

Lecture 2

3B1B Optimization

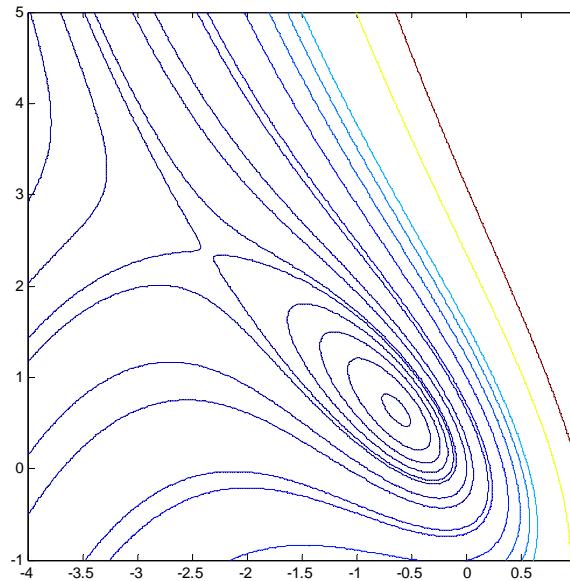
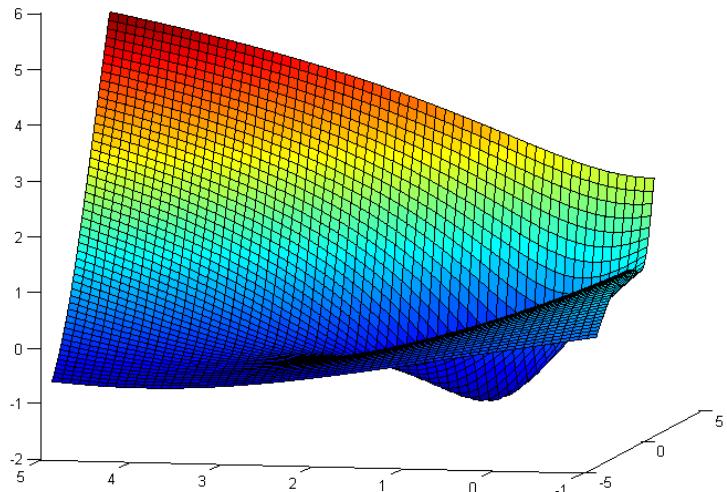
Michaelmas 2016

A. Zisserman

- Newton's method
 - Line search
- Quasi-Newton methods
- Least-Squares and Gauss-Newton methods
- Downhill simplex (amoeba) algorithm

Optimization for General Functions

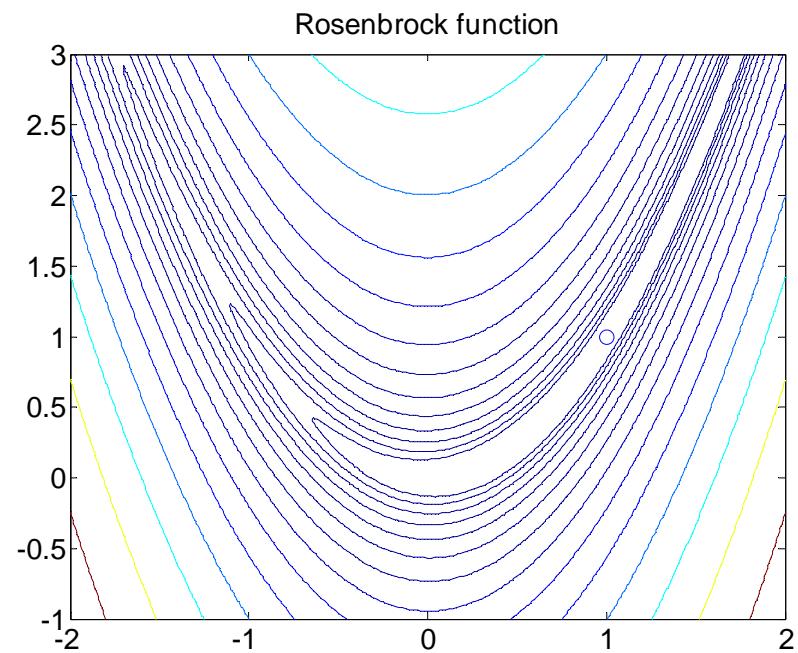
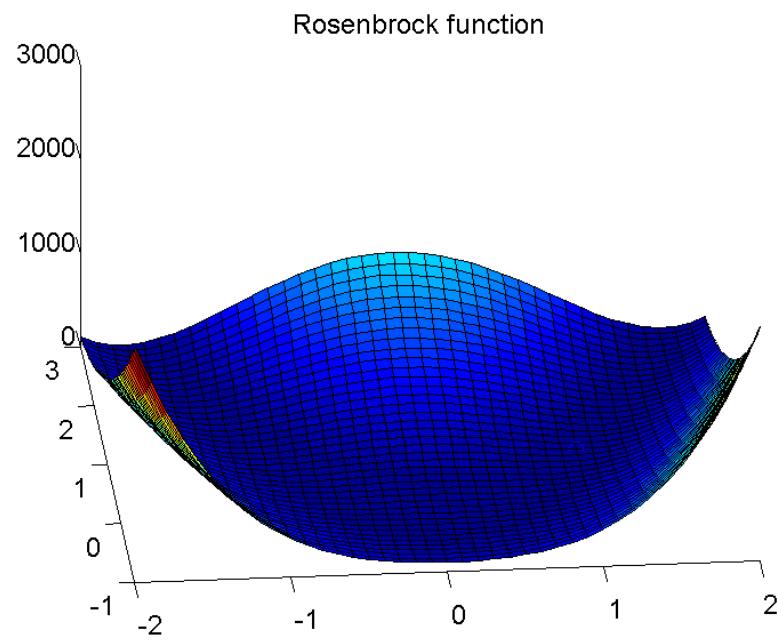
$$f(x, y) = \exp(x)(4x^2 + 2y^2 + 4xy + 2x + 1)$$



Apply methods developed using quadratic Taylor series expansion

Rosenbrock's function

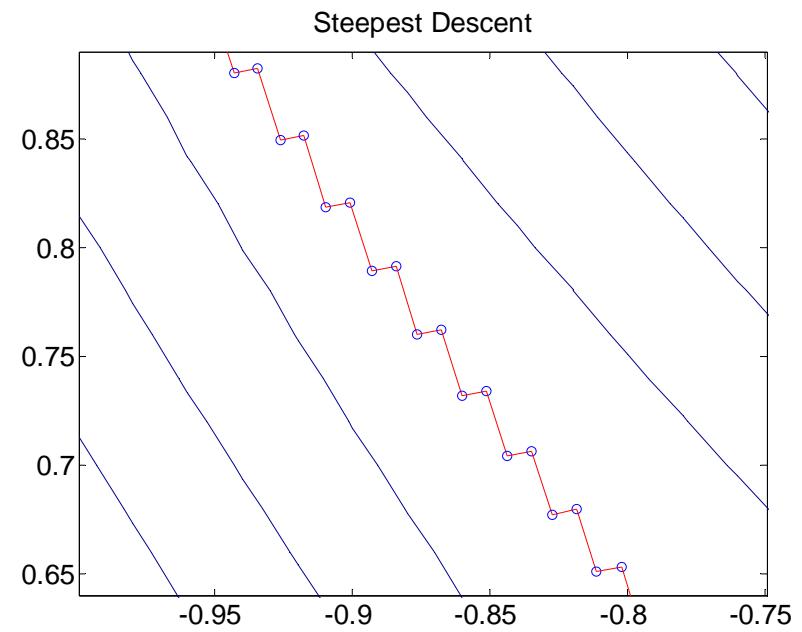
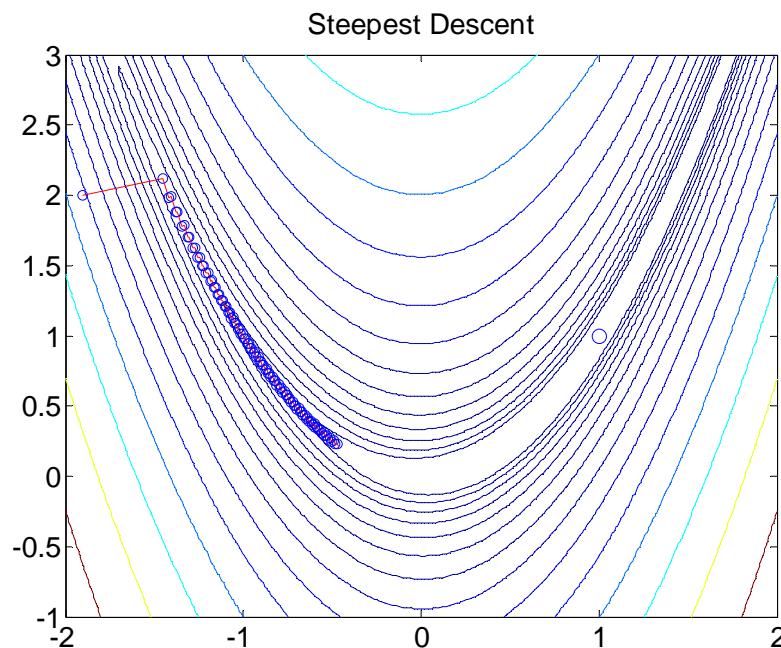
$$f(x, y) = 100(y - x^2)^2 + (1 - x)^2$$



Minimum is at [1, 1]

Steepest descent

- The 1D line minimization must be performed using one of the earlier methods (usually cubic polynomial interpolation)



- The zig-zag behaviour is clear in the zoomed view (100 iterations)
- The algorithm crawls along the valley

Performance issues for optimization algorithms

1. Number of iterations required
2. Cost per iteration
3. Memory footprint
4. Region of convergence

Recall from lecture 1: Newton's method in 1D

Fit a quadratic approximation to $f(x)$ using both gradient and curvature information at x .

- Expand $f(x)$ locally using a Taylor series

$$f(x + \delta x) = f(x) + \delta x f'(x) + \frac{\delta x^2}{2} f''(x) + \text{h.o.t}$$

- Find the δx which minimizes this local quadratic approximation

$$f'(x + \delta x) = f'(x) + \delta x f''(x) = 0$$

- and rearranging

$$\delta x = -\frac{f'(x)}{f''(x)}$$

- Update for x

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

Recall from lecture 1: Taylor expansion in 2D

A function may be approximated locally by its Taylor series expansion about a point \mathbf{x}_0

$$f(\mathbf{x}_0 + \boldsymbol{\delta}\mathbf{x}) \approx f(\mathbf{x}_0) + \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right) \begin{pmatrix} \delta x \\ \delta y \end{pmatrix} + \frac{1}{2} (\delta x, \delta y) \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix} \begin{pmatrix} \delta x \\ \delta y \end{pmatrix} + \text{h.o.t}$$

The expansion to second order is a [quadratic function](#)

$$f(\mathbf{x}_0 + \boldsymbol{\delta}\mathbf{x}) = a + \mathbf{g}^\top \boldsymbol{\delta}\mathbf{x} + \frac{1}{2} \boldsymbol{\delta}\mathbf{x}^\top \mathbf{H} \boldsymbol{\delta}\mathbf{x}$$

Newton's method in ND

Expand $f(\mathbf{x})$ by its Taylor series about the point \mathbf{x}_n

$$f(\mathbf{x}_n + \delta\mathbf{x}) \approx f(\mathbf{x}_n) + \mathbf{g}_n^\top \delta\mathbf{x} + \frac{1}{2} \delta\mathbf{x}^\top \mathbf{H}_n \delta\mathbf{x}$$

where the gradient is the vector

$$\mathbf{g}_n = \nabla f(\mathbf{x}_n) = \left[\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_N} \right]^\top$$

and the Hessian is the symmetric matrix

$$\mathbf{H}_n = \mathbf{H}(\mathbf{x}_n) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_N} \\ \vdots & \ddots & \\ \frac{\partial^2 f}{\partial x_1 \partial x_N} & & \frac{\partial^2 f}{\partial x_N^2} \end{bmatrix}$$

For a minimum we require that $\nabla f(\mathbf{x}) = \mathbf{0}$, and so

$$\nabla f(\mathbf{x}) = \mathbf{g}_n + \mathbf{H}_n \delta\mathbf{x} = \mathbf{0}$$

with solution $\delta\mathbf{x} = -\mathbf{H}_n^{-1} \mathbf{g}_n$. This gives the iterative update

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{H}_n^{-1} \mathbf{g}_n$$

Assume that H is positive definite (all eigenvalues greater than zero)

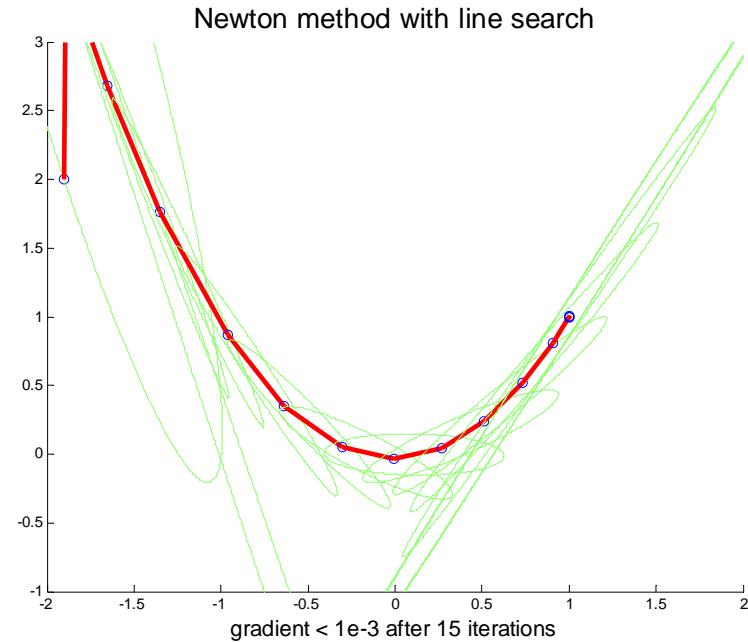
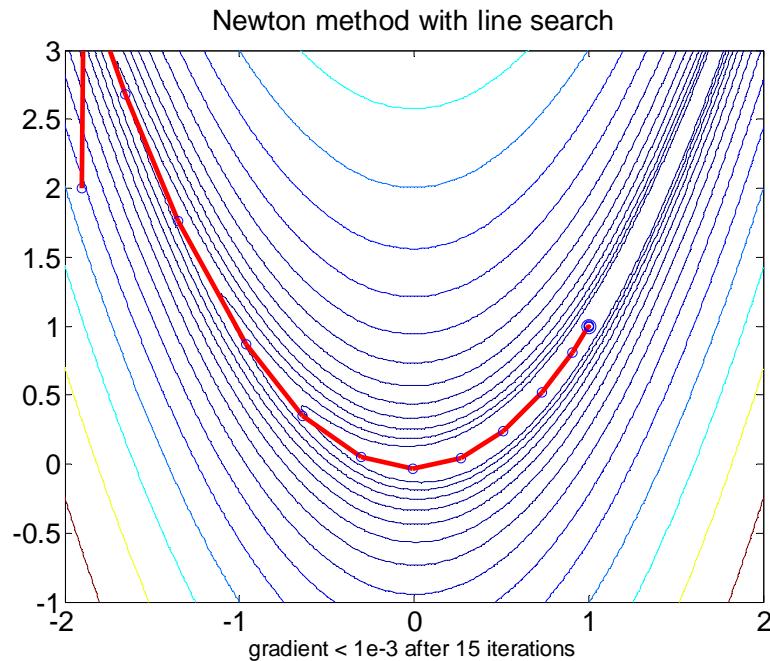
$$\begin{aligned}\mathbf{x}_{n+1} &= \mathbf{x}_n + \delta\mathbf{x} \\ &= \mathbf{x}_n - H_n^{-1}\mathbf{g}_n\end{aligned}$$

- If $f(\mathbf{x})$ is quadratic, then the solution is found in one step.
- The method has quadratic convergence (as in the 1D case).
- The solution $\delta\mathbf{x} = -H_n^{-1}\mathbf{g}_n$ is guaranteed to be a downhill direction (provided that H is positive definite)
- For numerical reasons the inverse is not actually computed, instead $\delta\mathbf{x}$ is computed as the solution of $H\delta\mathbf{x} = -\mathbf{g}_n$.
- Rather than jump straight to $\mathbf{x}_n - H_n^{-1}\mathbf{g}_n$, it is better to perform a line search which ensures global convergence

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \alpha_n H_n^{-1}\mathbf{g}_n$$

- If $H = I$ then this reduces to steepest descent.

Newton's method - example



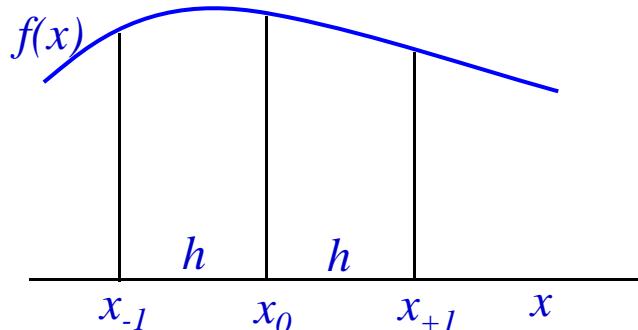
ellipses show successive quadratic approximations

- The algorithm converges in only 15 iterations compared to hundreds for steepest descent.
- **However**, the method requires computing the Hessian matrix at each iteration – this is not always feasible

Quasi-Newton methods

- If the problem size is large and the Hessian matrix is dense then it may be infeasible/inconvenient to compute it directly.
- Quasi-Newton methods avoid this problem by keeping a “rolling estimate” of $H(x)$, updated at each iteration using new gradient information.
- Common schemes are due to Broyden, Fletcher, Goldfarb and Shanno (BFGS), and also Davidson, Fletcher and Powell (DFP).

e.g. in 1D



First derivatives

$$f'(x_0 + \frac{h}{2}) = \frac{f_1 - f_0}{h} \quad \text{and} \quad f'(x_0 - \frac{h}{2}) = \frac{f_0 - f_{-1}}{h}$$

second derivative

$$f''(x_0) = \frac{\frac{f_1 - f_0}{h} - \frac{f_0 - f_{-1}}{h}}{h} = \frac{f_1 - 2f_0 + f_{-1}}{h^2}$$

For H_{n+1} build an approximation from $H_n, g_n, g_{n+1}, x_n, x_{n+1}$

Quasi-Newton: BFGS

- Set $H_0 = I$.
- Update according to

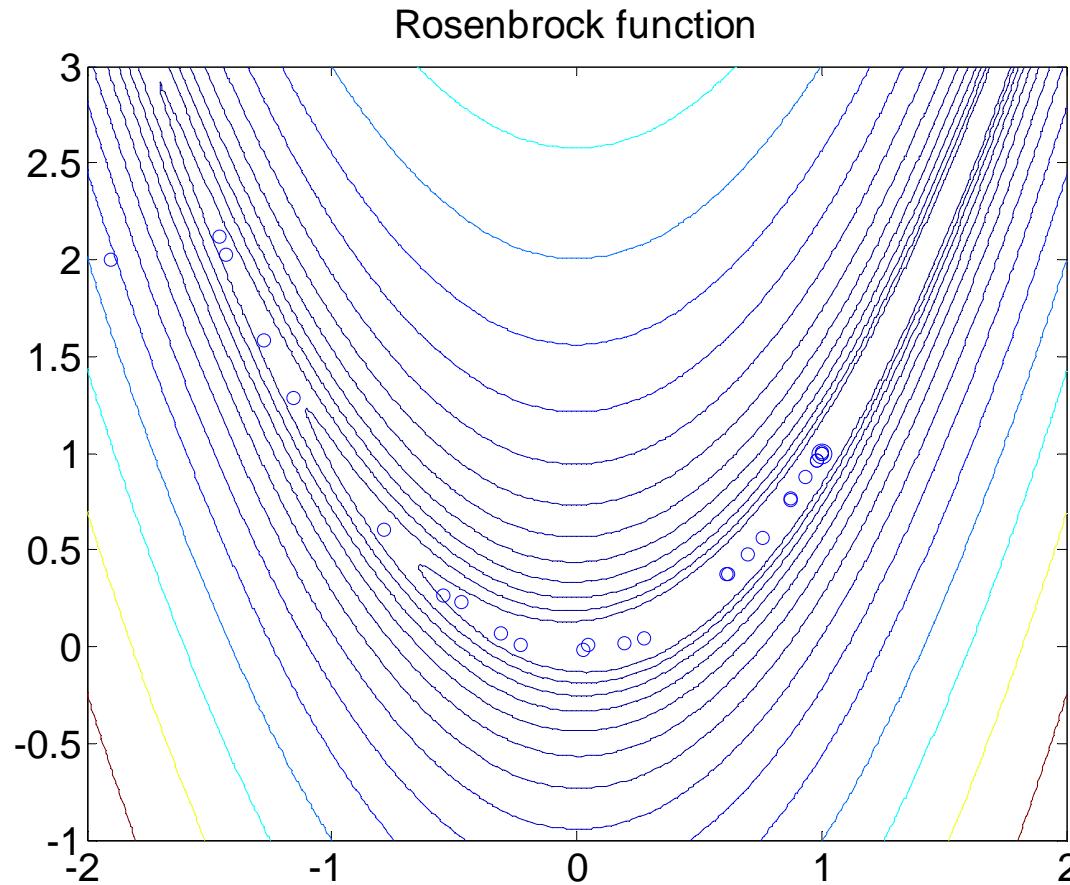
$$H_{n+1} = H_n + \frac{q_n q_n^\top}{q_n^\top s_n} - \frac{(H_n s_n) (H_n s_n)^\top}{s_n^\top H_n s_n}$$

where

$$\begin{aligned} s_n &= x_{n+1} - x_n \\ q_n &= g_{n+1} - g_n \end{aligned}$$

- The matrix itself is not stored, but rather represented compactly by a few stored vectors.
- The estimate H_{n+1} is used to form a local quadratic approximation as before.

Example



- The method converges in 25 iterations, compared to 15 for the full-Newton method
- In Matlab the optimization function '[fminunc](#)' uses a BFGS quasi-Newton method for medium scale optimization problems

Matlab – fminunc

```
>> f='100*(x(2)-x(1)^2)^2+(1-x(1))^2';  
  
>> GRAD=[100*(4*x(1)^3-4*x(1)*x(2))+2*x(1)-2; 100*(2*x(2)-2*x(1)^2)];
```

Choose options for BFGS quasi-Newton

```
>> OPTIONS=optimset('LargeScale','off', 'HessUpdate','bfgs' );  
>> OPTIONS = optimset(OPTIONS,'gradobj','on');
```

Start point

```
>> x = [-1.9; 2];  
  
>> [x,fval] = fminunc({f,GRAD},x,OPTIONS);
```

This produces

$x = 0.9998, 0.9996$ $fval = 3.4306e-008$

Non-linear least squares

- It is **very** common in applications for a cost function $f(\mathbf{x})$ to be the sum of a large number of squared residuals

$$f(\mathbf{x}) = \sum_{i=1}^M r_i^2$$

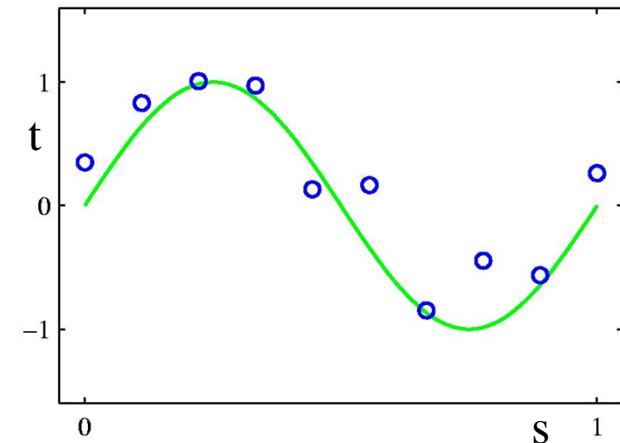
- If each residual depends **non-linearly** on the parameters \mathbf{x} then the minimization of $f(\mathbf{x})$ is a non-linear least squares problem.
- Examples arise in non-linear regression (fitting) of data

Linear least squares reminder

The goal is to fit a smooth curve to measured data points $\{s_i, t_i\}$ by minimizing the cost

$$f(\mathbf{x}) = \sum_{i=1} r_i^2 = \sum_{i=1} (y(s_i, \mathbf{x}) - t_i)^2$$

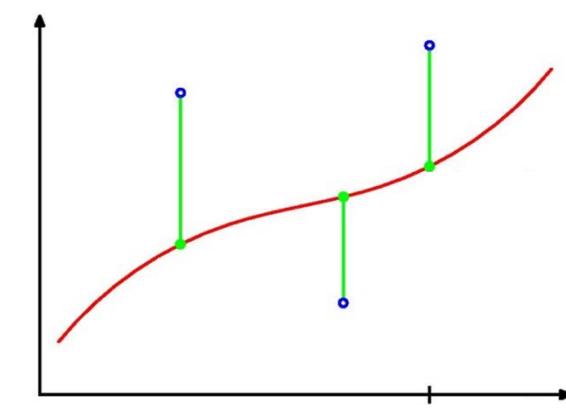
↑
target value



For example, the regression function $y(s_i, \mathbf{x})$ might be polynomial

$$y(s, \mathbf{x}) = x_0 + x_1 s + x_2 s^2 + \dots$$

In this case the function is linear in the parameter \mathbf{x} and there is a closed form solution. In general there will not be a closed form solution for non-linear functions $y(s, \mathbf{x})$.



Non-linear least squares example: aligning a 3D model to an image



Input:

3D textured face model, camera model, image $I(x, y)$.

Task:

Determine the 3D rotation and 3D translation that minimizes the error between image $I(x, y)$ and the projected 3D model



Cost function

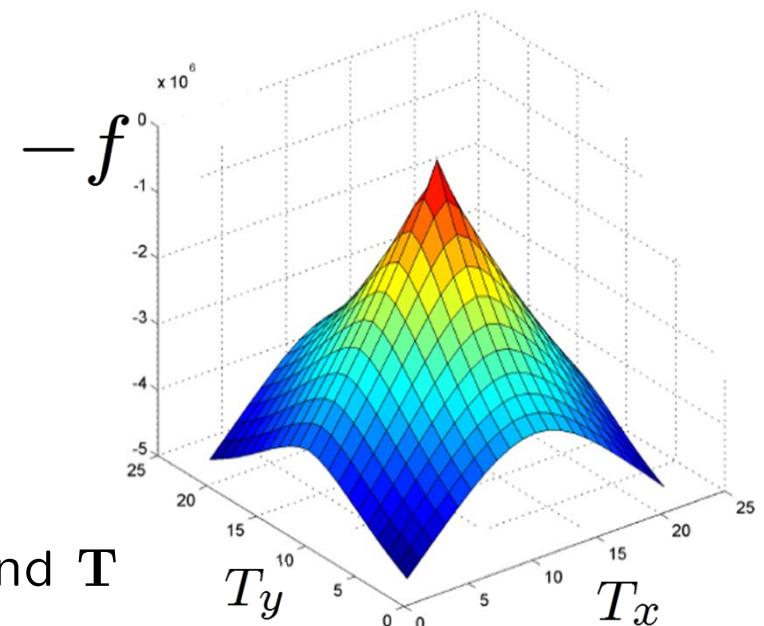
$$f(\mathbf{R}, \mathbf{T}) = \sum_{x,y}^M |\hat{I}_{\mathbf{R}, \mathbf{T}}(x, y) - I(x, y)|^2$$

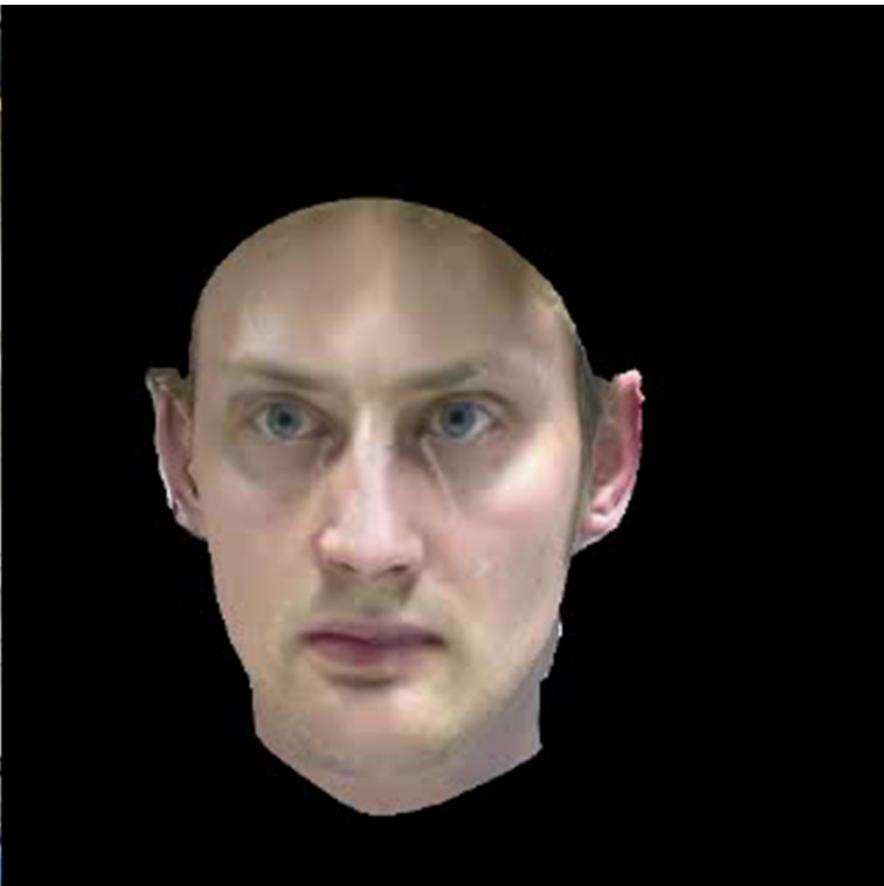
Transformation parameters:

- 3D rotation matrix \mathbf{R}
- translation 3-vector $\mathbf{T} = (T_x, T_y, T_z)^\top$

Image generation:

- rotate and translate 3D model by \mathbf{R} and \mathbf{T}
- project to generate image $\hat{I}_{\mathbf{R}, \mathbf{T}}(x, y)$







Non-linear least squares

$$f(\mathbf{x}) = \sum_{i=1}^M r_i^2 = \|\mathbf{r}\|^2$$

The $M \times N$ **Jacobian** of the vector of residuals \mathbf{r} is defined as

assume
 $M > N$

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} \frac{\partial r_1}{\partial x_1} & \cdots & \frac{\partial r_1}{\partial x_N} \\ \vdots & \ddots & \\ \frac{\partial r_M}{\partial x_1} & & \frac{\partial r_M}{\partial x_N} \end{pmatrix} \quad \boxed{\mathbf{J}}$$

Consider

$$\frac{\partial}{\partial x_k} \sum_i r_i^2 = \sum_i 2r_i \frac{\partial r_i}{\partial x_k}$$

Hence

$$\nabla f(\mathbf{x}) = 2\mathbf{J}^\top \mathbf{r} \quad \boxed{[}] = \left(\begin{array}{c} \mathbf{J}^\top \\ \vdots \end{array} \right) \boxed{[|]}$$

For the Hessian we require

$$\begin{aligned}\frac{\partial^2}{\partial x_l \partial x_k} \sum_i r_i^2 &= 2 \frac{\partial}{\partial x_l} \sum_i r_i \frac{\partial r_i}{\partial x_k} \\ &= 2 \sum_i \frac{\partial r_i}{\partial x_k} \frac{\partial r_i}{\partial x_l} + 2 \sum_i r_i \frac{\partial^2 r_i}{\partial x_k \partial x_l}\end{aligned}$$

Hence

$$H(\mathbf{x}) = 2J^\top J + 2 \sum_{i=1}^M r_i R_i$$
$$\left(\begin{array}{c} J^\top \\ J \end{array} \right) \left[\begin{array}{c} R_i \end{array} \right]$$

- Note that the second-order term in the Hessian $H(x)$ is multiplied by the residuals r_i .
- In most problems, the residuals will typically be small.
- Also, at the minimum, the residuals will typically be distributed with mean = 0.
- For these reasons, the second-order term is often ignored, giving the [Gauss-Newton](#) approximation to the Hessian :

$$H(x) = 2J^T J$$

- Hence, explicit computation of the full Hessian can again be avoided.

Example – Gauss-Newton

The minimization of the Rosenbrock function

$$f(x, y) = 100(y - x^2)^2 + (1 - x)^2$$

can be written as a least-squares problem with residual vector

$$\mathbf{r} = \begin{bmatrix} 10(y - x^2) \\ (1 - x) \end{bmatrix}$$

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} \frac{\partial r_1}{\partial x} & \frac{\partial r_1}{\partial y} \\ \frac{\partial r_2}{\partial x} & \frac{\partial r_2}{\partial y} \end{pmatrix} = \begin{pmatrix} -20x & 10 \\ -1 & 0 \end{pmatrix}$$

The true Hessian is

$$H(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix} = \begin{bmatrix} 1200x^2 - 400y + 2 & -400x \\ -400x & 200 \end{bmatrix}$$

The Gauss-Newton approximation of the Hessian is

$$2J^T J = 2 \begin{bmatrix} -20x & -1 \\ 10 & 0 \end{bmatrix} \begin{bmatrix} -20x & 10 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 800x^2 + 2 & -400x \\ -400x & 200 \end{bmatrix}$$

Summary: Gauss-Newton optimization

For a cost function $f(\mathbf{x})$ that is a sum of squared residuals

$$f(\mathbf{x}) = \sum_{i=1} r_i^2$$

The Hessian can be approximated as

$$\mathbf{H}(\mathbf{x}) = 2\mathbf{J}^\top \mathbf{J}$$

and the gradient is given by

$$\nabla f(\mathbf{x}) = 2\mathbf{J}^\top \mathbf{r}$$

So, the Newton update step

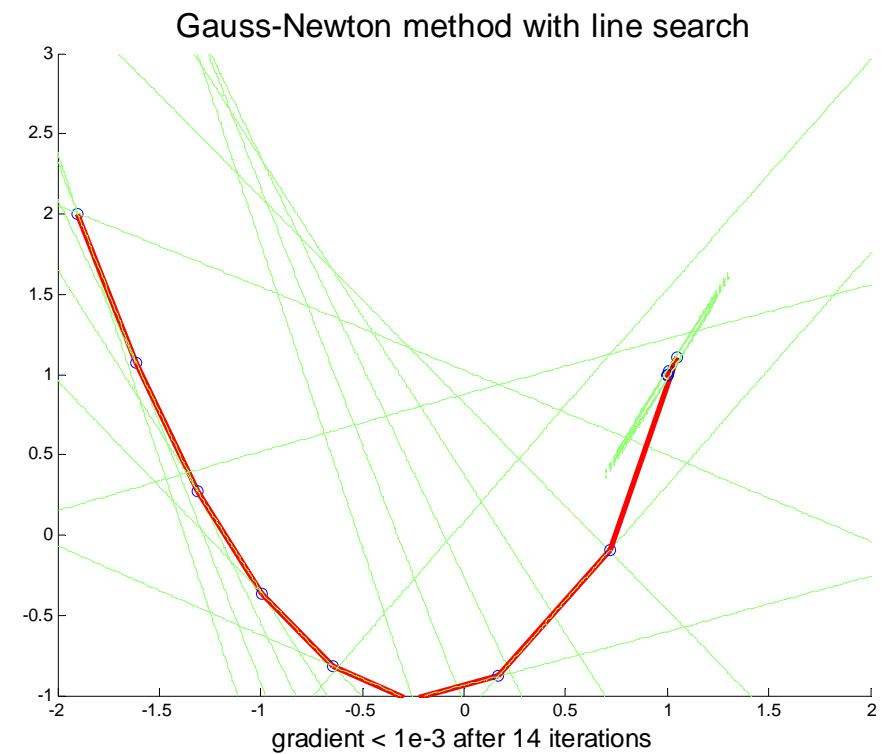
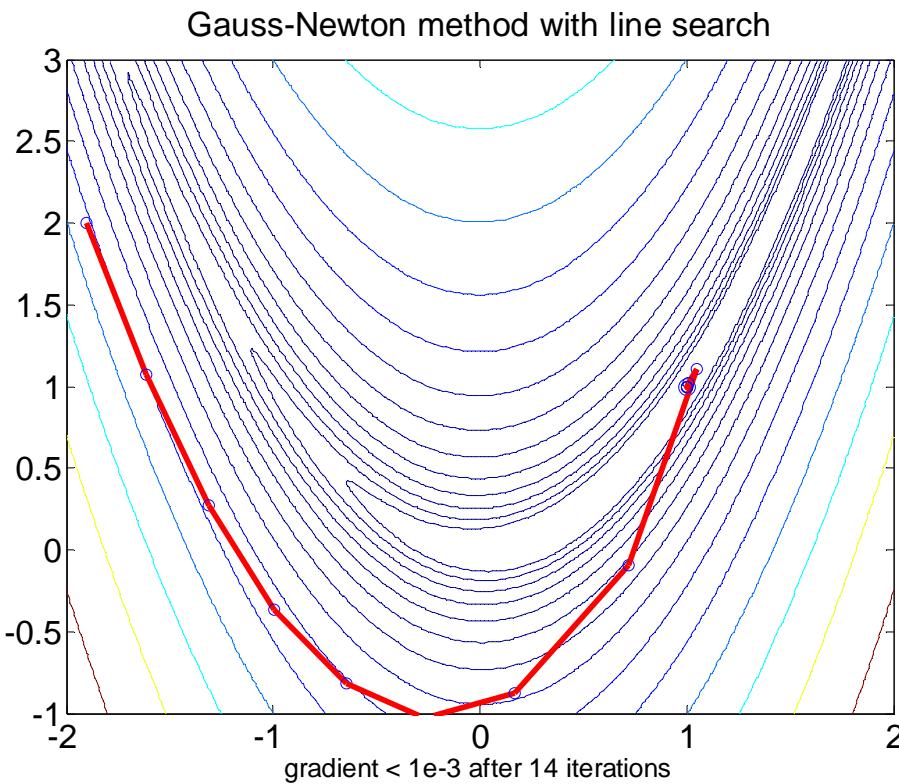
$$\begin{aligned}\mathbf{x}_{n+1} &= \mathbf{x}_n + \delta \mathbf{x} \\ &= \mathbf{x}_n - \mathbf{H}_n^{-1} \mathbf{g}_n\end{aligned}$$

computed as $\mathbf{H} \delta \mathbf{x} = -\mathbf{g}_n$, becomes

$$\mathbf{J}^\top \mathbf{J} \delta \mathbf{x} = -\mathbf{J}^\top \mathbf{r}$$

These are called the [normal equations](#).

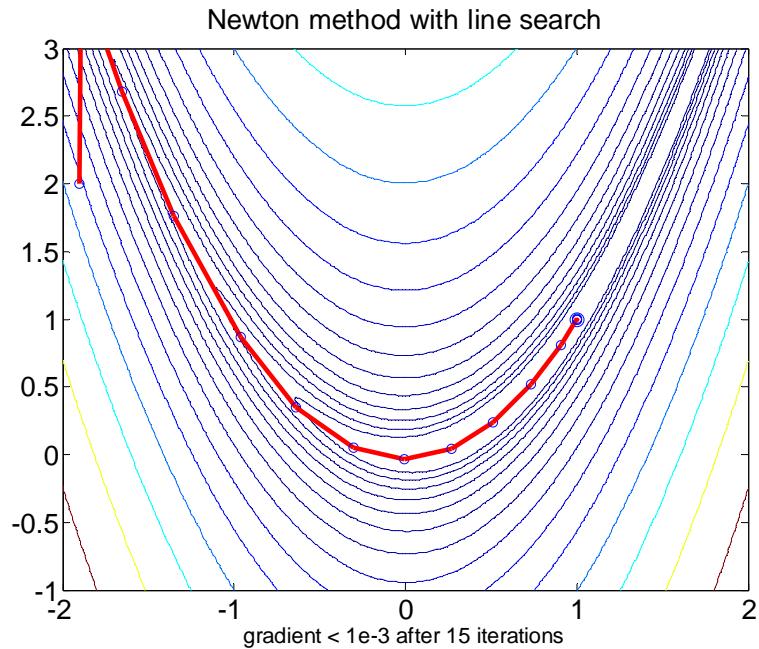
$$\mathbf{x}_{n+1} = \mathbf{x}_n - \alpha_n \mathbf{H}_n^{-1} \mathbf{g}_n \quad \text{with} \quad \mathbf{H}_n(\mathbf{x}) = 2\mathbf{J}_n^\top \mathbf{J}_n$$



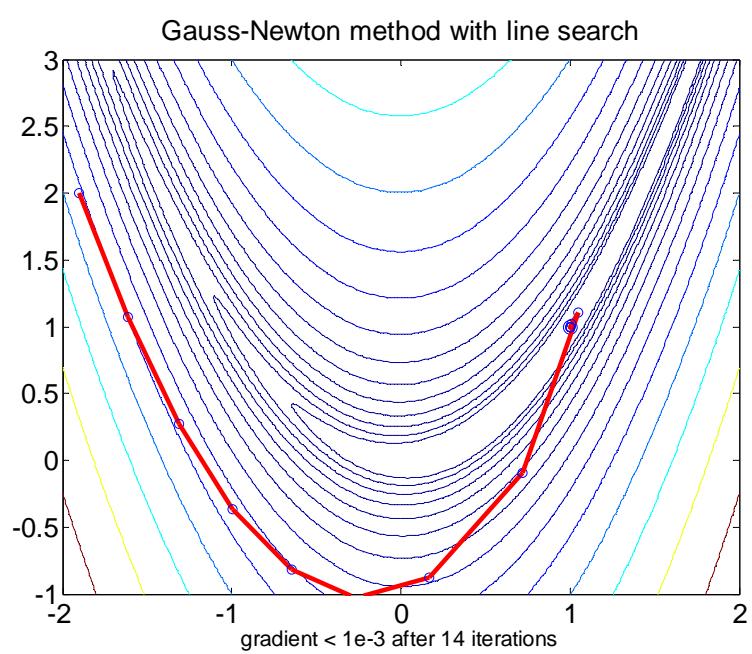
- minimization with the Gauss-Newton approximation with line search takes only 14 iterations

Comparison

Newton



Gauss-Newton



- requires computing Hessian (i.e. n^2 second derivatives)
- exact solution if quadratic

- approximates Hessian by Jacobian product
- requires only n first derivatives

Properties of methods

- **Gradient descent**

- will zig-zag – each new increment is perpendicular to previous.
- Requires 1D search
- Slow to converge.

- **Newton's method**

- requires computation of Hessian.
- Can converge to maximum or saddle as well as minimum.
- Can be unstable.

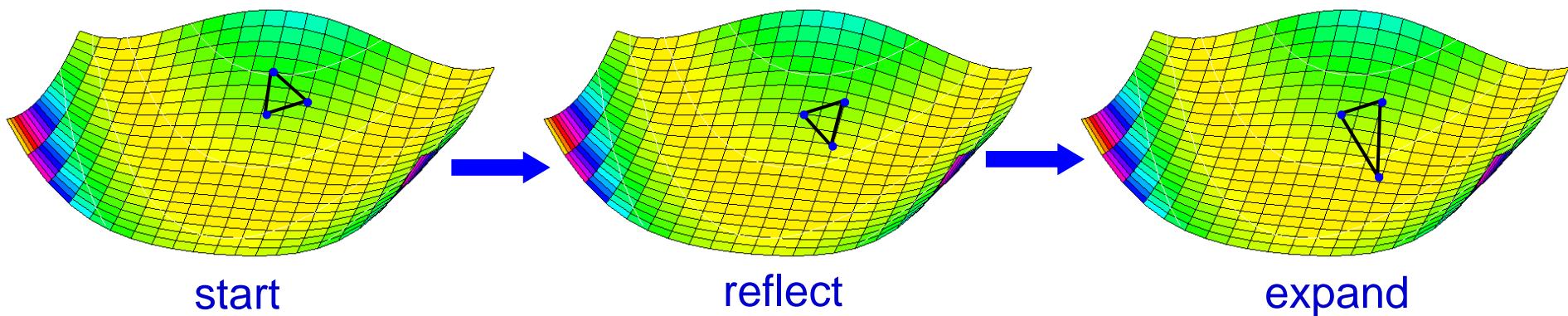
- **Gauss-Newton**

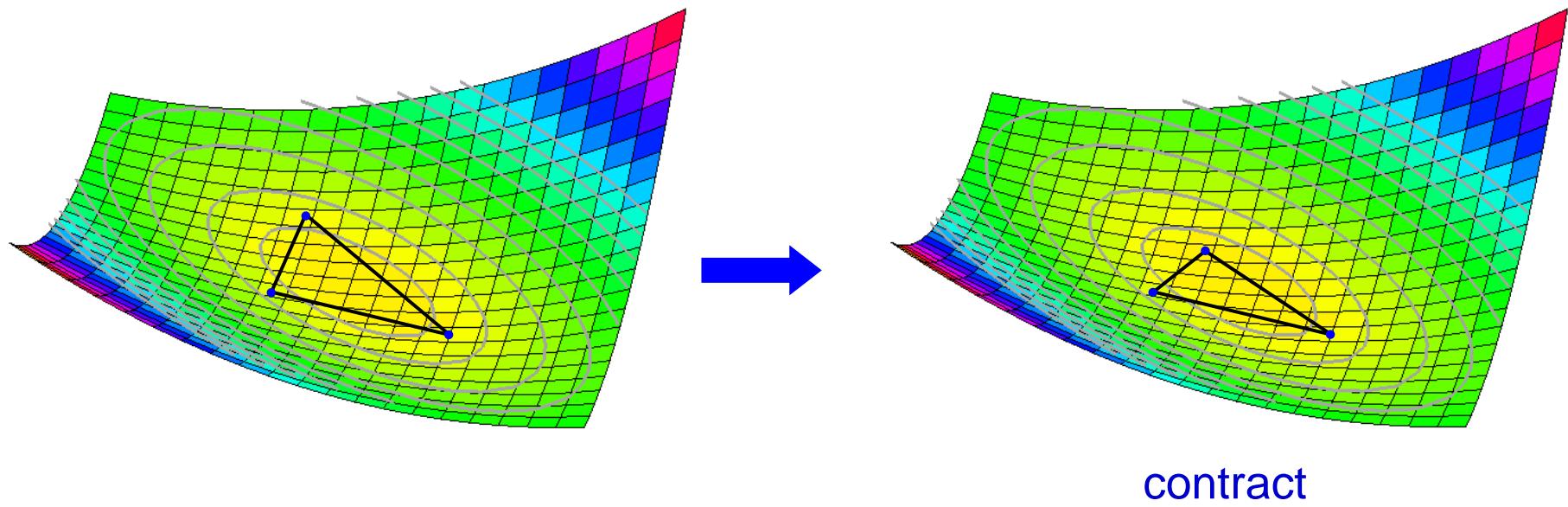
- Is a downhill method, so will not converge to maximum or saddle.
- Can be unstable, thus preferably needs line search.

The downhill simplex (amoeba) algorithm

The downhill simplex (amoeba) algorithm

- Due to Nelder and Mead (1965)
- A *direct* method: only uses function evaluations (no derivatives)
- a simplex is the polytope in N dimensions with $N+1$ vertices, e.g.
 - 2D: triangle
 - 3D: tetrahedron
- basic idea: move by reflections, expansions or contractions





One iteration of the simplex algorithm

- Reorder the points so that $f(\mathbf{x}_{n+1}) > f(\mathbf{x}_2) > f(\mathbf{x}_1)$ (i.e. \mathbf{x}_{n+1} is the worst point).
- Generate a trial point \mathbf{x}_r by *reflection*

$$\mathbf{x}_r = \bar{\mathbf{x}} + \alpha(\bar{\mathbf{x}} - \mathbf{x}_{n+1})$$

where $\bar{\mathbf{x}} = (\sum_i \mathbf{x}_i) / (N + 1)$ is the centroid and $\alpha > 0$. Compute $f(\mathbf{x}_r)$, and there are then 3 possibilities:

1. $f(\mathbf{x}_1) < f(\mathbf{x}_r) < f(\mathbf{x}_n)$ (i.e. \mathbf{x}_r is neither the new best or worst point), replace \mathbf{x}_{n+1} by \mathbf{x}_r .
2. $f(\mathbf{x}_r) < f(\mathbf{x}_1)$ (i.e. \mathbf{x}_r is the new best point), then assume direction of reflection is good and generate a new point by *expansion*

$$\mathbf{x}_e = \mathbf{x}_r + \beta(\mathbf{x}_r - \bar{\mathbf{x}})$$

where $\beta > 0$. If $f(\mathbf{x}_e) < f(\mathbf{x}_r)$ then replace \mathbf{x}_{n+1} by \mathbf{x}_e , otherwise, the expansion has failed, replace \mathbf{x}_{n+1} by \mathbf{x}_r .

3. $f(\mathbf{x}_r) > f(\mathbf{x}_n)$ then assume the polytope is too large and generate a new point by *contraction*

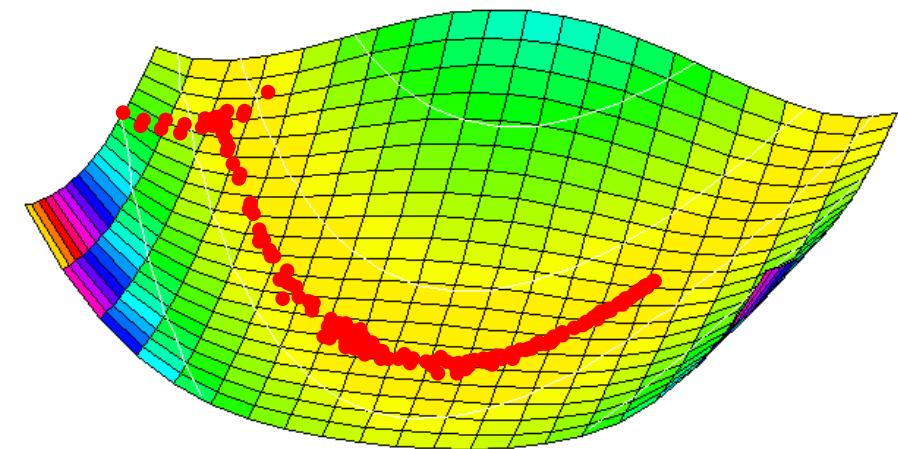
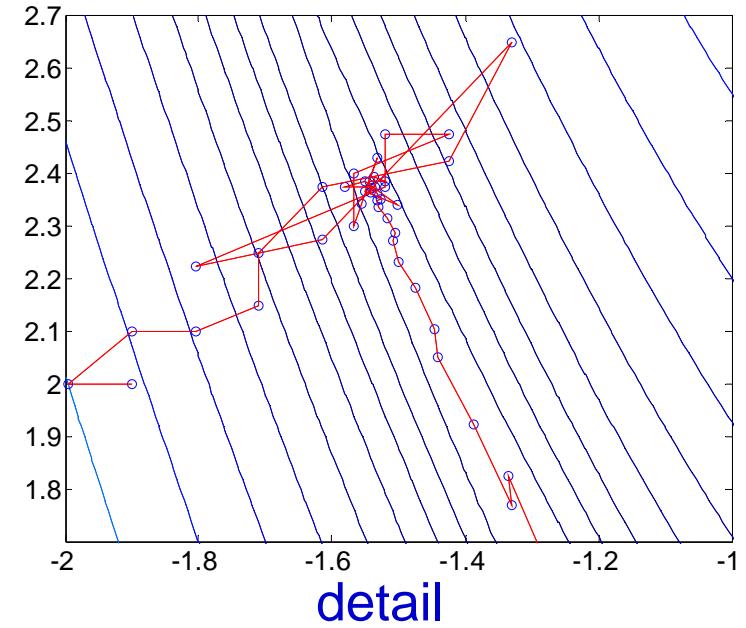
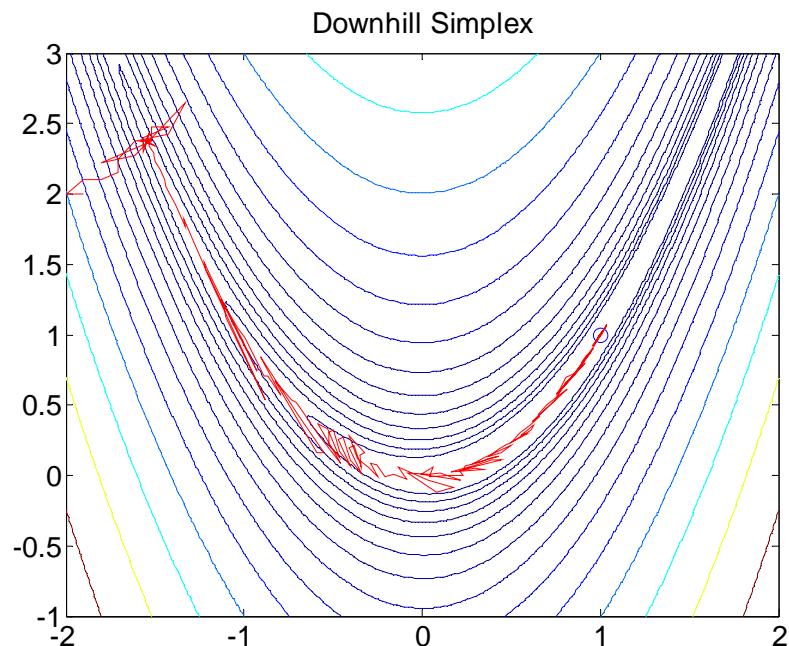
$$\mathbf{x}_c = \bar{\mathbf{x}} + \gamma(\mathbf{x}_{n+1} - \bar{\mathbf{x}})$$

where γ ($0 < \gamma < 1$) is the contraction coefficient. If $f(\mathbf{x}_c) < f(\mathbf{x}_{n+1})$ then the contraction has succeeded and replace \mathbf{x}_{n+1} by \mathbf{x}_c , otherwise contract again.

Standard values are $\alpha = 1, \beta = 1, \gamma = 0.5$.

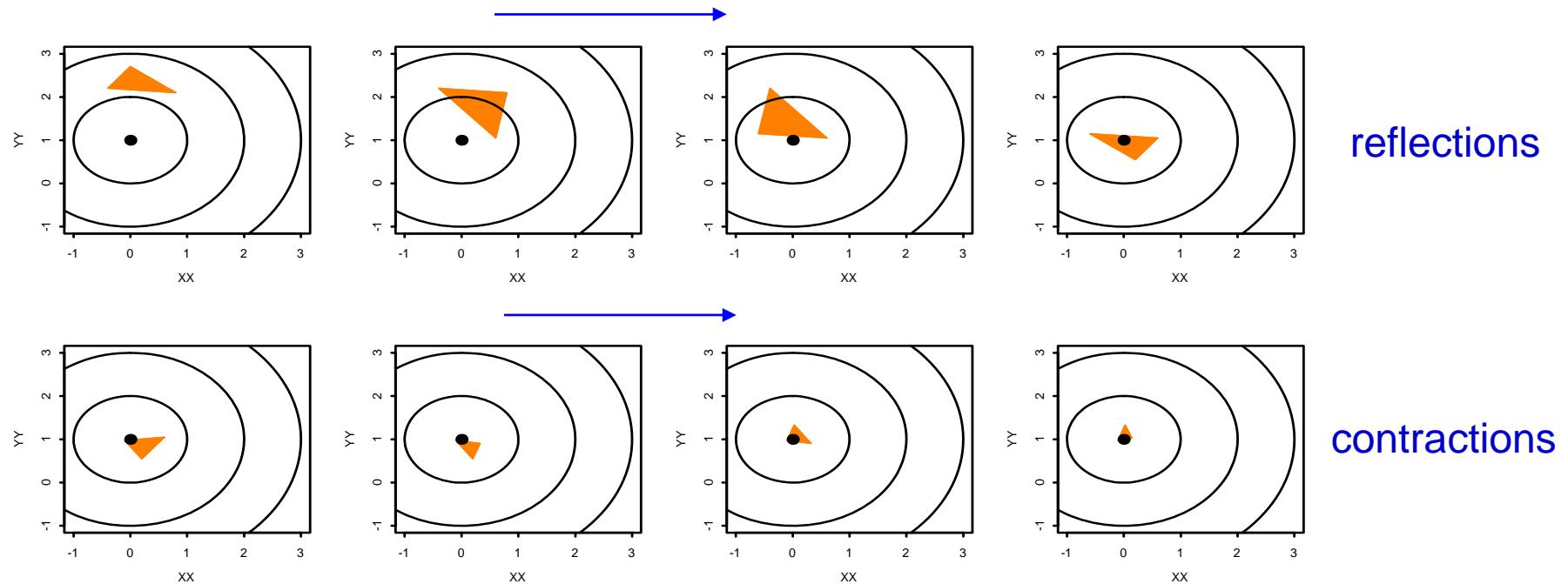
Example

Path of best vertex



Matlab fminsearch
with 200 iterations

Example 2: contraction about a minimum



Summary

- no derivatives required
- deals well with noise in the cost function
- is able to crawl out of some local minima (though, of course, can still get stuck)

Matlab – fminsearch

Nelder-Mead simplex direct search

```
>> banana = @(x)100*(x(2)-x(1)^2)^2+(1-x(1))^2;
```

Pass the function handle to fminsearch:

```
>> [x,fval] = fminsearch(banana,[-1.9, 2])
```

This produces

x = 1.0000 1.0000

fval = 4.0686e-010

Google to find out more on using this function

What is next?

- Move from general and quadratic optimization problems to linear programming
- Constrained optimization