• 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 $\delta x$ 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 $x_0$

$$f(x_0 + \delta x) \approx f(x_0) + \left( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right) \begin{pmatrix} \delta x \\ \delta y \end{pmatrix} + \frac{1}{2} (\delta x, \delta y) \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix} \begin{pmatrix} \delta x \\ \delta y \end{pmatrix} + \text{h.o.t}$$

The expansion to second order is a quadratic function

$$f(x_0 + \delta x) = a + g^\top \delta x + \frac{1}{2} \delta x^\top \mathbf{H} \delta x$$
Newton’s method in ND

Expand \( f(x) \) by its Taylor series about the point \( x_n \)

\[
f(x_n + \delta x) \approx f(x_n) + g_n^\top \delta x + \frac{1}{2} \delta x^\top H_n \delta x
\]

where the gradient is the vector

\[
g_n = \nabla f(x_n) = \begin{bmatrix} \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_N} \end{bmatrix}^\top
\]

and the Hessian is the symmetric matrix

\[
H_n = H(x_n) = \begin{bmatrix}
\frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_N} \\
\frac{\partial^2 f}{\partial x_2 \partial x_1} & \ddots & \cdots \\
\vdots & \ddots & \ddots \\
\frac{\partial^2 f}{\partial x_N \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_N^2}
\end{bmatrix}
\]

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

\[
\nabla f(x) = g_n + H_n \delta x = 0
\]

with solution \( \delta x = -H_n^{-1}g_n \). This gives the iterative update

\[
x_{n+1} = x_n - H_n^{-1}g_n
\]
Assume that \( H \) is positive definite (all eigenvalues greater than zero)

\[
x_{n+1} = x_n + \delta x = x_n - H_n^{-1} g_n
\]

- If \( f(x) \) is quadratic, then the solution is found in one step.
- The method has quadratic convergence (as in the 1D case).
- The solution \( \delta x = -H_n^{-1} g_n \) is guaranteed to be a downhill direction (provided that \( H \) is positive definite).
- For numerical reasons the inverse is not actually computed, instead \( \delta x \) is computed as the solution of \( H \delta x = -g_n \).
- Rather than jump straight to \( x_n - H_n^{-1} g_n \), it is better to perform a line search which ensures global convergence
  \[
x_{n+1} = x_n - \alpha_n H_n^{-1} g_n
\]
- If \( H = I \) then this reduces to steepest descent.
Newton’s method - example

- 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 $\mathbb{H}(x)$, 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 $\mathbb{H}_{n+1}$ build an approximation from $\mathbb{H}_n, g_n, g_{n+1}, x_n, x_{n+1}$
Quasi-Newton: BFGS

- Set $H_0 = I$.

- Update according to

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

where

$$s_n = x_{n+1} - x_n$$
$$q_n = g_{n+1} - g_n$$

- 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.
• 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

```matlab
>> 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 \( f(x) \) to be the sum of a large number of squared residuals

\[
f(x) = \sum_{i=1}^{M} r_i^2
\]

- If each residual depends **non-linearly** on the parameters \( x \) then the minimization of \( f(x) \) 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

\[
f(x) = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (y(s_i, x) - t_i)^2
\]

For example, the regression function \( y(s_i, x) \) might be polynomial

\[
y(s, x) = x_0 + x_1 s + x_2 s^2 + \ldots
\]

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

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

Transformation parameters:
- 3D rotation matrix \( R \)
- translation 3-vector \( T = (T_x, T_y, T_z)^\top \)

Image generation:
- rotate and translate 3D model by \( R \) and \( T \)
- project to generate image \( \hat{I}_{R,T}(x, y) \)
Non-linear least squares

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

The \( M \times N \) **Jacobian** of the vector of residuals \( r \) is defined as assume \( M > N \)

\[
J(x) = \begin{pmatrix}
\frac{\partial r_1}{\partial x_1} & \cdots & \frac{\partial r_1}{\partial x_N} \\
\vdots & \ddots & \vdots \\
\frac{\partial r_M}{\partial x_1} & \cdots & \frac{\partial r_M}{\partial x_N}
\end{pmatrix}
\]

Consider

\[
\frac{\partial}{\partial x_k} \sum_i r_i^2 = \sum_i 2r_i \frac{\partial r_i}{\partial x_k}
\]

Hence

\[
\nabla f(x) = 2J^T r
\]

\[
\begin{bmatrix}
\end{bmatrix} = \begin{bmatrix}
J^T
\end{bmatrix}
\]
For the Hessian we require

\[ \frac{\partial^2}{\partial x_l \partial x_k} \sum_i r_i^2 = 2 \frac{\partial}{\partial x_l} \sum_i r_i \frac{\partial r_i}{\partial x_k} \]

\[ = 2 \sum_i \frac{\partial r_i}{\partial x_k} \frac{\partial r_i}{\partial x_l} + 2 \sum_i r_i \frac{\partial^2 r_i}{\partial x_k \partial x_l} \]

Hence

\[ H(x) = 2J^T J + 2 \sum_{i=1}^{M} r_i \mathbf{R}_i \]
• Note that the second-order term in the Hessian $\mathbf{H}(\mathbf{x})$ 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:

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

• 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

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

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

$$2J^\top J = 2 \begin{bmatrix}
-20x & -1 \\
10 & 0
\end{bmatrix} \begin{bmatrix}
-20x & 10 \\
-1 & 0
\end{bmatrix} = \begin{bmatrix}
800x^2 + 2 & -400x \\
-400x & 200
\end{bmatrix}$$
Summary: Gauss-Newton optimization

For a cost function \( f(x) \) that is a sum of squared residuals

\[
f(x) = \sum_{i=1}^{r_i^2}
\]

The Hessian can be approximated as

\[
H(x) = 2J^\top J
\]

and the gradient is given by

\[
\nabla f(x) = 2J^\top r
\]

So, the Newton update step

\[
x_{n+1} = x_n + \delta x
\]

\[
= x_n - H_n^{-1}g_n
\]

computed as \( H \delta x = -g_n \), becomes

\[
J^\top J \delta x = -J^\top r
\]

These are called the normal equations.
\[ x_{n+1} = x_n - \alpha_n H_n^{-1} g_n \]  

with \[ H_n(x) = 2J_n^T J_n \]

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

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

Gauss-Newton
- approximates Hessian by Jacobian product
- requires only \( n \times M \) 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.
Comparison of methods

LM: Adaptive combination of Steepest Descent and Gauss-Newton
BFGS: Quasi-Newton
SGD: Steepest Descent
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
contract
One iteration of the simplex algorithm

- Reorder the points so that \( f(x_{n+1}) > f(x_2) > f(x_1) \) (i.e. \( x_{n+1} \) is the worst point).

- Generate a trial point \( x_r \) by \textit{reflection}

\[
x_r = \bar{x} + \alpha (\bar{x} - x_{n+1})
\]

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

1. \( f(x_1) < f(x_r) < f(x_n) \) (i.e. \( x_r \) is neither the new best or worst point), replace \( x_{n+1} \) by \( x_r \).

2. \( f(x_r) < f(x_1) \) (i.e. \( x_r \) is the new best point), then assume direction of reflection is good and generate a new point by \textit{expansion}

\[
x_e = x_r + \beta (x_r - \bar{x})
\]

where \( \beta > 0 \). If \( f(x_e) < f(x_r) \) then replace \( x_{n+1} \) by \( x_e \), otherwise, the expansion has failed, replace \( x_{n+1} \) by \( x_r \).

3. \( f(x_r) > f(x_n) \) then assume the polytope is too large and generate a new point by \textit{contraction}

\[
x_c = \bar{x} + \gamma (x_{n+1} - \bar{x})
\]

where \( \gamma \) (0 < \( \gamma < 1 \)) is the contraction coefficient. If \( f(x_c) < f(x_{n+1}) \) then the contraction has succeeded and replace \( x_{n+1} \) by \( x_c \), otherwise contract again.

Standard values are \( \alpha = 1, \beta = 1, \gamma = 0.5 \).
Example

Path of best vertex

Downhill Simplex

Matlab fminsearch with 200 iterations

detail
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