Endterm Zusammenfassung

Dienstag, 20. Dezember 2016 21:54

Convolutions

$y_i = H.row(i) * \vec{x} =$ nennt sich discrete Convolution$= \sum_{l=0}^{height(H)} x_l h_{j-l}$

n-periodic: height = n

Nonperiodic: height = 2n (attention: indexing from 0 to 2n-1)

Fourier

Def. Root unity: $\omega_n := e^{\frac{-2\pi i}{n}} = \cos\left(\frac{2\pi}{n}\right) - i\sin\left(\frac{2\pi}{n}\right)$

These are equally spaced in the complex plane in a circle around (0,0)

$\omega_n^n = 1,$

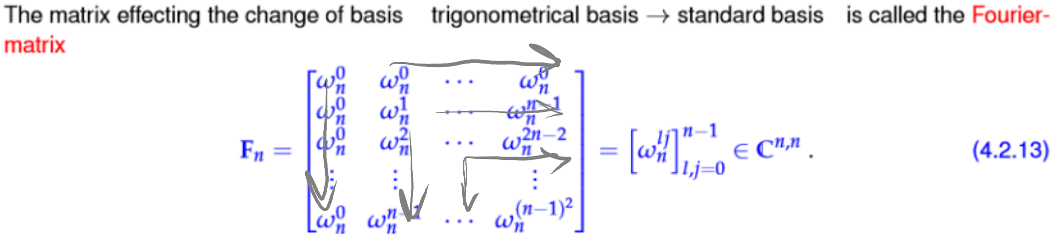
$\boldsymbol{\omega_n^{-l}} = e^{\frac{2\pi i}{-l}} = complex\ conjugate = \overline{\boldsymbol{\omega_n^l}}$

vectors v_k defined as $\left[\omega_n^{jk}\right]_{j=n}^{n-1}$

$$Cv_k = \omega_n^{kj} \sum_{l=0}^{n-1} u_l \omega_n^{-kl}$$

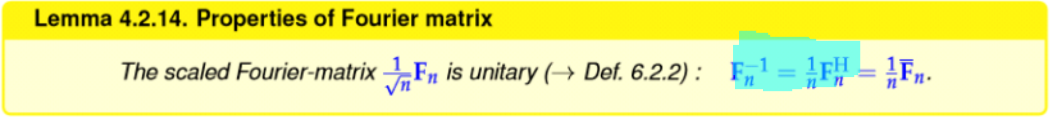
$$\Rightarrow \omega_n^{kj} = (v_k)_j = \lambda_k = Eigenwert\ von\ C\ and\ v_k\ is\ it's\ eigenvector$$

Fourier Matrix contains the Eigenvectors AND the Eigenvalues of any Circulant Matrix



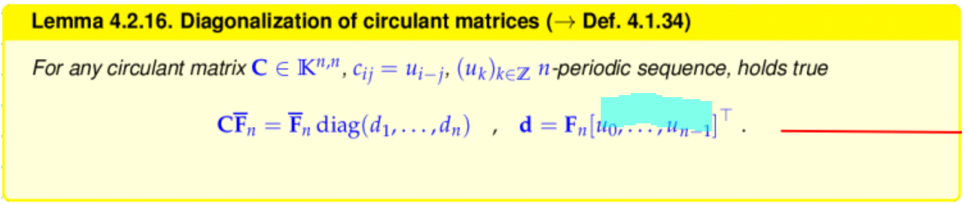
The matrix effecting the change of basis trigonometrical basis → standard basis is called the Fourier-matrix

$$\mathbf{F}_n = \begin{bmatrix} \omega_n^0 & \omega_n^0 & \cdots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \cdots & \omega_n^{n-1} \\ \omega_n^0 & \omega_n^2 & \cdots & \omega_n^{2n-2} \\ \vdots & \vdots & & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix} = \left[\omega_n^{lj}\right]_{l,j=0}^{n-1} \in \mathbb{C}^{n,n} . \qquad (4.2.13)$$



Lemma 4.2.14. Properties of Fourier matrix

The scaled Fourier-matrix $\frac{1}{\sqrt{n}}\mathbf{F}_n$ *is unitary (→ Def. 6.2.2) :* $\mathbf{F}_n^{-1} = \frac{1}{n}\mathbf{F}_n^H = \frac{1}{n}\overline{\mathbf{F}}_n$.



Lemma 4.2.16. Diagonalization of circulant matrices (→ Def. 4.1.34)

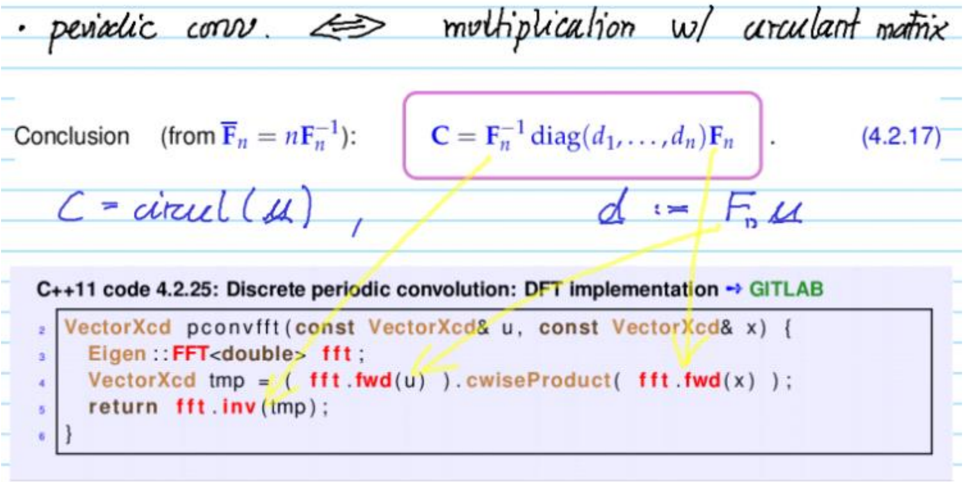
For any circulant matrix $\mathbf{C} \in \mathbb{K}^{n,n}$, $c_{ij} = u_{i-j}$, $(u_k)_{k\in\mathbb{Z}}$ *n-periodic sequence, holds true*

$$\mathbf{C}\overline{\mathbf{F}}_n = \overline{\mathbf{F}}_n \operatorname{diag}(d_1, \ldots, d_n) \quad , \quad \mathbf{d} = \mathbf{F}_n [u_0, \ldots, u_{n-1}]^\top .$$

$$\Rightarrow d = F_n \begin{pmatrix} u_0 \\ \cdots \\ u_{n-1} \end{pmatrix}$$

$$Cx = \overline{F_n} diag(d_1, .., d_n) \overline{F_n}^{-1} x = nF_n^{-1} diag \cdot \left(nF_n^{-1}\right)^{-1} x = F_n^{-1} diag \cdot F_n\ x$$

This is a periodic discrete convolution.



• periodic conv. ⟺ multiplication w/ circulant matrix

Conclusion (from $\overline{\mathbf{F}}_n = n\mathbf{F}_n^{-1}$): $\boxed{\mathbf{C} = \mathbf{F}_n^{-1}\operatorname{diag}(d_1,\dots,d_n)\mathbf{F}_n}$. (4.2.17)

$C = \mathrm{circul}(u)$, $d := F_n u$

C++11 code 4.2.25: Discrete periodic convolution: DFT implementation → GITLAB

```
VectorXcd pconvfft(const VectorXcd& u, const VectorXcd& x) {
  Eigen::FFT<double> fft;
  VectorXcd tmp = ( fft.fwd(u) ).cwiseProduct( fft.fwd(x) );
  return fft.inv(tmp);
}
```

Zero Padding

We now have a function that takes u and x as argument and performs a convolution $y = C(u) \cdot x = \overline{F_n} \cdot (F_n \vec{u}) \cdot F_n \cdot x$

In the beginning we had a H-Matrix (Filter) which is always circulant so this works always.

If we don't have periodicity, we can pad with zeros to remove interference:

$\vec{u}$ has size n
$C(u)\ has\ size\ n \times n\ for\ n-periodicity$
pad $u' \rightarrow size\ 2n-1$
$\Rightarrow C(u')\ has\ size\ (2n-1)\times(2n-1) \Rightarrow no\ interference$
$x\ has\ to\ fit\ the\ second\ dimension\ of\ C \Rightarrow x\ has\ size\ (2n-1)$

Frequencies

$$v_k = \left[\omega_n^{kj}\right]_{j=0}^{n-1} = \left[\cos\left(\frac{2\pi kj}{n}\right)\right]_{j=0}^{n-1} - i\left[\sin\left(\frac{2\pi kj}{n}\right)\right]_{j=0}^{n-1}$$

umformung der Definition $\omega_n = e^{i\left(-\frac{2\pi}{n}\right)}$ => frequenzen filtern: indem man die Mittleren Zeilen von $F_n\vec{x}$ auf Null setzt erhält man die hohen frequenzen. (Mittig, weil es nacher wieder langsamer wird, einfach mit richtungswechsel. Stichwort Stroboskop.)

Frequency identification

Some measurement with noise that might be repetitive => apply DFT => probably these frequencies that have a high number in the transformed Vector.

DFT in 2D

Same, but twice

Basis of 2D: because F is basis in 1D

$$V_{m,n} = \left\{\left[\omega_m^{lv}\right]_{l=0}^{m-1}\left(\left[\omega_n^{kp}\right]_{k=0}^{n}\right)^T\right\}_{\substack{v=0,\dots,m-1\\ p=0,\dots,n-1}}$$

$DFT\ of\ Y{:}\ C = F_m Y F_n$

$Inverse{:}\, Y = F_m^{-1} C F_n^{-1} = \frac{1}{mn}\overline{F_m} C \overline{F_n}$

//because scaled F is unitary. Remember: $\frac{1}{n}\overline{F_n} = F_n^{-1}$

C-wise DFT of Y: $(C)_{l,k} = \sum_{v=0}^{m-1}\sum_{\mu=0}^{n-1}(y)_{v,\mu}\omega_m^{lv}\omega_n^{\mu k}$

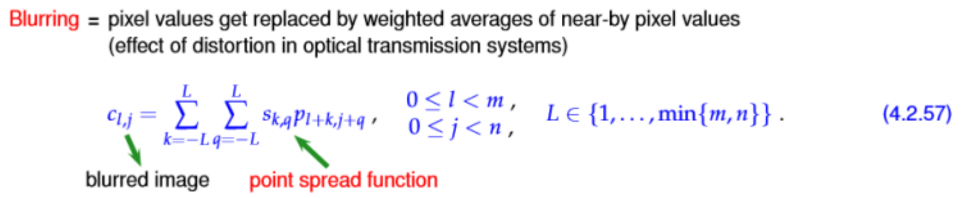
To calculate this, we usually don't do it with Matrix multiplication but with first row-wise and then column-wise FFT because this is in $O(n\log(n))$ solvable. This is mathematically the same as multiplying with F on both sides ($C = F_m Y F_n$). Think about it row-wise with F symmetric $\Rightarrow rows\ and\ columns\ are\ same\ in\ F.$

given Matrix Y. for all rows of Y:
tempMatrix.row(k) = fft.fwd(Y.row(k)).transpose();
then apply FFT again for all columns of this tempMatrix:

C.col(k) = fft.fwd(tempMatrix.col(k));

Deblurring

Blurring Operator B is given as pixelwise

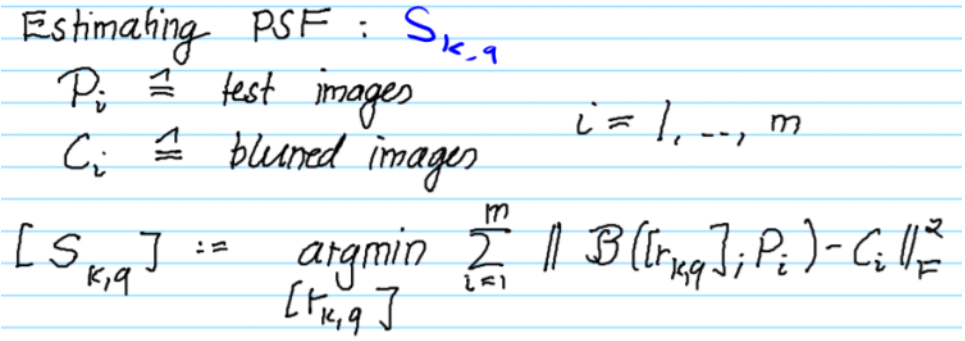


Blurring = pixel values get replaced by weighted averages of near-by pixel values (effect of distortion in optical transmission systems)

$$c_{l,j} = \sum_{k=-L}^{L}\sum_{q=-L}^{L} s_{k,q}\, p_{l+k,j+q}\,, \quad \begin{aligned} 0 \le l < m\,, \\ 0 \le j < n\,, \end{aligned} \quad L \in \{1,\dots,\min\{m,n\}\}\,. \qquad (4.2.57)$$

The solution for deblurring is to switch to Fourier space, divide cwise by s and then switch back. That works because of tricks - view Endterm Recap.

Estimating point spread function



Estimating PSF : $S_{k,q}$

$P_i \hat{=}$ test images

$C_i \hat{=}$ blurred images $\qquad i = 1, \dots, m$

$$[S_{k,q}] := \underset{[r_{k,q}]}{\operatorname{argmin}} \sum_{i=1}^{m} \| B([r_{k,q}]; P_i) - C_i \|_F^2$$

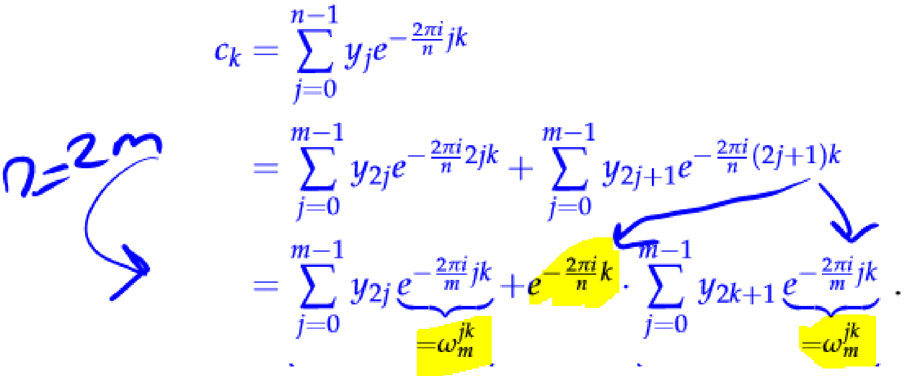
also Unterschied zwischen blurred und Blur(real) image minimieren.

Fast Fourier Transform

Divide Sums into a sum for the even and a sum for the odd indizes of the given Vector.

Recursion => $O(n\log(n))$

$for\ size\ of\ y\ called\ n = 2m$:

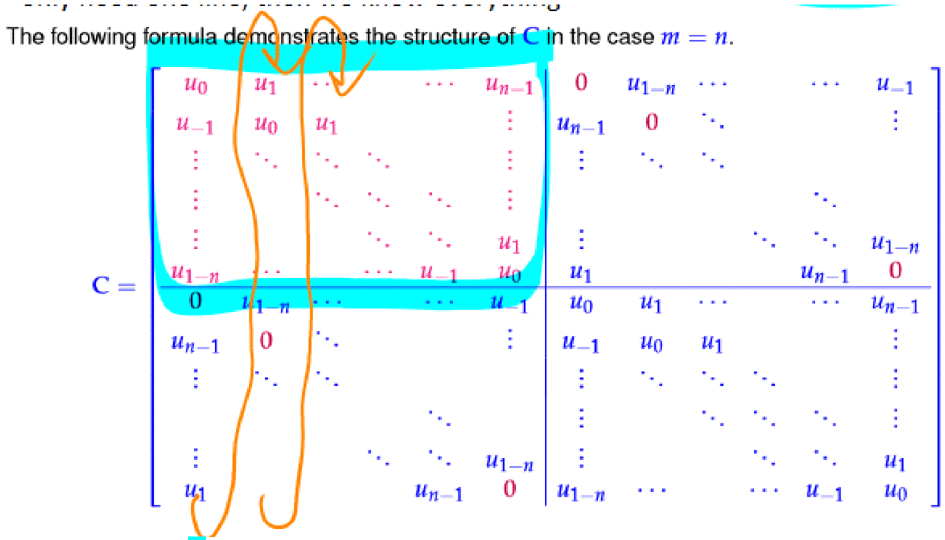


$$\begin{aligned} c_k &= \sum_{j=0}^{n-1} y_j e^{-\frac{2\pi i}{n} jk} \\ &= \sum_{j=0}^{m-1} y_{2j} e^{-\frac{2\pi i}{n} 2jk} + \sum_{j=0}^{m-1} y_{2j+1} e^{-\frac{2\pi i}{n}(2j+1)k} \\ &= \sum_{j=0}^{m-1} y_{2j} \underbrace{e^{-\frac{2\pi i}{m} jk}}_{=\omega_m^{jk}} + e^{-\frac{2\pi i}{n} k} \cdot \sum_{j=0}^{m-1} y_{2k+1} \underbrace{e^{-\frac{2\pi i}{m} jk}}_{=\omega_m^{jk}} . \end{aligned}$$

Toeplitz Multiplication with a Vector

constant diagonals => m+n-1 actual information content numbers ⇒ Toeplitz matrix can be displayed with a vector $u = (u_{-m+1}, \dots, u_{n-1})$

Extend to Circulant matrix of size 2m × 2n only need one line, then we know everything



The following formula demonstrates the structure of C in the case $m = n$.

$$\mathbf{C} = \left[\begin{array}{cccccc|cccccc}
u_0 & u_1 & \cdots & & \cdots & u_{n-1} & 0 & u_{1-n} & \cdots & & \cdots & u_{-1} \\
u_{-1} & u_0 & u_1 & & & \vdots & u_{n-1} & 0 & \ddots & & & \vdots \\
\vdots & \ddots & \ddots & \ddots & & \vdots & \vdots & \ddots & \ddots & & & \\
\vdots & & \ddots & \ddots & \ddots & \vdots & & & & & \ddots & \\
\vdots & & & \ddots & \ddots & u_1 & \vdots & & & \ddots & \ddots & u_{1-n} \\
u_{1-n} & \cdots & & \cdots & u_{-1} & u_0 & u_1 & & & & u_{n-1} & 0 \\
\hline
0 & u_{1-n} & \cdots & & \cdots & u_{-1} & u_0 & u_1 & \cdots & & \cdots & u_{n-1} \\
u_{n-1} & 0 & \ddots & & & \vdots & u_{-1} & u_0 & u_1 & & & \vdots \\
\vdots & \ddots & \ddots & & & & \vdots & \ddots & \ddots & \ddots & & \vdots \\
 & & & & \ddots & & \vdots & & \ddots & \ddots & \ddots & \vdots \\
\vdots & & & \ddots & \ddots & u_{1-n} & \vdots & & & \ddots & \ddots & u_1 \\
u_1 & & & & u_{n-1} & 0 & u_{1-n} & \cdots & & \cdots & u_{-1} & u_0
\end{array}\right]$$

Now instead of $Tx, we\ can\ calculate\ C\begin{pmatrix}\vec{x}\\0\end{pmatrix} = \begin{pmatrix}Tx\\idgaf\end{pmatrix}$

This is like a convolution, so we can solve this using FFT.

Solving Least Squares for minimizing Filter error

can be solved like this, because Normal equation contains $A^T A$, which is a Toeplitz Matrix when A circulant. $\Rightarrow A^T Ah = y\ to\ minimize\ the\ error \Rightarrow fast.$

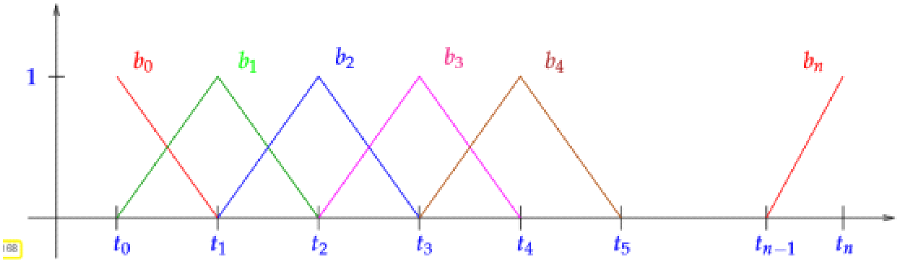
Interpolation in 1D

Piecewise Linear Interpolation

Connect measured points by lines.

for $Form\ ax = b$: $a = slope = \frac{y_i - y_{i-1}}{t_i - t_{i-1}},\ b = where\ line\ cuts\ y$

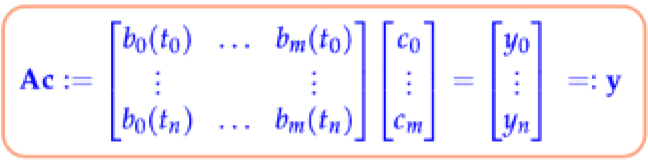
adding up tent functions: they have to be not influencing any other points and add up to a line in between points.



height 1 because they can be scaled using coefficients

Interpolation as linear Mapping

$BaseFunctions * Coefficients = measurements$



$$\mathbf{A}\mathbf{c} := \begin{bmatrix} b_0(t_0) & \ldots & b_m(t_0) \\ \vdots & & \vdots \\ b_0(t_n) & \ldots & b_m(t_n) \end{bmatrix} \begin{bmatrix} c_0 \\ \vdots \\ c_m \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix} =: \mathbf{y}\ .$$

=> need $m = n \Rightarrow Anzahl\ measurements = Anzahl\ base\ vectors$

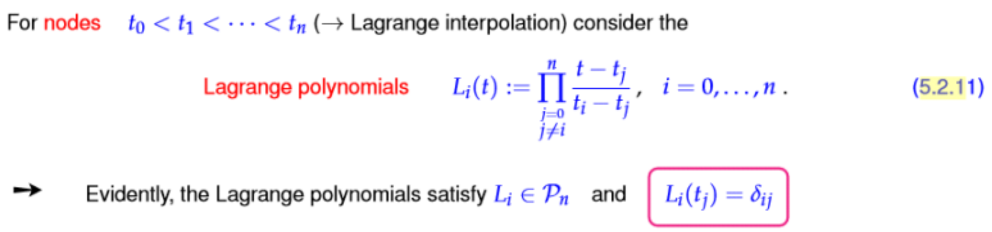
Polynomial Interpolation

Horner Scheme: Instead of calculating a polynomial $p(t)$ by calculating all powers of t, you can calculate it recursively

$$p(t) = t(\cdots t(t(\alpha_n t + \alpha_{n-1}) + \alpha_{n-2}) + \cdots + \alpha_1) + \alpha_0\ . \qquad (5.2.6$$

=> lineare zeit

Lagrange Polynomials as Cardinal Basis



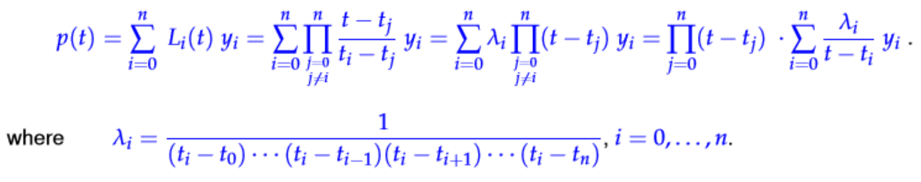
For nodes $t_0 < t_1 < \cdots < t_n$ (→ Lagrange interpolation) consider the

Lagrange polynomials $$L_i(t) := \prod_{\substack{j=0\\ j\neq i}}^{n} \frac{t-t_j}{t_i-t_j}, \quad i = 0,\dots,n\,. \qquad (5.2.11)$$

➙ Evidently, the Lagrange polynomials satisfy $L_i \in \mathcal{P}_n$ and $\boxed{L_i(t_j) = \delta_{ij}}$

and based on that Basis the Lagrange interpoland $p(t) = \sum_{i=0}^{n} y_i L_i(t)$ which is just the sum of all Lagrange polynomials. It fulfills $p(t_i) = y_i$.

fast by precomputing part of L_i



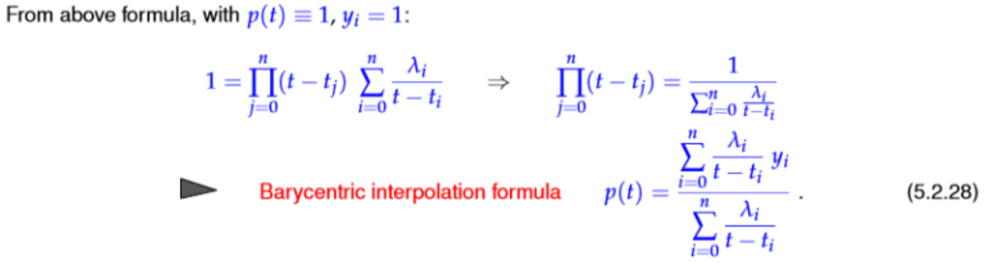
$$p(t) = \sum_{i=0}^{n} L_i(t)\, y_i = \sum_{i=0}^{n} \prod_{\substack{j=0\\ j\neq i}}^{n} \frac{t-t_j}{t_i-t_j}\, y_i = \sum_{i=0}^{n} \lambda_i \prod_{\substack{j=0\\ j\neq i}}^{n} (t-t_j)\, y_i = \prod_{j=0}^{n}(t-t_j) \cdot \sum_{i=0}^{n} \frac{\lambda_i}{t-t_i}\, y_i\,.$$

where $$\lambda_i = \frac{1}{(t_i-t_0)\cdots(t_i-t_{i-1})(t_i-t_{i+1})\cdots(t_i-t_n)}, \; i = 0,\dots,n.$$

=> precompute Lambda and sum or product

Laufzeit: without precomputing: Sum n, product n, evaluating all t_s N => $O(Nn^2)$

with precomputing: $O(nN)$



From above formula, with $p(t) \equiv 1$, $y_i = 1$:

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

▶ Barycentric interpolation formula $$p(t) = \frac{\sum_{i=0}^{n} \frac{\lambda_i}{t-t_i}\, y_i}{\sum_{i=0}^{n} \frac{\lambda_i}{t-t_i}}\,. \qquad (5.2.28)$$

this works because we know that p has to be 1 where 1 is 1

partial Lagrange interpolant

Aitken-Neville scheme: $k < l$. ***Good for single evaluation***

$p_{k,l}$
$= unique\ polynomial\ of\ degree\ (l - k)\ through\ the\ known\ points\ (t,y)_k \dots (t,y)_l$

First, set polynomials through just the k-th point:

$p_{k,k}(x) = y_k$

From these, derive interpolating polynomials through more points:

$$p_{k,l}(x) = \frac{1}{t_l - t_k}\left((x-t_k)p_{k+1,l}(x) - (x-t_l)p_{k,l-1}(x)\right)$$

so we weigh the polynomial in the interval to the right based on how much to the right x is, and the left polynomial based on how much to the left x is. (Assuming $t_k < x < t_l$, this will result in an addition. If x is not in the interval, then what?)

Dividing the whole thing to rescale it back to normal

Extrapolation to zero

same as interpolation but with x outside the interval.

Lagrangian

works well if function is even: $\phi(t) = \phi(-t)$ and $\phi\ behaves\ nicely\ around\ h$

Given: smooth function f in procedural form

Sought: approximation of f'

(unfinished. view Endterm Recap)