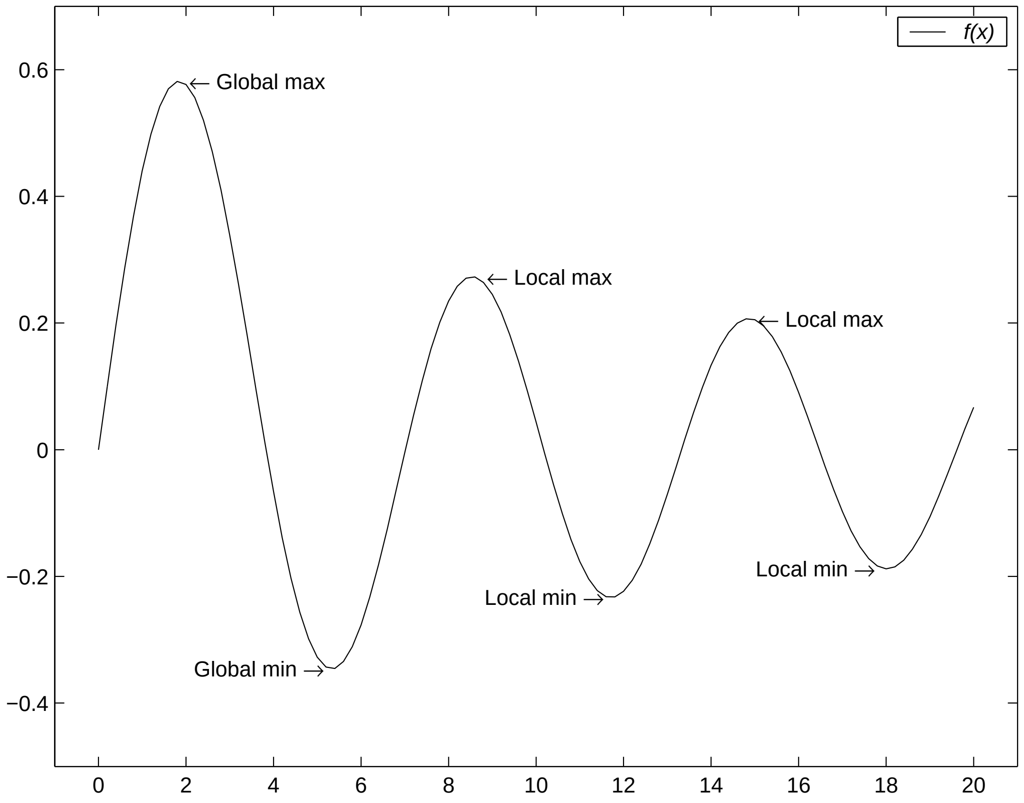
Purpose:

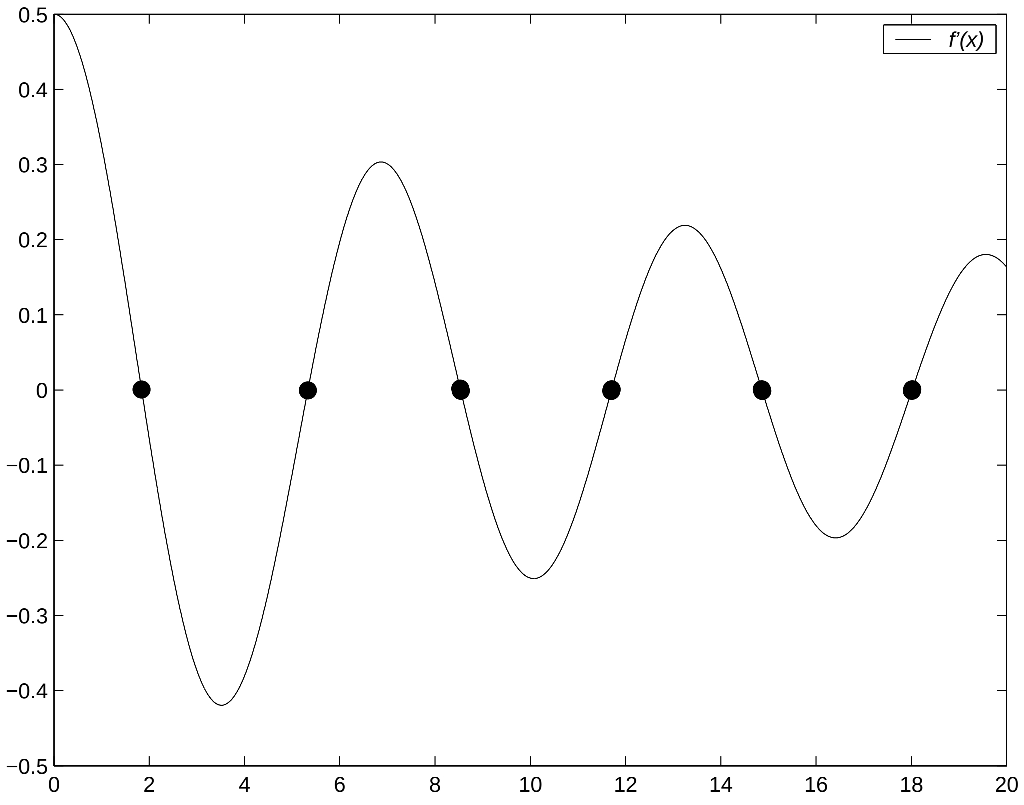
How to train an MLP neural network in MATLAB environment!

that is

For good computations,
we need good formulae
for good algorithms;
and good visualization
for good illustration
and proper testing
of good methods
and succesfull applications!

Critical values:





Theoretical bases of optimization problems:

$$\text{Minimize} \quad \mathcal{J}(\mathbf{u}) \quad \text{where } \mathbf{u} = \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix} \in \mathbf{R}^n. \tag{1}$$

- $\mathcal{J} : \mathbf{R}^n \to \mathbf{R}$ *cost function(al)* measuring the goodness of a solution candidate: NOTICE: good measure $\Rightarrow$ good problem $\Rightarrow$ useful solution ($A \Rightarrow B \equiv \neg B \to \neg A$!)
- we assume that $\mathcal{J}(\mathbf{u}) \geq 0 \quad \forall \mathbf{u} \in \mathbf{R}^n$
- NOTICE: $\max_{\mathbf{u}} \mathcal{J}(\mathbf{u}) \equiv \min_{\mathbf{u}} -\mathcal{J}(\mathbf{u})$
- we are seeking the values of $(u_1, \ldots, u_n)$ (unknowns)
- through the following definitions we introduce *precise characterization* of the visual intuition of the previous (and the following) figures

Definition 1. Vector $\mathbf{u}^*$ is the *(strict) global minimum* of problem (1) if

$$\mathcal{J}(\mathbf{u}^*) \leq \ (<) \ \mathcal{J}(\mathbf{u}) \quad \text{for all } \mathbf{u} \in \mathbf{R}^n.$$

Definition 2. Vector $\mathbf{u}^*$ is the *(strict) local minimum* of problem (1) if there exists a $\delta > 0$ such that

$$\mathcal{J}(\mathbf{u}^*) \leq \ (<) \ \mathcal{J}(\mathbf{u}), \quad \text{for all } \mathbf{u} \in \mathbf{R}^n \text{ such that } \|\mathbf{u} - \mathbf{u}^*\| \leq \delta.$$

Theorem 1. (**Weierstraas)** If function $\mathcal{J}$ in problem (1) is *continuous*, then there exists a minimum solution $\mathbf{u}^*$.

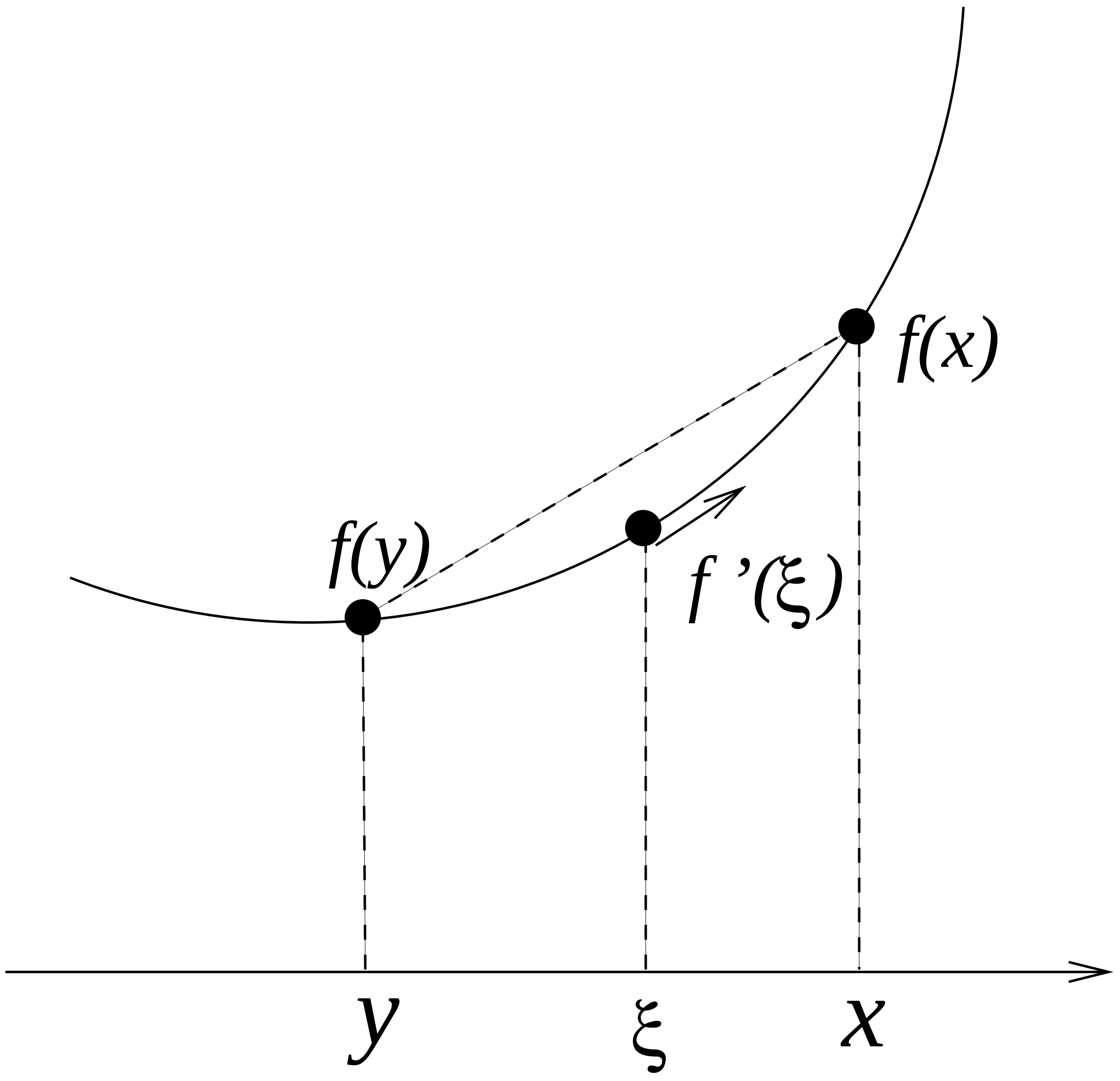
Definition 3. Function $\mathcal{J}$ is *convex* if

$$\mathcal{J}(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) \leq \lambda\mathcal{J}(\mathbf{x}) + (1-\lambda)\mathcal{J}(\mathbf{y}) \quad \forall \mathbf{x}, \mathbf{y} \in \mathbf{R}^n \text{ and } 0 \leq \lambda \leq 1.$$

Strict convexity requires $<$ instead of $\leq$ for $\mathbf{x} \neq \mathbf{y}$.

Theorem 2. For (locally) convex (and bounded from below) function $\mathcal{J}$ there exists a (local) minimum. If $\mathcal{J}$ is (locally) strictly convex, then the minimum point is (locally) unique.

Theoretical bases of gradient methods:



1D *mean value theorem of differential calculus*:

$$f(x) = f(y) + f'(\xi)(x - y) \quad \text{for some } \xi \in (y, x).$$

- through the following definitions we generalize both the concept of derivative and its relation to local function approximation in 1D into higher-order spaces

Theoretical bases of gradient methods II:

Definition 4. Function $\mathcal{J}$ is *(continuously) differentiable* at $\mathbf{u}$ ($\mathcal{J} \in C^1(\mathbf{R}^n)$), if there exists vector $\nabla\mathcal{J}(\mathbf{u}) \in \mathbf{R}^n$ and function $\varepsilon : \mathbf{R}^n \to \mathbf{R}$ such that

$$\mathcal{J}(\bar{\mathbf{u}}) = \mathcal{J}(\mathbf{u}) + \nabla\mathcal{J}(\mathbf{u})^T(\bar{\mathbf{u}} - \mathbf{u}) + \|\bar{\mathbf{u}} - \mathbf{u}\|\varepsilon(\mathbf{u}, \bar{\mathbf{u}} - \mathbf{u}) \tag{2}$$

for all $\bar{\mathbf{u}} \in \mathbf{R}^n$ and $\varepsilon(\mathbf{u}, \bar{\mathbf{u}} - \mathbf{u}) \to 0$ when $\bar{\mathbf{u}} \to \mathbf{u}$.

Vector $\nabla\mathcal{J}(\mathbf{u})$ is the *gradient* of $\mathcal{J}$ at $\mathbf{u}$ consisting of the *partial derivatives*:

$$\nabla\mathcal{J}(\mathbf{u}) = \begin{bmatrix} \frac{\partial\mathcal{J}(\mathbf{u})}{\partial u_1} \\ \vdots \\ \frac{\partial\mathcal{J}(\mathbf{u})}{\partial u_n} \end{bmatrix} \simeq \begin{bmatrix} \frac{\partial}{\partial u_1} \\ \vdots \\ \frac{\partial}{\partial u_n} \end{bmatrix} \mathcal{J}(\mathbf{u}). \tag{3}$$

Definition 5. Function $\mathcal{J}$ is *twice (continuously) differentiable* at $\mathbf{u}$ ($\mathcal{J} \in C^2(\mathbf{R}^n)$), if there exists vector $\nabla\mathcal{J}(\mathbf{u}) \in \mathbf{R}^n$ and symmetric $n \times n$-matrix $\mathbf{H}(\mathbf{u})$, the so-called *Hessian matrix*, and function $\varepsilon : \mathbf{R}^n \to \mathbf{R}$ such that

$$\mathcal{J}(\bar{\mathbf{u}}) = \mathcal{J}(\mathbf{u}) + \nabla\mathcal{J}(\mathbf{u})^T(\bar{\mathbf{u}} - \mathbf{u}) + \frac{1}{2}(\bar{\mathbf{u}} - \mathbf{u})^T\mathbf{H}(\mathbf{u})(\bar{\mathbf{u}} - \mathbf{u}) + \|\bar{\mathbf{u}} - \mathbf{u}\|^2\varepsilon(\mathbf{u}, \bar{\mathbf{u}} - \mathbf{u}), \tag{4}$$

where (again) $\varepsilon(\mathbf{u}, \bar{\mathbf{u}} - \mathbf{u}) \to 0$ when $\bar{\mathbf{u}} \to \mathbf{u}$.

Hessian matrix consists of the *second-order partial derivatives* $\frac{\partial^2\mathcal{J}(\mathbf{u})}{\partial u_i \partial u_j}$:

$$\mathbf{H}(\mathbf{u}) \left(\simeq \nabla(\nabla^T\mathcal{J}(\mathbf{u})) \simeq \nabla^2\mathcal{J}(\mathbf{u})\right) = \begin{bmatrix} \frac{\partial^2\mathcal{J}(\mathbf{u})}{\partial u_1^2} & \dots & \frac{\partial^2\mathcal{J}(\mathbf{u})}{\partial u_1 \partial u_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2\mathcal{J}(\mathbf{u})}{\partial u_n \partial u_1} & \dots & \frac{\partial^2\mathcal{J}(\mathbf{u})}{\partial u_n^2} \end{bmatrix}.$$

Definition 6. Vector $\mathbf{d} \in \mathbf{R}^n$ is *descent direction* for function $\mathcal{J}$ at $\bar{\mathbf{u}}$, if there exists $\delta > 0$ such that

$$\mathcal{J}(\bar{\mathbf{u}} + t\mathbf{d}) < \mathcal{J}(\bar{\mathbf{u}}) \quad \text{for all } t \in (0, \delta].$$

Definition 7. Let $\mathcal{J}$ be differentiable at $\bar{\mathbf{u}}$. If there exists a direction $\mathbf{d} \in \mathbf{R}^n$ such that $\nabla\mathcal{J}(\bar{\mathbf{u}})^T\mathbf{d} < 0$, then $\mathbf{d}$ is descent direction for $\mathcal{J}$ at $\bar{\mathbf{u}}$.

Theorem 3. Let $\mathcal{J}$ be differentiable at $\mathbf{u}^*$. If $\mathbf{u}^*$ is local minimum, then $\nabla\mathcal{J}(\mathbf{u}^*) = 0$ (i.e., $\mathbf{u}^*$ is a *critical value* of $\mathcal{J}$).

Theorem 4. Let $\mathcal{J}$ be twice differentiable at $\mathbf{u}^*$. If $\mathbf{u}^*$ is local minimum, then $\nabla\mathcal{J}(\mathbf{u}^*) = 0$ and the Hessian matrix $\mathbf{H}(\mathbf{u}^*)$ is positive semidefinite. If $\nabla\mathcal{J}(\mathbf{u}^*) = 0$ and $\mathbf{H}(\mathbf{u}^*)$ is positive definite, then $\mathbf{u}^*$ is *strict* local minimum.

Two examples:

As an example, we consider a few least-mean-squares (LMS) (quadratic) cost functionals and the corresponding optimization problems. Let $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ be a given set of (random) vectors such that $\mathbf{x}_i \in \mathbf{R}^n$ for all $1 \le i \le N$.

i) Mean:

$$\mathcal{J}(\mathbf{u}) = \sum_{i=1}^{N} \frac{1}{2}\|\mathbf{u} - \mathbf{x}_i\|^2 = \sum_{i=1}^{N} \frac{1}{2}(\mathbf{u} - \mathbf{x}_i)^T\,(\mathbf{u} - \mathbf{x}_i) = \sum_{i=1}^{N} \frac{1}{2}(\sum_{j=1}^{n}(u_j - (\mathbf{x}_i)_j)^2).$$

Because $\frac{1}{2}\frac{\partial(u_j-(\mathbf{x}_i)_j)^2}{\partial u_j} = (\mathbf{u} - \mathbf{x}_i)_j$ for all i, j, we obtain

$$\nabla\mathcal{J}(\mathbf{u}) = \sum_{i=1}^{N}(\mathbf{u} - \mathbf{x}_i) = N\,\mathbf{u} - \sum_{i=1}^{N}\mathbf{x}_i.$$

When $\mathbf{u}$ is solved from $\nabla\mathcal{J}(\mathbf{u}^*) = 0$, we get the sample mean

$$\mathbf{u}^* = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i = \bar{\mathbf{x}}.$$

Notice that if there is some (measurement, quantization) error like $\mathbf{x}_i = \tilde{\mathbf{x}}_i + \varepsilon_i$, then $\mathbf{u}^* = \frac{1}{N}\sum_{i=1}^{N}\tilde{\mathbf{x}}_i + \frac{1}{N}\sum_{i=1}^{N}\varepsilon_i$. Hence, when $N \to \infty$ or $\varepsilon_i \in \mathcal{N}(0, \delta^2)$ (in general, any symmetric error distribution with "enough samples"), $\mathbf{u}^*$ is a good estimate for the average behaviour of the given sample.

Finally, $\mathbf{H}(\mathbf{u}^*) = \nabla^T(\nabla\mathcal{J}(u)) = N\mathbf{I}$, so that $\bar{\mathbf{x}}$ is always unique.

ii) Linear fit: let $n = 2$ and

$$\mathcal{J}(\mathbf{u}) = \frac{1}{2}\sum_{i=1}^{N}|(\mathbf{x}_i)_2 - (u_2(\mathbf{x}_i)_1 + u_1)|^2.$$

Then

$$\frac{\partial\mathcal{J}(\mathbf{u})}{\partial u_1} = -\sum_{i=1}^{N}((\mathbf{x}_i)_2 - (u_2(\mathbf{x}_i)_1 + u_1))$$

$$\frac{\partial\mathcal{J}(\mathbf{u})}{\partial u_2} = -\sum_{i=1}^{N}((\mathbf{x}_i)_2 - (u_2(\mathbf{x}_i)_1 + u_1))\;(\mathbf{x}_i)_1$$

and

$$\mathbf{H}(\mathbf{u}) = \begin{bmatrix} N & \sum_{i=1}^{N}(\mathbf{x}_i)_1 \\ \sum_{i=1}^{N}(\mathbf{x}_i)_1 & \sum_{i=1}^{N}(\mathbf{x}_i)_1^2 \end{bmatrix}$$ Error in lecture notes!!!.

Basic Algorithm:

1. Choose a starting point $\mathbf{u}^0$. Set iteration counter $k = 0$.
2. Generate a descent direction $\mathbf{d}^k$.
3. Generate a step length t^k such that $\mathcal{J}(\mathbf{u}^k + t^k\mathbf{d}^k) < \mathcal{J}(\mathbf{u}^k)$.
4. Update $\mathbf{u}^{k+1} = \mathbf{u}^k + t^k\mathbf{d}^k$.
5. Stopping test. If need to continue, set $k = k + 1$ and go to 2.

When to Stop?

For chosen $\varepsilon > 0$:

- **(Absolute) critical point:** $\|\nabla\mathcal{J}(\mathbf{u}^{k+1})\| \le \varepsilon$.
- **(Relative) critical point:** $\|\nabla\mathcal{J}(\mathbf{u}^{k+1})\| \le \varepsilon\,\|\nabla\mathcal{J}(\mathbf{u}^0)\|$.
- **Change of solution:** $\|\mathbf{u}^{k+1} - \mathbf{u}^k\| = t^k\|\mathbf{d}^k\| \le \varepsilon$.
- **(Relative) change of cost functional:**
  $$\frac{\mathcal{J}(u^{k+1}) - \mathcal{J}(\mathbf{u}^k)}{\max(\delta, |\mathcal{J}(\mathbf{u}^k)|, |\mathcal{J}(\mathbf{u}^{k+1}|)} \le \varepsilon, \quad \text{where } \delta > 0.$$

Qualities of a good algorithm?

1. convergence (it solves the problem…)
2. speed of convergence (fastly…)
3. memory efficiency (with low memory consumption; usually contradicts 2.)

Stepsize determination:

- assume that a descent direction $\mathbf{d}^k$ is given
- we review different possibilities for selecting t^k appropriately
- starting point is to consider the following 1D minimization problem

$$\min_{t \in I} \mathcal{J}(\mathbf{u}^k + t\mathbf{d}^k) = j(t), \tag{5}$$

  where I is *a priori given search interval,* usually $I = [0,1]$ (cf. Definition 3 of convexity)
- in principle, any minimization method for (5) is sufficient (halfing method(?), regula-falsi, golden search, etc.), but one must try to cope with previous quality attributes of a good overall method
  $\Rightarrow$ $\underline{\text{compromise:}}$ compute quickly "good enough" solution for (5)!

Basic approaches:

- **Fixed stepsize:** choose by hand some stepsize $0 < t^* < 1$ and use it throughout the optimization iterations. Convergence questionable and slow, usual values, e.g. $t^* = 0.01,\ 0.05,\ 0.1$.
- **Armijo-rule:** Search smaller stepsizes consequtively by testing the sufficient decrease of cost functional

  0^o Fix constants s, β, σ such that $s > 0,\ \beta \in (0,1)$ and $\sigma \in (0, \frac{1}{2})$.

  1^o Try consequtively $k = \{0,1,2,\ldots\}$ and set $t = t^k = \beta^{m_k}$, where m_k is the first non-negative integer m, for which the so-called *Wolfe*-condition is satisfied:

$$\mathcal{J}(\mathbf{u}^k) - \mathcal{J}(\mathbf{u}^k + \beta^m\, s\, \mathbf{d}^k) \geq -\sigma\, \beta^m\, s\, \nabla\mathcal{J}(\mathbf{u}^k)^T\mathbf{d}^k.$$

  Choice of free parameters, e.g., as $s = 1.0,\ \beta = 0.4$ and $\sigma = 0.25$.

***Basic approaches (cont.)*:**

- **Quadratic interpolation:** Approximate function j using second-order polynomial $j(t) \simeq p(t) = a\,t^2 + bt + c$. Setting $p'(t) = 2at + b = 0$ yields to stepsize $t^* = -b/(2a)$ *when* $a \neq 0$.

  For determining the coefficients a, b and c usually two basic methods are applied.

  1. first approach is based on using values of j at three points, e.g.

     $$\begin{cases} t_0 = 0: & j_0 = \mathcal{J}(\mathbf{u}^k) \\ t_1 = \frac{1}{2}: & j_1 = \mathcal{J}(\mathbf{u}^k + \frac{1}{2}\cdot\mathbf{d}^k) \\ t_2 = 1: & j_2 = \mathcal{J}(\mathbf{u}^k + 1\cdot\mathbf{d}^k) \end{cases}$$

     Second-order polynomial that goes through the points $(t_i, j_i),\ i = 1, 2, 3$, is recovered by solving the resulting linear problem, whose solution

     $$\begin{cases} c &= j_0 \\ a &= 2(j_0 - 2j_1 + j_2) \\ b &= -3j_0 + 4j_1 - j_2 \end{cases}$$

     yields $t^* = \frac{-b}{2a} = \frac{3j_0 - 4j_1 + j_2}{4(j_0 - 2j_1 + j_2)}$.

  2. if gradient of $\mathcal{J}$ is also available, then by using $j'(t_0) = \nabla\mathcal{J}(\mathbf{u}^k)^T\mathbf{d}^k$ (cf. Definition 7), choosing $0 < t_1 \leq 1$ and setting $j_1 = \mathcal{J}(\mathbf{u}^k + t_1\mathbf{d}^k)$, we get

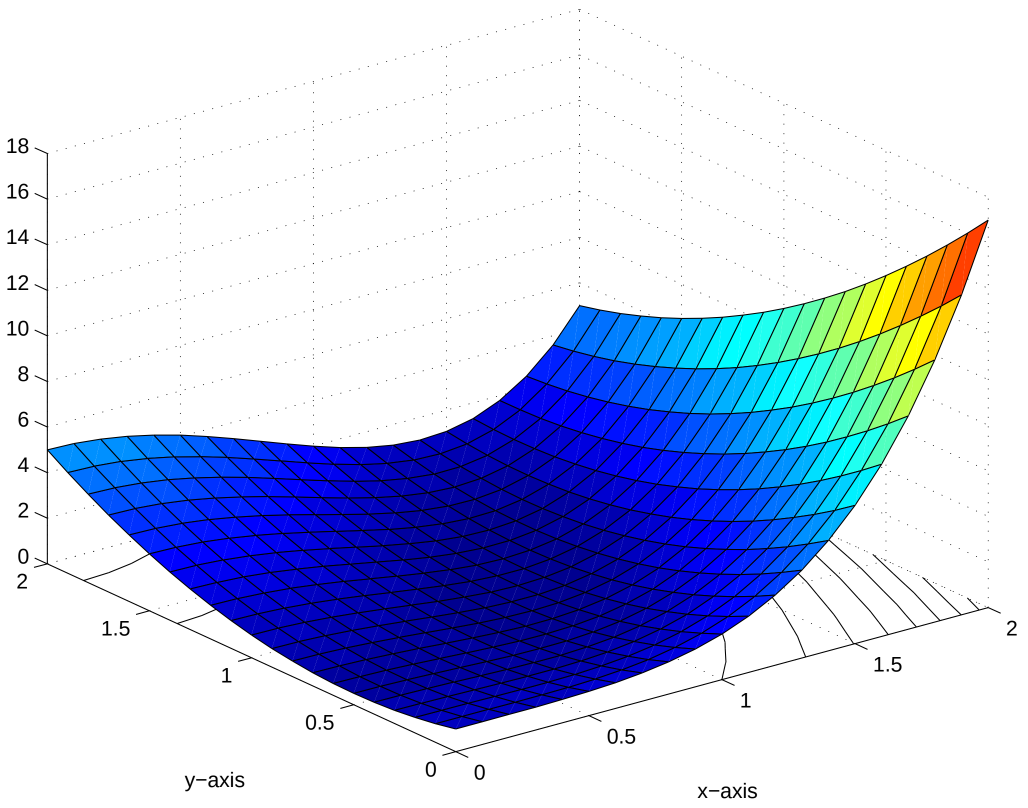
     $$\begin{cases} c &= j_0 \\ b &= \mathcal{J}(\mathbf{u}^k)^T\mathbf{d}^k \\ a &= \dfrac{j_1 - bt_1 - c}{t_1^2} \end{cases}.$$

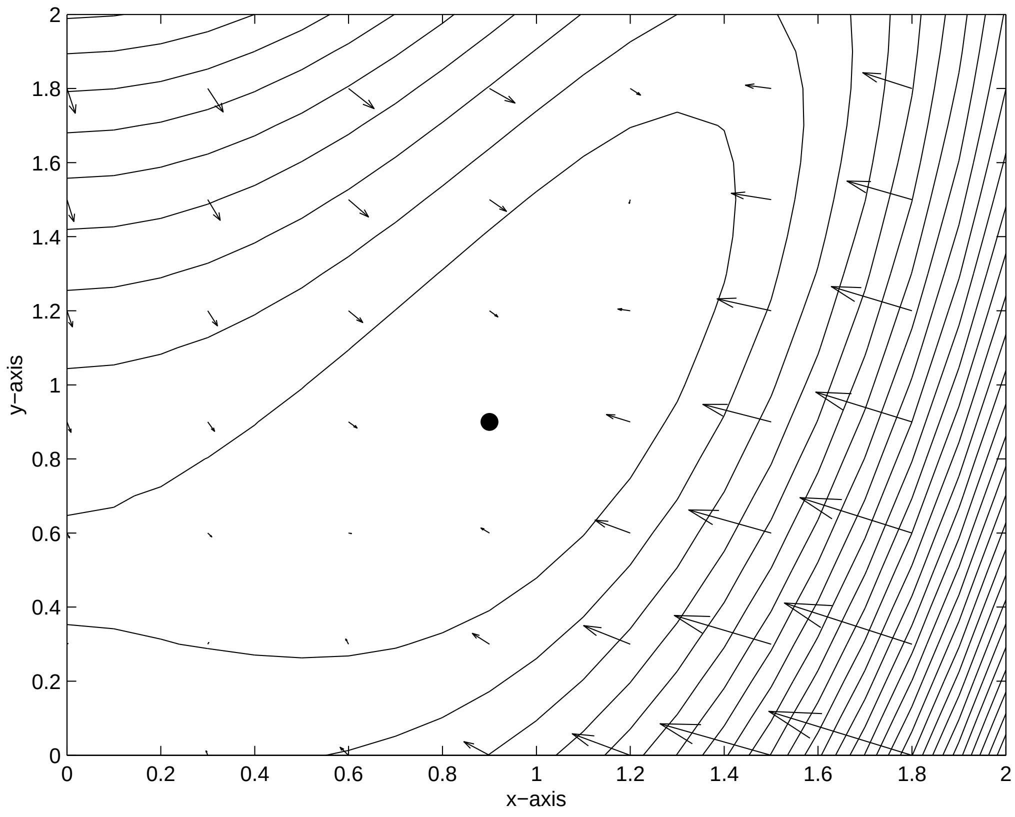
  Notice that if $a < 0$ then quadratic approximation is insufficient (too large search interval, bad search direction, etc.). Usually one then tries to decrease $I \longleftarrow 0.5 * I$ and repeat the process.

- **Cubic interpolation:** like the quadratic, but based on third-order polynomial approximation, which can be determined using four values of j or two set of value-derivative pairs. Notice the more restrictive conditions for appropriate values of coefficients.

- more advanced example routine in lecture notes, see also MATLAB *Optimization Toolbox*

Descent direction:





Descent direction (cont.):

- Starting point: from Theorem 7 it follows that

$$-\nabla\mathcal{J}(\mathbf{u}^k)^T\nabla\mathcal{J}(\mathbf{u}^k) = -\|\nabla\mathcal{J}(\mathbf{u}^k)\|^2 < 0.$$

- in fact, $-\nabla\mathcal{J}(\mathbf{u}^k)$ points to the direction of the *most rapid decrease*
$\Rightarrow$ good direction, but usually not the best length!

- **Newton's method**:

$$\mathbf{H}(\mathbf{u}^k)\mathbf{d}^k = -\nabla\mathcal{J}(\mathbf{u}^k) = -\mathbf{g}^k$$

  - well-defined when $\mathbf{H}(\mathbf{u}^k)$ positive definite (i.e., $\mathcal{J}$ strictly convex):

$$\nabla\mathcal{J}(\mathbf{u}^k)^T\mathbf{d}^k = -\mathbf{g}^{k^T}[\mathbf{H}(\mathbf{u}^k)]^{-1}\mathbf{g}^k < 0$$

  - BUT: analytic determination of $\mathbf{H}(\mathbf{u}^k)$ for real problems problematic!
  - BUT: Inversion of $\mathbf{H}(\mathbf{u}^k)$ for real problems expensive!

- **BFGS quasi-Newton method**: approximate $(\mathbf{H}(\mathbf{u}^{k+1}))^{-1}$ by

$$\mathbf{D}^{k+1} = \mathbf{D}^k + \left(1 + \frac{\mathbf{q}^T\mathbf{D}^k\,\mathbf{q}}{\mathbf{p}^T\mathbf{q}}\right)\,\frac{\mathbf{p}\,\mathbf{p}^T}{\mathbf{p}^T\mathbf{q}} - \frac{\mathbf{D}^k\mathbf{q}\,\mathbf{p}^T + \mathbf{p}\,(\mathbf{D}^k\mathbf{q})^T}{\mathbf{p}^T\mathbf{q}},$$

  where

$$\begin{aligned}\mathbf{p} &= \mathbf{u}^{k+1} - \mathbf{u}^k,\\ \mathbf{q} &= \mathbf{g}^{k+1} - \mathbf{g}^k,\end{aligned}$$

  and usually $\mathbf{D}^0 = \mathbf{D}^1 = \mathbf{I}$.

  - due to cumulation of errors reinitialization of $\mathbf{D}^k = \mathbf{I}$ after suitable number of iterations (usually after 20–50 iters.)

Some additional stuff:

- **Finite difference approximation of the gradient:**

$$\mathbf{g}_i^k \simeq \frac{\mathcal{J}(\mathbf{u}^k + h\,\delta_i) - \mathcal{J}(\mathbf{u}^k)}{h} \quad \text{forward difference, 1st order accuracy wrt } h,$$

$$\mathbf{g}_i^k \simeq \frac{\mathcal{J}(\mathbf{u}^k + h\,\delta_i) - \mathcal{J}(\mathbf{u}^k - h\,\delta_i)}{2\,h} \quad \text{central difference, 2nd order accuracy.}$$

  - the usual choice $h = \sqrt{\varepsilon}$, ε is the machine epsilon (MATLAB `eps`).
  - δ_i is the so-called *Knonecker's delta*

$$\delta_i = \begin{cases} 1 & i\text{th index}, \\ 0 & \text{for other indeces}, \end{cases}$$

- **Levenberg-Marquart**-method:

$$\text{minimize} \quad \mathcal{J}(\mathbf{u}) = \frac{1}{2}\sum_{i=1}^{N} \mathbf{e}_i(\mathbf{u})^2 = \frac{1}{2}\mathbf{E}(\mathbf{u})^T\mathbf{E}(\mathbf{u}), \quad \text{where } \mathbf{E}(\mathbf{u}) = \begin{bmatrix} \mathbf{e}_1(\mathbf{u}) \\ \vdots \\ \mathbf{e}_N(\mathbf{u}) \end{bmatrix}$$

  - gradient: $\nabla\mathcal{J}(\mathbf{u}) = \nabla(\frac{1}{2}\sum_i e_i(\mathbf{u})^2) = \sum_i \nabla e_i(\mathbf{u}) \cdot e_i(\mathbf{u}) = \mathbf{J}(\mathbf{u})^T\,\mathbf{E}(\mathbf{u})$, where $\mathbf{J}(\mathbf{u})$ is the so-called *Jacobian matrix*

$$\mathbf{J}(\mathbf{u}) = \begin{bmatrix} \frac{\partial \mathbf{e}_1(\mathbf{u})}{\partial \mathbf{u}_1} & \cdots & \frac{\partial \mathbf{e}_1(\mathbf{u})}{\partial \mathbf{u}_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{e}_N(\mathbf{u})}{\partial \mathbf{u}_1} & \cdots & \frac{\partial \mathbf{e}_N(\mathbf{u})}{\partial \mathbf{u}_n} \end{bmatrix} \in \mathbf{R}^{N\times n}.$$

  - iteration: $(\mathbf{J}(\mathbf{u}^k)^T\,\mathbf{J}(\mathbf{u}^k) + \mu^k\mathbf{I})\,\mathbf{d}^k = -\mathbf{J}(\mathbf{u}^k)^T\,\mathbf{E}(\mathbf{u}^k)$ for suitable $\mu^k > 0$.

Some additional stuff (cont.):

- **Conjugate gradient method á la *Polak-Ribière*:**

$$\begin{aligned}
\mathbf{d}^0 &= \mathbf{r}^0 = -\nabla\mathcal{J}(\mathbf{u}^0) \quad \text{(initialization)} \\
t^k &: \text{ 1D minimization of function} \mathcal{J}(\mathbf{u}^k + t^k\,\mathbf{d}^k) \\
\mathbf{u}^{k+1} &= \mathbf{u}^k + t^k\,\mathbf{d}^k \\
\mathbf{r}^{k+1} &= -\nabla\mathcal{J}(\mathbf{u}^{k+1}) \\
\beta^{k+1} &= \max\left\{\frac{(\mathbf{r}^{k+1})^T(\mathbf{r}^{k+1}-\mathbf{r}^k)}{(\mathbf{r}^k)^T\mathbf{r}^k}, 0\right\} \\
\mathbf{d}^{k+1} &= \mathbf{r}^{k+1} + \beta^{k+1}\,\mathbf{d}^k
\end{aligned}$$

  - better control of search directions on (nearly) flat error surface
  - *de facto* -method for solving SPD linear problems

- **About constrained optimization**

  - in many cases solution of an optimization problem should be constrained to a given *admissible set* C
  - e.g., production costs always positive $u_i \geq 0 \quad \forall i$ (*inequality constraint*), eigenvector's norm always one $\|\mathbf{u}^*\| = 1$ i.e. $\|\mathbf{u}^*\| - 1 = 0$ (*equality constraint*) etc.
  - most common approach is to complement the basic algorithm with a projection step:

    4.5 Project $\mathbf{u}^{k+1}$ into C by setting $\mathbf{u}^{k+1} = \mathcal{P}_C(\mathbf{u}^{k+1})$.

    Here $\mathcal{P}_C : \mathbf{R}^n \to \mathbf{R}^n$ is a projetion-operator, e.g.

$$\begin{aligned}
\mathcal{P}_{\{\mathbf{u}\geq 0\}}(\mathbf{u}) &\equiv \max(\mathbf{u}, 0) \quad \text{(componentwise)}, \\
\mathcal{P}_{\{\|\mathbf{u}\|=1\}}(\mathbf{u}) &\equiv \mathbf{u} = \frac{\mathbf{u}}{\|\mathbf{u}\|}.
\end{aligned}$$

  - generally constraint optimization is a hard discipline
  - Other basic approach is to use the so-called *(augmented) Lagrangian (merit) function* for combining cost function and constraints into one functional which is then minimized. This needs appropriate update rules for the resulting Lagrangian coefficients.