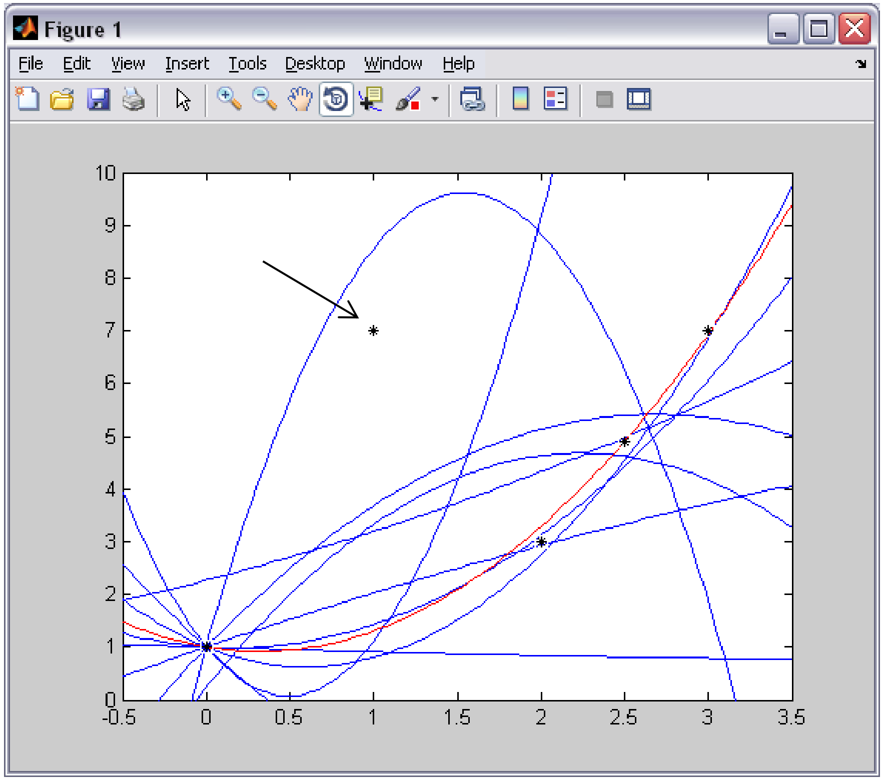
BRNO UNIVERSITY OF TECHNOLOGY

```
O = [0 1; 1 3; 2 3; 2.5 4.9; 3 7];
% observations of some 1D function
% O(:,1) are the arguments, a
% O(:,2) are the desired fit values, b

a = 1.345;
% guess some parameter for outlier rejection

for i = 1:10
   % [calculate J, db]

   MAD = median(abs(db));
   s = 1.4826 * MAD;
   % calculate MAD

   w = zeros(length(db), 1);
   for j = 1:length(db)
      u = db(j) / s; % apply scale
      if(abs(u) <= a)
         w(j) = 1 / u; % resolve division by zero
      else
         w(j) = a * sign(u) / u;
      end
   end
   W = diag(w); % calculate the robust weights

   % [solve]
end

% [plot stuff]
```



BRNO UNIVERSITY OF TECHNOLOGY

```
O = [0 1; 1 7; 2 3; 2.5 4.9; 3 7]; % turrn the second point into an OUTLIER
% observations of some 1D function
% O(:,1) are the arguments, a
% O(:,2) are the desired fit values, b

a = 1.345;
% guess some parameter for outlier rejection

for i = 1:10
   % [calculate J, db]

   MAD = median(abs(db));
   s = 1.4826 * MAD;
   % calculate MAD

   w = zeros(length(db), 1);
   for j = 1:length(db)
      u = db(j) / s; % apply scale
      if(abs(u) <= a)
         w(j) = 1 / u; % resolve division by zero
      else
         w(j) = a * sign(u) / u;
      end
   end
   W = diag(w); % calculate the robust weights

   % [solve]
end

% [plot stuff]
```



- What are the magic numbers (1.4826, 1.345)?
- Relative estimator efficiency

$$e = \frac{\mathrm{E}\left((T_2 - \boldsymbol{x})^2\right)}{\mathrm{E}\left((T_1 - \boldsymbol{x})^2\right)}$$

where $\boldsymbol{x}$ is true solution, $\mathrm{E}(\cdot)$ is expectation

- Typically compare to NLS, set efficiency to 95%

- Are there other kernel types?

Table 2.1: A few of the commonly used robust functions. Note that a, b and c are constant parameters of the individual functions (i.e. not the same variable).

	loss function $\rho(u)$	score function $\psi(u) = \frac{\partial \rho(u)}{\partial u}$
Ordinary LS	$\frac{1}{2}u^2$	u
Huber [88]	$\begin{cases} \frac{1}{2}u^2 & \text{if } \lvert u\rvert \le a \\ \frac{1}{2}a(2\lvert u\rvert - a) & \textit{otherwise} \end{cases}$	$\begin{cases} 1 & \text{if } \lvert u\rvert \le a \\ a \operatorname{sign}(u) & \textit{otherwise} \end{cases}$
Cauchy [83]	$\frac{a^2}{2}\log\left(1 + \left(\frac{u}{a}\right)^2\right)$	$\frac{u}{1+\left(\frac{u}{a}\right)^2}$
Tukey [15]	$\begin{cases} \frac{a^2}{6}\left(1 - \left(1 - \left(\frac{u}{a}\right)^2\right)^3\right) & \text{if } \lvert u\rvert \le a \\ \frac{a^2}{6} & \textit{otherwise} \end{cases}$	$\begin{cases} u\left(1 - \left(\frac{u}{a}\right)^2\right)^2 & \text{if } \lvert u\rvert \le a \\ 0 & \textit{otherwise} \end{cases}$
Hampel [78]	$\begin{cases} \frac{1}{2}u^2 & \text{if } \lvert u\rvert < a \\ a\lvert u\rvert - \frac{1}{2}a^2 & \text{if } a \le \lvert u\rvert < b \\ a\frac{c\lvert u\rvert - \frac{1}{2}u^2}{c-b} - \frac{7}{6}a^2 & \text{if } b \le \lvert u\rvert < c \\ a(b+c-a) & \textit{otherwise} \end{cases}$	$\begin{cases} u & \text{if } \lvert u\rvert < a \\ a \operatorname{sign}(u) & \text{if } a \le \lvert u\rvert < b \\ a\frac{c \operatorname{sign}(u) - u}{c-b} & \text{if } b \le \lvert u\rvert < c \\ 0 & \textit{otherwise} \end{cases}$

And now for something completely different …

INTERMEZZO II

- NLS boils down to solving linearized system

$$\boldsymbol{J}^T\boldsymbol{J}\Delta\boldsymbol{x} = \boldsymbol{J}^T\Delta\boldsymbol{b}$$

  - Most of time is spent there
  - In the remainder, let's assume $\boldsymbol{\Lambda x} = \boldsymbol{b}$
  - In here, Λ is sparse (most of its entries are zero) but many of the methods apply to dense matrices too

- Different methods
  - Direct methods - reduce the matrix to triangular
    - Proven complexity, mature algorithms
    - Need considerable amounts of memory
  - Iterative methods - do a lot of vector math, iteratively converge to the solution
    - Needs very little memory
    - Convergence depends on problem, preconditioner
    - Quite young field, not much proven
  - Subspace methods - use Eigenvalue-like algorithms to calculate subspace approximations, solve there
- [Davis, 2006, Direct Methods for Sparse Systems]
- [Saad, 2003, Iterative Methods for Sparse Systems]

- Consider the two following systems

$$\begin{bmatrix} 1 & 7 & 5 \\ 1 & 5 & 1 \\ 1 & 2 & 1 \end{bmatrix} \cdot \boldsymbol{x} = \begin{bmatrix} 9 \\ 6 \\ 9 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 7 & 5 \\ 0 & -2 & -4 \\ 0 & 0 & 6 \end{bmatrix} \cdot \boldsymbol{x} = \begin{bmatrix} 9 \\ -3 \\ 7.5 \end{bmatrix}$$

- In both cases, $\boldsymbol{x} = [9.75,\ -1,\ 1.25]$
- What happened?
- Pivoting for stability, cost. Only one r.h.s.

Gaussian Elimination, Backsubstitution

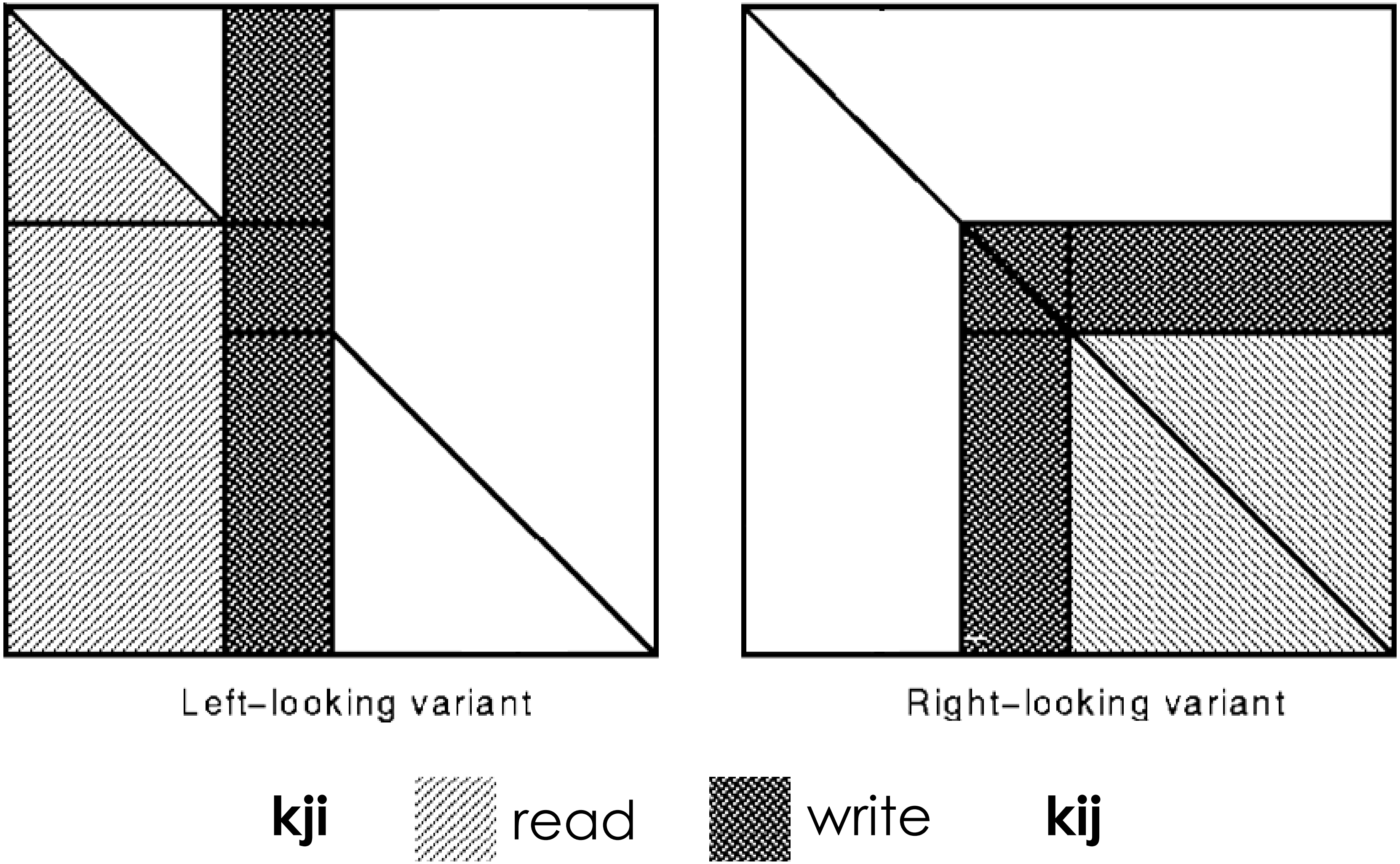
```
A = round(rand(3,3) * 10)
b = round(rand(3,1) * 10)

n = length(A);
for k=1:n-1
    for i=k+1:n
        x = A(i,k) / A(k,k);
        A(i,k+1:n) = A(i,k+1:n) - x * A(k,k+1:n); % row combine
        A(i,k) = 0; % we just eliminated it
        b(i) = b(i) - x * b(k);
    end
end
A
b
% Gaussian elimination

x = zeros(n,1);
for i=n:-1:1 % loop backwards
    r = b(i) - dot(A(i,i+1:n), x(i+1:n)');
    x(i) = r / A(i, i);
end
x
% back-substitution
```

LU decomposition

- Another old method
- Factorize $\Lambda = LU$
  - L being lower triangular with unit diagonal
  - U being upper triangular
- To solve $\Lambda \boldsymbol{x} = LU\boldsymbol{x} = \boldsymbol{b}$ (and so $U\boldsymbol{x} = L^{-1}\boldsymbol{b}$)
  - First solve L$\boldsymbol{y} = \boldsymbol{b}$ forward substitution w/o div
  - Then solve $U\boldsymbol{x} = \boldsymbol{y}$ back-substitution
- To work, Λ must be square, invertible
- Will require pivoting for stability
  - Partial $P\Lambda = LU$ (reorder rows)
  - Full $P\Lambda Q = LU$ (reorder rows, cols)

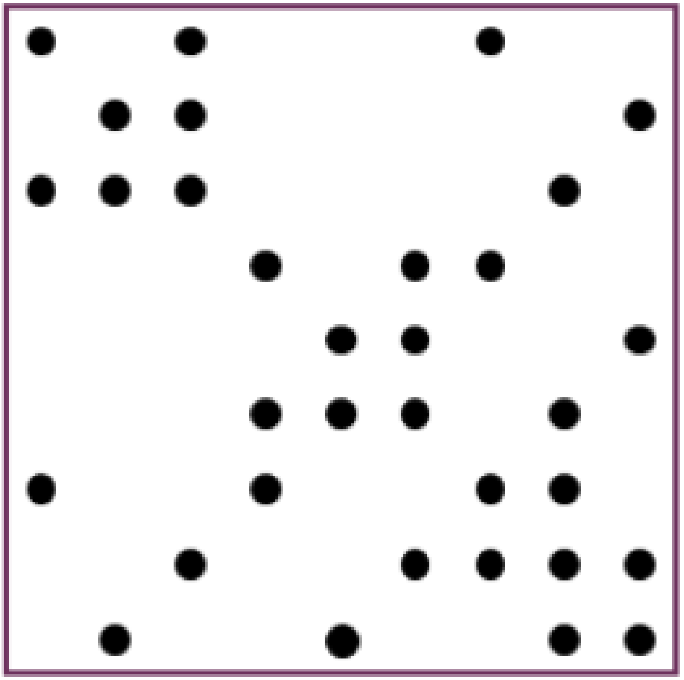
```
A = round(rand(3,3) * 10)

LU = A; % works in-place
for k = 1:n
    LU(k+1:n, k) = LU(k+1:n, k) / LU(k, k);
    % divide by pivot (modifies the rest of this column)

    for i = k+1:n
        x = LU(i, k);
        for j = k+1:n % explicit loop intended
            LU(i, j) = LU(i, j) - x * LU(k, j); % causes fill-in in sparse version
        end
        % reduce the rest of the matrix
    end
    % modify the lower-right submatrix
end

U = triu(LU)
L = tril(LU, -1) + eye(size(LU))
% unpack to L and U, add identity diagonal to L
```

- Several variants, based on ordering of loops **i j k**

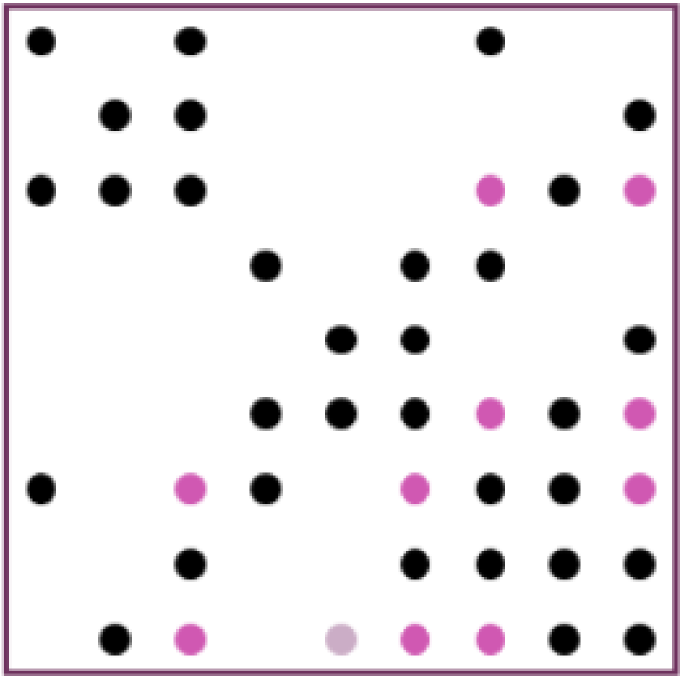


LU decomposition

- In sparse version, we care about fill-in



A

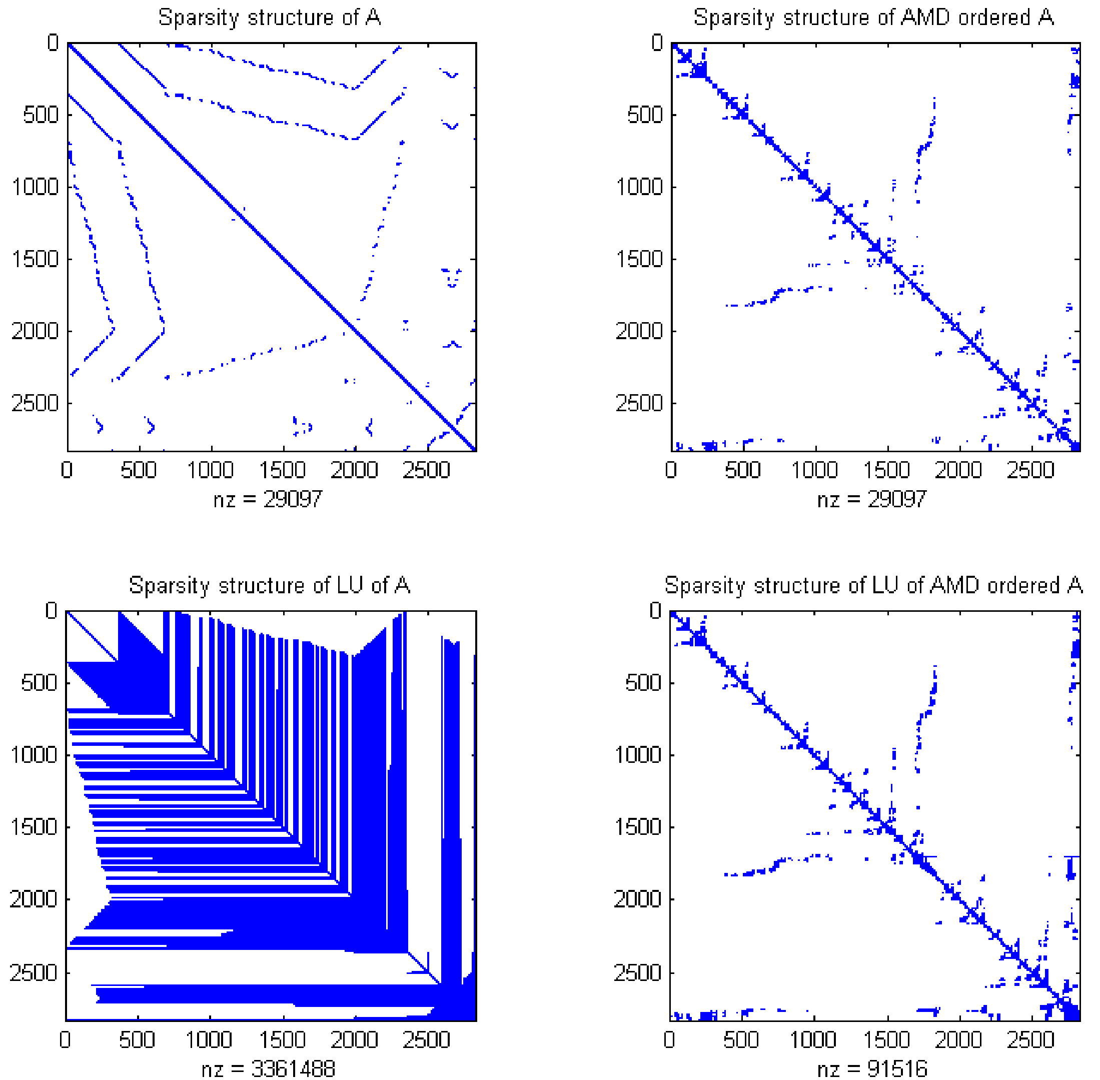


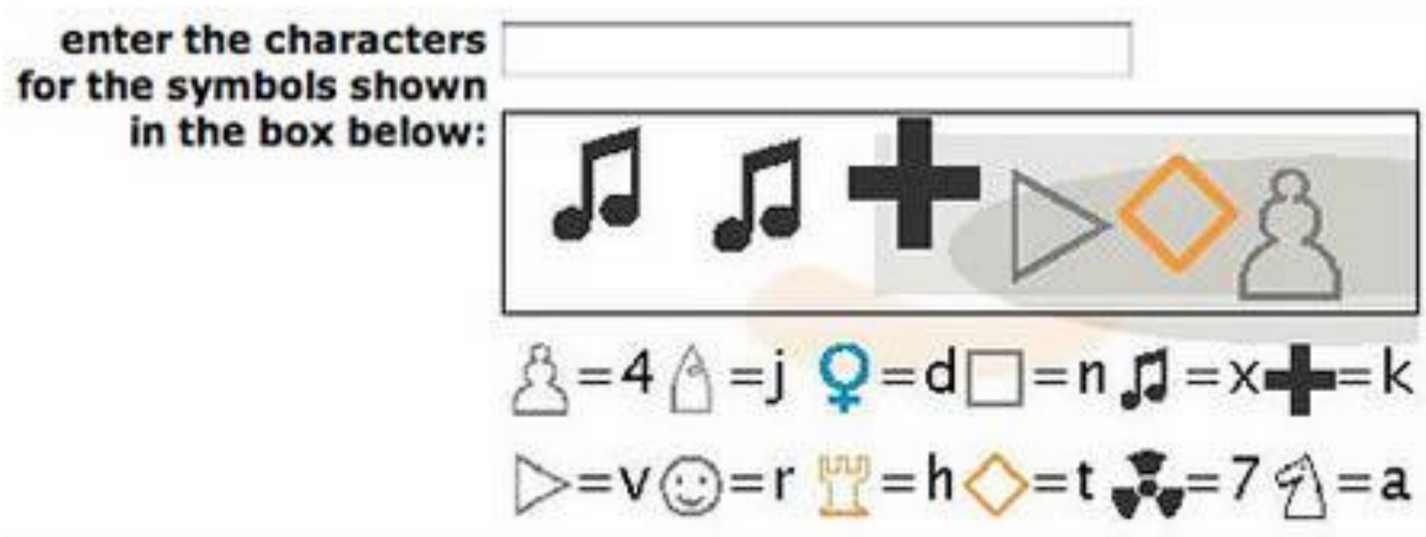
L+U

- Depends on order of rows / columns

AMD Ordering

- Approximate Minimum Degree [Amestoy 2004]





ENTER CAPTCHA IF YOURE AWAKE

- Factorize $\Lambda = R^T R$
  - R being upper triangular, with positive diagonal entries
- To solve $\Lambda \boldsymbol{x} = R^T R \boldsymbol{x} = \boldsymbol{b}$
  - First solve $R^T \boldsymbol{y} = \boldsymbol{b}$ forward substitution
  - Then solve $R\boldsymbol{x} = \boldsymbol{y}$ back-substitution
- To work, Λ must be square, symmetric and positive-definite (SPD)
  - NLS matrices are (up to numerical precision)
  - Adding small damping on the diagonal usually helps
  - Modified Cholesky factorization
- Does *not* need pivoting

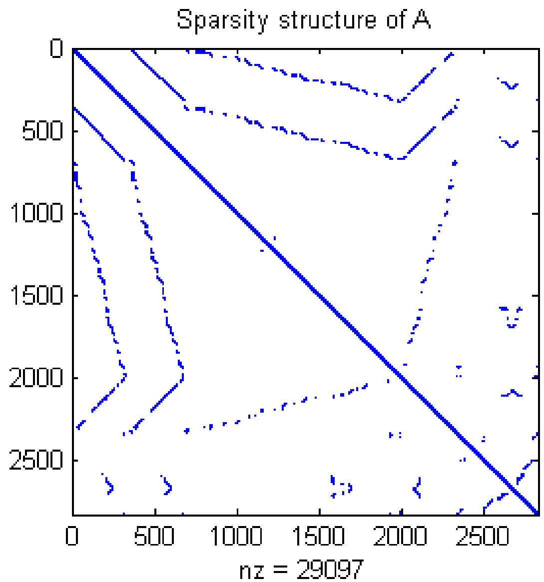
Cholesky factorization

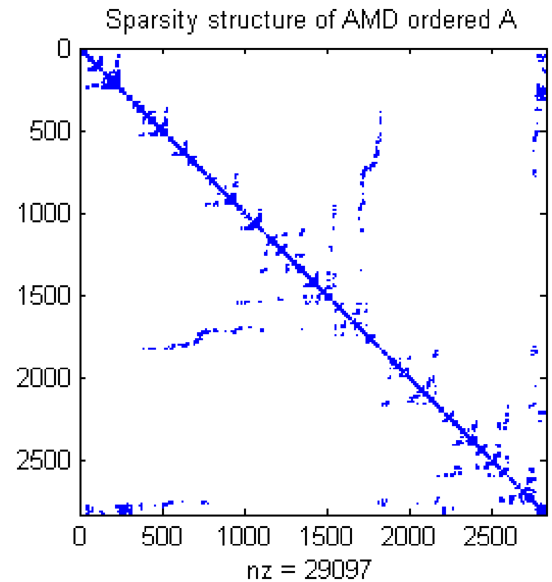
```
A = round(rand(3,3) * 10);
A = A' * A + eye(size(A)) * 10 % try to make it SPD

R = zeros(size(A));
for j = 1:n % for every column
    for k = 1:j-1 % for all prev cols that are nnz at row j (know those from etree)
        s = 0;
        for i = 1:k-1 % for all blocks above the diagonal in the prev column
            s = s + R(i, k) * R(i, j); % takes elements from two different columns
        end
        % cmod; causes fill-in in the current column

        R(k, j) = (A(k, j) - s) / R(k, k); % accesses upper diagonal of A
    end

    s = R(1:j-1, j)' * R(1:j-1, j);
    R(j, j) = sqrt(A(j, j) - s); % must be positive (or modified Cholesky)
    % cdiv
end
```

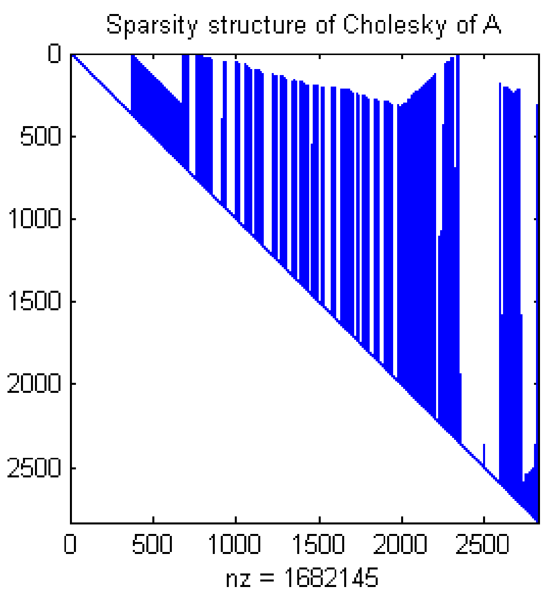
BRNO UNIVERSITY OF TECHNOLOGY

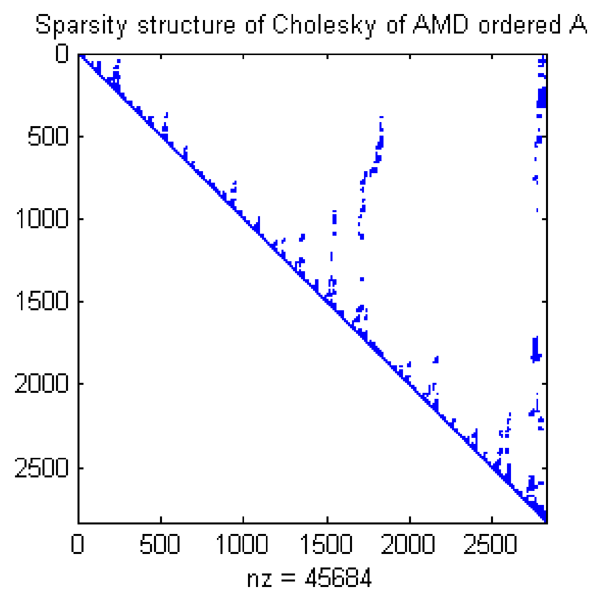
- Elimination tree theory
  - Calculate tree that drives loops in the factorization
  - Saves time
  - Particular orderings also yield elimination trees with independent and balanced subtrees (parallelism)
- Supernodal Cholesky
  - Sometimes, several columns in the factorization have identical nonzero structure - supernodes
  - Treating supernodes as dense allows parallelism, GPU acceleration
- Multifrontal Cholesky
  - Frontal matrices, conceptually similar to supernodes

Cholesky factorization - AMD again









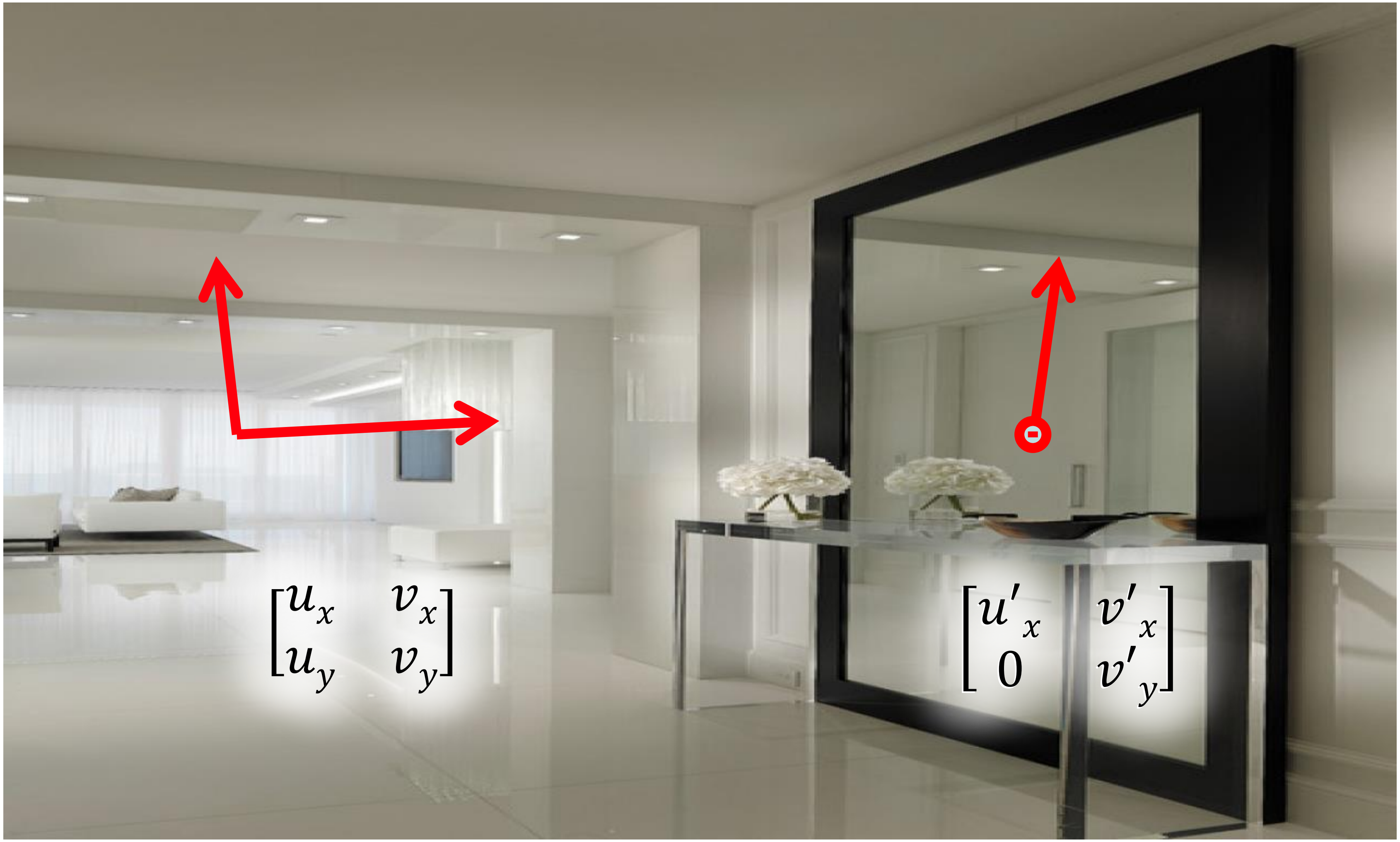
QR decomposition

- Relatively new method (in the 60's)
- Factorize $A = QR$
  - A can now be *rectangular*
  - Q being orthogonal ($Q^T = Q^{-1}$)
  - R being upper triangular (zero rows at the bottom)
- To solve $\mathrm{A}\boldsymbol{x} = QR\boldsymbol{x} = \boldsymbol{b}$
  - First solve $\boldsymbol{y} = Q^T\boldsymbol{b}$ multiplication*
  - Then solve $R\boldsymbol{x} = \boldsymbol{y}$ back-substitution
- No pivoting needed
- Fill-in depends on *column* ordering only
  - Out-of-core methods

Computing QR decomposition

- Similar to Gaussian elimination
- Householder reflections
  - Compute a reflection about a plane (mirror matrix) that zeroes a lower part of a column in R
  - Minor tricks (sign choice) for numerical stability
  - Record reflections rather than representing Q
- Givens rotations
  - Compute rotation matrix that zeros one element in R
  - Record rotation chains rather than representing Q

- Householder reflections



- *Householder transformation* has form

$$\boldsymbol{H} = \boldsymbol{I} - 2\frac{\boldsymbol{v}\boldsymbol{v}^T}{\boldsymbol{v}^T\boldsymbol{v}}$$

  where $\boldsymbol{v}$ is nonzero vector

- From definition, $\boldsymbol{H} = \boldsymbol{H}^T = \boldsymbol{H}^{-1}$, so $\boldsymbol{H}$ is both orthogonal and symmetric

- For given vector $\boldsymbol{a}$, choose $\boldsymbol{v}$ so that

$$\boldsymbol{H}\boldsymbol{a} = \begin{bmatrix} \alpha \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \alpha \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \alpha\boldsymbol{e}_1$$

- *Givens rotation* operates on pair of rows to introduce single zero
- For given 2-vector $\boldsymbol{a} = [a_1 \;\; a_2]^T$, if

$$c = \frac{a_1}{\sqrt{a_1^2 + a_2^2}}, \qquad s = \frac{a_2}{\sqrt{a_1^2 + a_2^2}}$$

then

$$\boldsymbol{G}\boldsymbol{a} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \end{bmatrix}$$

- Scalars c and s are cosine and sine of angle of rotation, and $c^2 + s^2 = 1$, so $\boldsymbol{G}$ is orthogonal

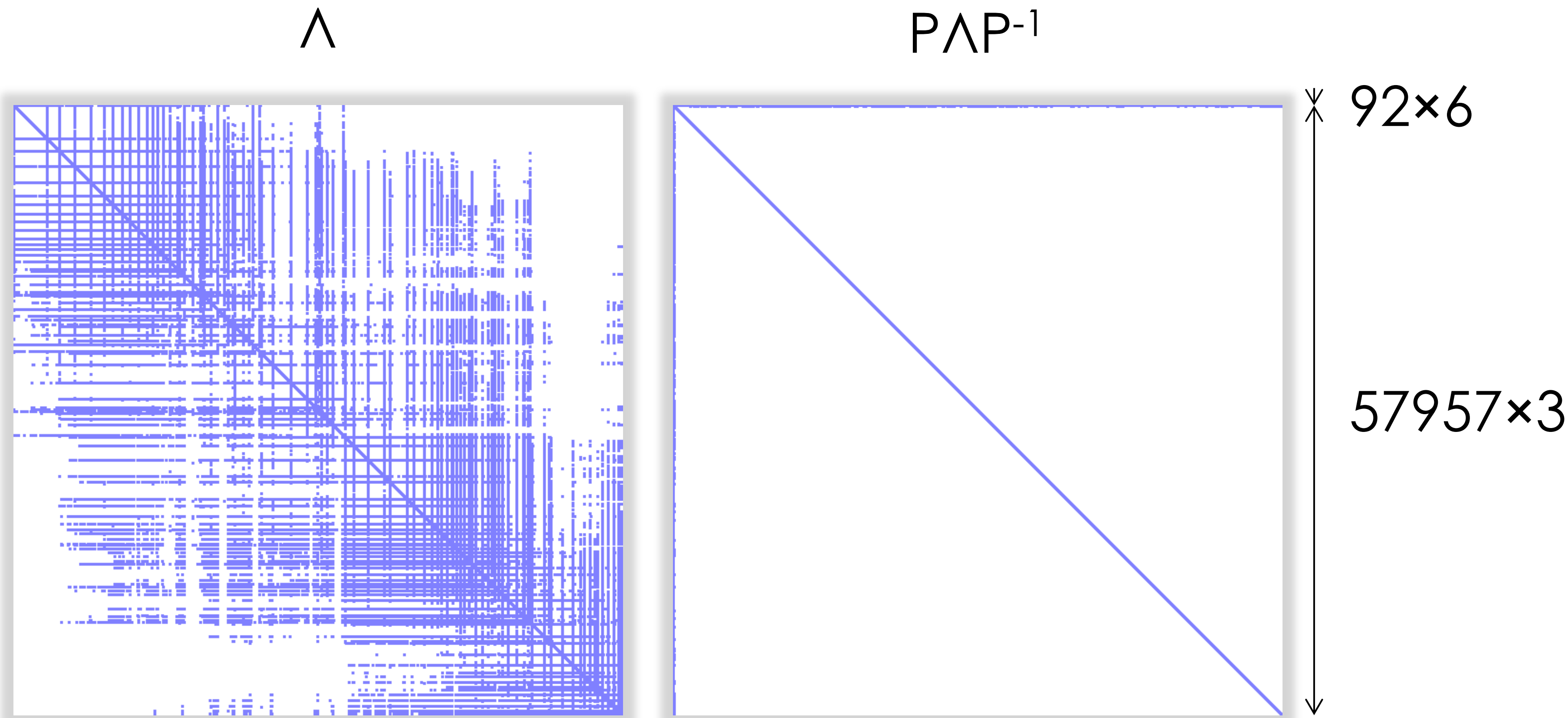
- Related to Cholesky
  - $A = QR$
  - $\Lambda = A^T A$
  - $\Lambda = R^T Q^T Q R = R^T R$
  - R is the same as in Cholesky factorization, up to the sign of the rows (Cholesky always has positive diag)
- Can directly solve NLS on $\boldsymbol{J}$ without forming $\boldsymbol{J}^T\boldsymbol{J}$
  - $\boldsymbol{J} = QR$
  - $R\Delta\boldsymbol{x} = Q^T \Delta\boldsymbol{b}$
  - Numerical benefits

```
M = round(rand(5, 3) * 10)

[m n] = size(M);
Q = eye(m, m); % crime against QR: explicit Q
R = M; % works inplace
elim = min(m - 1, n); % decide how many cols to eliminate to get triangular R
for i = 1:elim
    Aii = R(i, i);
    Ai_norm = sqrt(R(i:m, i)' * R(i:m, i));
    dii = abs(Ai_norm) * sign(sign(Aii) + 0.5); % !!! need zero-avoiding sign function !!!
    wii = Aii - dii;
    two_fi_inv_squared = -1 / (wii * dii);
    % calculate Householder reflection of the i-th column

    for j = i+1:n
        fj = wii * R(i, j); % the head of column i is replaced by wii
        fj = fj + R(i+1:m, i)' * R(i+1:m, j); % dot of lower-part of columns i and j
        fj = fj * two_fi_inv_squared;
        % calculate columns dot

        R(i, j) = R(i, j) - fj * wii;
        R(i+1:m, j) = R(i+1:m, j) - fj * R(i+1:m, i);
        % update jth column
    end
    % apply Householder reflections also to the other columns to the right
```

```
    for j = 1:m
      fj = wii * Q(j, i); % the head of column i is replaced by wii
      fj = fj + Q(j, i+1:m) * R(i+1:m, i); % dot of lower-part of columns i and j
      fj = fj * two_fi_inv_squared;
      % calculate columns dot

      Q(j, i) = Q(j, i) - fj * wii;
      Q(j, i+1:m) = Q(j, i+1:m) - fj * R(i+1:m, i)';
      % update jth column
    end
    % apply Householder reflections also to the r.h.s columns, in transpose!

    R(i, i) = dii;
    R(i+1:m, i) = 0; % finally, clear the rest of the current column
end
% eliminate lower triangle of M, column by column
```

Schur Complement Trick

- In BA, it is possible to reorder the system to have diagonal submatrices
  - bipartite ordering, MIS



*The sparsity is the same on the left / right. The nnz's are just very inflated to be visible and that makes the sparsity *appear* different.

Schur Complement Trick

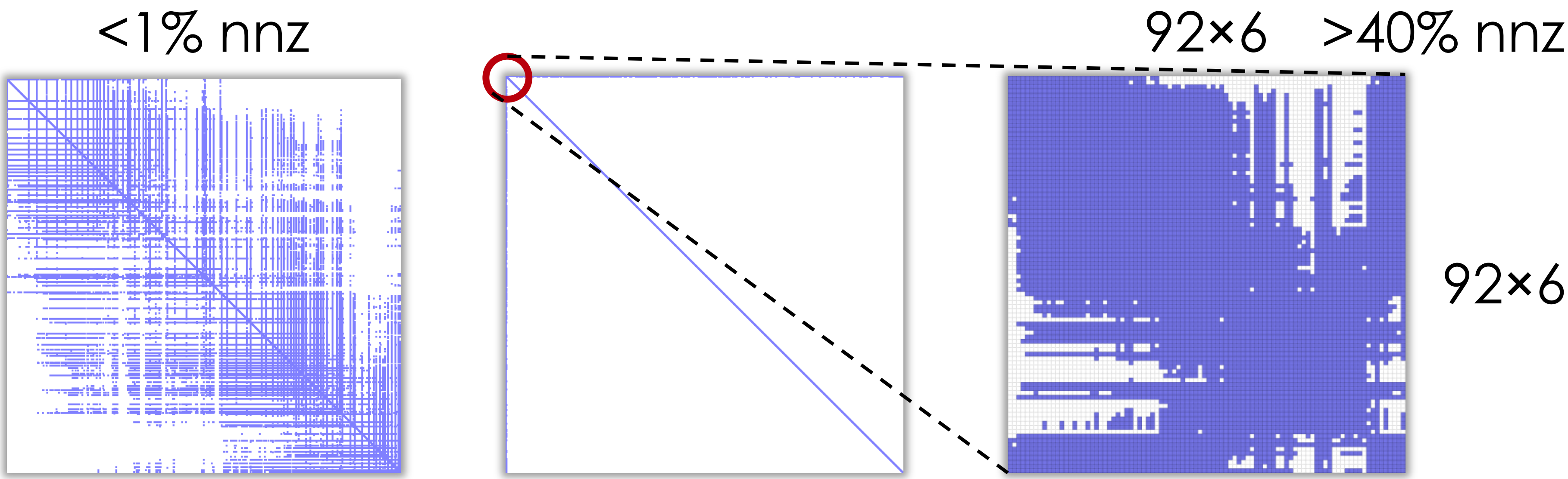
- Then the matrix is partitioned as

$$P\Lambda P^{-1} = \begin{bmatrix} A & U \\ U^T & D \end{bmatrix}$$

and the linear system $\Lambda \boldsymbol{x} = \boldsymbol{b}$ partitioned as

$$\begin{bmatrix} A & U \\ U^T & D \end{bmatrix} \begin{bmatrix} \boldsymbol{p} \\ \boldsymbol{l} \end{bmatrix} = \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{v} \end{bmatrix} \text{ with } \begin{bmatrix} \boldsymbol{p} \\ \boldsymbol{l} \end{bmatrix} = P\boldsymbol{x}\,,\ \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{v} \end{bmatrix} = P\boldsymbol{b}$$

- The Schur complement of A is $A - UD^{-1}U^T$

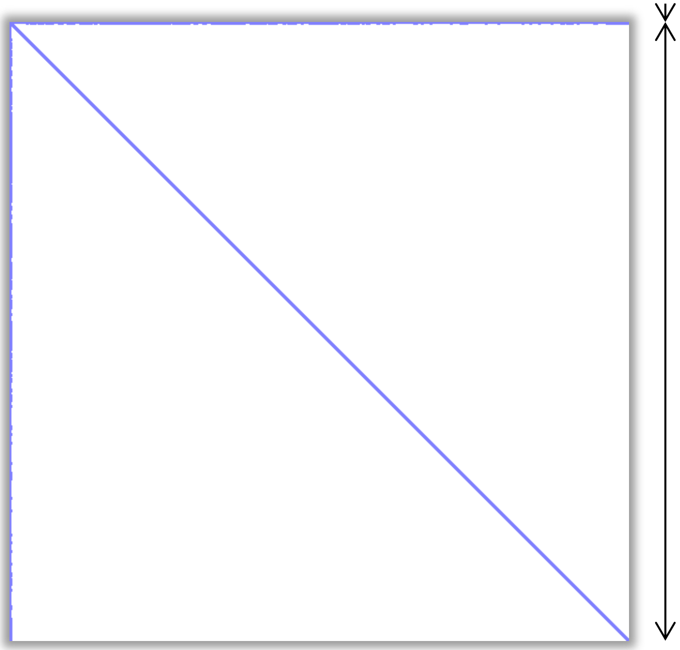


- The linear system

$$\begin{bmatrix} A & U \\ U^T & D \end{bmatrix} P^{-1} \begin{bmatrix} \boldsymbol{p} \\ \boldsymbol{l} \end{bmatrix} = P^{-1} \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{v} \end{bmatrix}$$

can be solved as

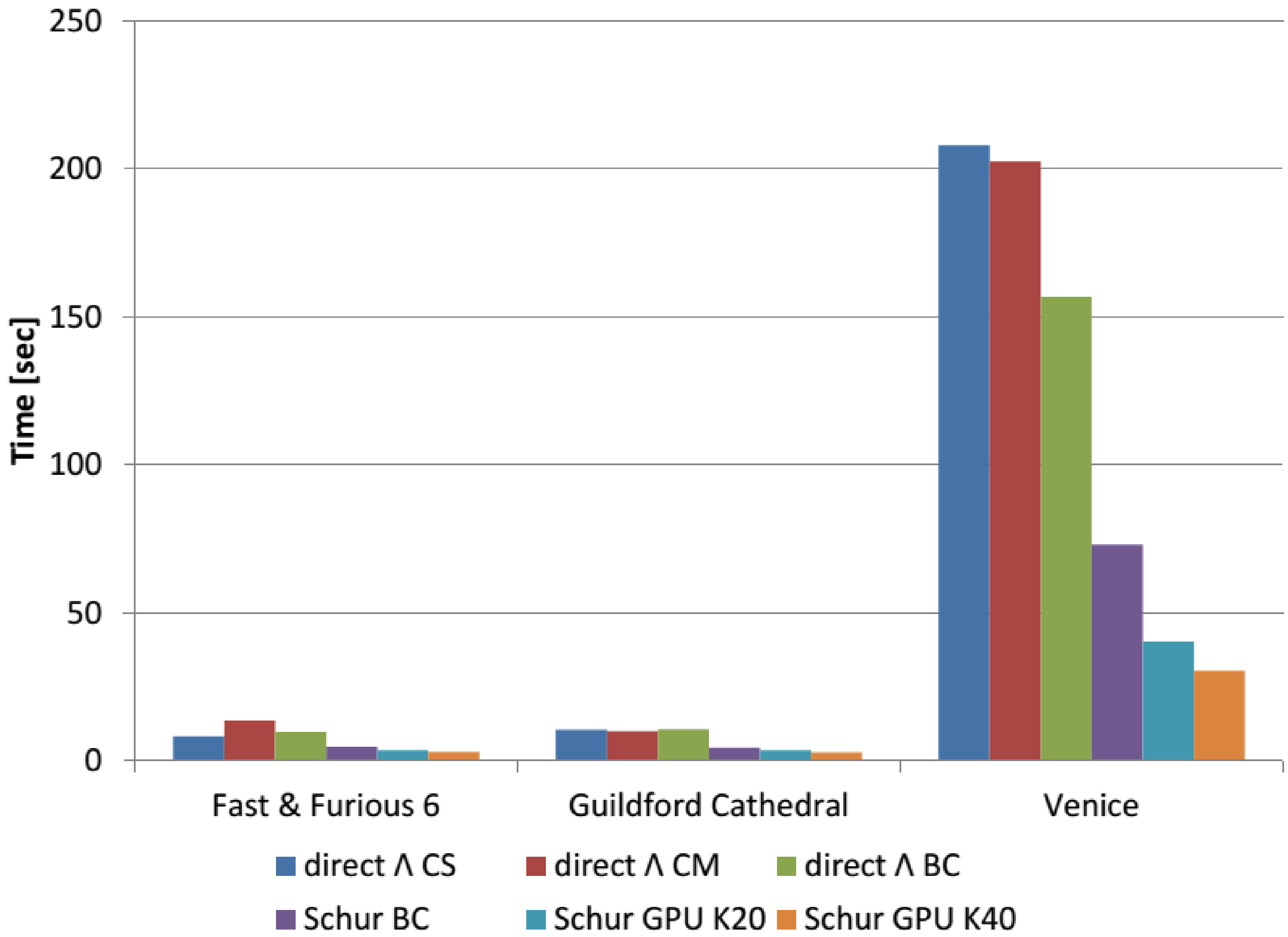
$$(A - UD^{-1}U^T)\boldsymbol{p} = \boldsymbol{u} - UD^{-1}\boldsymbol{v}$$
$$\boldsymbol{l} = D^{-1}(\boldsymbol{v} - U^T\boldsymbol{p})$$



92×6

57957×3

Schur Complement Trick



Who will do seminar on

ITERATIVE METHODS?

Who will do seminar on

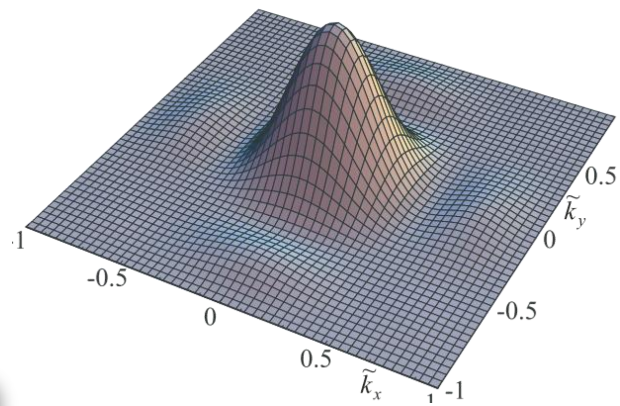
SUBSPACE METHODS?

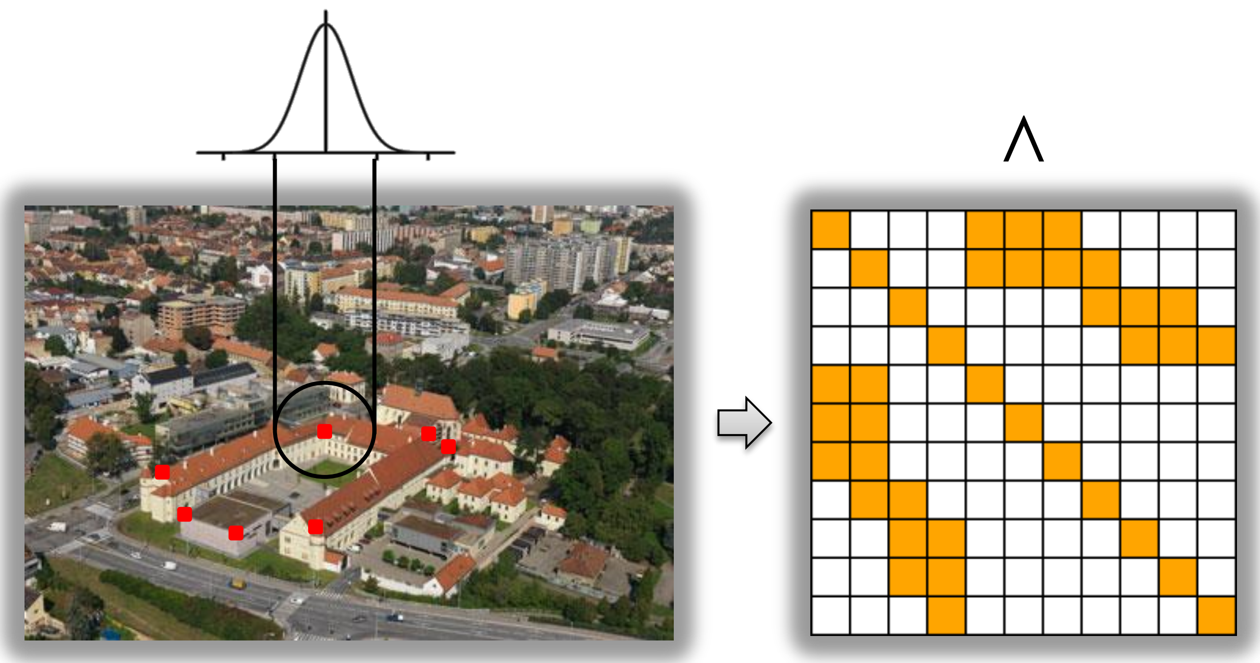
And now …

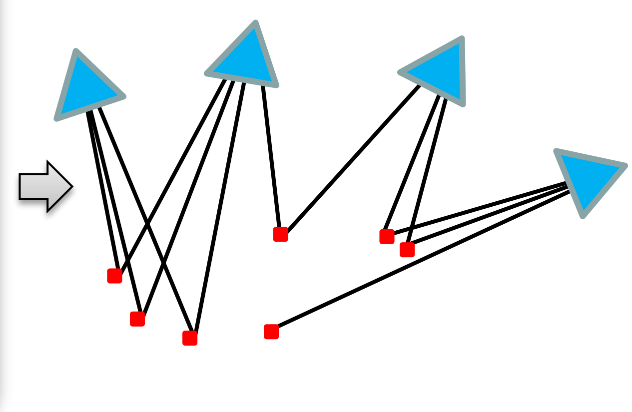
BACK TO BUNDLING

Calculating Covariances

- System covariances (variable covariances)
  - Different from edge (observation) covariances
- Obtained by inverting the information matrix Λ
  - Inverse $\Sigma = \Lambda^{-1}$ is fully dense







Calculating Covariances

- Use Cholesky factorization & backsubstitution, keep only parts of the covariance to save memory
  - The way Google's Ceres does it, gruesome performance
- Use SVD
  - Another dead end explored by Google™

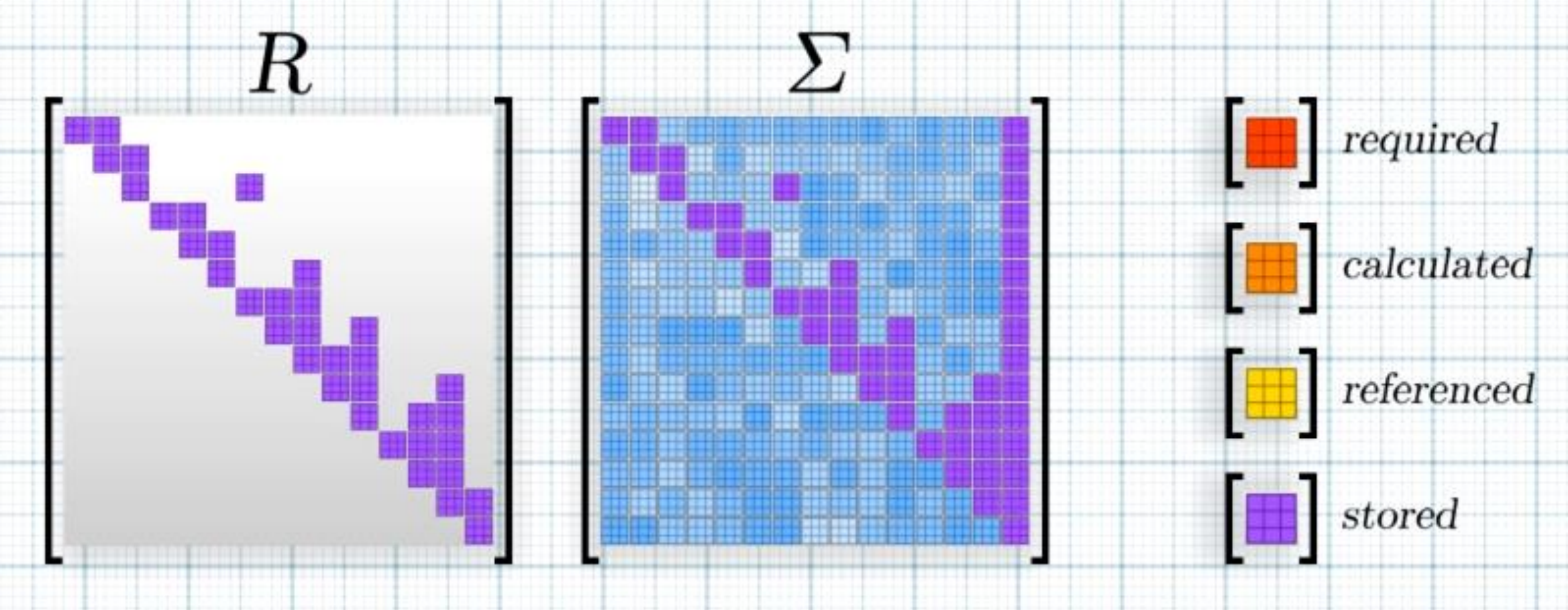
- Use recursive formula for calculating only some parts of the inverse

$$\Sigma_{ii} = \frac{1}{R_{ii}}\left(\frac{1}{R_{ii}} - \sum_{k=i+1, R_{ik}\neq 0}^{n} R_{ik}\Sigma_{ki}\right)$$

$$\Sigma_{ij} = \frac{1}{R_{ii}}\left(- \sum_{k=i+1, R_{ik}\neq 0}^{n} R_{ik}\Sigma_{kj} - \sum_{k=j+1, R_{ik}\neq 0}^{n} R_{ik}\Sigma_{jk}\right)$$

- [Björck, 1996, Numerical methods for least squares problems, SIAM]

- Use recursive formula for calculating only some parts of the inverse



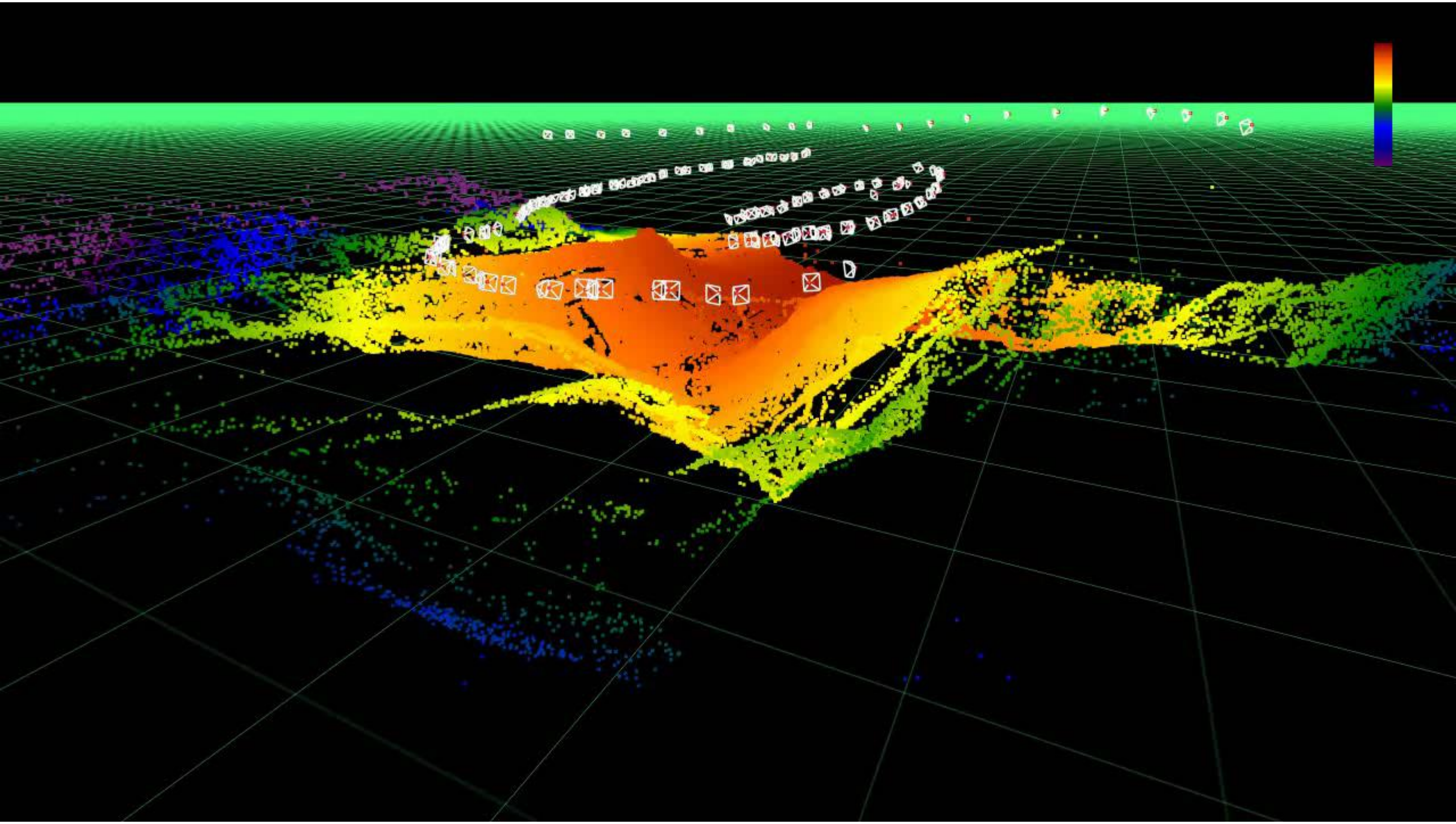
- [Ila, 2015, Fast Covariance Recovery in Incremental Nonlinear Least Square Solvers, ICRA]

- In SLAM, we often only have small increment
- Updating covariance cheaper than recalculating from scratch

$$\Delta\Sigma = \hat{\Sigma} {A_u}^T \left(I - A_u \hat{\Sigma} {A_u}^T\right)^{-1} A_u \hat{\Sigma}$$
$$\Delta\Sigma = -\Sigma {A_u}^T \left(I + A_u \Sigma {A_u}^T\right)^{-1} A_u \Sigma$$

where $\hat{\Lambda} = \hat{A}^T \hat{A} = \Lambda + {A_u}^T A_u$ with $\hat{A} = A + A_u$

- Need only a few elements of $\hat{\Sigma}$ (can use backsubstitution)
- [Ila, 2015, Fast Covariance Recovery in Incremental Nonlinear Least Square Solvers, ICRA]



The end

ARE YOU STILL ALIVE?