# 1. Computing with Matrices and Vectors

## Machine Numbers

- $\mathbb{M} \times \mathbb{M} \to \mathbb{M}$ (finite)
- Standard for machine-numbers: floating-point representation

## Fixed point representation

- Fixed decimal point
- Less accurate

## Floating point representation

- $0,73125 \cdot 10^{2}$ (Mantissa · Base, Exponent)
- Normalized, when mantissa starts with 0
- Good for quickly changing the scale (KM -> mm)
- Decimal point isn't fixed -> defined through exponent

## Error of rounding

- Must round to next nearest machine-number
- Absolute error: $|x - \tilde{x}|$ - Relative Error: $\frac{|x-\tilde{x}|}{|x|}$
- Machine precision "EPS": $10 + EPS = 10$
- Solutions for computing relative Error x
  1. Foreward error: $x_{exact} - x_{approximate}$ (cannot be computed)
  2. Backward error: $b_{exact} - b_{approximate}$
     - $Ax = b_{exact}$ => compute $x_{approximation}$
     - $b_{approximation} = Ax_{approximation}$
     - Backward error: $b_{exact} - b_{approximate}$

## S.p.d matrix

- Symmetric positive definite matrix

## Eigen

- MatrixXd (Dynamic double matrix)
- Matrix3d (3x3 double matrix)
- Matrix<double, dynamic, 5> (dynamic x 5 double matrix)
- MatrixXd y(6,9); (6x9 dynamic double matrix) (can be resized)
- Row-major: 1|2|3|4|5|6

| 1 | 2 | 3 |
|---|---|---|
| 4 | 5 | 6 |

- Column-major: 1|4|2|5|3|6

---

## Computational effort

(= Number of basic operations that the algorithm is performing)

- Computational effort ≠ runtime
  - ↳ e.g. same number of operations, but row/column-major
- Dot-product / Skalarprodukt / Äusseres-Produkt: $O(n)$ $(x^T \cdot y)$
- Tensor-product / Äusseres-Produkt: $O(m \cdot n)$ $(x \cdot y^T)$
- Matrix-vector-product: $O(n^2)$
- Matrix-product: $O(m \cdot n \cdot k)$ $(A \cdot B)$
- Multiplying by a diagonal matrix: $O(n)$

---

## Hidden summation

- A, B matrices, we want to compute $y = \text{upper\_triangular}(A \cdot B^T) \cdot x$

1. Approach: $y = \text{upper\_triangular}(A \cdot B^T) \cdot x \Rightarrow O(n^2 \cdot p)$

2. Approach: Decompose and regroup first: for # columns = 1

$$y = \begin{bmatrix} a_1 & & \\ & \ddots & \\ & & a_n \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & \cdots & 1 \\ & 1 & \cdots & 1 \\ & & \ddots & \vdots \\ 0 & & & 1 \end{bmatrix} \cdot \begin{bmatrix} b_1 & & \\ & \ddots & \\ & & b_n \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

$= z$ $O(n)$ (matrix-vector)

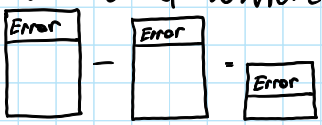
$O(n)$ (matrix-vector)

$O(n)$ (Diagonal-Matrix-vector)

↳ For 1 column: $O(n)$
For $p$ columns: $O(p \cdot n)$

## Cancellation

- Subtraction of two numbers, which are approximatly the same size




- Errors get amplified
- Stable computation of roots: $\frac{-\alpha + \sqrt{\alpha^2 - 4\beta}}{2}$

  - $p(\xi) = 1\xi^2 + \alpha\xi + \beta$
  1. $\xi_1 = \frac{-\alpha - \sqrt{\alpha^2 - 4\beta}}{2 \cdot 1}$
  2. $\beta = \xi_1 \cdot \xi_2 \Rightarrow \xi_2 = \beta / \xi_1$

Vieta's formula: $x^2 + bx + c = (x-p)\cdot(x-q)$

$x^2 + bx + c = x^2 - \underbrace{(p+q)}_{=b} x + \underbrace{pq}_{=c}$

---

## Numerical stability

- When an algorithm is numericaly stable, if you give me a x, the computer converts x to a slightly other $\tilde{x}$, because x couldn't be in the machine-numbers
- The error is close to EPS
- Algorithm $\tilde{F}$ is outputting the exact solution to nearly the right answer, as if you rounded your input data
- Mixed stability

  $\frac{\|x - \tilde{x}\|}{x} \leq O(EPS)$

- Numerical stability is only telling us, it is good at implementing F(x), but F(x) could be a bad function

---

## Condition-number

- How an output value can change for a small change in the input (=Sensitivity)
- Ratio of smallest/largest Eigenvalue of Matrix

# 2. Direct methods for solving linear systems of equations

## Gauss Elimination

1. Elimination: $O(n^3)$
2. Backsubstitution: $O(n^2)$

$\Big\} O(n^3)$

- Solving triangular-matrix only via Backsubstitution: $O(n^2)$
- For each right-hand-side, the whole computation must be done another time

---

## LU-Decomposition

$P \cdot A = L \cdot U$

($P$: permutation Matrix; $L$: Lower triangular with 1s; $U$: Upper triangular)

1. LU-Decomposition: $A = LU$ $O(n^3)$
2. Forward-Substitution: $Lz = b$ $O(n^2)$
3. Backward-Substitution: $Ux = z$ $O(n^2)$

$\Big\} O(n^3)$

- Overall complexity is the same as Gauss
- But we can solve for different right-hand sides, and only compute $A = LU$ once $\Rightarrow$ new RHS $O(n^2)$
- Low-rank-modification of A of an LSE:
  - After solving $Ax = b$, we solve $\tilde{A}\tilde{x} = b$ with different $A$
  - General rank-1-modification: $\tilde{A} = A + u \cdot v^T$
  - Trick: use block-elimination: $\begin{bmatrix} A & u \\ v^T & -1 \end{bmatrix} \cdot \begin{bmatrix} \tilde{x} \\ \xi \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}$

① $\begin{cases} \tilde{A}\tilde{x} + u\xi = b \\ v^T \cdot \tilde{x} - \xi = 0 \Leftrightarrow \xi = v^T \cdot \tilde{x} \end{cases} \Longrightarrow A\tilde{x} + u \cdot v^T \cdot \tilde{x} = b \Leftrightarrow (A + uv^T) \cdot \tilde{x} = b$

② $\{\tilde{x} = A^{-1} \cdot (b - u\xi)\} \Leftrightarrow (v^T)^{-1} + \xi = A^{-1} \cdot (b - u\xi) \Leftrightarrow v^T \cdot A^{-1} \cdot (b - u\xi) - \xi = 0$

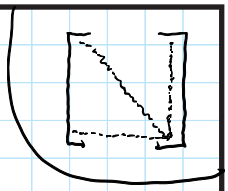
③ $\begin{cases} \tilde{x} = A^{-1}\tilde{b} - A^{-1} \cdot u \cdot \xi \\ \Leftrightarrow \tilde{x} = A^{-1} \cdot b - A^{-1} \cdot u \cdot \dfrac{v^T \cdot A^{-1} \cdot b}{v^T \cdot A^{-1} \cdot u + 1} \end{cases}$

$\Leftrightarrow (v^T A^{-1} \cdot u + 1) \cdot \xi = v^T \cdot A^{-1} \cdot b$

$\Leftrightarrow \xi = \dfrac{v^T \cdot A^{-1} \cdot b}{v^T \cdot A^{-1} \cdot u + 1}$

└ Assuming, LU-Decomposition of A already available.

# Arrow-Matrices



$$A_{Arrow} = \begin{bmatrix} D & c \\ b^T & \alpha \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ z \end{bmatrix} = \begin{bmatrix} b_1 \\ \beta \end{bmatrix} \Leftrightarrow \begin{bmatrix} D & c \\ b^T & \alpha \end{bmatrix} \begin{bmatrix} b_1 \\ \beta \end{bmatrix}$$

($\alpha$: scalar, $z$: scalar, $\beta$: scalar)

- $A_{11}$ is diagonal, good for solving LSE
- We can also use block-matrices to compute $A \cdot A \cdot x$
  - Normal: Matrix-Matrix-Vector $O(n^3)$
  - Block-Matrix-Multiplication: Vector-Vector $O(n^2)$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

- Block-LU-Decomposition: $A = LU = \begin{bmatrix} I & 0 \\ b^T D^{-1} & 1 \end{bmatrix} \cdot \begin{bmatrix} D & C \\ 0 & -b^T \cdot D^{-1} \cdot c \end{bmatrix}$
- Block-Gaussian-Decomposition:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \longrightarrow \left[\begin{array}{cc|c} A_{11} & A_{12} & b_1 \\ 0 & A_{22} - A_{21} \cdot A_{11}^{-1} A_{12} & b_2 - A_{21} A_{11}^{-1} b_1 \end{array}\right]$$

$$\longrightarrow \left[\begin{array}{cc|c} 1 & 0 & A_{11}^{-1}(b_1 - A_{12} \cdot S^{-1} \cdot b_S) \\ 0 & 1 & S^{-1} \cdot b_S \end{array}\right] \quad S = A_{22} - A_{21} \cdot A_{11}^{-1} A_{12}, \quad b_S = b_2 - A_{21} A_{11}^{-1} b_1$$

$$\begin{cases} x_1 = A_{11}^{-1}(b_1 - A_{12} S^{-1} b_S) \\ x_2 = S^{-1} b_S \end{cases}$$

---

# Sparse Matrices

- Most entries of matrix $= 0$
- Number of nonzero-Elements: $nnz(A)$
- Required memory should only depend on $nnz(A)$
- Example: Arrow-Matrix
- COO: Triplet-Format
  - (rows, columns, value)
  - if two locations are the same → summation
- CRS: Compressed-Row-Storage
- CCS: Compressed-Column-Storage

- Example CRS:

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}$$

val-vector:

| 10 | -2 | 3 | 9 | 3 | 7 | 8 | 7 | 3...9 | 13 | 4 | 2 | -1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

col_ind-array:

| 1 | 5 | 1 | 2 | 6 | 2 | 3 | 4 | 1...5 | 6 | 2 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

row_ptr-array:

| 1 | 3 | 6 | 9 | 13 | 17 | 20 |
|---|---|---|---|---|---|---|

- Build Sparse-Matrix
  1. Use triplet format and at the end compress to CRS/CSS
  2. Reserve enough storage in each row
  ↳ Because inserting into CRS/CSS is cost-intense, because all 3 arrays must be moved

---

## 3. Direct methods for linear least-squares problem

- Linear regression
  - When we have n parameters and n measurements
  - Problematic: Overfitting: When parameters fit good for the less measurements
    ↳ Get more measurements ⇒ Overdetermined system

- Normal equation

$$\cdot\ Ax = b \quad / \cdot A^T$$

$$\Leftrightarrow A^T \cdot A \cdot x = A^T \cdot b \quad / (A^T \cdot A)^{-1}$$

$$\Leftrightarrow x = (A^T \cdot A)^{-1} \cdot b$$

- QR-Decomposition
  - $A = \underbrace{Q}_{\text{Orthogonal Unit vectors}} \cdot R$ } upper diagonal matrix
  - Full QR-decomposition: $[A] = [Q] \cdot \begin{bmatrix} R \\ 0 \end{bmatrix}$
  - Thin QR-decomposition: $[A] = [Q] \cdot [R]$

1. Approach: Gram-Schmidt
2. Approach: Householder-Transformation

## - Singular-Value-decomposition

- $A = U \cdot \Sigma \cdot V^H$
- $U = A \cdot \Sigma^{-1} \cdot V^{H^{-1}}$
- $\Sigma$: Singular values in ascending, diagonal form
- $V$: Normalized Eigenvectors
- Singular value = $\sqrt{\text{Eigenvalue}}$

---

## Approximation/Compression of Matrix via SVD

1. Compute SVD of A
2. Take only the first k rows

- If we ignore the small singular-values, it's O.K. because they don't contribute much to the matrix

---

## Principle Component analysis

- Shows the trend of data, e.g. stock-prices

1. Get some data $\begin{bmatrix} | & | & | \\ d_1 & d_2 & d_3 \cdots \\ | & | & | \end{bmatrix}$
2. Compute the mean (=Durchschnitt) and substract it from each data-input $A = \begin{bmatrix} | & | & | \\ d_1\text{-mean} & d_2\text{-mean} & d_3\text{-mean} \cdots \\ | & | & | \end{bmatrix}$
3. Calculate the covariance matrix $A \cdot A^T$
4. Compute the Eigenvectors $A \cdot A^T$ via SVD U-Matrix

# Total least squares

- Also system Matrix $A$ can be perturbed
- Because $A$ and $B$ is perturbed, we use the SVD-k-approximation trying to eliminate the meassurement errors
- $[\hat{A}\ \hat{b}] \approx [A\ b]$
  - ↳ Best n-rank-approximation

1. $[A\ b] = U \cdot \Sigma \cdot V^T$
2. $[\hat{A}\ \hat{b}] = \sum_{i=1}^{n} \sigma_i \cdot (U_{:,i}) \cdot (V_{:,i})^T$
3. $[\hat{A}\ \hat{b}] \cdot (V)_{:,n+1} = 0$ / $(V)_{:,n+1}$ is orthogonal

$\Leftrightarrow \hat{A} \cdot (V)_{1:n,\,n+1} + \hat{b}\,(V)_{n+1,\,n+1} = 0$ / $-\hat{b}\,(V)_{n+1,\,n+1}$

$\Leftrightarrow \hat{A} \cdot (V)_{1:n,\,n+1} = -\hat{b}\,(V)_{n+1,\,n+1}$ / $:(V)_{n+1,\,n+1}$

$-\hat{b} = \hat{A} \cdot \frac{(V)_{1:n,\,n+1}}{(V)_{n+1,\,n+1}}$ / $\hat{A}^{-1}$ / $\cdot(-1)$

$\hat{A}^{-1} \cdot \hat{b} = -\frac{(V)_{1:n,\,n+1}}{(V)_{n+1,\,n+1}}$

4. $x = -\hat{A}^{-1} \cdot \hat{b}$

$x = -\frac{(V)_{1:n,\,n+1}}{(V)_{n+1,\,n+1}}$

# Constrained least-squares / Lagrange-Multipliers

- Least-Squares with side-condition
- $Ax = b$ in least squares
- $Cx = d$ exact

} x is the same

- Solve via Lagrange-Multipliers:

1. $L(y, m) := \frac{1}{2} \cdot \|Ay - b\|_2^2 + m^T(Cy - d)$
   - $x = \operatorname{argmin} \max L(y, m) \Rightarrow$ Saddlepoint problem $\Rightarrow$ Derivatives vanish
2. $\frac{\partial f}{\partial x}(x, m) = A^T(Ax - b) + C^T \cdot m = 0$
3. $\frac{\partial f}{\partial m}(x, m) = Cx - d = 0$
4. Augmented normal equation: $\begin{bmatrix} A^TA & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} x \\ m \end{bmatrix} = \begin{bmatrix} A^Tb \\ d \end{bmatrix}$ (A, x, b)
5. Solve augmented normal equation, just as normal

Jonas B.

# 4. Filtering Algorithms

- Impulse response: · $h = (h_0, \dots, h_{n-1})^T$
  · Output when the filter is fed with a single impulse of strength one, corresponding to input $e_1$
- Properties of a filter: · Finite length input => finite length output
  · Time invariant
  · Linearity
  · Output depends only on past and present
- n-th order filter: Impulse response is n long
- Given input signal $x$, output $y$ can be computed as follows
  ↳ $y_k = \sum_{j=0}^{n-1} \underbrace{h_{k-j} \cdot x_j}_{1.\text{variant}} = \sum_{j=0}^{n-1} \underbrace{h_j \cdot x_{k-j}}_{2.\text{variant}}$, $h_j = 0$ for $j < 0$
- Discrete-Convolution: · Given Input-signal $x = (x_0, \dots, x_{n-1})^T$
  · Given filter $h = (h_0, \dots, h_{n-1})^T$
  · $y = h * x \iff y_k = \sum_{j=0}^{n-1} h_{k-j} \cdot x_j$
- n-periodic: · When the Input-signal repeats itself after n Inputs
  · When Input is periodic, so is the output
- Discrete-periodic-convolution: · Output is also periodic
  · Satisfy $x_j = x_{j+n}$
  · We must zero-padd to match:
  ↳ length of $x$ = length of $h$
  · $y_k = p_k * x_k$
  $= \sum_{j=0}^{n-1} \underbrace{p_{k-j}}_{\text{Periodic impulse response}} \cdot x_j = \sum_{j=0}^{n-1} x_{k-j} \cdot p_j$
- Periodic impulse response: $p_j = \sum_{i \in \mathbb{N}} h_{j+i \cdot n} \mid j \in \{0, \dots, n-1\}$
  ↳ L-periodic extension of $h$

- Periodic signal can be represented as following

$$\begin{bmatrix} y_0 \\ \vdots \\ y_{n-1} \end{bmatrix} = \underbrace{\begin{bmatrix} p_0 & p_{n-1} & p_{n-2} & \cdots & & \cdots & p_1 \\ p_1 & p_0 & p_{n-1} & & & & \vdots \\ p_2 & p_1 & p_0 & \ddots & & & \\ \vdots & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ \vdots & & & & \ddots & \ddots & p_{n-1} \\ p_{n-1} & \cdots & & & & p_1 & p_0 \end{bmatrix}}_{=:P} \begin{bmatrix} x_0 \\ \vdots \\ x_{n-1} \end{bmatrix}.$$

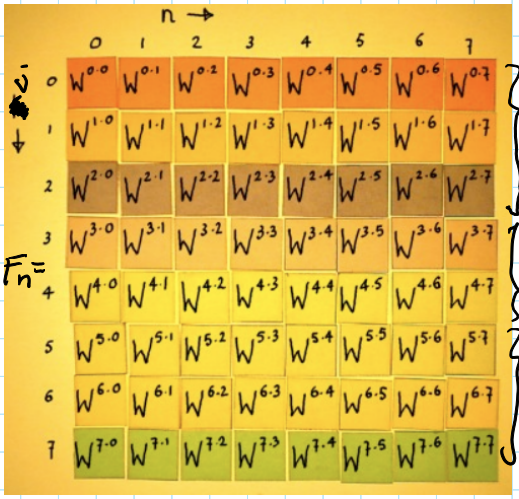
- Circulant-Matrix:
  - Have constant main/sub/super diagonals
  - Used to represent $r_x$
  - Can be determined by a vector

---

## Discrete Fourier Transformation (DFT)

### Fourier-Matrix

- All Circulant-matrices with the same dimensions have the same set of eigenvectors (= orthogonal trigonometric basis of $\mathbb{C}^n$)
- Discrete convolution ⇒ discrete circular convolution ⇒ discrete fourier transformation
- Computes the spectrum of a periodic signal
- $\omega_n = e^{\frac{-2\pi i}{n}}$ / $\omega_n^{n \cdot j} = \left(e^{\frac{-2\pi i}{n}}\right)^{n \cdot j} = e^{-2\pi i j}$
- The scaled Fourier matrix $\frac{1}{\sqrt{n}} \cdot F_n$ is unitary




### Fourier Transformation

- We get the spectrum by calculating: $C = F_n(y)$ ($y$ = signal)

$$C = F_n \cdot y$$

### Discrete convolution via DFT

- DC can be written as multiplication with circulant-matrix
- We can use Fourier-transform instead
- $c_k = \sum_{j=0}^{n-1} y_j \cdot \omega_n^{n \cdot j}$ / $F_n^{-1} = \frac{1}{n} \cdot \overline{F_n}$
- $y_i = \frac{1}{n} \cdot \sum_{k=0}^{n-1} c_k \cdot \omega_n^{-n \cdot j}$

$$\begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} \cdot \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{bmatrix}$$

$C$ $F_n$ $y$

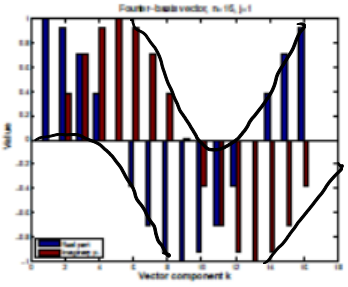
## Fast-Fourier-Transformation

[tells you, how much periodicity you have in your signal, the more peaks, the more periodic.]

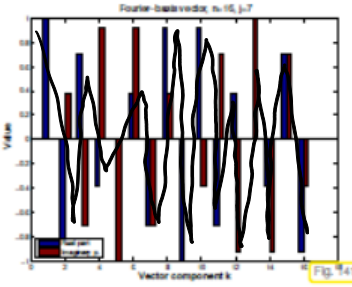
- Normal DFT: $O(n^2)$
- Because $\omega_n^{2j} = \omega_m^j$ $(n = 2m)$, DFT can be implemented using Divide and Conquer in $O(n \cdot \log(n))$

## Frequency Filtering

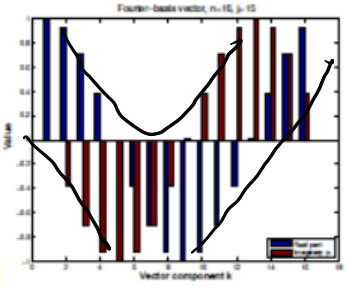
- k-th row of $F_n$: $(F_n x)_k = v_k^T \cdot x$ ($v$ = trigonometric basis)
- If $k$ is close to 0 mod(n) => slow frequency
- If $k$ is in the middle mod(n) => high frequency
- If $k$ is close to n mod(n) => slow frequency



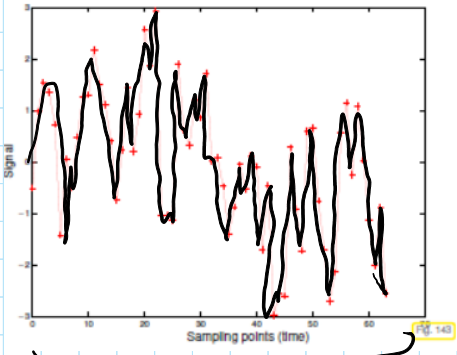
"slow oscillation/low frequency"



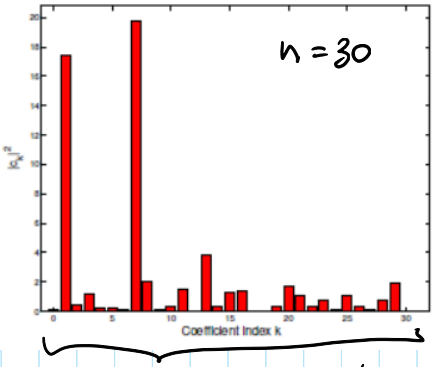
"fast oscillation/high frequency"



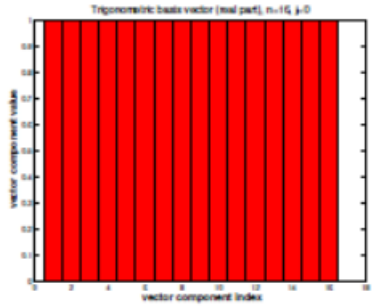
"slow oscillation/low frequency"




signal

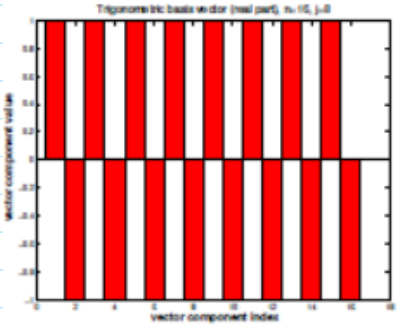



spectrum of signal



$\mathrm{Re}(\mathbf{F}_{16})_{:,0}$

k near 0 mod(16) => slow



$\mathrm{Re}(\mathbf{F}_{16})_{:,8}$

k middle mod(n) => fast

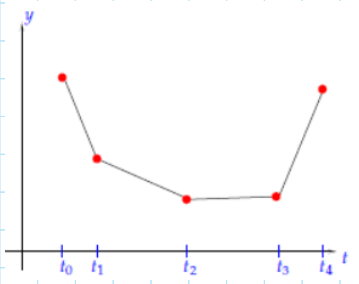
"Low pass filter"

=> Filter high frequency / noise out

=> cut them off

# 5. Data interpolation and Data fitting in 1D

- Given a set of data points $(t_i, y_i)$, find an interpolant function $f(x)$ with $f(t_i) = y_i$




## Piecewise linear interpolation

- Just connect the points

## Global polynomial interpolation

## Lagrange interpolation

- We use Lagrange-Polynomials: $L_i(t) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{t - t_j}{t_i - t_j}$
- Interpolation: $p(t) = \sum_{i=0}^{n} y_i \cdot L_i(t)$
  - ↳ Polynom for a set of nodes is unique
- Polynom of degree $n$ is determined by $(n+1)$-data-points
- Evaluating $L_i(t)$ is $\Theta(n)$ with Horner-Scheme
- Evaluating $p(t)$ is $O(n^2)$, because we have to evaluate $n$ polynomials
- Reusing computation for different y-values ⇒ Barycentric-formula

$$p(t) = \sum_{i=0}^{n} L_i(t) \cdot y_i = \sum_{i=0}^{n} \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{t - t_j}{t_i - t_j} = \sum_{i=0}^{n} \lambda_i \prod_{\substack{j=0 \\ j \neq i}}^{n} (t - t_j) \cdot y_i = \prod_{j=0}^{n} (t - t_j) \cdot \sum_{i=0}^{n} \frac{\lambda_i}{t - t_i} \cdot y_i$$

$$\lambda_i = \frac{1}{(t_i - t_0) \cdots (t_i - t_{i-1})(t_i - t_{i+1}) \cdots (t_i - t_n)}$$

$$1 = \prod_{j=0}^{n} (t - t_j) \cdot \sum_{i=0}^{n} \frac{\lambda_i}{t - t_i} \quad / : \sum_{i=0}^{n} \frac{\lambda_i}{t - t_i}$$

$$\prod_{j=0}^{n} (t - t_j) = \frac{1}{\sum_{i=0}^{n} \frac{\lambda_i}{t - t_i}} \quad / *$$

$$p(t) = \frac{\sum_{i=0}^{n} \frac{\lambda_i}{t - t_i} \cdot y_i}{\sum_{i=0}^{n} \frac{\lambda_i}{t - t_i}}$$ } Barycentric Formula

$$p(t) = \prod_{j=0}^{n} (t - t_j) \cdot \sum_{j=0}^{n} \frac{d_i}{t - t_i} \cdot y_i \quad / : \sum_{j=0}^{n} \frac{d_i}{t - t_i} \cdot y_i$$

$$* \prod_{j=0}^{n} (t - t_i) = \frac{p(t)}{\sum_{j=0}^{n} \frac{d_i}{t - t_i} \cdot y_i}$$

- Advantage: $\lambda_i$ is independent of $t$ and $y_i$ ⇒ precompute
- Computation of $\lambda_i$, $i = 0, \ldots, n$: $\Theta(n^2)$ (only once!)
- Cost of $\Theta(n)$ for every subsequent evaluation of $p$

- Update-friendly for adding/changing points: Aitken-Neville
  - We do interpolation on pieces of data, and gradually get to the whole interpolation

- $p_{k,k}(x) \equiv y_k$ ("constant polynomial"), $k = 0, \ldots, n$,
- $p_{k,\ell}(x) = \dfrac{(x - t_k)p_{k+1,\ell}(x) - (x - t_\ell)p_{k,\ell-1}(x)}{t_\ell - t_k}$

| $n =$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| $t_0$ | $y_0 =: p_{0,0}(x)$ | $\to p_{0,1}(x)$ | $\to p_{0,2}(x)$ | $\to p_{0,3}(x)$ |
| $t_1$ | $y_1 =: p_{1,1}(x)$ | $\to p_{1,2}(x)$ | $\to p_{1,3}(x)$ | |
| $t_2$ | $y_2 =: p_{2,2}(x)$ | $\to p_{2,3}(x)$ | | |
| $t_3$ | $y_3 =: p_{3,3}(x)$ | | | |
| $+1$ | $y_4 =: p_{4,4}(x)$ | | | |

($t_4$: new point)

## Newton Basis

- Good for many interpolation-points
- $N_0(t) = 1$
- $N_1(t) = (t - t_0)$
- $N_n(t) = \prod_{i=0}^{n-1} (t - t_i)$

$$\left.\begin{array}{l} \\ \\ \\ \end{array}\right\} a_0 N_0(t_i) + a_1 N_1(t_i) + \ldots + a_n N_n(t_i) = y_i$$

- $n$ = amount of points
- Find interpolant $p(t) = \sum_{i=0}^{n} a_i N_i(t)$ ($a_i$: coefficients need to be determined)
  - $p(t_0) = a_0 + 0 + 0 + \ldots + 0$
  - $p(t_1) = a_0 + a_1 N_2(t_0) + 0 + 0 + \ldots + 0$

} Lower triangular system of equations

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & N_1(t_1) & 0 & \cdots & 0 \\ 1 & N_1(t_2) & N_2(t_2) & \cdots & 0 \\ \vdots & & & & \\ 1 & N_1(t_n) & N_2(t_n) & \cdots & N_n(t_n) \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

$\Rightarrow$ You are computing the same polynomial, only with different ways, because polynom is unique.

## Spline-Interpolation

mesh

- Each subinterval $[t_{i-1}, t_i]$ has an own polynomial of degree $d$
- Match the first $(d-1)$ derivatives at the nodes of the previous polynomial
- The first 3 spline-spaces
  - $d=1$: $M$-piecewise constant discontinous function
  - $d=2$: $M$-piecewise linear continous function
  - $d=3$: continously differentiable $M$-piecewise quadratic function
    - These are relevant, because $C^2$-function is smooth

## Cubic spline interpolation ($d=3$)

- For each subinterval: $S_j = a_j + b_j t + c_j t^2 + d_j t^3$
  - $\hookrightarrow 4n$ coefficients to determine!
- 1. Interpolating condition: $S_j(t_{j-1}) = \gamma_{j-1}$ & $S_j(t_j) = \gamma_j$
  - $a_j + b_j t_{j-1} + c_j \cdot t_{j-1}^2 + d_j \cdot t_{j-1}^3 = \gamma_{j-1}$
  - $a_j + b_j \cdot j + c_j \cdot j + d_j \cdot j = \gamma_j$

  } $2n$-conditions
- 2. Interpolating condition: Smoothness condition I (Match 1. derivative)
  - $S_j'(t_j) = S_{j+1}'(t_j)$ $(j = 1, \dots, n-1)$
  - $b_j + 2c_j \cdot t_j + 3d_j t_j^2 = b_{j+1} + 2c_{j+1} \cdot t_j + 3d_{j+1} \cdot t_j^2$

  } $n-1$ conditions
- 3. Interpolating Condition: Smoothness condition II (Match 2. derivative)
  - $S_j''(t_j) = S_{j+1}''(t_j)$ $(j = 1, \dots, n-1)$
  - $2c_j + 6d_j t_j = 2c_{j+1} + 6d_{j+1} \cdot t_j$

  } $n-1$ conditions
- 4. Interpolating condition: Natural / simple Cubic-Spline
  - $S_1''(t_0) = 0$
  - $S_n''(t_n) = 0$

  } 2 conditions

  - $\hookrightarrow$ Altogether: LSE for coefficients

# - Economical / Smart implementation of spline-interpolation

$h_j := t_j - t_{j-1}$

$\sigma_j :=$ unknown

- $a_j = y_{j-1}$
- $b_j = \frac{y_j - x_{j-1}}{n} - \frac{h_j \left[2 \cdot \sigma_{j-1} + \sigma_j\right]}{6}$
- $c_j = \frac{\sigma_{j-1}}{2}$
- $d_j = \frac{\sigma_j - \sigma_{j-1}}{6 h_j}$

$\hookrightarrow \sigma_{j-1} \cdot \frac{h_j}{6} + \sigma_j \cdot \frac{h_j + h_{j+1}}{3} + \sigma_{j+1} \cdot \frac{h_{j+1}}{6} = r_j$

$$\begin{bmatrix} \frac{h_1 + h_2}{3} & \frac{h_2}{6} & & & \\ \frac{h_2}{6} & \frac{h_2 + h_3}{3} & \frac{h_3}{6} & & \\ & \ddots & \ddots & \ddots & \\ & & \frac{h_{n-1}}{6} & \frac{h_{n-1} + h_n}{3} & \end{bmatrix} \cdot \begin{bmatrix} \sigma_1 \\ \vdots \\ \sigma_{n-1} \end{bmatrix} = \begin{bmatrix} r_1 \\ \vdots \\ r_{n-1} \end{bmatrix}$$

- $\sigma_0 = 0 = \sigma_n$

1. Solve tridiagonal system for $\sigma_j$
2. Compute $a_j$ with $y_{j-1}$
3. Insert $\sigma_j / h_j$ into $b_j / c_j / d_j$

# 6. Approximation of functions

- Given a function $f$, find a "simple"-approximation-func. $\tilde{f}$
- Minimize error $\|f - \tilde{f}\|$
- General idea: 1. Get samples from the function $f$
  2. Interpolate points
  3. Get an approximation function $\tilde{f}$
- Estimate of error-norm $\|f - \tilde{f}\|$ for some norm (e.g. $L^\infty$ = greatest error)
  - $\|f - \tilde{f}\| \leq C \cdot T(n)$ for $n \to \infty$
  - Algebraic convergence with rate $p > 0$: $\exists p > 0 : T(n) \leq n^{-p}$
  - Exponential convergence : $\exists 0 < p < 1 : T(n) \leq q^n$
  - ↳ We can get the convergence by measuring $(n_i, \epsilon_i)$, where $n_i$ is the polynomial degree and $\epsilon_i$ is the measured error
  - Algebraic convergence: $\log(\epsilon_i) \approx \log(c) - p \cdot \log(n_i) \Rightarrow$ linear in log-log-scale
  - Exponential convergence: $\log(\epsilon_i) \approx \log(c) - \beta \cdot n_i \Rightarrow$ linear in lin-log-scale

  } plotting

$$\|f - L_T f\|_{L^\infty(I)} \leq \frac{\|f^{(n+1)}\|_{L^\infty(I)}}{(n+1)!} \max_{t \in I} |(t - t_0) \cdot \dots \cdot (t - t_n)| \quad .$$

} Approximation of error

## Taylor-Polynom-Approximation

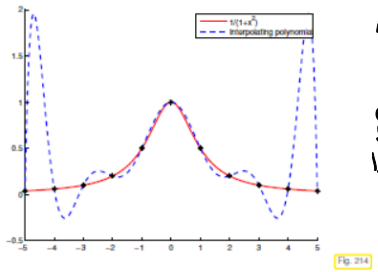
- $f(t) \approx \sum_{j=0}^{k} \frac{f^{(j)}(t_0)}{j!} \cdot (t - t_0)^j$
- Approximates $f(t)$ in a neighbourhood of $t_0$
  ↳ Only provides approximation in a small interval

## Uniform Bernstein Approximation

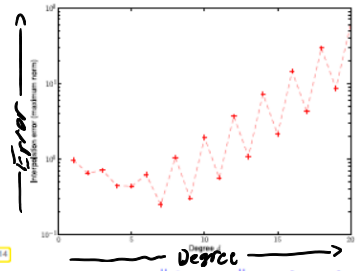
- Approximates $f(t)$ on a closed interval
- $p_n(t) = \sum_{j=0}^{n} f(j/n) \cdot \binom{n}{j} \cdot t^j (1-t)^{n-j}$
- As $n \to \infty$, the error $\|f - p_n\|_\infty \to 0$
- We need a high-degree-polynomial to precisely estimate $f(x)$
  ↳ Inefficient!

## Equidistant Lagrange-Interpolation Approximation

- Take some nodes with the same distance to each other
- Use these nodes for Lagrange Interpolation
- As more points we use, the more precision we get ↯
  - ↳ Runges-Experiment
    - Strong oscillation of Lagrange-Interpolation near the endpoints of the interval



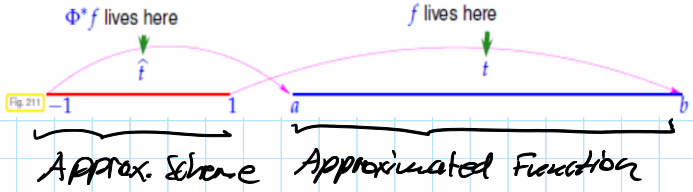
Interpolating polynomial, $n = 10$




Approximate $\|f - L_{T_n} f\|_\infty$ on $[-5, 5]$

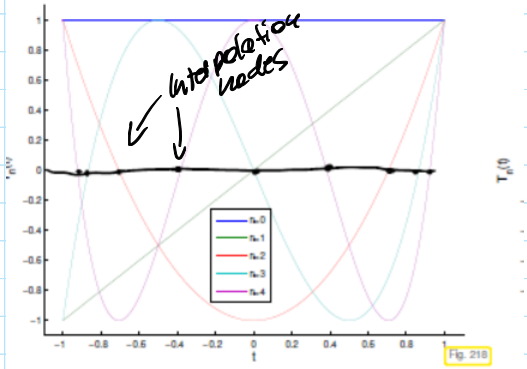
## Transformation of polynomial approximation schemes

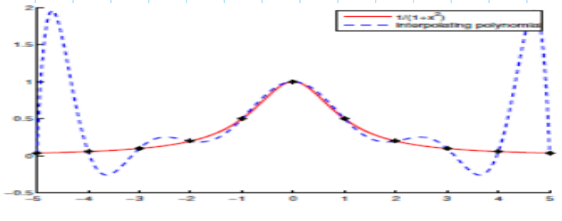
- If approx. scheme is only defined on $[-1,1]$, then by the following trick, we can transfer any interval $[a,b]$ to $[-1,1]$
- $\Phi : [-1,1] \to [a,b]$
- $\tilde{\Phi}(t) = a + \frac{1}{2}(t+1)\cdot(b-a)$

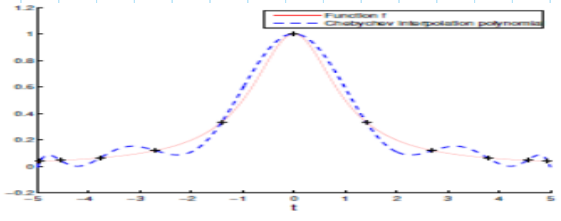



## Chebychev-Interpolation

- Optimal choice of interpolation nodes independent of interpolant
- The $n^{th}$ chebychev polynomial is $T_n(t) = \cos(n \cdot \arccos(t)) \quad (-1 \le t \le 1)$
- Recursive definition: $T_0 = 1 \,/\, T_1(t) = t \,/\, T_{n+1}(t) = 2t \cdot T_n(t) - T_{n-1}(t)$
- The polynomials $T_n$ minimize the supremum/$L^\infty$-Norm
- The zeros of $T_n$ are $t_k = \cos\left(\frac{2k+1}{2n} \cdot \pi\right)$
- When we use Chebychev nodes for polynomial interpolation, we call the resulting Lagrangian-approx. scheme "Chebychev interpolation"



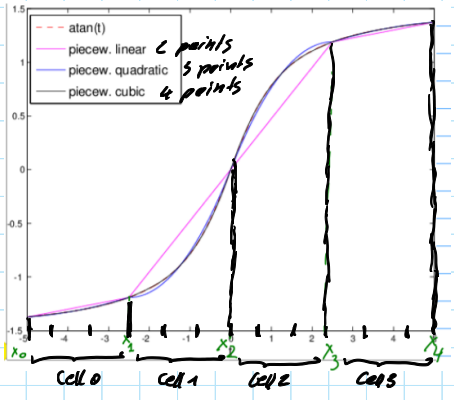



- We observe, that the Chebychev nodes cluster at the endpoints of the interval, which prevents oscillation
- Because $\{T_0, \dots, T_n\}$ is a basis of $\mathcal{P}_n$, we can write the polynomial as a Chebychev-expansion: $p = \sum_{j=0}^{n} \alpha_j \cdot T_j$, $\alpha \in \mathbb{R}$
- Error approximation: $\|f - L_j(f)\|_{L^\infty(I)} \leq \frac{2 \cdot 2^{n-1}}{(n+1)!} \cdot |I|^{n+1} \cdot \|f^{(n+1)}\|_{L^\infty(I)}$

## Piecewise polynomial Lagrange-Interpolation

- We make a mesh $\mathcal{M}$ out of the interval $[a,b]$
  - $x_j$: nodes of the mesh $\mathcal{M}$
  - $[x_{j-1}, x_j]$: cells of the mesh $\mathcal{M}$
  - $h_{\mathcal{M}}$: Maximal mesh width
  - $n_j$: local degree of the interpolating polynomial for each cell
  - $T^j$: set of local interpolating points (we need $(n_j+1)$ of them)
- We need to satisfy: $t^j_{n_j} = t^{j+1}_0$ for continuity
- We can improve error by decreasing mesh width
- Error: $\|f - s\|_{L^\infty([x_0, x_{m_j}])} \leq \frac{h_{\mathcal{M}}^{n+1}}{(n+1)!} \cdot \|f^{(n+1)}\|_{L^\infty}$



# 7. Numerical Quadrature

- Approximate $\int_a^b f(t)\,dt$ using only point evaluations of $f(t)$
- General quadrature formula: $\int_a^b f(t)\,dt \approx Q_n(f) := \sum_{j=1}^{n} w_j^n \cdot f(c_j^n)$
  - $w_j^n$: weight, $\in \mathbb{R}$
  - $f(c_j^n)$: Nodes $\in [a,b]$

## Affine Pullback

$$\int_a^b f(t)\,\mathrm{d}t \approx \tfrac{1}{2}(b-a)\sum_{j=1}^{n} \widehat{w}_j \widehat{f}(\widehat{c}_j) = \sum_{j=1}^{n} w_j f(c_j) \quad \text{with} \quad c_j = \tfrac{1}{2}(1-\widehat{c}_j)a + \tfrac{1}{2}(1+\widehat{c}_j)b, \quad w_j = \tfrac{1}{2}(b-a)\widehat{w}_j.$$

($\widehat{f}$: $[-1,1]$, $f$: $[a,b]$)

$= a + \frac{1}{2}(\hat{c}_j + 1)(b-a)$

## Quadrature by approximation scheme

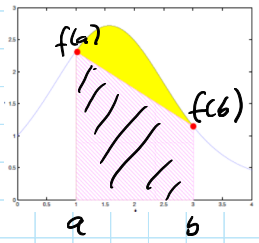
- $\int_a^b f(t)\,dt \approx \int_a^b I[f(t_0), \dots, f(t_n)](t)\,dt = \sum_{j=1}^{n} f(t_j) \int_a^b I[e_j](t)\,dt = \sum_{j=1}^{n} f(t_j) \cdot w_j^n$ ($w_j^n$)
  - $f$: Echte Funktion
  - $I[\dots]$: Approximierende-Funktion mittels Interpolation
- Interpolation $\Rightarrow$ Approximation $\Rightarrow$ Quadrature

## Polynomial quadrature by approximation scheme

- $\int_a^b f(t)\,dt \approx \int_a^b p_{n-1}(t)\,dt = \sum_{i=0}^{n-1} f(t_i) \cdot \int_a^b L_i(t)\,dt = \sum_{i=0}^{n-1} f(t_i) \cdot w_i$
  - $p_{n-1}(t)$: Polynomial Lagrange interpolant of degree $(n-1)$
  - $t_i$: Nodes
  - $\int_a^b L_i(t)\,dt$: Weights
  - $f(t_i)$: Node, $w_i$: Weight

## Newton-Cotes formulas

- Lagrange-Interpolation with equidistant nodes
- $t_j := a + \frac{b-a}{n-1} \cdot j$
- $n = 2$: Trapezoidal-Rule: $\int_a^b f(t)\,dt \approx 1 \cdot \frac{b-a}{2}(f(a)+f(b))$ (1: weight)
- $n = 3$: Simpson-Rule




$\Rightarrow$ Lagrange-Interpolation is unstable with equidistant nodes $\lightning$

# Gauss Quadrature

- Order: · Quality measurement, for how good the rule is
  · Max. degree +1 of polynomials, for which the quadrature rule is guaranteed to be correct
  · Depends on the node set $\mathcal{T} = \{t_1, \dots, t_n\}$
  · The maximal order of an n-point quad.-rule is $2n$
  ↳ With n points we can integrate a polynomial, using Lagrange-Interpolation, with max degree of $2n-1$
- Legendre-Polynomials: $\int_{-1}^{1} P_n(t) \cdot q(t) \cdot dt = 0$ / $P_n(1) = 1$
- The zeros of the Legendre-Polynomials form the Gauss-Points

**Definition 7.3.29. Gauss-Legendre quadrature formulas**

The $n$-point Quadrature formulas whose nodes, the Gauss points, are given by the zeros of the $n$-th Legendre polynomial (→ Def. 7.3.27), and whose weights are chosen according to Thm. 7.3.5, are called Gauss-Legendre quadrature formulas.

- Evaluation points from Legendre: $t_j, j = 1, \dots, n$
- Weights for points from Lagrange: $w_j = \int_a^b \underbrace{L_{j-1}(t)}_{\text{Polynom}} \cdot dt$ (Positive Constant)

$$\int_a^b f(t)\,dt = \sum_{j=1}^{n} w_j \cdot f(t_j)$$

- The gauss-points (Legendre-roots) and gauss-weights (Lagrange-Poly.) can be computed with the Golub-Welsch-algorithm.

**Lemma 7.3.42. Quadrature error estimates for $C^r$-integrands** (Error-estimation)

For every $n$-point quadrature rule $Q_n$ as in (7.1.2) of order $q \in \mathbb{N}$ with weights $w_j \geq 0$, $j = 1, \dots, n$ we find that the quadrature error $E_n(f)$ for and integrand $f \in C^r([a,b])$, $r \in \mathbb{N}_0$, satisfies

$$\text{in the case } q \geq r: \quad E_n(f) \leq C\, q^{-r} |b-a|^{r+1} \left\| f^{(r)} \right\|_{L^\infty([a,b])}, \tag{7.3.43}$$

$$\text{in the case } q < r: \quad E_n(f) \leq \frac{|b-a|^{q+1}}{q!} \left\| f^{(q)} \right\|_{L^\infty([a,b])}, \tag{7.3.44}$$

with a constant $C > 0$ independent of $n$, $f$, and $[a,b]$.

# Composite Quadrature

- Diride into small intervals
- Guess at least as good as composite QF

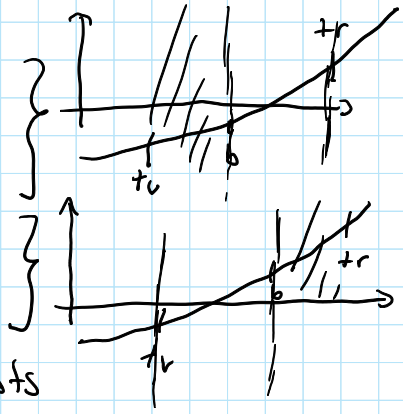
# 8. Iterative methods for nonlinear systems of equations

- Iterative methods for finding approximations to the solution of a nonlinear system

⇒ How to solve $f(x^*) = 0$ (= root)

## Bisection algorithm

```
while(n ≤ n_max) do        /iterate n_max times
    b = (t_l + t_r)/2      /compute the middle x^(k)
    if (|f(b)| < Tol_1 || |t_l - t_r| < Tol_2)   /check if approximation
        return x* = b                            /is sufficient
    n++                    /increase counter
    if (sign(f(t_r)) == sign(f(b)))
        t_r = b
    if (sign(f(t_l)) ≠ sign(f(b)))
        t_l = b
```




- We always search, where a root exists
- Iterative error: $e^{(k)} = x^{(k)} - x^*$
- Rate of convergence: $x^{(k)} \to x^*$
  - For bisection method: $|e^{(1)}| \leq \frac{1}{2}|a-b|$, $|e^{(2)}| \leq \frac{1}{2^2}|a-b|$ } "linear type" convergence by factor 1/2

## Fixed Point Convergence in 1D

- $f(x) = 0 \iff \Phi(x) := f(x) + x$ (= finding fixed point)
- $x^*$ is a fixed point of $\Phi$: • $\Phi(x^*) = x^*$
  • $f(x^*) = 0$
- Requirement: $\Phi$ is Lipschitz stetig on $[a,b]$
  ↳ $\exists L > 0 \ \forall x, y \in [a,b]: |\Phi(x) - \Phi(y)| \leq L \cdot |x-y|$
  • $L < 1$ ($\Phi$ is a contractive mapping)

- General construction of fixed-point iteration

1. Rewrite equivalently: $F(x) = 0 \Leftrightarrow \Phi(x) = x$

↳ $x \cdot e^x - 1 = y$ $\qquad$ $x \cdot e^x - 1 = y$

$x \cdot e^x - 1 = 0$ $\qquad$ $x \cdot e^x - 1 = 0$

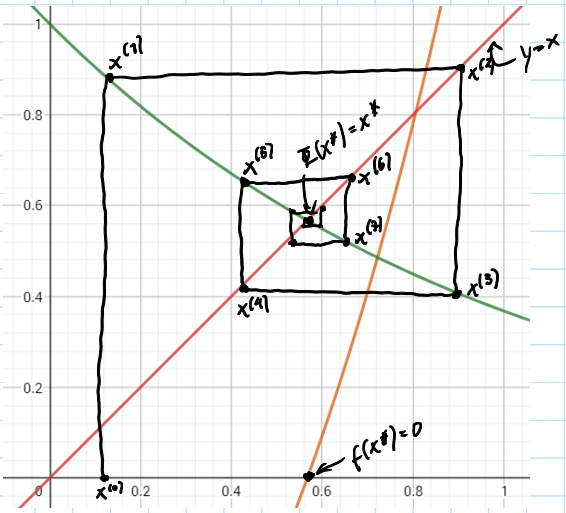
$x = e^{-x}$ $\qquad$ $x = \frac{1+x}{1+e^x}$

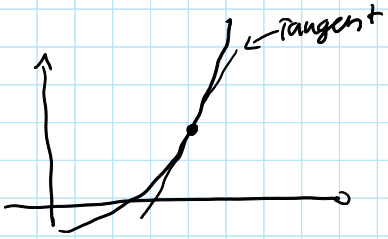
$(y = e^{-x})$ $\qquad$ $(y = \frac{1+x}{1+e^x})$

2. Start with initial guess $x^{(0)} = 0.1$

3. Use the fixed point iteration: $x^{(k+1)} = \Phi(x^{(k)})$

↳ $x^{(k+1)} = e^{-x^{(k)}}$ $\qquad$ $x^{(k+1)} = \frac{1+x^{(k)}}{1+e^{x^{(k)}}}$



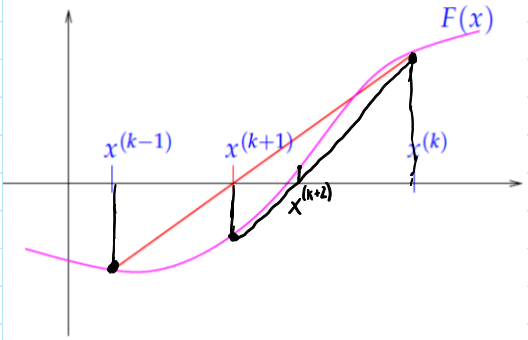

## Newton iteration in 1D

- Gives us quadratic convergence
- Requirement: $f \in C^1$
- Intuition: Approx. around $x^{(k)}$: $f(x) \approx f(x^{(k)}) + (x - x^{(k)}) \cdot f'(x^{(k)})$

$0 = f(x^{(k)}) + (x^{(k+1)} - x^{(k)}) \cdot f'(x^{(k)})$

- $x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}$
- Newton's method converges quadratically, if $f'(x^*) \neq 0$
- Requires computation of $f'(x^{(k)})$ for each iteration → costly!
- Alternative: Secant method
  - Replace $f'(x^{(k)})$ by an approximation

- $f'(x^{(k)}) \approx \frac{f(x^{(k)}) - f(x^{(k-1)})}{x^{(k)} - x^{(k-1)}}$
- It's called a 2-point-method
  ↳ Depends on the most 2 recent computed points




## Nonlinear systems of equations

- n equations, n unknowns
- Convergence in $\mathbb{R}^n$ independent of choice of norm
  - Local convergence: Only convergence in a neighbourhood → good 1st guess
  - Global convergence: 1st guess isn't important
- Fixed point iteration in higher dimensions → Lipschitz-steig
  - Requirements: - As in 1D, the mapping must be contractive
    - $\Phi$ converges at least local linearly, if $\|D\Phi(x^*)\| < 1$ ($D\Phi(x)$ = Jacobian-Matrix)
    - $\Phi$ converges at least linearly, if everywhere $\|D\Phi(x^*)\| < 1$
  - Termination criteria: - $\|F(x^{(k)})\| \leq \tau$
    - $\|x^{(k+1)} - x^{(k)}\| \leq \tau \cdot \|x^{(k+1)}\|$
  - But: $\|F(x^{(k)}) - F(x^*)\|$ small $\not\Rightarrow$ $\|x^{(k)} - x^*\|$ small
    ↳ Ultimate goal: guarantee of form $\|x^{(k)} - x^*\| \leq \tau$

$$\|x^* - x^{(k)}\| \leq \frac{L^{k-l}}{1-L}\|x^{(l+1)} - x^{(l)}\| \quad \} \text{Approximation (8.2.21)}$$

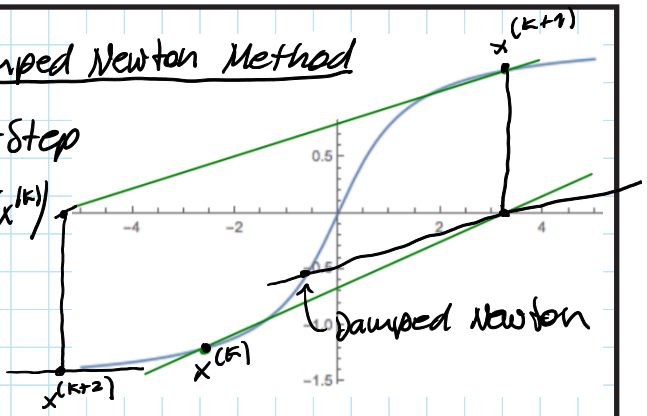
| Set $l = 0$ in (8.2.21) | Set $l = k-1$ in (8.2.21) |
|---|---|
| a priori termination criterion | a posteriori termination criterion |
| $\|x^* - x^{(k)}\| \leq \frac{L^k}{1-L}\|x^{(1)} - x^{(0)}\| \leq \tau$ | $\|x^* - x^{(k)}\| \leq \frac{L}{1-L}\|x^{(k)} - x^{(k-1)}\| \leq \tau$ |
| Compute, how many steps | compute at every step |

- Newton's method in higher dimensions

  - 1D: $f'(x^{(k)}) \neq 0$
  - Higher dimensions: Invertibility of Jacobian-Matrix
  - $x^{(k+1)} := x^{(k)} - \underbrace{DF(x^{(k)})^{-1}}_{\text{Jacobian}} \cdot F(x^{(k)})$
  - Newton correction $:= -DF(x^{(k)})^{-1} \cdot F(x^{(k)})$
    - ↳ To compute Newton-correction: solve LGS: $DF(x^{(k)}) \cdot y = -F(x^{(k)})$
  - Stopping criteria: $\|DF(x^{(k-1)})^{-1} \cdot F(x^{(k)})\| \leq \tau \cdot \|x^{(k)}\|$
    - No need to compute $x^{(k)}$, even if $x^{(k-1)}$ was a good enough approximation
    - $DF(x^{(k)}) \approx DF(x^{(k+1)})$ at the last steps
  - Problem: Newton correction is costly because of Jacobian
    - Secant method in higher dimensions
    - Broyden's quasi Newton Method
      - $x^{(k+1)} = x^{(k)} - \underbrace{\left[J_{k-1} + \frac{F(x^{(k)}) \cdot (x^{(k)} - x^{(k-1)})^T}{\|x^{(k)} - x^{(k-1)}\|_2^2}\right]^{-1}}_{J_k^{-1} \approx DF(x^{(k)})^{-1}} \cdot F(x^{(k)})$

- Problem: Overshooting / Solution: Damped Newton Method

  - Idea: Damp the Newton-Step
  - $x^{(k+1)} := x^{(k)} - \lambda^{(k)} \cdot DF(x^{(k)})^{-1} \cdot f(x^{(k)})$
  - $\lambda^{(k)}$: Damping factor of the $k^{th}$-Iteration




  - Strategy: Largest possible $\lambda^{(k)}$ so that distance between iterate points decrease

**Affine invariant damping strategy**

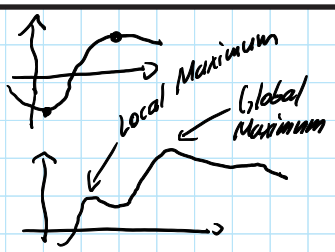
Choice of damping factor: affine invariant natural monotonicity test [?, Ch. 3]:

choose "maximal" $0 < \lambda^{(k)} \leq 1$: $\left\| \Delta\bar{\mathbf{x}}(\lambda^{(k)}) \right\| \leq (1 - \frac{\lambda^{(k)}}{2}) \left\| \Delta\mathbf{x}^{(k)} \right\|_2$ (8.4.57)

where
$\Delta\mathbf{x}^{(k)} := DF(\mathbf{x}^{(k)})^{-1} F(\mathbf{x}^{(k)})$ → current Newton correction ,
$\Delta\bar{\mathbf{x}}(\lambda^{(k)}) := DF(\mathbf{x}^{(k)})^{-1} F(\mathbf{x}^{(k)} + \lambda^{(k)} \Delta\mathbf{x}^{(k)})$ → tentative simplified Newton correction .

1. Set $\lambda^{(k)} = 1$
2. check natural monotonicity test
3. repeatedly take $\lambda^{(k)} = \frac{\lambda^{(k)}}{2}$ until NMT passes for the first time and calculate $x^{(k)}$ with damping-factor

# 8.5 Unconstrained Optimization




- General question: How to find min/max of $f(x)$?
- Maximizing $f(x) \Leftrightarrow$ minimizing $-f(x)$
  ↳ Therefore only consider minimization problems

## Minimization/Optimization with differentiable functions

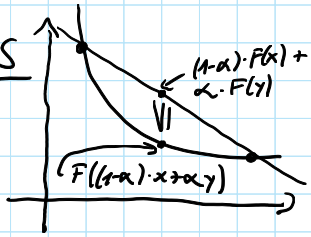
- $F: \mathbb{R}^n \to \mathbb{R}$ differentiable
- $\nabla F$: direction of greatest increase
- $-\nabla F$: direction of greatest descent
- $\nabla f(x) = 0$: local/global max/min or saddle point
- If $F$ is twice differentiable, check Hessian-Matrix at stationary point where $F(x) = 0$, because

$$F(x) \approx F(\bar{x}) + \underbrace{\nabla F(\bar{x})^T (x-\bar{x})}_{=0 \text{ at stationary point}} + \frac{1}{2}(x-\bar{x})^T \cdot H_F(\bar{x}) \cdot (x-\bar{x})$$

- At stationary point: $F(x) \approx F(\bar{x}) + \frac{1}{2} \cdot (x-\bar{x})^T \cdot H_F(\bar{x}) \cdot (x-\bar{x})$
  - $H_F(\bar{x})$ negative definite: $\bar{x}$ local maximum
  - $H_F(\bar{x})$ indefinite : $\bar{x}$ saddle point
  - $H_F(\bar{x})$ positive definite: $\bar{x}$ local minimum
    ↳ Check by Cholesky factorization / Eigenvalues

## Minimization/Optimization with convex functions

- Convex: $F((1-\alpha)\cdot x + \alpha y) \leq (1-\alpha)\cdot F(x) + \alpha \cdot F(y)$
  ↳ Local minimum $\hat{=}$ global minimum

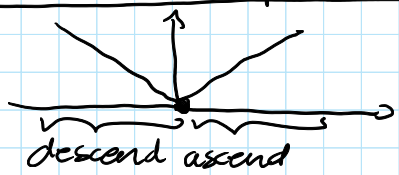



## Methods in 1D

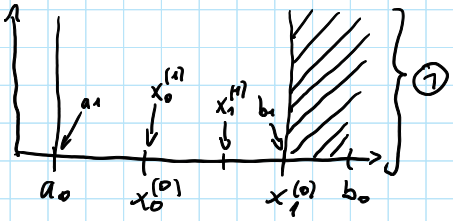
- Newton's method for $C^2$ functions
  - Applied to $f'(x)$ to search for $f'(x) = 0 \Rightarrow$ Min/Max
  - $x_{k+1} = x_k - \frac{f'(x)}{f''(x)}$

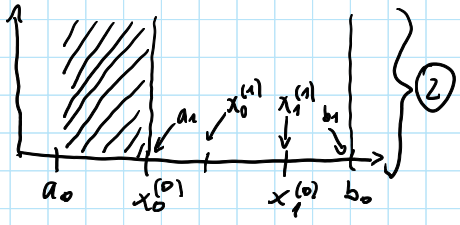
## - Golden-Section-Search for non-differentiable functions

- Function must be unimodal:




- In each iteration, interval size is reduced by factor $\lambda$
  ↳ Linear-type-convergence
- In each iteration, only one point is computed
- while $|b-a| >$ tolerance




if $f_1 > f_0$ ①

$$b = x_1$$
$$x_1 = x_0$$
$$f_1 = f_0$$
$$x_0 = a + (1-\lambda)(b-a)$$
$$f_0 = f(x_0)$$

if $f_0 \geq f_1$ ②

$$a = x_0$$
$$x_0 = x_1$$
$$f_0 = f_1$$
$$x_1 = a + \lambda(b-a)$$
$$f_1 = f(x_1)$$

## Methods in higher dimensions

### - Gradient descend

- Greatest descend direction: $-\nabla F$
- $x^{(k+1)} = x^{(k)} - t^{(k)} \cdot \nabla f(x^{(k)})$

1. Start with initial guess $x^{(0)}$
2. while $\|\nabla F\|_2 <$ tol
   3. Find stepsize $t^{(k)}$ through line search
   4. $x^{(k+1)} = x^{(k)} - t^{(k)} \cdot \nabla F(x^{(k)})$

- How to find stepsize $t^{(k)}$ in each iteration?
  ↳ with backtracking!

- Problem: line search in each iteration

- Helper: Backtracking Line search
  - $F(x - t \cdot \nabla F(x)) \overset{!}{\leq} F(x) - t \cdot \|\nabla F(x)\|^2$
    $< F(x) - \alpha \cdot t \cdot \|\nabla F(x)\|^2$
  1. iterate until good decrease is reached
  2. start with $t = 1$, fix $\alpha \in (0, 0.5]$, $\beta \in (0,1)$
  3. while $F(x - t \cdot \nabla F(x)) > F(x) - \alpha \cdot t \cdot \|\nabla F(x)\|^2$
     4. $t = \beta \cdot t$
- Newton's method
  - As in 1D: if $F$ is twice differentiable
  - $x^{(k+1)} \overset{!}{\leq} F(x^{(k)}) + \nabla F(x^{(k)})^T \cdot (x - x^{(k)}) + \frac{1}{2} \cdot (x - x^{(k)}) \cdot H_F[x^{(k)}] \cdot (x - x^{(k)})$
  - Problem: computing $H_F$ in each iteration
- BFGS-Method (= Quasi Newton)
  - Approximate $H_F(x^{(k)})$ by $B_k$
  - $B_{k+1}$ is obtained from simple update of $B_k$
  - $B_{k+1} \cdot \underbrace{(x^{(k+1)} - x^{(k)})}_{= s^{(k)}} = \underbrace{\nabla F(x^{(k+1)}) - \nabla F(x^{(k)})}_{y^{(k)}}$

$$\begin{bmatrix} \cdot\ B_{k+1} \cdot s^{(k)} = y^{(k)} \\ \cdot\ B_{k+1} = B_k + \alpha \cdot u \cdot u^T + \beta \cdot v \cdot v^T \\ \cdot\ u = y^{(k)} \\ \cdot\ v = B_k \cdot s^{(k)} \end{bmatrix}$$

$$\hookrightarrow B_{k+1} = B_k + \frac{y^{(k)} \cdot y^{(k)T}}{y^{(k)T} \cdot s^{(k)}} - \frac{B_k \cdot s^{(k)} \cdot s^{(k)T} \cdot B_k^T}{s^{(k)T} \cdot B_k \cdot s^{(k)}}$$