ETH Lecture 401-0663-00L Numerical Methods for CSE

Numerical Methods for Computational Science and Engineering

Prof. R. Hiptmair, SAM, ETH Zurich

(with contributions from Prof. P. Arbenz and Dr. V. Gradinaru)

Autumn Term 2016

(C) Seminar für Angewandte Mathematik, ETH Zürich

URL: http://www.sam.math.ethz.ch/~hiptmair/tmp/NumCSE/NumCSE16.pdf

V. Data Interpolation and Fitting in 1D

Given: data points

$(t_i, y_i) \in \mathbb{R}^2$, $i = 0, \ldots, n$

t_i: nodes, y_i: values

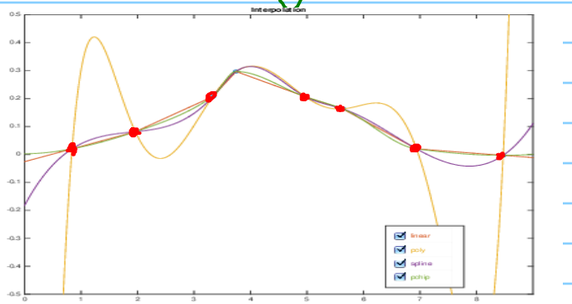


Fig. 165

Seek interpolant, a function $\in C^0(I)$

$f : I \subset \mathbb{R} \to \mathbb{R}$: $f(t_i) = y_i$ [interpolation condition] I.C.

$t_i \in I$

Want $f \in S \subset C^0(I)$ [admissible function]

More specifically: $S \subset C^0(I) \hat{=} m+1$-dim. subspace $(*)$, $m \in \mathbb{N}$

Application: Reconstruction of constitutive relations from measurements

Examples: t and y could be

t	y
voltage U	current I
pressure p	density ρ
magnetic field H	magnetic flux B
...	...

Known: several accurate $(*)$ measurements

(t_i, y_i), $i = 1, \ldots, m$

↳ justifies I.C.

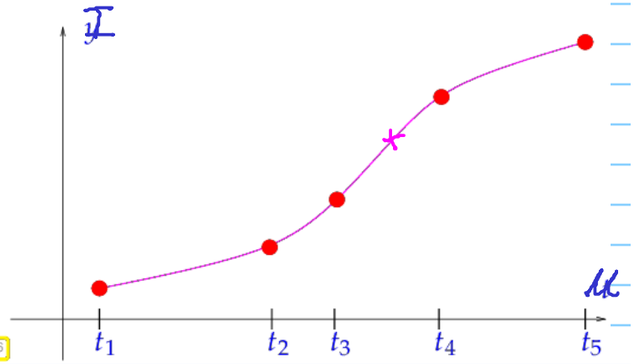


Fig. 166

continuous functions $I \to \mathbb{R}$, $b_j = b_j(t)$

$(*) \Rightarrow S = \mathrm{Span}\{b_0, \ldots, b_m\}$ [basis]

$\Rightarrow \exists c_j \in \mathbb{R}$: $f = \sum_{j=0}^{m} \alpha_j b_j$

(2)

Example: Piecewise linear interpolation

Piecewise linear interpolation

= connect data points (t_i, y_i), $i = 0, \ldots, n$, $t_{i-1} < t_i$, by line segments

➢ interpolating polygon

Piecewise linear interpolant of data ▷

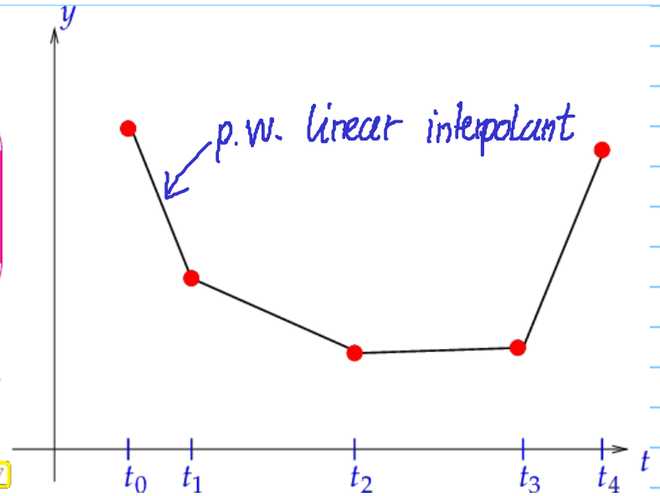


Fig. 167

$I = [t_0, t_n]$

Space S:

$S := \{ f \in C^0(I) : f(t) = \gamma_i t + \beta_i, \gamma_i, \beta_i \in \mathbb{R}, \text{ in } [t_{i-1}, t_i] \}$

$\dim S = n+1$, because $n+1$ values y_i can be imposed freely

Basis functions for p.w. linear interp.: [clearly linearly independent]

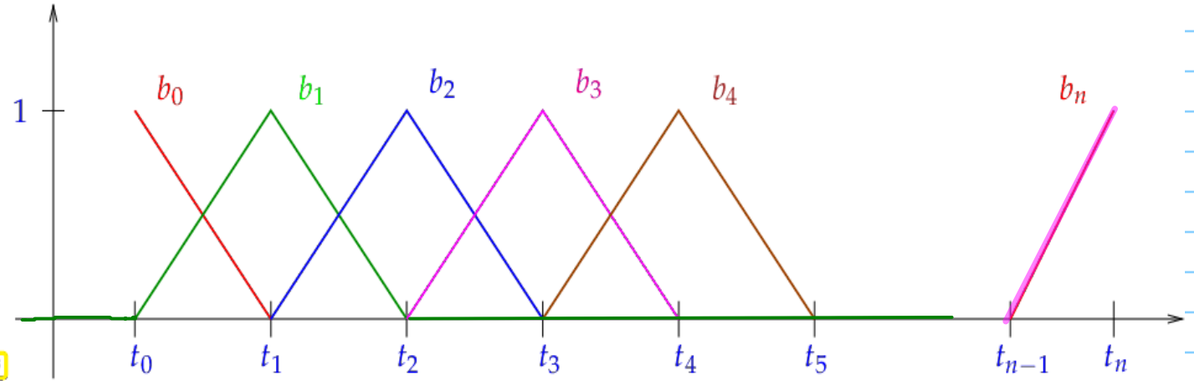


Fig. 168

$$b_0(t) = \begin{cases} 1 - \frac{t-t_0}{t_1-t_0} & \text{for } t_0 \le t < t_1 , \\ 0 & \text{for } t \ge t_1 . \end{cases}$$

$$b_j(t) = \begin{cases} 1 - \frac{t_j - t}{t_j - t_{j-1}} & \text{for } t_{j-1} \le t < t_j , \\ 1 - \frac{t-t_j}{t_{j+1}-t_j} & \text{for } t_j \le t < t_{j+1} , \\ 0 & \text{elsewhere in } [t_0, t_n] . \end{cases} \quad , \quad j = 1, \ldots, n-1 , \qquad (5.1.9)$$

$$b_n(t) = \begin{cases} 1 - \frac{t_n - t}{t_n - t_{n-1}} & \text{for } t_{n-1} \le t < t_n , \\ 0 & \text{for } t < t_{n-1} . \end{cases}$$

satisfy $b_j(t_i) = \delta_{ij}$ (*)

▷ Interpolant in S: $f = \sum_{j=0}^{n} y_j b_j$ by (*)

A basis satisfying (*) ≙ cardinal basis

Note: S and b_j may depend on nodes

Note: - Infinitely many possibilities to choose a basis
- but cardinal basis unique in this example

Back to general setting:

(5.1.1) & (5.1.7) $\Rightarrow$ $f(t_i) = \sum_{j=0}^{m} \alpha_j b_j(t_i) = y_i\,, \quad i = 0,\ldots,n\,,$ (5.1.12)

I.C. basis repr.

$$\Updownarrow$$

$$\mathbf{A}\mathbf{c} := \begin{bmatrix} b_0(t_0) & \ldots & b_m(t_0) \\ \vdots & & \vdots \\ b_0(t_n) & \ldots & b_m(t_n) \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \vdots \\ \alpha_m \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix} =: \mathbf{y} \,. \qquad (5.1.13)$$

This is an $(m+1) \times (n+1)$ linear system of equations !

Necessary condition for existence & uniqueness of interpolant for *any* y_i : $m = n$

Note : $\begin{cases} \mathbb{R}^{n+1} \longrightarrow C^0(I) \\ y \longrightarrow f \text{ (interpolant)} \end{cases}$ is linear

independent of basis $\rightarrow$ $f := \sum_{j=0}^{n} \underbrace{(A^{-1}y)_j}_{=\alpha_j} b_j$ (assumed to exists)

- If $\{b_0,\ldots,b_n\}$ = cardinal basis $\Rightarrow$ $A = I$
- Whether A is regular, depends on – nodes t_i – on S

(regular: independent of the choice of basis !)

3.2. (Global) Polynomial Interpolation

3.2.1. Polynomials

Notation: Vector space of the polynomials of degree $\leq k$, $k \in \mathbb{N}$:

$$\mathcal{P}_k := \{t \mapsto \alpha_k t^k + \alpha_{k-1} t^{k-1} + \cdots + \alpha_1 t + \alpha_0 \,|\, \alpha_j \in \mathbb{R}\}\,. \qquad (5.2.1)$$

leading coefficient

monomial basis $\{t \to t^\ell\}_{\ell=0}^{k}$

$\dim \mathcal{P}_k = k+1$

Advantages of polynomials

(i) easily derived & integrated

(ii) easily evaluated : Horner scheme

$p(t) = t(\cdots t(\alpha_k t + \alpha_{k-1}) + \alpha_{k-2}) + \ldots + \alpha_1) + \alpha_0$

C++-code 5.2.7: Horner scheme (vectorized version)

```
/* Efficient evaluation of a polynomial in monomial representation
 * using the Horner scheme \eqref{intp:Horner}
 * IN: p = vector of \com{monomial coefficients}, length = degree + 1
 *     x = vector of evaluation points  <-> t
 * OUT: y = polynomial evaluated at x */
void horner(const VectorXd& p, const VectorXd& x, VectorXd& y) {
  const VectorXd::Index n = x.size();
  y.resize(n); y = p(0)*VectorXd::Ones(n);      -> vectorized version
  for (unsigned i = 1; i < p.size(); ++i) {
    y = x.cwiseProduct(y) + p(i)*VectorXd::Ones(n);
  }
}
/* SAM_LISTING_END_0 */
```

[Convention : leading coefficient $\leftrightarrow$ $p(0)$]

(4)

Effort = $O(k)$

(iii) Polynomials can approximate functions well (Taylor sums!)

3.2.2. Theory

Lagrange polynomial interpolation problem

Given the simple nodes $t_0, \ldots, t_n$, $n \in \mathbb{N}$, $-\infty < t_0 < t_1 < \cdots < t_n < \infty$ and the values $y_0, \ldots, y_n \in \mathbb{R}$ compute $p \in \mathcal{P}_n$ such that

$$p(t_j) = y_j \quad \text{for} \quad j = 0, \ldots, n\,. \tag{5.2.9}$$

$\Leftrightarrow$ LSE with $(n+1)\times(n+1)$ coefficient matrix

A cardinal basis for $\mathcal{P}_n$: Lagrange polynomials

$$L_i(t) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{(t-t_j)}{(t_i-t_j)} \in \mathcal{P}_n : \quad L_i(t_j) = \delta_{ij}$$

$j, i = 0, \ldots, n$

▷ Existence ($\Rightarrow$ uniqueness) of interpolant $\forall y_j$

Remark: Hermite interpolation:

Given $t_0 < t_1 < \cdots < t_n$, $y_j, c_j \in \mathbb{R}$ find $p \in \mathcal{P}_{2n+1}$:

$p(t_j) = y_j$ (I.C.) & $p'(t_j) = c_j$, $j = 0, \ldots, n$

3.2.3. Algorithms

An "interpolant class"

C++-code 5.1.18: C++ class representing an interpolant in 1D

```
class Interpolant {
  private:
    // Various internal data describing f
    // Can be the coefficients of a basis representation (5.1.7)
  public:
    // Constructor: computation of coefficients c_j of representation
       (5.1.7)
    Interpolant(const vector<double>& t, const vector<double>& y);
    // Evaluation operator for interpolant f
    double operator() (double t) const;;
};
```

→ returns $f(t)$

→ required to be very efficient!

→ can require much effort

3.2.3.1 Multiple evaluations

C++-code 5.2.26: Polynomial Interpolation

```
class PolyInterp {
  private:
    // various internal data describing p
    Eigen::VectorXd t;
  public:
    // Constructors taking node vector (t_0,...,t_n) as argument
    PolyInterp(const Eigen::VectorXd &_t);
    template <typename SeqContainer>
             PolyInterp(const SeqContainer &v);
    // Evaluation operator for data (y_0,...,y_n); computes
    // p(x_k) for x_k's passed in x , K = 1,...,N
    Eigen::VectorXd eval(const Eigen::VectorXd &y,
                         const Eigen::VectorXd &x) const;
};
```

} must be very efficient

⑤

Approach: $p(x_k) = \sum_{j=0}^{n} y_j L_j(x_k)$, $k = 1, \ldots, N$

Cost $= O(Nn^2)$ for $n, N \to \infty$

Idea: Precompute (parts of the L_i's) in constructor

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

where $\lambda_i = \dfrac{1}{(t_i - t_0)\cdots(t_i - t_{i-1})(t_i - t_{i+1})\cdots(t_i - t_n)}$, $i = 0, \ldots, n$. ← Zero factor dropped

Observe: $y_i = 1 \Rightarrow p \equiv 1$

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

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

C++-code 5.2.32: Interpolation class: multiple point evaluations

```
template <typename NODESCALAR>
template <typename RESVEC, typename DATAVEC>
RESVEC BarycPolyInterp<NODESCALAR>::eval
  (const DATAVEC &y, const nodeVec_t &x) const {
  const idx_t N = x.size(); // No. of evaluation points
  RESVEC p(N); // Ouput vector
  // Compute quotient of weighted sums of λ_i/(t-t_i), effort O(n)
  for (unsigned i = 0; i < N; ++i) {
    nodeVec_t z = (x(i)*nodeVec_t::Ones(n) - t); ← vector of [(x_k - t_i)]_i

    // check if we want to evaluate at a node <-> avoid division by zero
    NODESCALAR* ptr = std::find(z.data(), z.data() + n,
      NODESCALAR(0)); //
    if (ptr != z.data() + n) { // if ptr = z.data + n = z.end no zero was found
      p(i) = y(ptr - z.data()); // ptr - z.data gives the position of the zero
    }
    else {
      const nodeVec_t mu = lambda.cwiseQuotient(z); → compute weights (5.2.28)
      p(i) = (mu.cwiseProduct(y)).sum()/mu.sum();
    }
  } // end for
  return p;
}
```

Cost $= O(Nn)$ for $n, N \to \infty$

↳ Operations on vectors of length n

Note: Cost (Compute λ_i's) $= O(n^2)$

3.2.3.2 Single evaluation

nodes t_i ↓ values y_i ↓ evaluation point ↓

```
double eval(const VectorXd &t, const VectorXd &y, double x)
```

▷ Aitken-Neville scheme

"partial Lagrange interpolant": $p_{k,\ell} \in \mathcal{P}_{\ell-k} : p(t_j) = y_j$, $j = k, \dots, \ell$

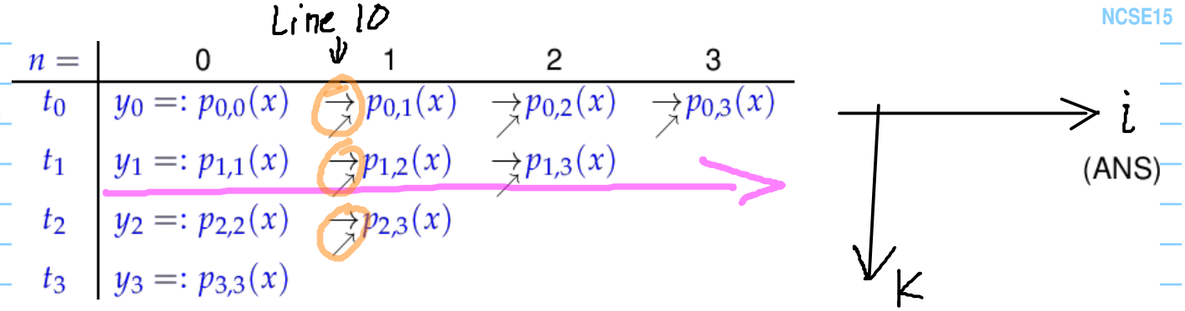
▷ recursion:

$$p_{k,k} \equiv y_k$$

$$p_{k,\ell}(x) = \underbrace{\frac{(x-t_k)p_{k+1,\ell}(x) - (x-t_\ell)p_{k,\ell-1}(x)}{t_\ell - t_k}}_{\in \mathcal{P}_{\ell-k}}$$

$$= \begin{cases} y_k & \text{at } x = t_k \\ y_\ell & x = t_\ell \\ y_i & x = t_i \end{cases}, \quad k < i < \ell$$

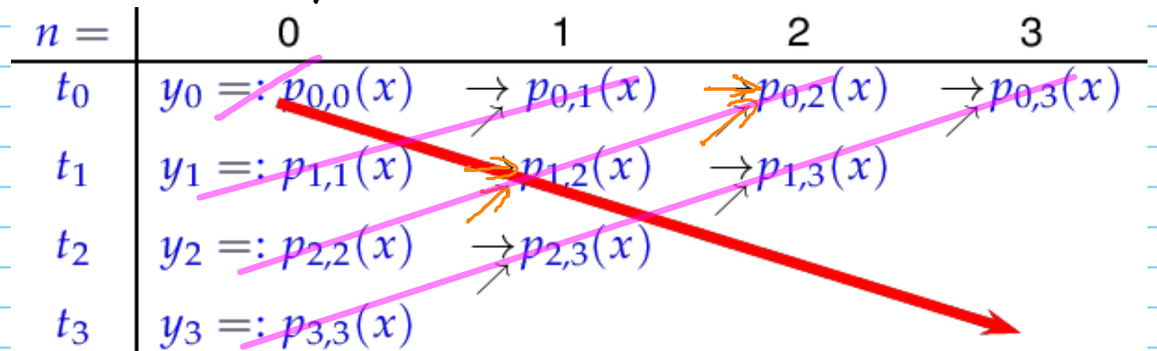
Note: Lagr. Intp. $p = p_{0,n}$



C++-code 5.2.35: Aitken-Neville algorithm

```
// Aitken-Neville algorithm for evaluation of interpolating polynomial
// IN: t, y:  (vectors of) interpolation data points
// x:  (single) evaluation point
// OUT: value of interpolant in x
double ANipoleval(const VectorXd& t, VectorXd y, const double x) {
  for (int i = 0; i < y.size(); ++i) {
    for (int k = i - 1; k >= 0; --k) {
      // Recursion (5.2.34)
      y(k) = y(k + 1) + (y(k + 1) - y(k))*(x - t(i))/(t(i) - t(k));
    }
  }
  return y(0);
}
```

A.N. scheme very useful for simultaneous evaluations and data updates



C++-code 5.2.37: Single point evaluation with data updates

```
class PolyEval {
  private:
    // evaluation point and various internal data describing the
       polynomials
  public:
    // Constructor taking the evaluation point as argument
    PolyEval(double x);
    // Add another data point and update internal information
    void addPoint(t,y);
    // Value of current interpolating polynomial at x
    double eval(void) const;
};
```

3.2.3.3. Extrapolation to Zero

Compute: $\lim_{h\to 0} \Psi(h)$ without evaluating Ψ for small $|h|$ [because of numerical instability]

Example: Ψ = symmetric difference quotient

$\Psi(h) = \frac{1}{2h}(f(x+h) - f(x-h))$: cancellation for $h \approx 0$

$f(x) = \arctan(x)$

h	Relative error
2^{-1}	0.20786640808609
2^{-6}	0.00773341103991
2^{-11}	0.00024299312415
2^{-16}	0.00000759482296
2^{-21}	0.00000023712637
2^{-26}	0.00000001020730
2^{-31}	0.00000005960464
2^{-36}	0.00000679016113

$f(x) = \sqrt{x}$

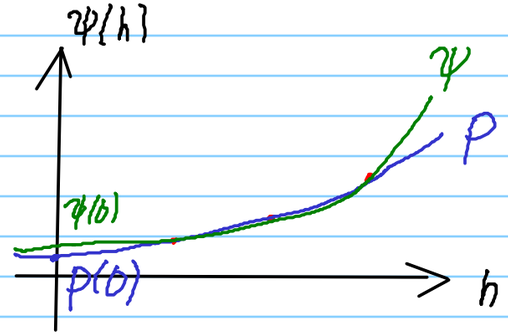
h	Relative error
2^{-1}	0.09340033543136
2^{-6}	0.00352613693103
2^{-11}	0.00011094838842
2^{-16}	0.00000346787667
2^{-21}	0.00000010812198
2^{-26}	0.00000001923506
2^{-31}	0.00000001202188
2^{-36}	0.00000198842224

$f(x) = \exp(x)$

h	Relative error
2^{-1}	0.29744254140026
2^{-6}	0.00785334954789
2^{-11}	0.00024418036620
2^{-16}	0.00000762943394
2^{-21}	0.00000023835113
2^{-26}	0.00000000429331
2^{-31}	0.00000012467100
2^{-36}	0.00000495453865

Idea: computing inaccessible limit by extrapolation to zero

① Pick $h_0, \ldots, h_n$ for which ψ can be evaluated "safely".
② evaluation of $\psi(h_i)$ for different h_i, $i = 0, \ldots, n$, $|t_i| > 0$.
③ $\psi(0) \approx p(0)$ with interpolating polynomial $p \in \mathcal{P}_n$, $p(h_i) = \psi(h_i)$.




```
auto f = [](double x) {return std::atan(x);}
diffex(f,1.1,0.5)
```

Degree	Relative error
0	0.04262829970946
1	0.02044767428982
2	0.00051308519253
3	0.00004087236665
4	0.00000048930018
5	0.00000000746031
6	0.00000000001224

```
auto g = [](double x) {return std::sqrt(x);}
diffex(g,1.1,0.5)
```

Degree	Relative error
0	0.02849215135713
1	0.01527790811946
2	0.00061205284652
3	0.00004936258481
4	0.00000067201034
5	0.00000001253250
6	0.00000000004816
7	0.00000000000021

```
auto h = [](double x) {return std::exp(x);}
diffex(h,1.1,0.5)
```

Degree	Relative error
0	0.04219061098749
1	0.02129207652215
2	0.00011487434095
3	0.00000825582406
4	0.00000000589624
5	0.00000000009546
6	0.00000000000002

$h_0 = 1/2$, $h_1 = 1/4$, $h_2 = 1/8$ .....

C++-code 5.2.46: Numerical differentiation by extrapolation to zero

```
// Extrapolation based numerical differentation
// with a posteriori error control
// f:  handle of a function defined in a neighbourhood of x ∈ ℝ
// x:  point at which approximate derivative is desired
// h0:  initial distance from x
// rtol:  relative target tolerance, atol:  absolute tolerance
template <class Function>
double diffex(Function& f, const double x, const double h0, const
    double rtol, const double atol) {
  const unsigned nit = 10; // Maximum depth of extrapolation
  VectorXd h(nit); h(0) = h0; // Widths of difference quotients
  VectorXd y(nit); // Approximations returned by difference quotients
  y(0) = (f(x + h0) - f(x - h0))/(2*h0); // Widest difference quotients

  // using Aitken-Neville scheme with x = 0, see Code 5.2.35
  for (unsigned i = 1; i < nit; ++i) {
    // create data points for extrapolation
    h(i) = h(i-1)/2; // Next width half a big
    y(i) = ( f(x + h(i)) - f(x - h(i)) )/h(i - 1);
    // Aitken-Neville update
    for (int k = i - 1; k >= 0; --k)
      y(k) = y(k+1) - (y(k+1)-y(k))*h(i)/(h(i)-h(k));
    // termination of extrapolation when desired tolerance is reached
    const double errest = std::abs(y(1)-y(0)); // error indicator
    if ( errest < rtol*std::abs(y(0)) || errest < atol ) //
      break;
  }
  return y(1);
}
```

data point loop

$n=$	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	$y_3 =: p_{3,3}(x)$			

values of polynomial interpolants

← Compare $p_{0,n}(0)$ and $p_{1,n}(0)$ [both approximations of $\Psi(0)$] to decide when to stop. a posteriori error control!

∇ Works well for: • Ψ smooth
• $\Psi(h) = \Psi(-h)$

(9)

3.2.3.4. Newton basis & divided difference

C++code 5.2.47: Polynomial evaluation

```
class PolyEval {
  private:
    // evaluation point and various internal data describing the
       polynomials
  public:
    // Idle Constructor
    PolyEval();
    // Add another data point and update internal information
    void addPoint(t,y);
    // Evaluation of current interpolating polynomial at x
    Eigen::VectorXd operator () (const Eigen::VectorXd &x) const;
};
```

double

Note: If data point (t_{n+1}, y_{n+1}) added

$\Rightarrow$ No change of $a_0, \ldots, a_n$!

Efficient point evaluation for Newton representation

$$p(t) = \sum_{j=0}^{n} a_j N_j(t)$$

$$= a_0 + (t-t_0)\{a_1 + (t-t_1)(a_2 + \ldots\ldots + a_n))\ldots)$$

"Update friendly" Newton basis of $\mathcal{P}_n$

$$N_0(t) \equiv 1\,, \quad N_k(t) = \prod_{i=0}^{k-1} (t-t_i) \in \mathcal{P}_k$$

$$\Rightarrow N_k(t_j) = 0\,, \quad j = 0,\ldots,k-1 \qquad k = 1,\ldots,n$$

+ interpolation condition

"Interpolation matrix"

$$\begin{bmatrix} N_0(t_0) & \cdots & N_n(t_0) \\ \vdots & & \vdots \\ N_0(t_n) & \cdots & N_n(t_n) \end{bmatrix}$$

$$a_j \in \mathbb{R}: \quad a_0 N_0(t_j) + a_1 N_1(t_j) + \cdots + a_n N_n(t_j) = y_j, \quad j = 0,\ldots,n.$$

$\Leftrightarrow$ triangular linear system : solve by forward subst.

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 1 & (t_1-t_0) & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ 1 & (t_n-t_0) & \cdots & \prod_{i=0}^{n-1}(t_n-t_i) \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

C++-code 5.2.58: Divided differences evaluation by modified Horner scheme

```
// Evaluation of polynomial in Newton basis (divided differences)
// IN: t = nodes (mutually different)
// y = values in t
// x = evaluation points (as Eigen::Vector)  --> vectorized version
// OUT: p = values in x */
void evaldivdiff(const VectorXd& t, const VectorXd& y, const
    VectorXd& x, VectorXd& p) {
  const unsigned n = y.size() - 1;

  // get Newton coefficients of polynomial (non in-situ
     implementation!)
  VectorXd coeffs; divdiff(t, y, coeffs);

  // evaluate
  VectorXd ones = VectorXd::Ones(x.size());
  p = coeffs(n)*ones;
  for (int j = n - 1; j >= 0; --j) {
    p = (x - t(j)*ones).cwiseProduct(p) + coeffs(j)*ones;
  }
}
```

"Horner-like" evaluation

(10)

Define : $a_{\ell,m} \hat{=}$ leading coeff. of $p_{\ell,m}$

$p_{k,k}(x) \equiv y_k$ ("constant polynomial"), $k = 0,\ldots,n$, $\Rightarrow$ $a_{k,k} = y_k$

$$p_{k,\ell}(x) = \frac{(x-t_k)p_{k+1,\ell}(x) - (x-t_\ell)p_{k,\ell-1}(x)}{t_\ell - t_k} \qquad (5.2.34)$$

$$= p_{k+1,\ell}(x) + \frac{x-t_\ell}{t_\ell - t_k}(p_{k+1,\ell}(x) - p_{k,\ell-1}(x)), \quad 0 \le k \le \ell \le n,$$

▷ recursion for leading coeffs

$$a_{k,\ell} = \frac{a_{k+1,\ell} - a_{k,\ell-1}}{t_\ell - t_k} \qquad (*)$$

↑ highest power $x^{k-\ell}$

Note: leading coefficient of $N_k = 1$

$a_{0,m} = a_m$

Divided difference scheme

$$\begin{array}{c|ccccc} t_0 & y[t_0] & \overset{(*)}{>} & y[t_0,t_1] & & \\ & & & > & y[t_0,t_1,t_2] & \\ t_1 & y[t_1] & & > & & y[t_0,t_1,t_2,t_3], \\ & & & y[t_1,t_2] & > & \\ t_2 & y[t_2] & & > & y[t_1,t_2,t_3] & \\ & & & y[t_2,t_3] & & \\ t_3 & y[t_3] & & & & \end{array} \qquad (5.2.54)$$

$y[t_{k,\ldots},t_\ell] := a_{k,\ell}$

C++-code 5.2.55: Divided differences, recursive implementation, in situ computation

```
// IN: t = node set (mutually different)
// y = nodal values
// OUT: y = coefficients of polynomial in Newton basis
void divdiff(const VectorXd& t, VectorXd& y) {
  const unsigned n = y.size() - 1;
  // Follow scheme (5.2.54), recursion (5.2.51)
  for (unsigned l = 0; l < n; ++l)
    for (unsigned j = l; j < n; ++j)
      y(j+1) = (y(j+1)-y(l))/(t(j+1)-t(l));
}
```

5.6. Trigonometric Interpolation

Measuring 1-periodic (time/angle-dependent) quantity

$\Rightarrow$ 1-periodic data (t_i, y_i), $i = 0,\dots,n$, $0 \le t_i < 1$

Want: 1-periodic interpolant $f \in S \subset C^0(\mathbb{R})$, $f(t+1) = f(t)$ $\forall t \in \mathbb{R}$

5.6.1. Trigonometric Polynomials

Definition 5.6.3. Space of trigonometric polynomials

The vector space of 1-periodic trigonometric polynomials of degree $2n$, $n \in \mathbb{N}$, is given by

$$S = \mathcal{P}_{2n}^{T} := \operatorname{Span}\{t \mapsto \cos(2\pi jt), t \mapsto \sin(2\pi jt)\}_{j=0}^{n} \subset C^\infty(\mathbb{R}) .$$

$\hookrightarrow \dim \mathcal{P}_{2n}^{T} = 2n+1$

$$q(t) = \alpha_0 + \sum_{j=1}^{n} \alpha_j \cos(2\pi jt) + \beta_j \sin(2\pi jt) \quad \in \mathcal{P}_{2n}^{T} \quad , \alpha_j, \beta_j \in \mathbb{R}$$

$$= \alpha_0 + \frac{1}{2}\Big\{\sum_{j=1}^{n} (\alpha_j - i\beta_j) e^{2\pi \imath jt} + (\alpha_j + i\beta_j) e^{-2\pi \imath jt}\Big\} \qquad \text{[Euler formula!]}$$

$$= \alpha_0 + \frac{1}{2}\sum_{j=-n}^{-1} (\alpha_{-j} + \imath\beta_{-j}) e^{2\pi \imath jt} + \frac{1}{2}\sum_{j=1}^{n} (\alpha_j - \imath\beta_j) e^{2\pi \imath jt}$$

$$= e^{-2\pi \imath nt} \sum_{j=0}^{2n} \gamma_j e^{2\pi \imath jt} \quad , \quad \text{with } \gamma_j = \begin{cases} \frac{1}{2}(\alpha_{n-j} + \imath\beta_{n-j}) & \text{for } j = 0,\dots,n-1 , \\ \alpha_0 & \text{for } j = n , \\ \frac{1}{2}(\alpha_{j-n} - \imath\beta_{j-n}) & \text{for } j = n+1,\dots,2n . \end{cases} \tag{5.6.6}$$

$$p(z) := \sum_{j=0}^{2n} \gamma_j z^j$$

$$\triangleright \quad q(t) = e^{-2\pi \imath nt} p(z)\big|_{z = e^{2\pi \imath t}}$$

trig. pol. $\widehat{=}$ std. pol./scaling evaluated on $\mathbb{S}^1$

$\mathbb{S}^1 := \{z \in \mathbb{C} : |z| = 1\}$

5.6.2. Reduction to Lagrange interpolation

α_j, β_j : $q(t_k) = y_k$ $\qquad z_k := e^{2\pi \imath t_k} \in \mathbb{C}$

$\Leftrightarrow$ γ_j : $p(z_k) = e^{2\pi \imath n t_k} y_k$

Std. Lagr. interpolation in $\mathbb{C}$

$\Downarrow$

Existence & uniqueness of trig. intp.

$$\alpha_j = \tfrac{1}{2}(\gamma_{n-j} + \gamma_{n+j})$$
$$\beta_j = \tfrac{1}{2\imath}(\gamma_{n-j} - \gamma_{n+j})$$

5.6.3. Equidistant trigonometric interpolation

Now: $t_k = \frac{k}{2n+1}$, $k = 0, \ldots, 2n$

$q \hat{=}$ trig. pol. as above

$\Rightarrow$ LSE from interpolation conditions

$q(t_k) = y_k \quad k = 0, \ldots, 2n \iff p(z_k) = e^{2\pi i t_k} y_k$

$$\sum_{j=0}^{2n} \gamma_j \underbrace{\exp(2\pi i \frac{jk}{2n+1})}_{=(z_k)^j} = b_k := \exp(2\pi i \frac{nk}{2n+1}) y_k , \quad k = 0, \ldots, 2n .$$

$$\Updownarrow$$

$$\overline{\mathbf{F}}_{2n+1}\mathbf{c} = \underline{b} , \quad \mathbf{c} = [\gamma_0, \ldots, \gamma_{2n}]^\top \quad \overset{\text{Lemma 4.2.14}}{\Rightarrow} \quad \mathbf{c} = \frac{1}{2n+1}\mathbf{F}_{2n+1} \cdot \underline{b} \qquad (5.6.13)$$

$(2n+1) \times (2n+1)$ (conjugate) Fourier matrix, see (4.2.13)

$\overline{F}_{2n+1}^{-1} = \frac{1}{2n+1} F_{2n+1}$

$\hat{=}$ DFT, implement via FFT

Cost $= O(n \log n)$

Evaluation of trig. pol. at equidistant points

Compute $q(\tau_\ell)$, $\tau_\ell = \frac{\ell}{N}$, $\ell = 0, \ldots, N-1$

$N \gg n$

$$q(\tau_\ell) = e^{-2\pi i n \tau_\ell} p(w_\ell) , \quad w_\ell = e^{2\pi i \frac{\ell}{N}}$$

$$p(w_\ell) = \sum_{j=0}^{2n} \gamma_j e^{2\pi i \frac{\ell j}{N}} = \sum_{j=0}^{N} \tilde{\gamma}_j e^{2\pi i \frac{\ell j}{N}}$$

$$\tilde{\gamma}_j = \begin{cases} \gamma_j , & j = 0, \ldots, 2n \\ 0 , & j = 2n+1, \ldots, N-1 \end{cases}$$

DFT !

Cost $O(N \log N)$