# LECTURE 8: Optimization in Higher Dimensions

- Optimization (maximization/minimization) is of huge importance in data analysis and is the basis for recent breakthroughs in machine learning and big data
- A lot of it is application dependent and there is a vast number of methods developed: we cannot cover them all in this lecture

# LECTURE 8: Optimization in Higher Dimensions

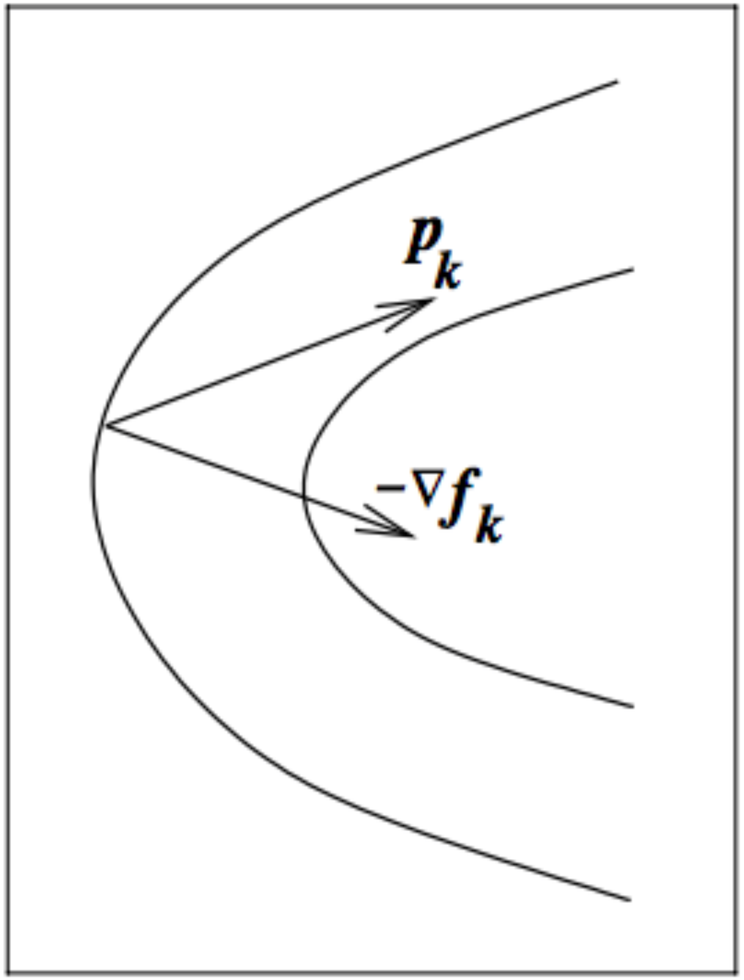
- Broadly can be divided into 1st order (derivatives are available, but not Hessian) and 2nd order (approximate Hessian or full Hessian evaluation)
- 0th order: no gradients available: use finite difference to get the gradient or use downhill simplex (Nelder & Mead method). Very slow and we will not discuss them here.

# Preparation of Parameters

- Often the parameters are not unconstrained: they may be positive (or negative), or bounded to an interval
- First step is to make optimization unconstrained: map the parameter to a new parameter that is unbounded. For example, if a variable is positive, $x > 0$, use $z = log(x)$ instead of $x$.
- One can also change the prior so that it reflects the original prior: $p_{pr}(z)dz = p_{pr}(x)dx$
- If $x > 0$ has uniform prior in x then $p_{pr}(z) = dx/dz = x = e^z$

# General Strategy

- We want to descend down a function $J(a)$ (if minimizing) using iterative sequence of steps at $a_t$. For this we need to choose a direction $p_t$ and move in that direction: $J(a_t+\eta p_t)$
- A few options: fix $\eta$
- line search: vary $\eta$ until $J(a_t+\eta p_t)$ is minimized
- Trust region: construct an approximate quadratic model for $J$ and minimize it but only within trust region where quadratic model is approximately valid



# Line Search Directions and Backtracking

- Gradient descent: Gradient $- \nabla_a J(a,x_t)$
- Newton: Inverse Hessian $H^{-1}$ times gradient $-H^{-1} \nabla_a J(a)$
- Quasi-Newton: approximate $H^{-1}$ with $B^{-1}$ (SR1 and BFGS)
- Nonlinear conjugate gradient: $p_t = - \nabla_a J(a,x_t)+\beta_t p_{t-1}$ , where $p_{t-1}$ and $p_t$ are conjugate
- Step length with backtracking: choose first proposed length
- If it does not reduce the function value reduce it by some factor, check again
- Repeat until step length is ε, at that point switch to gradient descent

# Trust Region Method

- Multi-dim parabola method: define approximate quadratic function, but limit the step

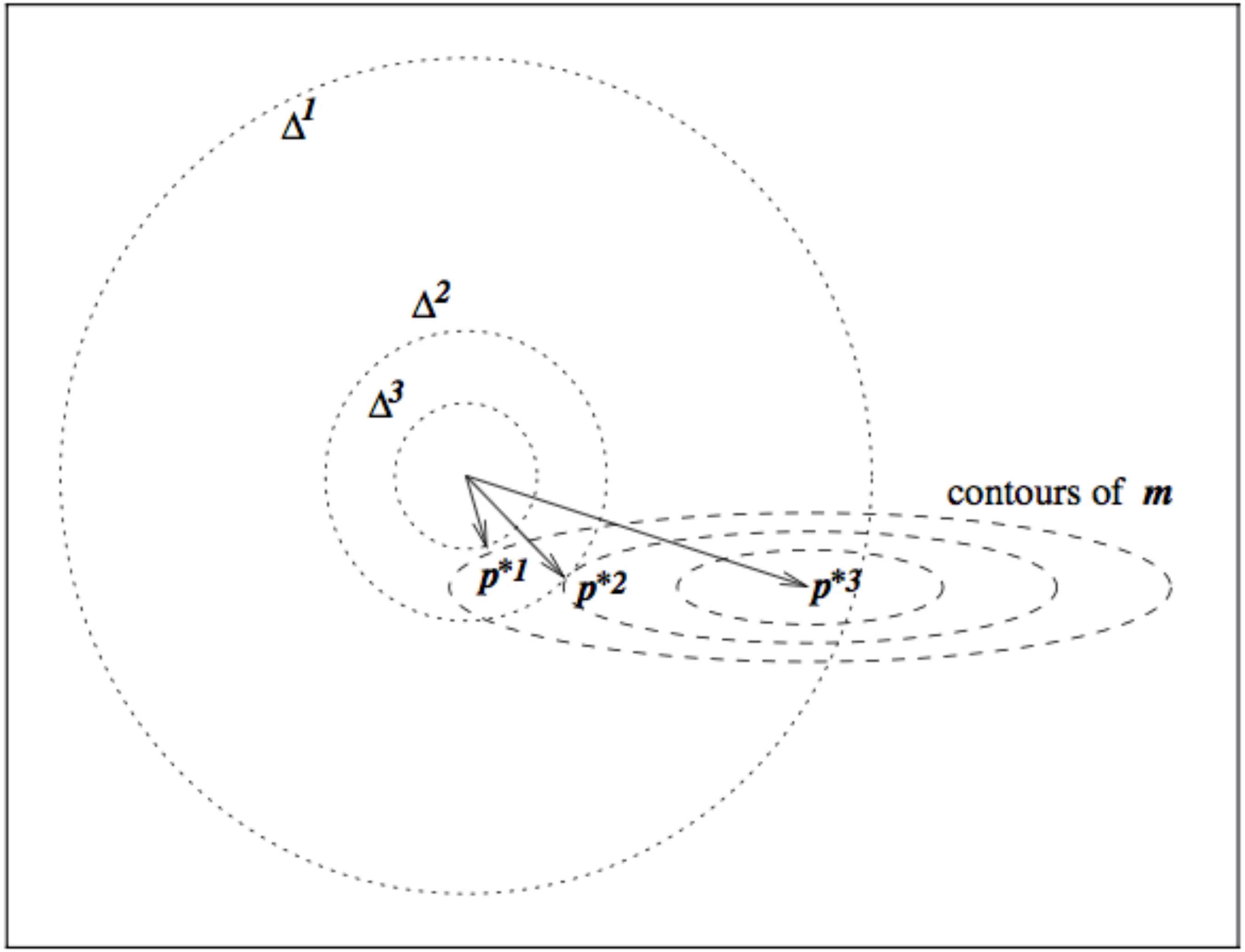
$$\min_{p\in \mathbb{R}^n} m_k(p) = f_k + g_k^T p + \tfrac{1}{2} p^T B_k p \qquad \text{s.t. } \|p\| \leq \Delta_k$$

- Here $\Delta_k$ is trust region radius
- Evaluate at previous iteration and compare the actual reduction to predicted reduction

$$\rho_k = \frac{f(x_k) - f(x_k + p_k)}{m_k(0) - m_k(p_k)}$$

- If $\rho_k$ around 1 we can increase $\Delta_k$
- If close to 0 or negative we shrink $\Delta_k$

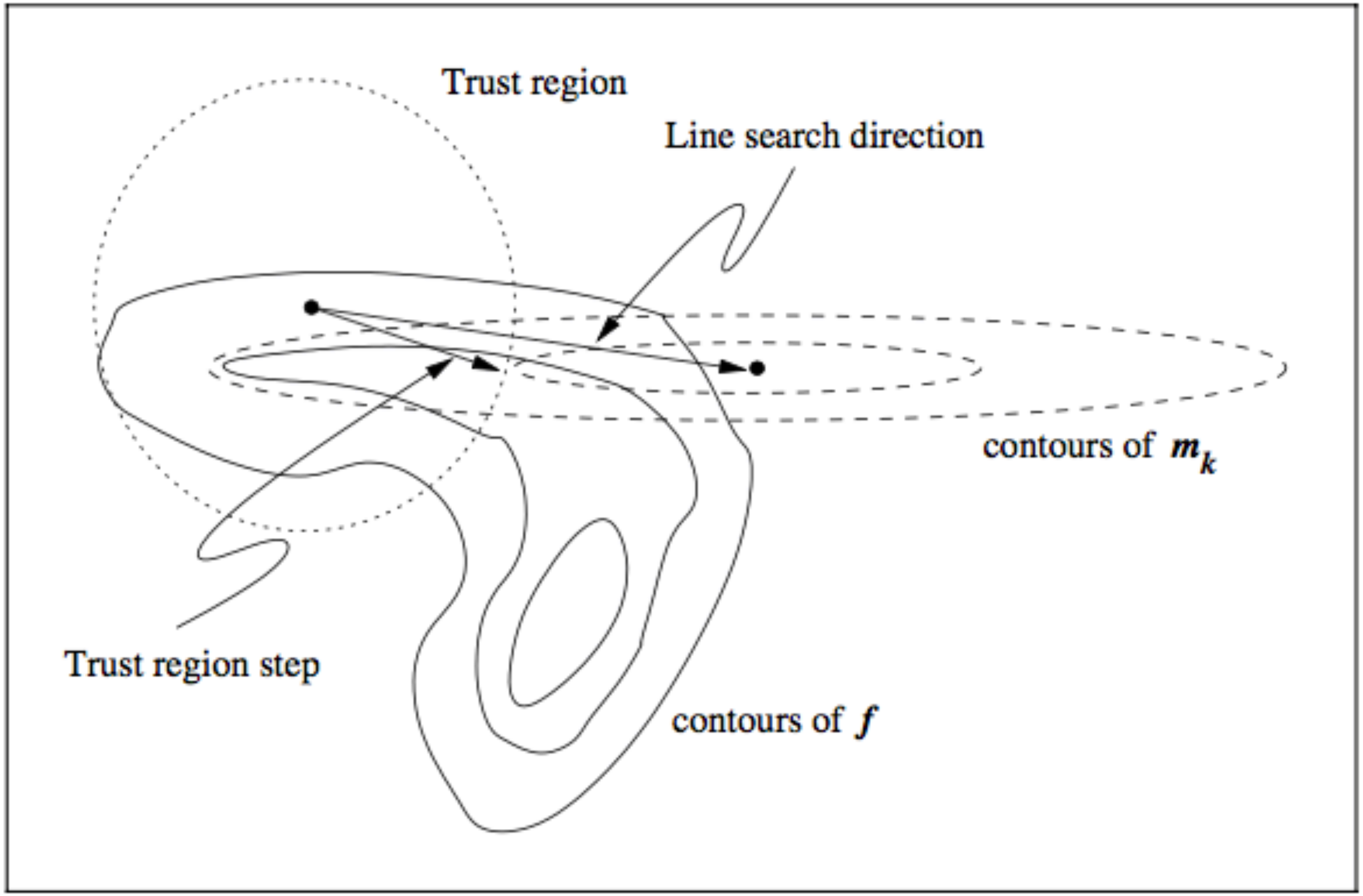
- If trust region covers *m* center step there
- Otherwise direction of step changes



# Constrained Optimization: Lagrange Multiplier Method

- If the center of $m$ is inside trust region step there
- Otherwise we must solve constrained optimization
- We solve this optimization with Lagrange multiplier method: minimize $f + g^T p + p^T B p + \lambda\ (p^2 - \Delta^2)$ with respect to $p$ and $\lambda$. Gradient w.r.t. $\lambda$ gives the constraint $p^2 = \Delta^2$, thus the constraint is automatically satisfied. This determines the value of $\lambda$.
- Minimization with respect to $p$ now includes $\lambda p^2$
- As a result the step direction is not towards center of $m$ when trust region does not cover it: see picture on next slide

# Line Search vs. Trust Region



# $1^{st}$ Order: Gradient Descent

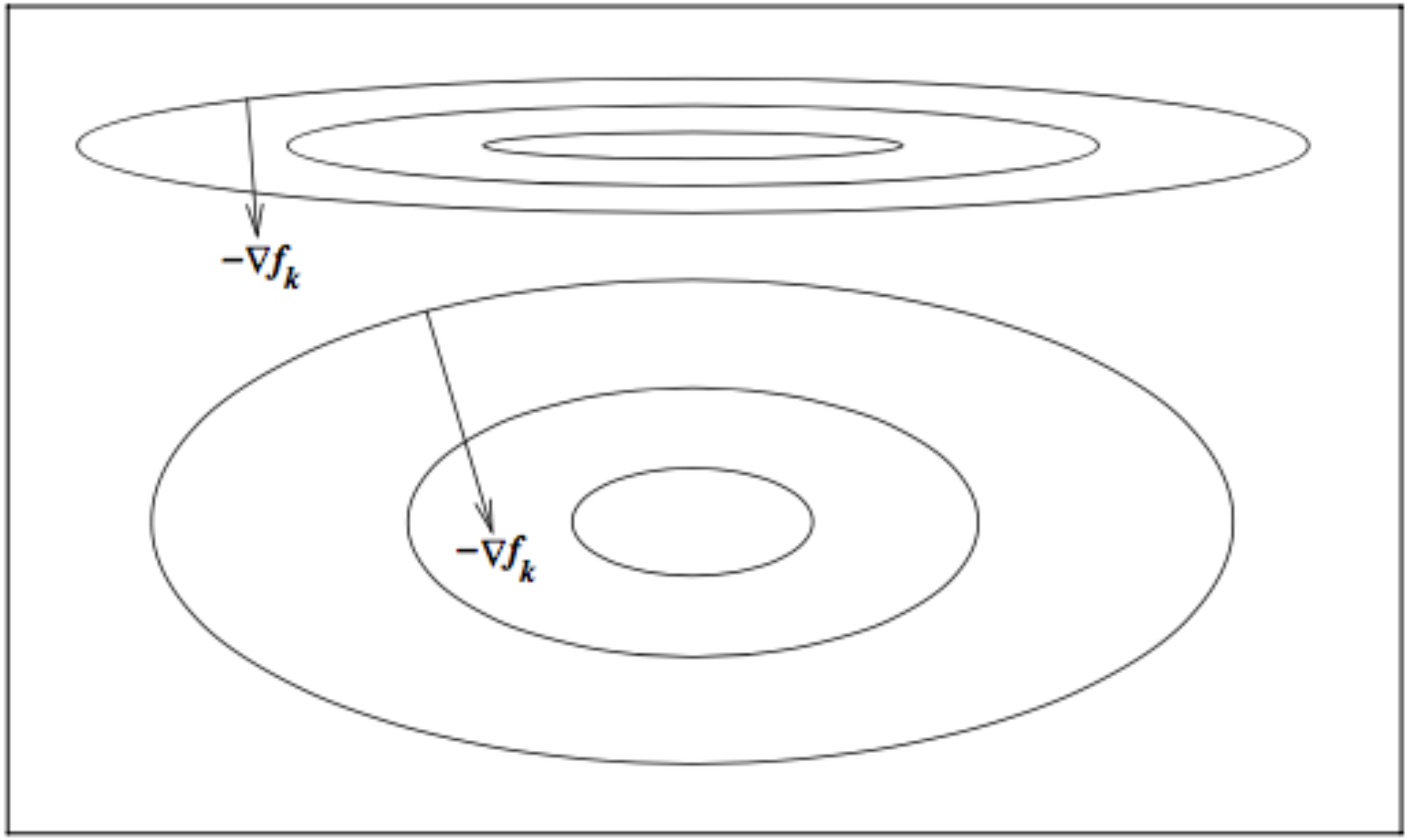
- We have a vector of parameters a and a scalar loss (cost) function *J(a,x,y)* which is a function of a data vector (*x,y*) we want to optimize (say minimize). This could be a nonlinear least square loss function: $J = \chi^2$

$$\chi^2(\mathbf{a}) = \sum_{i=0}^{N-1} \left[ \frac{y_i - y(x_i|\mathbf{a})}{\sigma_i} \right]^2$$

- (Batch) gradient descent updates all the variables at once: $\delta a = - \eta \nabla_a J(a)$: in ML. $\eta$ is called learning rate
- It gets stuck on saddle points, where gradient is 0 everywhere (see animation later)

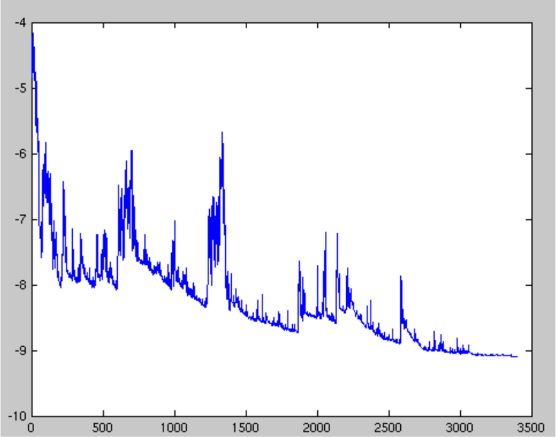
# Scaling

- Change variables to make surface more circular
- Example: change of dimensions



# Stochastic Gradient Descent

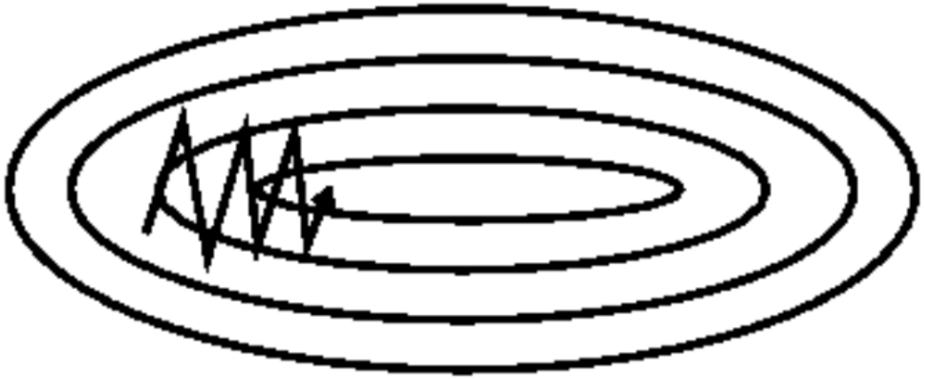
- Stochastic gradient descent: do this just for one data pair $x_i$, $y_i$:
  $\delta a = -\eta \nabla_a J(a, x_i, y_i)$
- This saves on computational cost, but is noisy, so one repeats it by randomly choosing data $i$
- Has large fluctuations in the cost function



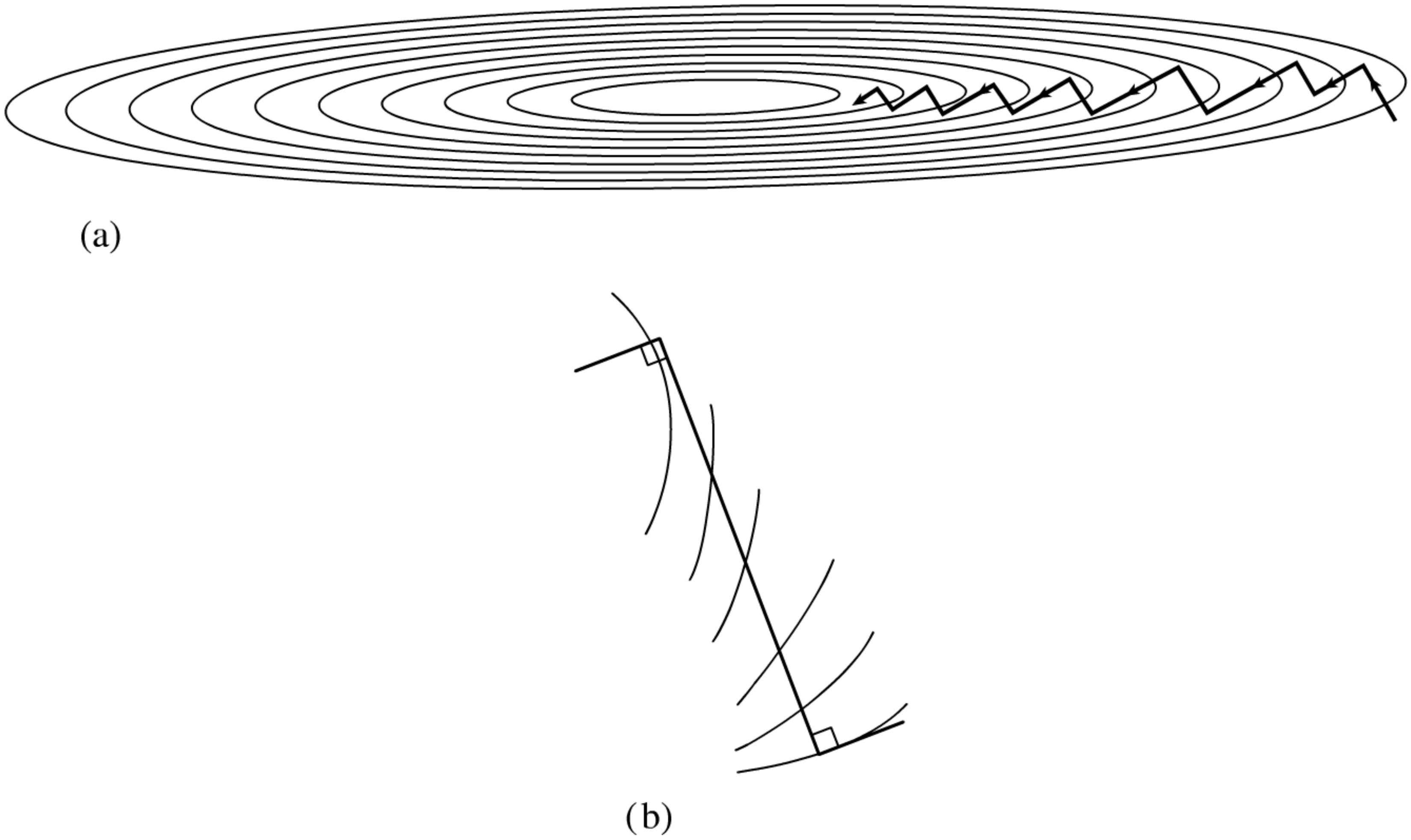
- This is potentially a good thing: it may avoid getting stuck in the local minima (or saddle points)
- Learning rate is slowly reduced
- Has revolutionized machine learning

# Mini-batch Stochastic Gradient

- Mini-batch takes advantage of hardware and software implementations where a gradient w.r.t. to a number of data points can be evaluated as fast as a single data (e.g. mini-batch of $N = 256$)
- Challenges of (stochastic) gradient descent: how to choose learning rate (in 2nd order methods this is given by Hessian)
- Ravines:

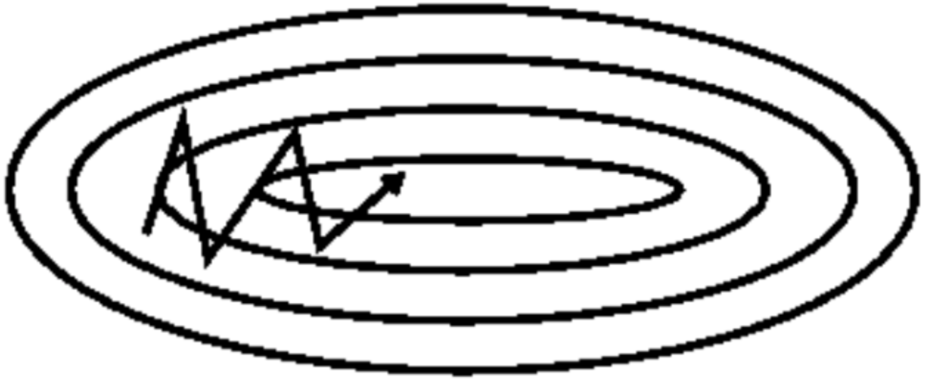


# Ravines



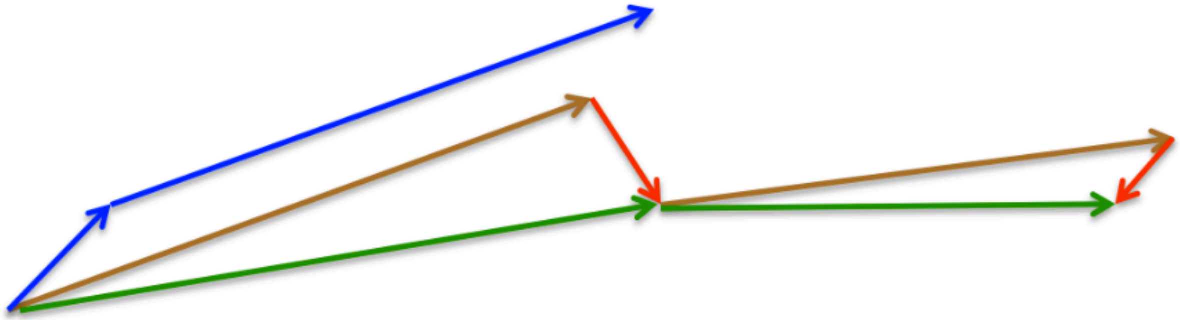
# Adding Momentum: Rolling down the hill

- We can add momentum and mimic a ball rolling down the hill
- Use previous update as the direction
- $v_t = \gamma v_{t-1} + \eta \nabla_a J(a)$, $da = -v_t$ with $\gamma$ of order 1 (e.g. 0.9)
- Momentum increases for directions where gradient does not change



# Nesterov Accelerated Gradient

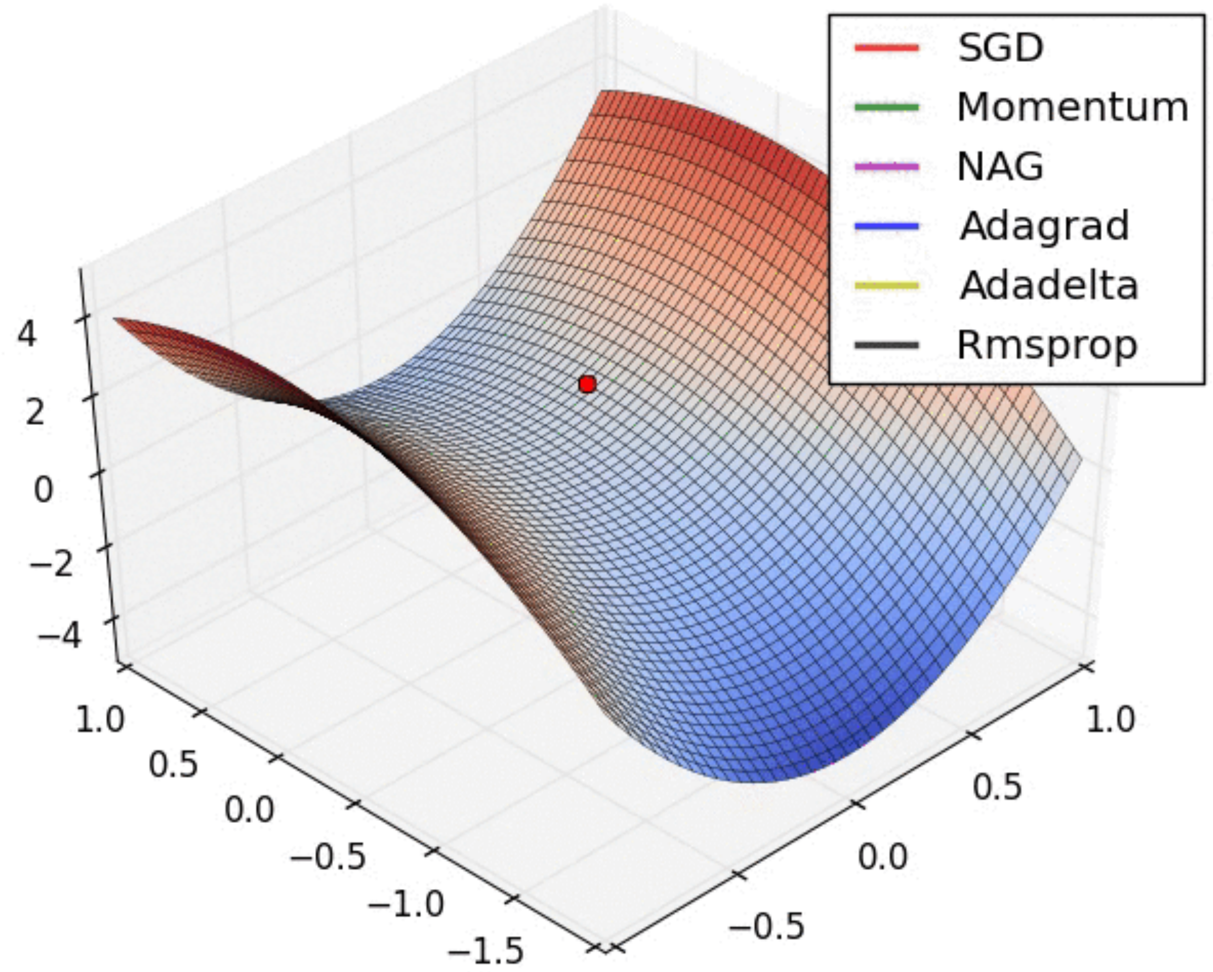
- We can predict where to evaluate next gradient using previous velocity update
- $v_t = \gamma v_{t-1} + \eta \nabla_a J(a - \gamma v_{t-1})$, $\delta a = -v_t$
- Momentum (blue) vs NAG (brown+red=green)



- See https://arxiv.org/abs/1603.04245 for theoretical justification of NAG based on a Bregman divergence Lagrangian

# Adagrad, Adadelta, Rmsprop, ADAM, …

- Make the learning rate h dependent on $a_i$
- Use past gradient information to update h
- Example ADAM: ADAptive Momentum estimation
- $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \qquad g_t = \nabla_a J(a)$
- $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
- bias correction: $m_t' = m_t/(1-\beta_1)$, $v_t' = v_t/(1-\beta_2)$
- Update rule: $\delta a = -\eta/(v_t'^{1/2} + \varepsilon)$
- Recommended values $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$
- The methods are empirical (show animation)


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
4
2
0
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.0
0.5
0.0
−0.5

# $2^{nd}$ Order Method: Newton

- We have seen that there is no natural way to choose learning rate in 1st order methods
- But Newton's method provides a clear answer what the learning rate should be:
- $J(a+\delta a) = J(a) + \delta a \nabla_a J(a) + \delta a \delta a' \nabla_a \nabla_{a'} J(a)/2 \ldots$
- Hessian $H_{ij} = \nabla_{a\_i} \nabla_{a\_j} J(a)$
- At the extremum we we want $\nabla_a J(a) = 0$ so a Newton update step is $\delta a = -H^{-1} \nabla_a J(a)$
- We do not need to guess the learning rate
- We do need to evaluate Hessian and invert it (or use LU): expensive in many dimensions!
- In many dimensions we use iterative schemes to solve this problem

# Quasi-Newton

- Computing Hessian and inverting it is expensive, but one can approximate it with a low rank tensor

- Symmetric rank 1 (SR1) $s_k = x_{k+1} - x_k, \qquad y_k = \nabla f_{k+1} - \nabla f_k.$

$$B_{k+1} = B_k + \frac{(y_k - B_k s_k)(y_k - B_k s_k)^T}{(y_k - B_k s_k)^T s_k} \qquad B_{k+1} s_k = y_k$$

- BFGS (rank 2 update, positive definite)

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}$$

- Inverse (Woodburry formula)

$$B_{k+1}^{-1} = \left(I - \rho_k s_k y_k^T\right) B_k^{-1} \left(I - \rho_k y_k s_k^T\right) + \rho_k s_k s_k^T, \qquad \rho_k = \frac{1}{y_k^T s_k}$$

# L-BFGS

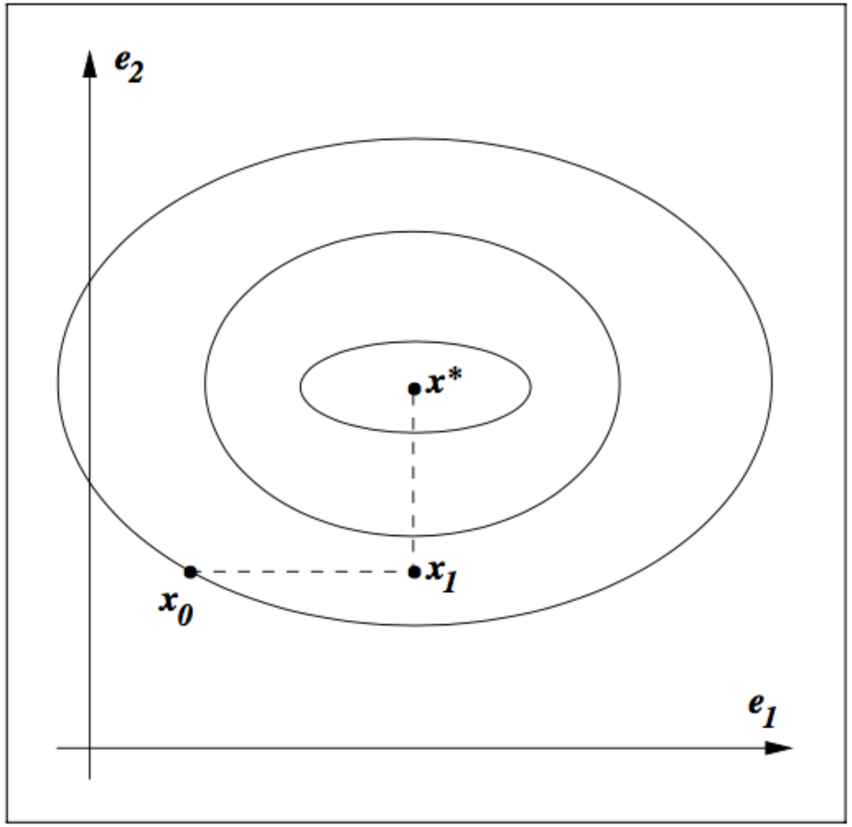
- For large problems this gets too expensive. Limited memory BFGS updates only based on last *N* iterations (*N* of order 10-100)
- In practice increasing *N* often does not improve the results
- Historical note: quasi-Newton methods originate from W.C. Davidon's work in 1950s, a physicist at Argonne national lab.

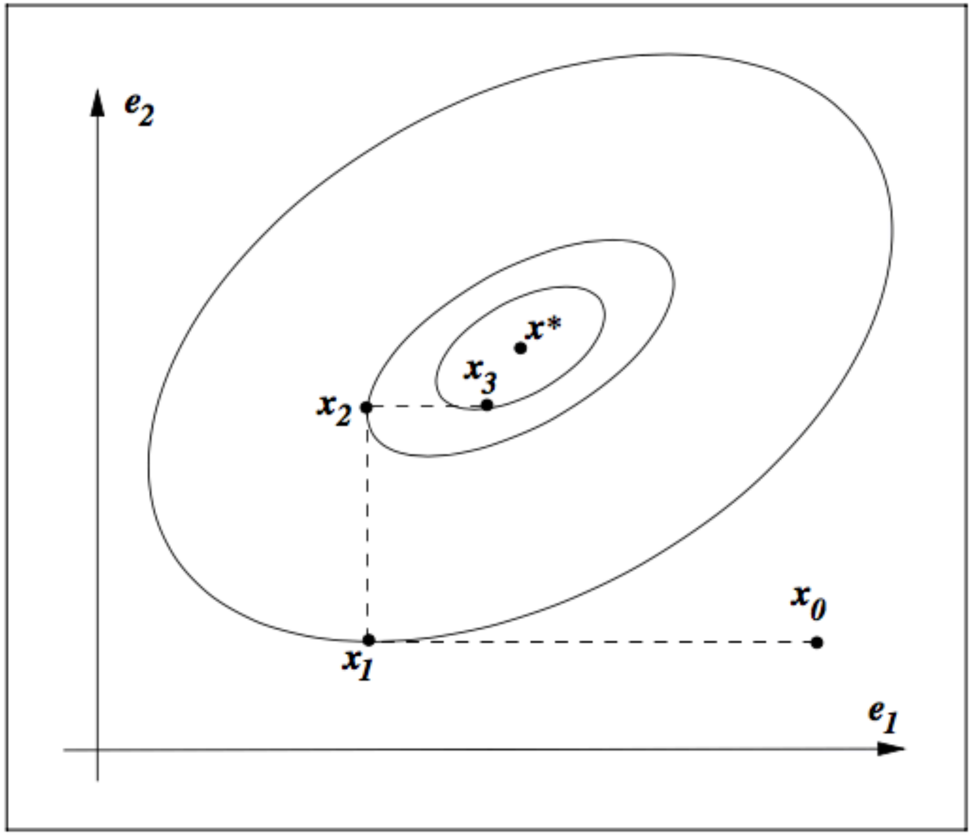
# Linear Conjugate Direction

- Is an iterative method to solve $\mathbf{Ax} = \mathbf{b}$ (so belongs to linear algebra)
- Can be used for optimization: min $\mathbf{J} = \mathbf{x}^T\mathbf{Ax} - \mathbf{b}^T\mathbf{x}$
- Conjugate vectors: $\mathbf{p_i A p_j} = 0$ for all *i, j* not equal *i*
- Construction similar to Gram-Schmidt (QR), where A plays the role of scalar product norm: $\mathbf{x_{k+1}} = \mathbf{x_k} + \alpha_k \mathbf{p_k}$ where $\alpha_k = -\boldsymbol{r}_k^T \boldsymbol{p}_k / (\boldsymbol{p}_k^T \boldsymbol{A} \boldsymbol{p}_k)$ and $\boldsymbol{r}_k = \boldsymbol{A}\boldsymbol{x}_k - \boldsymbol{b}$
- Essentially we are taking a dot product (with **A** norm) of the vector with previous vectors to project it perpendicular to previous vectors
- Since the space is *N*-dim after *N* steps we have spanned the full space and converged to true solution, $r_N$=0.

# Conjugate Direction

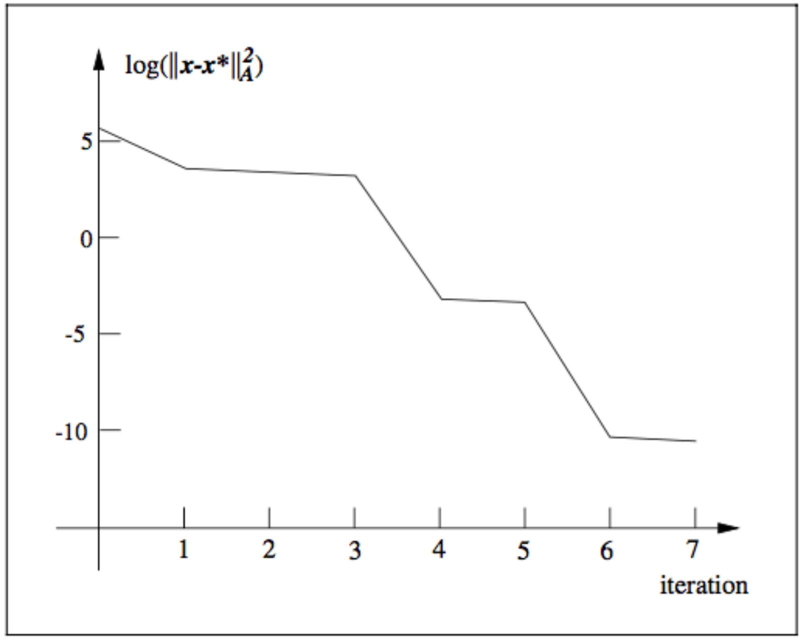
- If we have the matrix **A** in diagonal form so that basis vectors are orthogonal we can find the minimum trivially along the axes, otherwise not





# Linear Conjugate Gradient

- Computes $\mathbf{p}_k$ from $\mathbf{p}_{k-1}$
- We want the step to be linear combination of residual $-\mathbf{r}_k$ and previous direction $\mathbf{p}_{k-1}$
- $\mathbf{p}_k = -\mathbf{r}_k + \beta_k \mathbf{p}_{k-1}$ premultiply by $\mathbf{p}^T_{k-1}\mathbf{A}$
- $\beta_k = (\mathbf{r}_k \mathbf{A} \mathbf{p}_{k-1})/(\mathbf{p}^T_{k-1}\mathbf{A}\mathbf{p}_{k-1})$ imposing $\mathbf{p}^T_{k-1}\mathbf{A}\mathbf{p}_k = 0$
- Converges rapidly for similar eigenvalues, not so much if condition number is high



# Preconditioning

- Tries to improve condition number of **A** by multiplying by another matrix **C** that is simple

$$\hat{x} = Cx.$$

$$\hat{\phi}(\hat{x}) = \tfrac{1}{2}\hat{x}^T(C^{-T}AC^{-1})\hat{x} - (C^{-T}b)^T\hat{x}.$$

$$(C^{-T}AC^{-1})\hat{x} = C^{-T}b$$

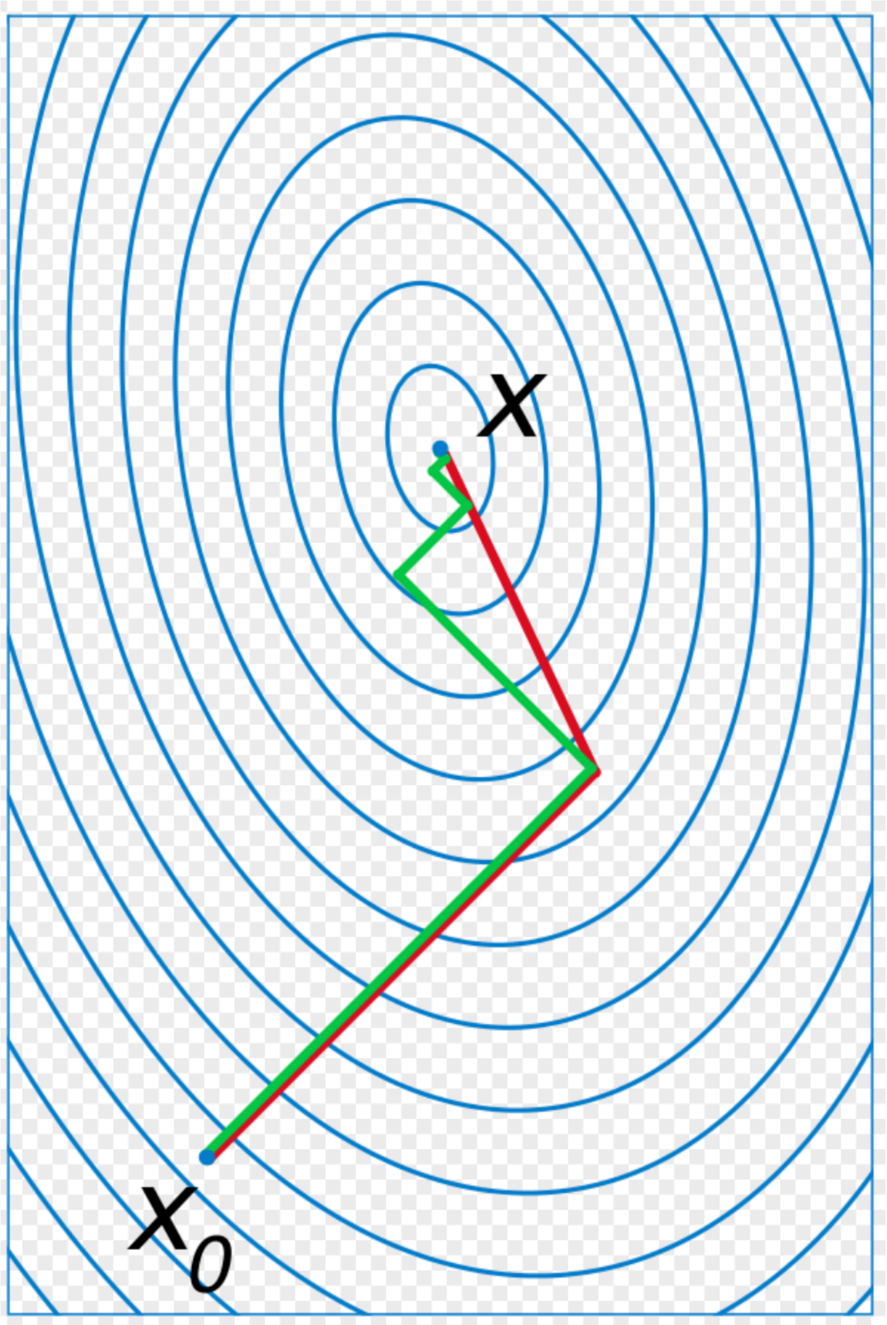
- We wish to reduce condition number of $C^{-T}AC^{-1}$
- Example: incomplete Cholesky **A** = **LL**$^{T}$ by computing only a sparse **L**
- Preconditioners are very problem specific

# Nonlinear Conjugate Gradient

- Replace $\mathbf{a_k}$ with line search that minimizes $\mathbf{J}$, and use $\mathbf{x_{k+1}} = \mathbf{x_k} + \mathbf{a_k p_k}$
- Replace $\mathbf{r_k} = \mathbf{Ax_k} - \mathbf{b}$ with gradient of $\mathbf{J}$: $\bigtriangledown_a \mathbf{J}$
- This is Fletcher-Reeves version, Polak-Ribiere modifies $\beta$
- CG is one of the most competitive methods, but requires the Hessian to have low condition number
- Typically we do a few CG steps at each *k*, then move on to a new gradient evaluation

# CG vs. Gradient Descent

- In 2-d CG has to converge in 2 steps



# Gauss-Newton for Nonlinear Least Squares

$$\chi^2(\mathbf{a}) = \sum_{i=0}^{N-1} \left[ \frac{y_i - y(x_i|\mathbf{a})}{\sigma_i} \right]^2$$

$$\frac{\partial \chi^2}{\partial a_k} = -2 \sum_{i=0}^{N-1} \frac{[y_i - y(x_i|\mathbf{a})]}{\sigma_i^2} \frac{\partial y(x_i|\mathbf{a})}{\partial a_k} \qquad k = 0, 1, \ldots, M-1$$

$$\frac{\partial^2 \chi^2}{\partial a_k \partial a_l} = 2 \sum_{i=0}^{N-1} \frac{1}{\sigma_i^2} \left[ \frac{\partial y(x_i|\mathbf{a})}{\partial a_k} \frac{\partial y(x_i|\mathbf{a})}{\partial a_l} - [y_i - y(x_i|\mathbf{a})] \frac{\partial^2 y(x_i|\mathbf{a})}{\partial a_l \partial a_k} \right]$$

$$\beta_k \equiv -\frac{1}{2} \frac{\partial \chi^2}{\partial a_k} \qquad \alpha_{kl} \equiv \frac{1}{2} \frac{\partial^2 \chi^2}{\partial a_k \partial a_l} \qquad \sum_{l=0}^{M-1} \alpha_{kl}\, \delta a_l = \beta_k$$

$$\alpha_{kl} = \sum_{i=0}^{N-1} \frac{1}{\sigma_i^2} \left[ \frac{\partial y(x_i|\mathbf{a})}{\partial a_k} \frac{\partial y(x_i|\mathbf{a})}{\partial a_l} \right]$$

We drop 2nd term in Hessian because residual $r = y_i - y$ is small, fluctuates around 0 and because y'' may be small (or zero for linear problems)

Line search in direction $\delta a$

# Gauss-Newton + Trust Region = Levenberg-Marquardt Method

- Solving $\mathbf{A}^T\mathbf{A}\delta\mathbf{a} = \mathbf{A}^T\mathbf{b}$ is equivalent to minimize $|\mathbf{A}\delta\mathbf{a}\text{-}\mathbf{b}|^2$
- if trust region is within the solution just solve this equation
- If not we need to impose $||\delta\mathbf{a}||=\Delta_k$
- Lagrange multiplier minimization equivalent to $(\mathbf{A}^T\mathbf{A}+ \lambda\mathrm{I})\ \delta\mathbf{a} = \mathbf{A}^T\mathbf{b}$ and $\lambda(\Delta\text{-}||\delta\mathbf{a}||) = 0$
- For small $\lambda$ this is Gauss-Newton (use close to minimum), for large $\lambda$ this is steepest descent (use far from minimum)
- A good method for nonlinear least squares

# Summary

- Optimization one of key numerical methods of modern data analysis. Typical examples are nonlinear least square problem and ML parameters (e.g. neural networks etc.)
- If at this point you are confused which methods you should use you are not alone: it depends on application and often the best way to answer is to try
- Some general guidances: if there is a lot of parameters (e.g. ML) and likelihood evaluations are cheap then use 1st order methods

# Summary

- If the data is independent and there is a lot of data then use stochastic 1st order methods, e.g. ADAM
- If the likelihood evaluation is slow and number of parameters low use Newton or Gauss-Newton (e.g. Levenberg-Marquardt)
- If likelihood slow and number of parameters large use approximate Newton or Gauss-Newton (e.g. Steihaug with nonlinear CG), or use quasi-Newton (e.g. L-BFGS)
- Choosing a method is not enough: you also need to choose line search method (e.g. backtracking, Wolfe conditions) or trust region determination
- Typically these methods only find local minimum. Non-convex problems are hard: we will look at some stochastic methods (e.g. simulated annealing) in next lecture

# Literature

- *Numerical Recipes*, Press et al., Chapter 9, 10, 15
- *Computational Physics,* M. Newman, Chapter 6
- Nocedal and Wright, Optimization
- https://arxiv.org/abs/1609.04747