Interpolation

• In interpolation or extrapolation we usually want to do the following

- We have data points (x_i, y_i), $i = 1, 2, \ldots, N$
- We want to know the y value at $x \neq x_i$

- In interpolation $x_1 < x < x_N$ and in extrapolation $x < x_1$ or $x > x_N$
- Extrapolation is dangerous; it is used e.g. in solving differential equations.

- Data set may have noise: the interpolate should go smoothly through the data set not necessarily through all points.

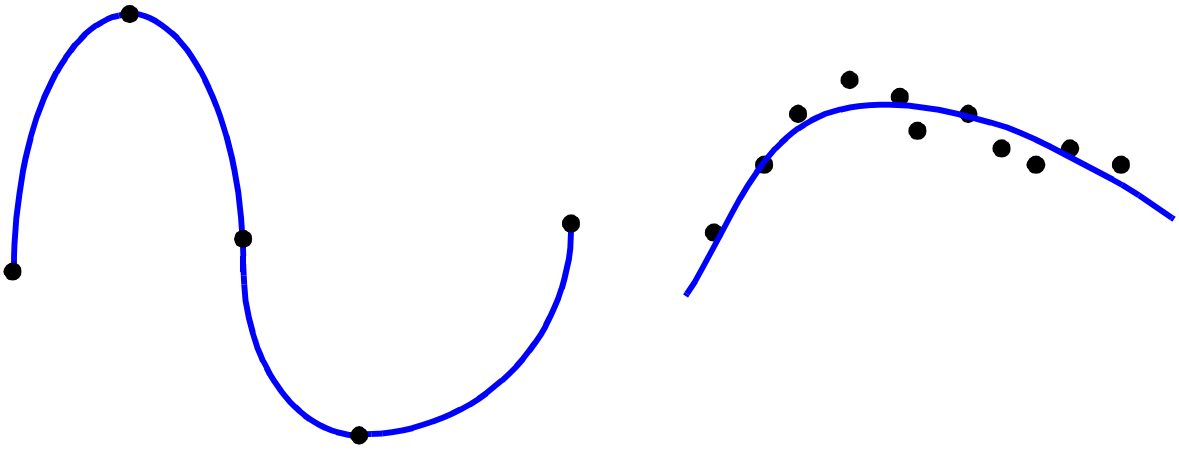
- One application is approximating (special) functions
  - In this case we have an infinite number of points available.

- In some cases interpolation is done by using a few points in the neighborhood of x.
  - This may result in noncontinuous derivative of the interpolate.
  - In spline interpolation one condition is that also the derivative is continuous.

- In polynomial interpolation one is not particularly interested in the polynomial coefficients only in its values.
  - Calculating coefficients is rather error prone.

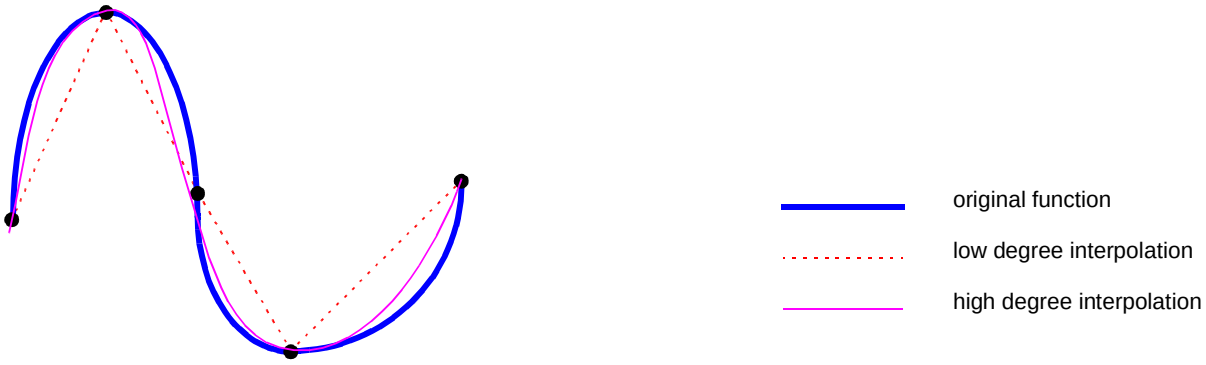
Interpolation

• Interpolation vs. curve fitting:

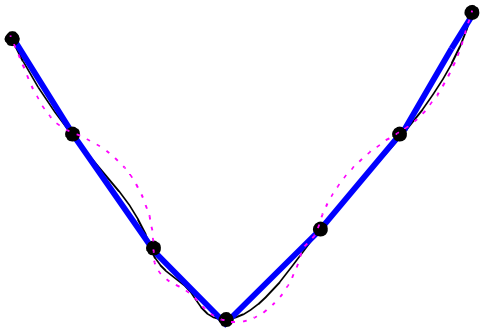


Interpolation

• Degree of interpolation is (number of points used)-1.
  - Below an example of interpolation of a smooth function



  - When the original function has sharp corners an interpolation polynomial with a lower degree may work better



Interpolation: polynomials

• We have a data set (x_i, y_i), $y_i = f(x_i)$, $i = 1, \ldots, N$

  - We have to find a polynomial $P_{N-1}(x)$ that fulfills the condition

$$P_{N-1}(x_i) = y_i, \quad i = 1, \ldots, N$$

  - It is easy to show that the polynomial is at most of degree $N-1$ and it is unique if all x_i are different.

  - A straightforward way to determine the coefficients is to use the methods we have already learned

    - Let the polynomial be of the form

$$y = c_1 + c_2 x + c_3 x^2 + \ldots + c_N x^{N-1}$$

    - From the above condition we get a group of linear equations

$$\begin{cases} c_1 + c_2 x_1 + c_3 x_1^2 + \ldots + c_N x_1^{N-1} = y_1 \\ c_1 + c_2 x_2 + c_3 x_2^2 + \ldots + c_N x_2^{N-1} = y_2 \\ \qquad\qquad \ldots \\ c_1 + c_2 x_N + c_3 x_N^2 + \ldots + c_N x_N^{N-1} = y_N \end{cases}$$

Interpolation: polynomials

• Theorem: **existence of interpolating polynomial**:

If points $x_1, x_2, \ldots, x_N$ are distinct, then for arbitrary real values $y_1, y_2, \ldots, y_N$ there is a unique polynomial P of degree $\leq N-1$ such that $P(x_i) = y_i$ for $1 \leq i \leq N$.

- Proof by induction:

Suppose we have already a polynomial P that reproduces a part of the data set: $P(x_i) = y_i$, $1 \leq i \leq k$,
(For example: this can be a constant polynomial for data point 1: $P(x) = y_1$.)

Then we add another term to P so that it will go through the point (x_{k+1}, y_{k+1}):

$$Q(x) = P(x) + c(x-x_1)(x-x_2)\ldots(x-x_k)$$

Q reproduces data points $1, 2, \ldots, k$ because P does and the added term is zero for all these points.

Now we adjust constant c so that Q reproduces the data point $k+1$:

$$Q(x_{k+1}) = P(x_{k+1}) + c(x_{k+1}-x_1)(x_{k+1}-x_2)\ldots(x_{k+1}-x_k) = y_{k+1}.$$

From this equation we can solve c because all x_i are distinct.

QED.

Interpolation: polynomial

- In matrix form:

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{N-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{N-1} \\ 1 & x_3 & x_3^2 & \ldots & x_3^{N-1} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & x_N & x_N^2 & \ldots & x_N^{N-1} \end{bmatrix} \cdot \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \ldots \\ c_N \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \ldots \\ y_N \end{bmatrix}$$

- This square matrix has a name of its own: Vandermonde matrix and its a little bad behaving:

```
>> v=1:5
v =
     1     2     3     4     5
>> m=vander(v)
m =
     1     1     1     1     1
    16     8     4     2     1
    81    27     9     3     1
   256    64    16     4     1
   625   125    25     5     1
>> rcond(m)
ans =
   2.2699e-05
```

```
>> v=1:8
v =
     1     2     3     4     5     6     7     8
>> m=vander(v)
m =
         1         1         1         1         1         1   ...
       128        64        32        16         8         4   ...
      2187       729       243        81        27         9   ...
     16384      4096      1024       256        64        16   ...
     78125     15625      3125       625       125        25   ...
    279936     46656      7776      1296       216        36   ...
    823543    117649     16807      2401       343        49   ...
   2097152    262144     32768      4096       512        64   ...
>> rcond(m)
ans =
   6.0171e-10
```

Interpolation: polynomial

- A better way is to calculate P_{N-1} by using so called Lagrange's polynomials.

  - Let the polynomials $l_1, l_2, \ldots, l_N$ be defined as

$$l_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^{N} \left(\frac{x - x_j}{x_i - x_j} \right), \quad i = 1, \ldots, N$$

  - These functions have the property

$$l_i(x_j) = \delta_{ij}$$

  - Now we can write $P_{N-1}(x)$ as

$$P_{N-1}(x) = \sum_{i=1}^{N} f(x_i) l_i(x)$$

  - It is easy to check that $P_{N-1}(x)$ goes through all the data points (x_i, y_i).
  - Because functions l_i have degree less than N it follows that P_{N-1} also has degree less than N.

Interpolation: polynomial

- Example:

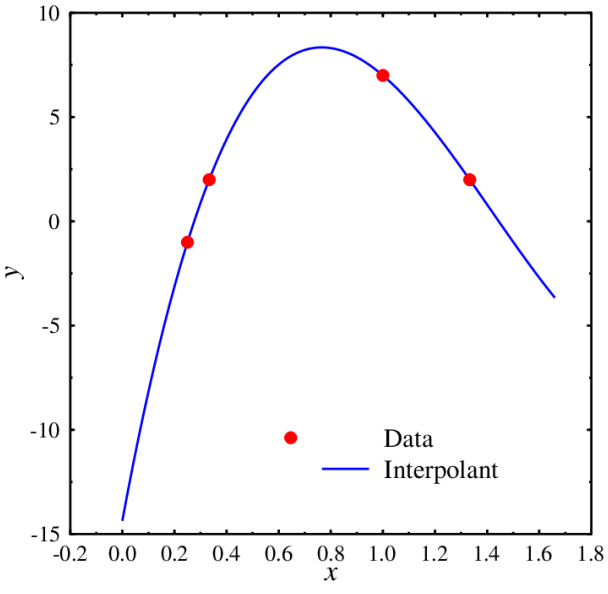
x:	$1/3$	$1/4$	1	$4/3$
y:	2	-1	7	2

$$l_1(x) = \frac{\left(x-\frac{1}{4}\right)(x-1)\left(x-\frac{4}{3}\right)}{\left(\frac{1}{3}-\frac{1}{4}\right)\left(\frac{1}{3}-1\right)\left(\frac{1}{3}-\frac{4}{3}\right)} = 18x^3 - \frac{93}{2}x^2 + \frac{69}{2}x - 6$$

$$l_2(x) = \frac{\left(x-\frac{1}{3}\right)(x-1)\left(x-\frac{4}{3}\right)}{\left(\frac{1}{4}-\frac{1}{3}\right)\left(\frac{1}{4}-1\right)\left(\frac{1}{4}-\frac{4}{3}\right)} = -\frac{192}{13}x^3 + \frac{512}{13}x^2 - \frac{1216}{39}x + \frac{256}{39}$$

$$l_3(x) = \frac{\left(x-\frac{1}{3}\right)\left(x-\frac{1}{4}\right)\left(x-\frac{4}{3}\right)}{\left(1-\frac{1}{3}\right)\left(1-\frac{1}{4}\right)\left(1-\frac{4}{3}\right)} = -6x^3 + \frac{23}{2}x^2 - \frac{31}{6}x + \frac{2}{3}$$

$$l_4(x) = \frac{\left(x-\frac{1}{3}\right)\left(x-\frac{1}{4}\right)(x-1)}{\left(\frac{4}{3}-\frac{1}{3}\right)\left(\frac{4}{3}-\frac{1}{4}\right)\left(\frac{4}{3}-1\right)} = \frac{36}{13}x^3 - \frac{57}{13}x^2 + \frac{24}{13}x - \frac{3}{13}$$



$$P_3(x) = 2l_1(x) - l_2(x) + 7l_3(x) + 2l_4(x) = \frac{186}{13}x^3 - \frac{1577}{26}x^2 + \frac{5281}{78}x - \frac{560}{39}$$

Interpolation: polynomials

- So we end up with the interpolation polynomial (let's leave the subscript out)

$$P(x) = \frac{(x-x_2)(x-x_3)\dots(x-x_N)}{(x_1-x_2)(x_1-x_3)\dots(x_1-x_N)}y_1 + \frac{(x-x_1)(x-x_3)\dots(x-x_N)}{(x_2-x_1)(x_2-x_3)\dots(x_2-x_N)}y_2 + \dots + \frac{(x-x_1)(x-x_2)\dots(x-x_{N-1})}{(x_N-x_1)(x_N-x_2)\dots(x_N-x_{N-1})}y_N$$

- The error estimation of the above polynomial can be given as follows:

  - Let $x_1, x_2, \dots, x_N$ be distinct numbers in $[a, b]$ and let's assume that f has N continuous derivatives in $[a, b]$
  - Then for each $x \in [a, b]$ $\exists \xi(x) \in (a, b)$ so that

$$f(x) = P(x) + \frac{f^{(N)}(\xi(x))}{N!}\prod_{i=1}^{N}(x-x_i) \qquad \textbf{(1)}$$

Interpolation: polynomials

- Proof:

- For $x = x_k$, $k = 1, 2, \dots, N$ $f(x_k) = P(x_k)$ and any $\xi(x_k) \in (a, b)$ fulfills **(1)**
- For $x_k \neq x$ we define function $g(t)$ as

$$g(t) = f(t) - P(t) - [f(x) - P(x)]\prod_{i=1}^{N}\frac{(t-x_i)}{(x-x_i)} \qquad \textbf{(2)}$$

- Since f has N continuous derivatives and P has all derivatives continuous and $x \neq x_k \rightarrow g(t)$ has N continuous derivatives in $[a, b]$
- For $t = x_k$

$$g(x_k) = f(x_k) - P(x_k) - [f(x) - P(x)]\prod_{i=1}^{N}\frac{(x_k-x_i)}{(x-x_i)} = 0$$

- For $t = x$

$$g(x) = f(x) - P(x) - [f(x) - P(x)]\prod_{i=1}^{N}\frac{(x-x_i)}{(x-x_i)} = 0$$

- Thus g vanishes at $N+1$ points $x, x_1, x_2, \dots, x_N$ in $[a, b]$.
- Generalized Rolle's theorem says that $\exists \xi \equiv \xi(x)$ in (a, b) for which $g^{(N)}(\xi) = 0$.
- From **(2)** we get

$$0 = g^{(N)}(\xi) = f^{(N)}(\xi) - P^{(N)}(\xi) - [f(x) - P(x)]\frac{d^N}{dt^N}\left\{\prod_{i=1}^{N}\frac{(t-x_i)}{(x-x_i)}\right\}\Bigg|_{t=\xi} \qquad \textbf{(3)}$$

Generalized Rolle's theorem:

Assume
1. $f(x)$ continuous on $[a, b]$,
2. derivatives $f^{(1)}(x), \dots, f^{(N)}(x)$ exist in (a, b),
3. $x_0, x_1, \dots, x_N \in [a, b]$,
4. $f(x_j) = 0$, for $j = 0, 1, \dots, N$.

Then $\exists c$, $a < c < b$, such that $f^{(N)}(c) = 0$.

Interpolation: polynomials

- Now P is at most of degree $N-1 \rightarrow P^{(N)} \equiv 0$
- The product term is a polynomial of degree $N \rightarrow$

$$\prod_{i=1}^{N} \frac{(t-x_i)}{(x-x_i)} = \left[\prod_{i=1}^{N} (x-x_i)\right]^{-1} t^N + O(t^{N-1}) \quad \rightarrow \quad \frac{d^N}{dt^N}\left\{\prod_{i=1}^{N} \frac{(t-x_i)}{(x-x_i)}\right\} = \frac{N!}{\prod_{i=1}^{N}(x-x_i)}$$

- **(3)** now becomes

$$0 = f^{(N)}(\xi) - [f(x) - P(x)]\frac{N!}{\prod_{i=1}^{N}(x-x_i)}$$

- And solving for $f(x)$ we get

$$f(x) = P(x) + \frac{f^{(N)}(\xi)}{N!}\prod_{i=1}^{N}(x-x_i)$$

QED.

- Note the analogy between the error formula of the Taylor series:

$$\frac{f^{(N)}(\xi)}{N!}(x-x_0)^N$$

and

$$\frac{f^{(N)}(\xi)}{N!}(x-x_1)(x-x_2)\ldots(x-x_N)$$

Interpolation: polynomials

- Example:
  - Prepare a table for function $f(x) = e^x$, $x \in [0,1]$.
  - Precision d decimals, step size h
  - What step size is needed for linear interpolation to give absolute error no more than 10^{-6}?

  - Let $x \in [0,1]$ and $x_j \le x \le x_{j+1}$
  - Error is now

$$|f(x) - P(x)| \le \left|\frac{f^{(2)}(\xi)}{2!}(x-x_j)(x-x_{j+1})\right| = \frac{|f^{(2)}(\xi)|}{2}|(x-x_j)(x-x_{j+1})| = \frac{|f^{(2)}(\xi)|}{2}|(x-jh)(x-(j+1)h)|$$

$$|f(x) - P(x)| \le \frac{1}{2}\max_{\xi\in[0,1]}|f^{(2)}(\xi)| \max_{x_j \le x \le x_{j+1}}|(x-jh)(x-(j+1)h)|$$

Maximum of $|(x-jh)(x-(j+1)h)|$ is at $x = (j+1/2)h$ with value $h^2/4$

$$\rightarrow \qquad |f(x) - P(x)| \le \frac{eh^2}{8}$$

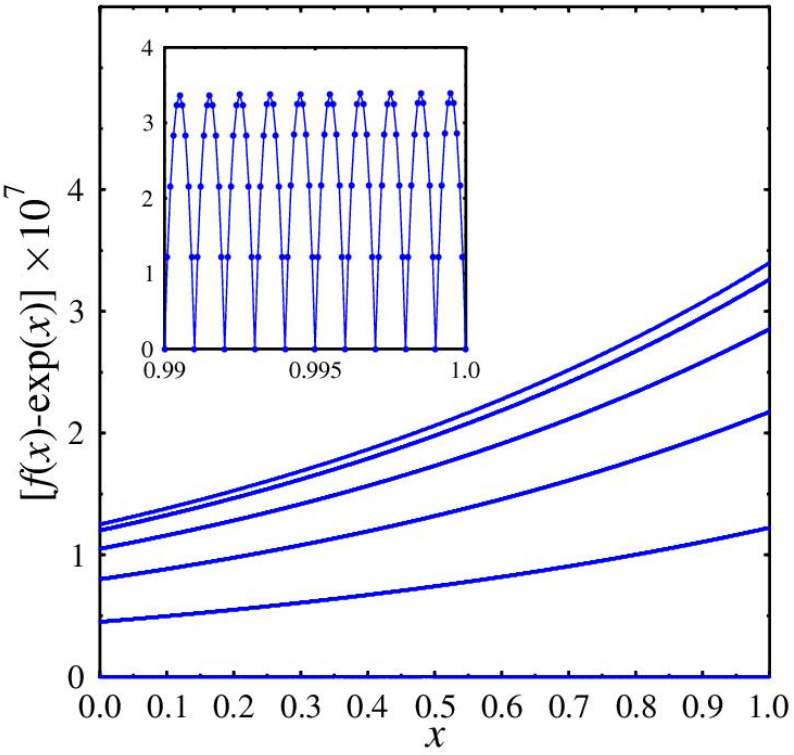
  - We want the error to be less than 10^{-6}:

$$\frac{eh^2}{8} \le 10^{-6} \Rightarrow h^2 \le \frac{8}{e}10^{-6} \Rightarrow h < 1.72\times10^{-3}$$

  - So we could choose $h = 0.001$

Interpolation: polynomials

- As a figure:



Interpolation: polynomials

- In principle one could calculate the interpolating values from the above equation we obtained:

$$P(x) = \frac{(x-x_2)(x-x_3)\ldots(x-x_N)}{(x_1-x_2)(x_1-x_3)\ldots(x_1-x_N)}y_1 + \frac{(x-x_1)(x-x_3)\ldots(x-x_N)}{(x_2-x_1)(x_2-x_3)\ldots(x_2-x_N)}y_2 + \ldots + \frac{(x-x_1)(x-x_2)\ldots(x-x_{N-1})}{(x_N-x_1)(x_N-x_2)\ldots(x_N-x_{N-1})}y_N$$

- However, there are more efficient ways to do that: so called **Neville's algorithm**:
  - Let P_1 be a zero-degree polynomial going throught the first point (x_1, y_1); i.e. $P_1 = y_1$.
  - In the same way define $P_2, P_3, \ldots, P_N$
  - Let P_{12} be a first-degree polynomial going through points (x_1, y_1) and (x_2, y_2),
  - In the same way define $P_{23}, P_{34}, \ldots, P_{(N+1)N}$
  - Going further we can define polynomials with higher degrees until we get $P_{123\ldots N}$ which is what we want.

Interpolation: polynomials

- Different P's can be written as an array with parents and children; for $N = 3$:

$$\begin{array}{lll} x_1 & y_1 = P_1 & \\ & & P_{12} \\ x_2 & y_2 = P_2 & P_{123} \\ & & P_{23} \\ x_3 & y_3 = P_3 & \end{array}$$

- Neville's algorithm fills the above array from left to right one column at a time.

- The recursion formula for the P's is

$$P_{i(i+1)\ldots(i+m)} = \frac{(x - x_{i+m})P_{i(i+1)\ldots(i+m-1)} + (x_i - x)P_{(i+1)\ldots(i+m)}}{x_i - x_{i+m}}$$

- This recursion works because the parents have the same values at points $x_{i+1} \cdots x_{i+m-1}$

Interpolation: polynomials

- Neville's algorithm is based on Aitken's lemma:

Let $P_{1n}(x)$ be a Lagrange's polynomial with degree $n-1$ which satisfies
$P_{1n}(x_i) = y_i,\ i = 1, \ldots, n$

Let $P_{2,n+1}(x)$ be a Lagrange's polynomial with degree $n-1$ which satisfies
$P_{2,n+1}(x_i) = y_i,\ i = 2, \ldots, n+1$

Now the polynomial $P_{1,n+1}(x)$ formed from the data set $(x_i, y_i),\ i = 1, \ldots, n+1$ can be obtained from the formula

$$P_{1,n+1}(x) = \frac{(x - x_1)P_{2,n+1}(x) - (x - x_{n+1})P_{1n}(x)}{x_{n+1} - x_1}$$

- This is trivially true since $P_{1,n+1}(x)$ is a polynomial of degree n satisfying $P_{1,n+1}(x_i) = y_i,\ i = 1, \ldots, n+1$

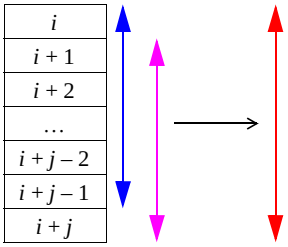
- This can be generalized to any interval i, j:

$$P_{i(i+1)\ldots(i+j)} = \frac{(x - x_{i+j})P_{i(i+1)\ldots(i+j-1)} + (x_i - x)P_{(i+1)(i+2)\ldots(i+j)}}{x_i - x_{i+j}}$$

Interpolation: polynomials

- Graphically:

$$\left.\begin{matrix} P_{i(i+1)\ldots(i+j-1)} \\ P_{(i+1)(i+2)\ldots(i+j)} \end{matrix}\right\} \Rightarrow P_{i(i+2)\ldots(i+j)}$$



- For practical calculations it is better to simplify the notation:

$$S_{ij} = P_{i(i+1)(i+2)\ldots(i+j)}$$

- Now the recursion relation becomes

$$S_{ij} = \frac{(x - x_{i+j})S_{i(j-1)} + (x_i - x)S_{(i+1)(j-1)}}{x_i - x_{i+j}} \quad \text{with } S_{i0} = y_i.$$

- For example

$$\begin{array}{lllll} y_1 = S_{10} & & & & \\ & S_{11} & & & \\ y_2 = S_{20} & & S_{12} & & \\ & S_{21} & & S_{13} & \\ y_3 = S_{30} & & S_{22} & & \\ & S_{31} & & & \\ y_4 = S_{40} & & & & \end{array}$$

Interpolation: polynomials

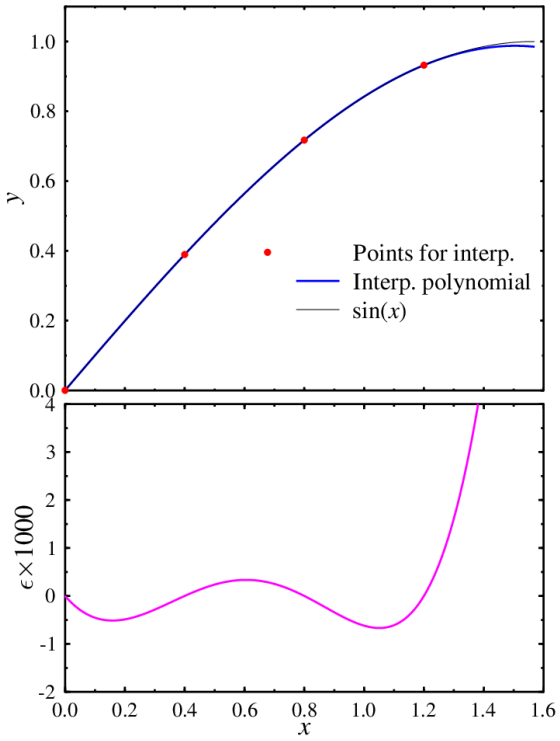
- In Fortran90 this looks like

```
program neville
  implicit none
  integer,parameter :: rk=selected_real_kind(15,100)
  real(rk),allocatable :: xa(:),ya(:),s(:,:)
  real(rk) ::  x,y,x1,x2,dx
  integer :: n,i,j,ix,ixmax
  character(len=80) :: argu
  if (iargc()/=3) then
     call getarg(0,argu)
     write(0,'(a,a,a)') 'usage: ',trim(argu),' x1 x2 dx'
     stop
  end if
  call getarg(1,argu); read(argu,*) x1 ! First x
  call getarg(2,argu); read(argu,*) x2 ! Last x
  call getarg(3,argu); read(argu,*) dx ! Step of x
  read(5,*) n                          !
  allocate(xa(n),ya(n),s(n,0:n-1)) ! Read in
  do i=1,n                             ! data (x_i,y_i)
     read(5,*) xa(i),ya(i)             !
  end do                               !
  s(1:n,0)=ya
  ixmax=(x2-x1)/dx
  do ix=0,ixmax     ! Loop over x
     x=x1+dx*ix
     do j=1,n-1     ! Neville's recursion loops to calculate S_ij
        do i=1,n-j
           s(i,j)=((x-xa(i+j))*s(i,j-1)+(xa(i)-x)*s(i+1,j-1))/(xa(i)-xa(i+j))
        end do
     end do
     write(6,*) x,s(1,n-1)
  end do
  stop
end program neville
```

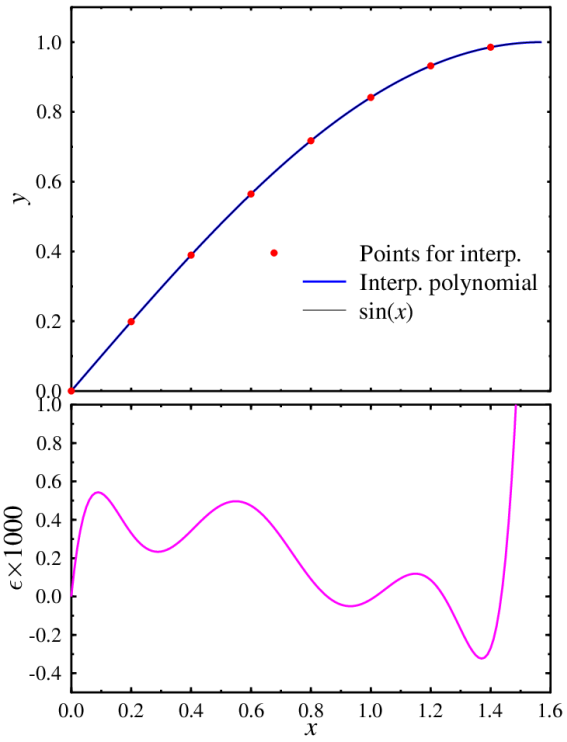
Interpolation: polynomials

- Example: $\sin(x)$

4 points

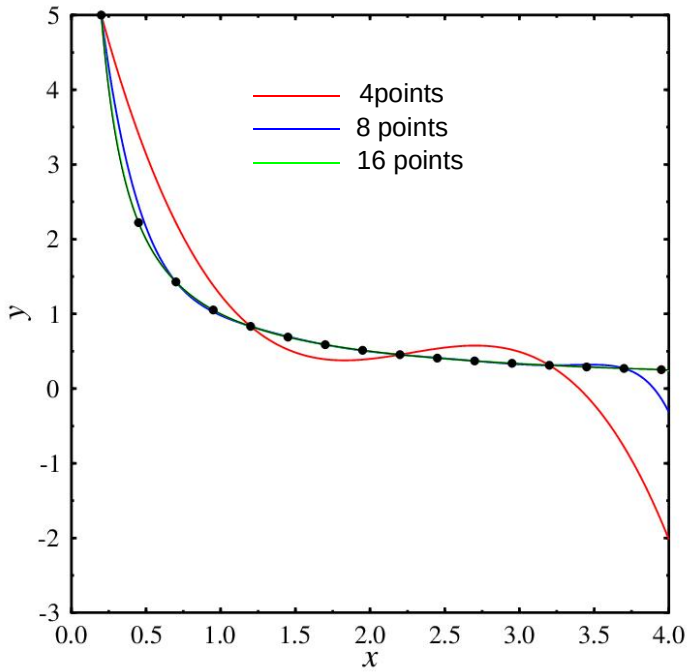


8 points



Interpolation: polynomials

- Another example $\frac{1}{x}$



Interpolation: polynomials

- In practice the polynomial is often calculated not using the previous recursion relation but using the differences $C_{m,i}$ and $D_{m,i}$:

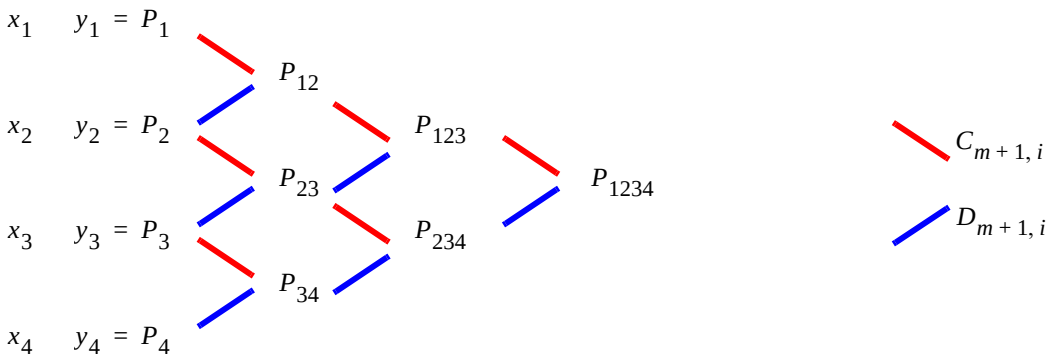
$$C_{m,i} = P_{i\ldots(i+m)} - P_{i\ldots(i+m-1)}, \qquad D_{m,i} = P_{i\ldots(i+m)} - P_{(i+1)\ldots(i+m)}$$

- Based on the recursion relation for the P's it is easy to derive the recursion relation for these differences

$$D_{m+1,i} = \frac{(x_{i+m+1} - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}}$$

$$C_{m+1,i} = \frac{(x_i - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}}$$

- On level m the corrections D and C take the polynomial one degree higher.
- Final polynomial $P_{12\ldots N}$ is thus obtained by traversing the array starting from any y_i and summing up the corrections.



Interpolation: polynomials

- An example implementation:

```
void polint(double xa[], double ya[], int n, double x, double *y, double *dy)
{
   int i,m,ns=1;
   double den,dif,dift,ho,hp,w;
   double *c,*d;
   dif=fabs(x-xa[1]);
   c=vector(1,n);
   d=vector(1,n);
   for (i=1;i<=n;i++) {
      if ( (dift=fabs(x-xa[i])) < dif) { // Find the table entry nearest to x
         ns=i;
         dif=dift;
       }
       c[i]=ya[i];
       d[i]=ya[i];
   }
   *y=ya[ns--];
   for (m=1;m<n;m++) {
      for (i=1;i<=n-m;i++) {
         ho=xa[i]-x;
         hp=xa[i+m]-x;
         w=c[i+1]-d[i];
         den=ho-hp;
         den=w/den;
         d[i]=hp*den;
         c[i]=ho*den;
       }
       *dy = 2*ns<(n-m) ? c[ns+1] : d[ns--];
       *y += *dy;
   }
free_vector(d,1,n);
free_vector(c,1,n);
}
```

Interpolation: polynomials

• Neville's algorithm does not give us the coefficients of the polynomial.
- They can be calculated by using so called divided differences.

- The proof of the theorem on existence of interpolating polynomial provides us a method for constructing it:
Newton's algorithm

- We start at zero degree polynomial:
$P_0(x) = y_0$

Note: Now indexing of the data points starts from 0: (x_i, y_i), $i = 0, 1, 2, \ldots, N$.

- Add the second term:
$P_1(x) = P_0(x) + a_1(x - x_0) = y_0 - a_1(x - x_0)$;

$P_1(x_1) = y_1 \Rightarrow a_1 = \dfrac{y_1 - y_0}{x_1 - x_0}$

- Add the third term:
$P_2(x) = P_1(x) + a_2(x - x_0)(x - x_1) = y_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1)$
By setting $P_2(x_2) = y_2$ we obtain constant a_2.

- Iteration for the kth polynomial is
$P_k(x) = P_{k-1}(x) + a_k(x - x_0)(x - x_1)\ldots(x - x_{k-1})$

- The final polynomial going through all the points is
$P_N(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \ldots + a_N(x - x_0)(x - x_1)\ldots(x - x_{N-1})$, where $a_0 = y_0$.

Interpolation: polynomials

- Example:

Find the Newton form of the interpolating polynomial for the following table of values:

x	$1/3$	$1/4$	1
y	2	-1	7

We will construct three successive polynomials p_0, p_1 and p_2. The first one is

$$p_0(x) = 2$$

The next one is given by

$$p_1(x) = p_0(x) + c(x - x_0) = 2 + c(x - 1/3)$$

Using the interpolation condition $p_1(1/4) = -1$, we obtain $c = 36$, and

$$p_1(x) = 2 + 36(x - 1/3)$$

Finally,

$$\begin{aligned} p_2(x) &= p_1(x) + c(x - x_0)(x - x_1) \\ &= 2 + 36(x - 1/3) + c(x - 1/3)(x - 1/4) \end{aligned}$$

The interpolation condition gives $p_2(1) = 7$, and thus $c = -38$.

The Newton form of the interpolating polynomial is

$$p_2(x) = 2 + 36(x - 1/3) - 38(x - 1/3)(x - 1/4)$$

Interpolation: polynomials

- The same data using the Lagrange polynomial:

Example.

Find the Lagrange form of the interpolating polynomial for the following table of values:

x	$1/3$	$1/4$	1
y	2	-1	7

The cardinal functions are:

$$l_0(x) = \frac{(x-1/4)(x-1)}{(1/3-1/4)(1/3-1)} = -18(x-1/4)(x-1)$$
$$l_1(x) = \frac{(x-1/3)(x-1)}{(1/4-1/3)(1/4-1)} = 16(x-1/3)(x-1)$$
$$l_2(x) = \frac{(x-1/3)(x-1/4)}{(1-1/3)(1-1/4)} = 2(x-1/3)(x-1/4)$$

Therefore, the interpolating polynomial in Lagrange's form is

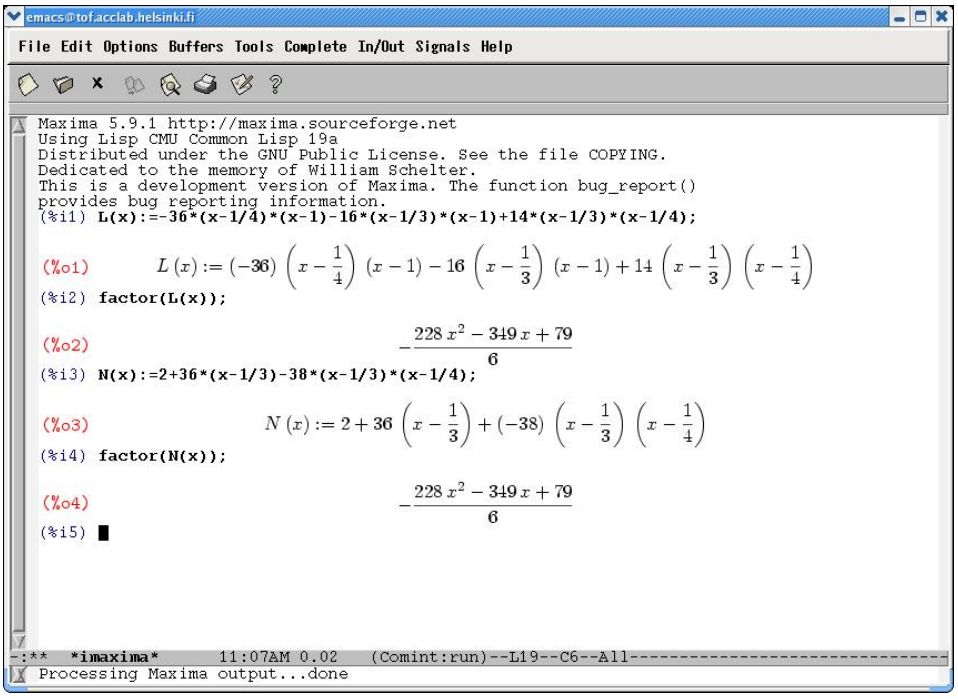
$$\begin{aligned} p_2(x) = &-36(x-1/4)(x-1) \\ &-16(x-1/3)(x-1) \\ &+14(x-1/3)(x-1/4) \end{aligned}$$

Interpolation: polynomials

- Due to the uniqueness of the interpolating polynomial the Lagrange's and Newton's polynomials are the same:

Lagrange: $P(x) = -36\left(x-\frac{1}{4}\right)(x-1) - 16\left(x-\frac{1}{3}\right)(x-1) + 14\left(x-\frac{1}{3}\right)\left(x-\frac{1}{4}\right)$

Newton: $P(x) = 2 + 36\left(x-\frac{1}{3}\right) - 38\left(x-\frac{1}{3}\right)\left(x-\frac{1}{4}\right)$



Emacs package **imaxima**: uses TeX to output **maxima** results.

Interpolation: polynomials

- An efficient way to evaluate Newton's polynomial is **nested multiplication**.

- Newton's polynomial $P(x) = a_0 + a_1(x-x_0) + a_2(x-x_0)(x-x_1) + \ldots + a_N(x-x_0)\ldots(x-x_N)$ can be written in the form

$$P(x) = a_0 + \sum_{i=1}^{N} a_i \left[\prod_{j=0}^{i-1} (x-x_j) \right]$$

- Using the common terms $(x-x_i)$ for factoring we obtain the **nested form**

$$P(x) = a_0 + (x-x_0)(a_1 + (x-x_1)(a_2 + (x-x_2))(a_3 + \ldots(x-x_{N-1})a_N))\ldots)$$

- For evaluating $P(x)$ for $x = t$ we begin from the innermost parenthesis

$$\begin{aligned} v_0 &= a_N \\ v_1 &= v_0(t-x_{N-1}) + a_{N-1} \\ v_2 &= v_1(t-x_{N-2}) + a_{N-2} \\ &\ldots \\ v_N &= v_{N-1}(t-x_0) + a_0 \end{aligned}$$

- v_N is the value of the polynomial.

Interpolation: polynomials

- An efficient way to compute the coefficients of the Newton's polynomial is based on **divided differences**.

- We have $N+1$ points $(x_i, f(x_i))\ \ i = 0, \ldots, N$

- Newton's polynomial interpolating these points is

$$P_N(x) = \sum_{i=0}^{N} a_i \left[\prod_{j=0}^{i-1} (x-x_j) \right] \qquad \left(\prod_{j=0}^{-1} (x-x_j) \text{ is interpreted as } 1.\right)$$

- Coefficients a_i do not depend on N: P_N is obtained from P_{N-1} by adding one more term without changing P_{N-1}.

- A systematic way to obtain a_i is to set x equal to points x_i one at a time:

$$\begin{cases} f(x_0) = a_0 \\ f(x_1) = a_0 + a_1(x_1 - x_0) \\ f(x_2) = a_0 + a_1(x_1 - x_0) + a_2(x_2 - x_0)(x_2 - x_1) \\ \qquad \ldots \end{cases}$$

or in compact form

$$f(x_k) = \sum_{i=0}^{k} a_i \left[\prod_{j=0}^{i-1} (x_k - x_j) \right] \qquad 0 < k < N$$

Interpolation: polynomials

- We can now see that a_k depends on the values of f at points $x_0, x_1, \ldots, x_k$; formally:

$$a_k = f[x_0, x_1, \ldots, x_k]$$

- Quantity $f[x_0, x_1, \ldots, x_k]$ is called the **divided difference** of order k for f.

Example. Determine the quantities $f[x_0]$, $f[x_0, x_1]$, and $f[x_0, x_1, x_2]$ for the following table

x	1	-4	0
$f(x)$	3	13	-23

The system of equations for the coefficients a_k:

$$\begin{cases} 3 & = a_0 \\ 13 & = a_0 + a_1(-4-1) \\ -23 & = a_0 + a_1(0-1) + a_2(0-1)(0+4) \end{cases}$$

We get

$$\begin{cases} a_0 & = 3 \\ a_1 & = [13 - a_0]/(-4-1) = -2 \\ a_2 & = [-23 - a_0 - a_1(0-1)]/(0+4) = 7 \end{cases}$$

Thus for this function, $f[1] = 3$, $f[1, -4] = -2$, and $f[1, -4, 0] = 7$.

Interpolation: polynomials

- Newton's polynomial now takes the form

$$P_N(x) = \sum_{i=0}^{N} \left\{ f[x_0, x_1, \ldots, x_i] \prod_{j=0}^{i-1} (x - x_j) \right\}$$

- Divided differences can be computed recursively:

$$f(x_k) = \sum_{i=0}^{k} a_i \left[\prod_{j=0}^{i-1} (x_k - x_j) \right] = f[x_0, \ldots, x_k] \prod_{j=0}^{k-1} (x_k - x_j) + \sum_{i=0}^{k-1} f[x_0, \ldots, x_i] \prod_{j=0}^{i-1} (x_k - x_j)$$

$$\rightarrow \quad f[x_0, \ldots, x_k] = \frac{f(x_k) - \sum_{i=0}^{k-1} f[x_0, \ldots, x_i] \prod_{j=0}^{i-1} (x_k - x_j)}{\prod_{j=0}^{k-1} (x_k - x_j)}$$

- Possible algorithm:
  1. Set $f[x_0] = f(x_0)$
  2. For $k = 1, 2, \ldots, N$ compute $f[x_0, \ldots, x_k]$ from the equation above.

- Well, there are more efficient ways to do this.

Interpolation: polynomials

- Recursive property of divided differences:

$$f[x_0, \ldots, x_k] = \frac{f[x_1, \ldots, x_k] - f[x_0, \ldots, x_{k-1}]}{x_k - x_0}$$

- Uniqueness of the polynomial: no change if points (x_i, y_i) permuted

→ The divided difference $f[x_0, \ldots, x_k]$ is invariant under permutations of arguments $x_0, \ldots, x_k$.

- This means that the recursive relation can be written as

$$f[x_i, x_{i+1}, \ldots, x_{j-1}, x_j] = \frac{f[x_{i+1}, \ldots, x_j] - f[x_i, \ldots, x_{j-1}]}{x_j - x_i}$$

- Now the divided differences can be evaluated as follows:

$$f[x_i] = f(x_i) \qquad f[x_{i+1}] = f(x_{i+1}) \qquad f[x_{i+2}] = f(x_{i+2})$$

$$f[x_i, x_{i+1}] = \frac{f[x_{i+1}] - f[x_i]}{x_{i+1} - x_i} \qquad f[x_{i+1}, x_{i+2}] = \frac{f[x_{i+2}] - f[x_{i+1}]}{x_{i+2} - x_{i+1}}$$

$$f[x_i, x_{i+1}, x_{i+2}] = \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i}$$

Interpolation: polynomials

- We can now construct a similar table for $f[\ldots]$'s as in the Neville's method:

$$\begin{array}{llll}
x_0 & f[x_0] & & \\
 & & f[x_0, x_1] & \\
x_1 & f[x_1] & & f[x_0, x_1, x_2] \\
 & & f[x_1, x_2] & & f[x_0, x_1, x_2, x_3] \\
x_2 & f[x_2] & & f[x_1, x_2, x_3] \\
 & & f[x_2, x_3] & \\
x_3 & f[x_3] & &
\end{array}$$

- To summarize the story so far:

Interpolating polynomial

$$P_N(x) = \sum_{i=0}^{N} a_i \left[\prod_{j=0}^{i-1} (x - x_j) \right] = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \ldots + a_N(x - x_0)\ldots(x - x_{N-1})$$

where

$a_k = f[x_0 \ldots x_k]$,

$$f[x_0, \ldots, x_k] = \frac{f[x_1, \ldots, x_k] - f[x_0, \ldots, x_{k-1}]}{x_k - x_0}$$

$f[x_k] = f(x_k)$.

Interpolation: polynomials

- Example:

Construct a divided differences table and write out the Newton form of the interpolating polynomial for the following table of values:

x_i	1	3/2	0	3
$f(x_i)$	3	13/4	3	5/3

Step 1.

The second column of entries is given by $f[x_i] = f(x_i)$. Thus

x	$f[\,]$	$f[\,,\,]$	$f[\,,\,,\,]$	$f[\,,\,,\,,\,]$
1	3			
		$f[x_0,x_1]$		
3/2	13/4		$f[x_0,x_1,x_2]$	
		$f[x_1,x_2]$		$f[x_0,x_1,x_2,x_3]$
0	3		$f[x_1,x_2,x_3]$	
		$f[x_2,x_3]$		
2	5/3			

Step 2.

The third column of entries is obtained using the values in the second column:

$$f[x_i,x_{i+1}] = \frac{f[x_{i+1}] - f[x_i]}{x_{i+1} - x_i}$$

For example,

$$f[x_0,x_1] = \frac{f[x_1] - f[x_0]}{x_1 - x_0} = \frac{13/4 - 3}{3/2 - 1} = \frac{1}{2}$$

x	$f[\,]$	$f[\,,\,]$	$f[\,,\,,\,]$	$f[\,,\,,\,,\,]$
1	3			
		1/2		
3/2	13/4		$f[x_0,x_1,x_2]$	
		1/6		$f[x_0,x_1,x_2,x_3]$
0	3		$f[x_1,x_2,x_3]$	
		-2/3		
2	5/3			

Interpolation: polynomials

Step 3.

The first entry in the fourth column is obtained using the values in the third column:

$$f[x_0,x_1,x_2] = \frac{f[x_1,x_2] - f[x_0,x_1]}{x_2 - x_0} = \frac{1/6 - 1/2}{0 - 1} = \frac{1}{3}$$

The table with a completed third column is

x	$f[\,]$	$f[\,,\,]$	$f[\,,\,,\,]$	$f[\,,\,,\,,\,]$
1	3			
		1/2		
3/2	13/4		1/3	
		1/6		$f[x_0,x_1,x_2,x_3]$
0	3		-5/3	
		-2/3		
2	5/3			

Interpolation: polynomials

Step 4.

The final entry in the table is

$$f[x_0,x_1,x_2,x_3] = \frac{f[x_1,x_2,x_3]-f[x_0,x_1,x_2]}{x_3-x_0} = \frac{-5/3-1/3}{2-1} = -2$$

The completed table is

x	$f[\,]$	$f[\,,\,]$	$f[\,,\,,\,]$	$f[\,,\,,\,,\,]$
1	**3**			
		1/2		
3/2	13/4		**1/3**	
		1/6		**-2**
0	3		-5/3	
		-2/3		
2	5/3			

Interpolation: polynomials

- Using the bold face values from the final table we can construct the Newton's polynomial

$$P_N(x) = \sum_{i=0}^{3}\left\{f[x_0, x_1, \ldots, x_i]\prod_{j=0}^{i-1}(x-x_j)\right\}$$

$$= f[x_0] + f[x_0, x_1](x-x_0) + f[x_0, x_1, x_2](x-x_0)(x-x_1) + f[x_0, x_1, x_2, x_3](x-x_0)(x-x_1)(x-x_2)$$

$$= \left(3 + \frac{1}{2}(x-1) + \frac{1}{3}(x-1)\left(x-\frac{3}{2}\right) - 2(x-1)\left(x-\frac{3}{2}\right)(x-0)\right) = -\frac{1}{3}(6x^3 - 16x^2 + 10x - 9)$$

```
(%i5) P(x):=3+1/2*(x-1)+1/3*(x-1)*(x-3/2)-2*(x-1)*(x-3/2)*x;
```

$$(\%o5)\qquad P(x) := 3 + \frac{1}{2}(x-1) + \frac{1}{3}(x-1)\left(x-\frac{3}{2}\right) + (-2)(x-1)\left(x-\frac{3}{2}\right)x$$

```
(%i6) factor(P(x));
```

$$(\%o6)\qquad -\frac{6x^3 - 16x^2 + 10x - 9}{3}$$

```
(%i7)
```

Interpolation: polynomials

- Algorithm that computes all the divided differences can now be constructed.

  - There is no need to store all $f[\ldots]$ because only $f[x_0]$, $f[x_0, x_1]$, ..., $f[x_0, \ldots, x_N]$ are needed.

  - Denote $a_i = f[x_0, x_1, \ldots, x_i]$ Newton's polynomial can be written as

$$P_N(x) = \sum_{i=0}^{N} a_i \prod_{j=0}^{i-1} (x - x_j)$$

  - We can use 1D array **`ai(0:N)`** to store divided differences
  - At each step the new divided difference is obtained by

$$a_i \leftarrow \frac{a_i - a_{i-1}}{x_i - x_{i-j}}$$

Interpolation: polynomials

- And in C:

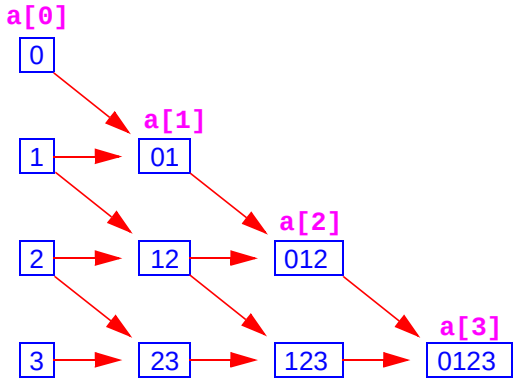
```
void Coeffs(int n, double *x, double *y, double *a) {
  int i, j;

  for (i=0;i<=n;i++) a[i]=y[i];

    for (j=1;j<=n;j++)
      for (i=n;i>=j;i--)
        a[i]=(a[i]-a[i-1])/(x[i]-x[i-j]);

  }
```

A simpler but more memory consuming way:

```
double x[n+1],y[n+1],a[n+1][n+1];
for (i=0;i<=n;i++) a[i]=y[i];
for(j=1;j<=n;j++)
   for(i=0;i<=n-j;i++)
       a[i][j]=(a[i+1][j+1]-a[i][j-1])/
                 (x[i+j]-x[i]);
```

- Evaluation of the polynomial:

```
double Evaluate(int n, double *x,
          double *a, double t) {
  int i;
  double pt;
  pt = a[n];
  for (i=n-1;i>=0;i--)
     pt=pt*(t-x[i])+a[i];
  return(pt);
}
```

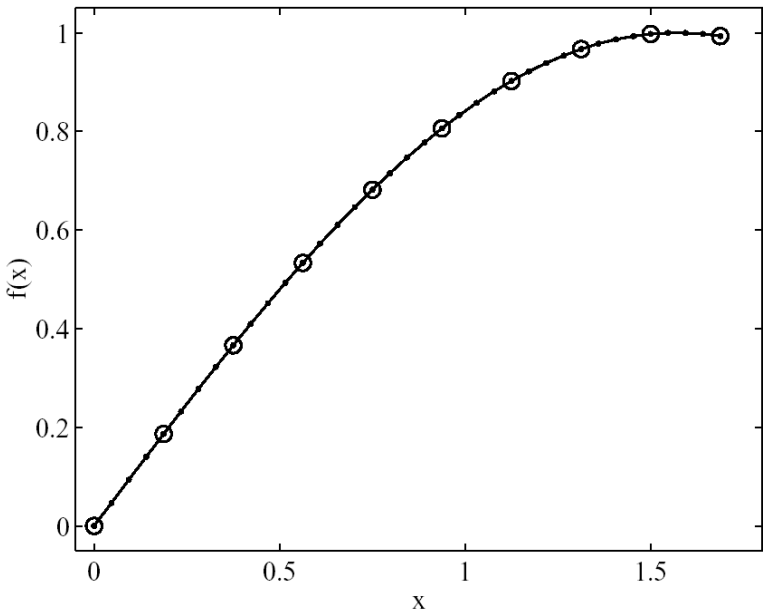


Interpolation: polynomials

- Example:
  - $\sin x$ in $[0, 1.6875]$
  - Interpolation based on 10 equidistant points

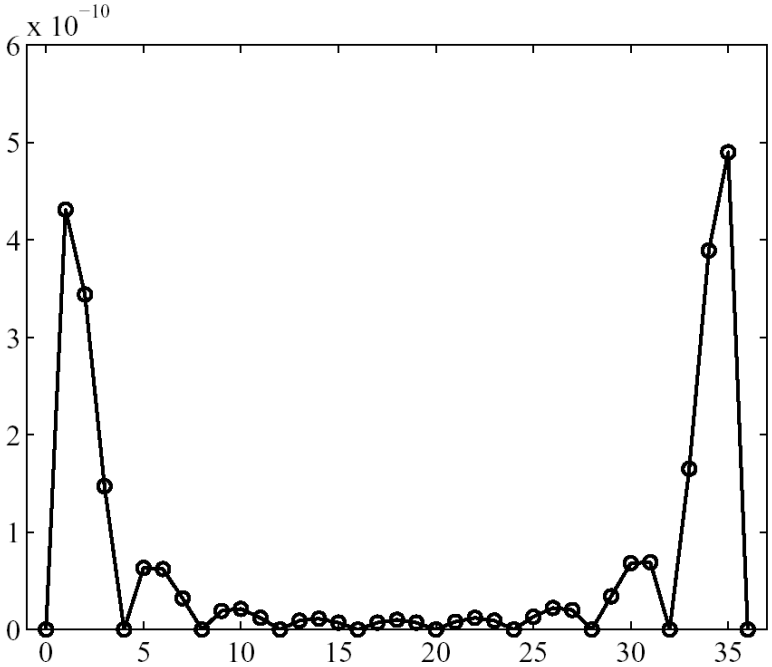
Polynomial coefficients

i	ai
0	0.00000000
1	0.99415092
2	-0.09292892
3	-0.15941590
4	0.01517217
5	0.00738018
6	-0.00073421
7	-0.00015560
8	0.00001671
9	0.00000181



Interpolation: polynomials

- Interpolation error:



Interpolation: polynomials

• Inverse interpolation:

- Approximation to inverse of a function
- Values $y_i = f(x_i)$ calculated for $x_0, x_1, \ldots, x_N$

- Form the interpolation polynomial

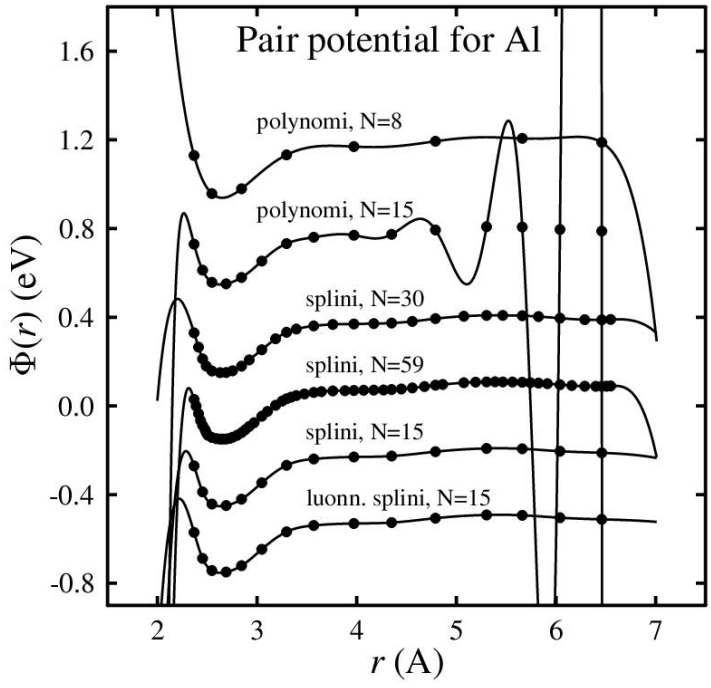
$$P_N(y) = \sum_{i=0}^{N} c_i \prod_{j=0}^{i-1} (y - y_j)$$

- It is easy to use the abovementioned routines to compute c_i

- Inverse interpolation can also be used to locate the root of a given function.

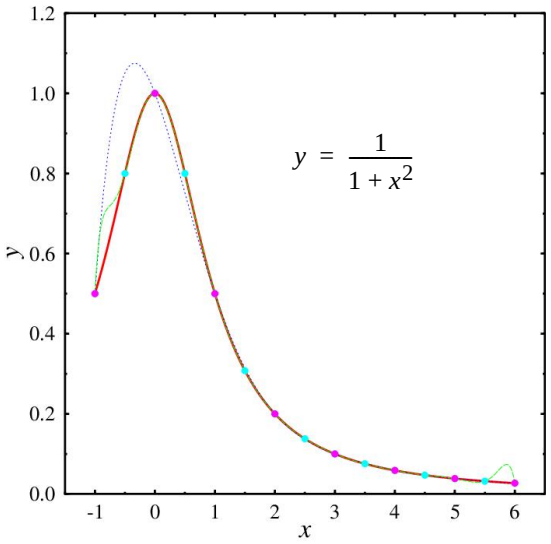
Interpolation: polynomials

• Finally a word of warning on using high-degree polynomials:
they do go through the data points but may oscillate wildly between them:



Interpolation: rational functions

- In many cases it is impossible to get a good interpolation using only polynomials
  - This means that the polynomial oscillates wildly between points
  - This may be caused by a singularity in the function to be interpolated.
    - It need not be in the interpolating interval; it may be near it or not at all on the real axis.
  - In these cases a good choice might be a rational function



  - We define the rational function $R_{i(i+1)\ldots(i+m)}$ so that it goes through points $(x_i, y_i), \ldots, (x_{i+m}, y_{i+m})$:

$$R_{i(i+1)\ldots(i+m)}(x) = \frac{P_\mu(x)}{Q_\nu(x)} = \frac{p_0 + p_1 x + \ldots + p_\mu x^\mu}{q_0 + q_1 x + \ldots + q_\nu x^\nu}$$

  - You have to give the order of the numerator and denominator when specifying the interpolating function.
  - Since there are $\mu + \nu + 1$ unknowns we must have $m + 1 = \mu + \nu + 1$
  - It is possible to derive a recursion relation as in the case of Neville's algorithm:

$$R_{i(i+1)\ldots(i+m)} = R_{(i+1)\ldots(i+m)} + \frac{R_{(i+1)\ldots(i+m)} - R_{i\ldots(i+m-1)}}{\left(\frac{x - x_i}{x - x_{i+m}}\right)\left(1 - \frac{R_{(i+1)\ldots(i+m)} - R_{i\ldots(i+m-1)}}{R_{(i+1)\ldots(i+m)} - R_{(i+1)\ldots(i+m-1)}}\right) - 1}$$

  - It is started with $R_i = y_i$ and $R_{i(i+1)\ldots(i+m)} = 0$, when $m = -1$.

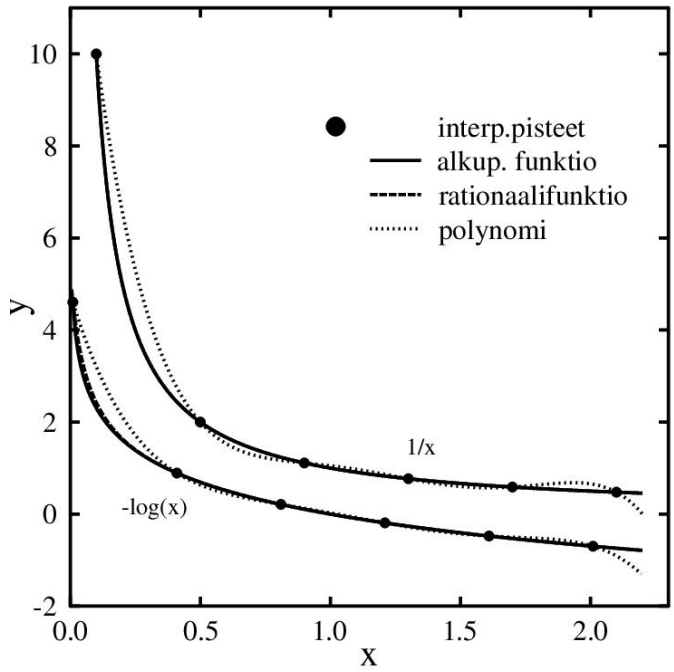
Interpolation: polynomials

- A sidenote: As expected, polynomial interpolation is used in computing transcendental functions in math libraries
  - E.g. function $\sin x$ from the **`libm`** sources:

```
/* @(#)k_sin.c 1.3 95/01/18 */
/*
* ====================================================
* Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
*
* Developed at SunSoft, a Sun Microsystems, Inc. business.
* Permission to use, copy, modify, and distribute this
* software is freely granted, provided that this notice
* is preserved.
* ====================================================
*/

/* __kernel_sin( x, y, iy)
* kernel sin function on [-pi/4, pi/4], pi/4 ~ 0.7854
* Input x is assumed to be bounded by ~pi/4 in magnitude.
* Input y is the tail of x.
* Input iy indicates whether y is 0. (if iy=0, y assume to be 0).
*
* Algorithm
* 1. Since sin(-x) = -sin(x), we need only to consider positive x.
* 2. if x < 2^-27 (hx<0x3e400000 0), return x with inexact if x!=0.
* 3. sin(x) is approximated by a polynomial of degree 13 on
* [0,pi/4]
*                  3            13
* sin(x) ~ x + S1*x + ... + S6*x
* where
*
* |sin(x)         2     4     6     8     10    12 |     -58
* |----- - (1+S1*x +S2*x +S3*x +S4*x +S5*x +S6*x ) | <= 2
* | x                                              |
*
* 4. sin(x+y) = sin(x) + sin'(x')*y
*    ~ sin(x) + (1-x*x/2)*y
 * For better accuracy, let
*      3      2      2      2      2
* r = x *(S2+x *(S3+x *(S4+x *(S5+x *S6))))
* then      3       2
* sin(x) = x + (S1*x + (x *(r-y/2)+y))
*/
```

Interpolation: rational functions

- Below two examples where rational function interpolation works better than polynomial:



Interpolation: numerical differentiation

• **Numerical differentiation** is not a trivial task.

- Subtracting almost equal numbers causes loss of significant figures.

• The crudest approximation is based on the definition of the derivative f' :

$$f'(x) \approx \frac{f(x+h)-f(x)}{h}$$

- Error truncation estimation can be obtained from the remainder term

$$f'(x) = \frac{f(x+h)-f(x)}{h} - \frac{1}{2}hf^{(2)}(\xi)$$

- This converges as $O(h)$; for $O(h^2)$ we have to take more terms.

- Consider the following Taylor's series

$$\begin{cases} f(x+h) = f(x) + hf'(x) + \frac{1}{2!}h^2f^{(2)}(x) + \frac{1}{3!}h^3f^{(3)}(x) + \frac{1}{4!}h^4f^{(4)}(x) + \dots \\ f(x-h) = f(x) - hf'(x) + \frac{1}{2!}h^2f^{(2)}(x) - \frac{1}{3!}h^3f^{(3)}(x) + \frac{1}{4!}h^4f^{(4)}(x) + \dots \end{cases}$$

- By subtraction we get

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{2}{3!}h^3f^{(3)}(x) + \frac{2}{5!}h^5f^{(5)}(x)$$

$$\rightarrow \quad f'(x) = \frac{f(x+h)-f(x-h)}{2h} - \frac{1}{3!}h^2f^{(3)}(x) - \frac{1}{5!}h^4f^{(5)}(x) - \dots \qquad \text{i.e. } O(h^2)\ !$$

Interpolation: numerical differentiation

- Including the truncation term we get

$$f'(x) = \frac{f(x+h)-f(x-h)}{2h} - \frac{1}{6}h^2 f^{(3)}(\xi) \quad , \qquad \xi \in [x-h, x+h]$$

- We can write

$$f'(x) = \frac{f(x+h)-f(x-h)}{2h} - \frac{1}{3!}h^2 f^{(3)}(x) - \frac{1}{5!}h^4 f^{(5)}(x) - \dots$$

as $$f'(x) = \frac{f(x+h)-f(x-h)}{2h} + a_2 h^2 + a_4 h^4 + a_6 h^6 + \dots$$

- Here the coefficients a_{2n} only depend on f and x *not on* h.
  - A method called **Richardson extrapolation** is based on this fact.
  - Let's define function ϕ in a following way (f and x now constants)

$$\phi(h) = \frac{f(x+h)-f(x-h)}{2h}$$

  - This is the approximation to derivative $f'(x)$ and its error behaves as $O(h^2)$.
  - Our goal is to compute the limit

$$\lim_{h \to 0} \phi(h) = f'(x)$$

Interpolation: numerical differentiation

- We can calculate ϕ for any set of h_n and use these as the approximation to $f'(x)$.
- However, we are smart and want to utilize the previous results in the series h_n to calculate the next member.

- Assume we have approximations $\phi(h)$ and $\phi(h/2)$.
- As we have $f'(x) = \frac{f(x+h)-f(x-h)}{2h} + a_2 h^2 + a_4 h^4 + a_6 h^6 + \dots$ we can write

$$\begin{cases} \phi(h) = f'(x) - a_2 h^2 - a_4 h^4 - a_6 h^6 - \dots & (1) \\ \phi(\frac{h}{2}) = f'(x) - a_2 \left(\frac{h}{2}\right)^2 - a_4 \left(\frac{h}{2}\right)^4 - a_6 \left(\frac{h}{2}\right)^6 - \dots & (2) \end{cases}$$

- Now calculate (1) – 4× (2)

$$\rightarrow \quad \phi(h) - 4\phi(\frac{h}{2}) = -3f'(x) - \frac{3}{4}a_4 h^4 - \frac{15}{16}a_6 h^6 - \dots$$

$$\rightarrow \quad \phi(\frac{h}{2}) + \frac{1}{3}\left[\phi(\frac{h}{2}) - \phi(h)\right] = f'(x) + \frac{1}{4}a_4 h^4 + \frac{5}{16}a_6 h^6 + \dots$$

- So, by adding the term $(1/3)[\phi(h/2) - \phi(h)]$ to $\phi(h/2)$ we have obtained a more accurate estimation to $f'(x)$!

- We could go on by killing the error term $a_4 h^4/4$ from the series above.
- This is called **Richardson extrapolation (RE)**.
  (Extrapolation in the sense $h \to 0$.)

Interpolation: numerical differentiation

- Generally: Assume we have a function $\phi(h)$:

$$\phi(h) = L - \sum_{k=1}^{\infty} a_{2k} h^{2k} \qquad \textbf{(1)}$$

- Coefficients a_{2k} are unknown.

- Goal is to approximate quantity L using function $\phi(h)$: we want to calculate the limit

$$L = \lim_{h \to 0} \phi(h)$$

- Choose a value for h and calculate the numbers

$$D(n, 1) = \phi\left(\frac{h}{2^n}\right), \quad n = 1, 2, \ldots$$

- According to **(1)**

$$D(n, 1) = L + \sum A(k, 1)\left(\frac{h}{2^n}\right)^{2k},$$

where $A(k, 1) = -a_{2k}$

- $D(n, 1)$ give a rough estimate for L.

Interpolation: numerical differentiation

- We can get more accurate estimates by **RE** using the recursion relation:

$$D(n, m+1) = \frac{4^m}{4^m - 1} D(n, m) - \frac{1}{4^m - 1} D(n-1, m)$$

- It can be shown that the quantities $D(n, m)$ have the form

$$D(n, m) = L + \sum_{k=m}^{\infty} A(k, m)\left(\frac{h}{2^n}\right)^{2k}$$

- Essential here is that the sum over k starts with term $k = m$, so that the first error term is $O(h^{2m})$.

- Numbers $D(n, m)$ can be arranged into triangular form:

$$\begin{array}{lllll} D(1,1) & & & & \\ D(2,1) & D(2,2) & & & \\ D(3,1) & D(3,2) & D(3,3) & & \\ \ldots & \ldots & \ldots & & \\ D(N,1) & D(N,2) & D(N,3) & \ldots & D(N,N) \end{array}$$

Interpolation: numerical differentiation

- A practical algorithm for **RE** is the following:

1. Write a subroutine for function $\phi(h)$.
2. Choose suitable values for N and h.
3. For $i = 1, 2, \ldots, N$ compute $D(i,1) = \phi(h/2^i)$
4. For $1 \le i \le j \le N$ compute $D(i,j+1) = D(i,j) + (4^j - 1)^{-1}[D(i,j) - D(i-1,j)]$

- The algorithm is easy to implement:

```
void Derivative(double (*f)(double x), double x, int N, double h, double D[NMAX][NMAX])
     {
       int i, j;
       double hh;
       hh=h;
       for (i=0; i<=N; i++) {
         D[i][0]=(f(x+hh)-f(x-hh))/(2.0*hh);
         for (j=0; j<=i-1; j++)
           D[i][j+1] = D[i][j]+(D[i][j]-D[i-1][j])/(pow(4.0,j+1.0)-1.0);
         hh = hh/2.0;
       }
     }
```


Interpolation: numerical differentiation

- And the main program:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NMAX 10
double f(double x);
void Derivative(double (*f)(double x), double x, int N, double h, double D[NMAX][NMAX]);

main (int argc, char **argv)
{
  double x,h,D[NMAX][NMAX];
  int N,i,j;
  if (argc!=4) {
    fprintf(stderr,"Usage: %s N x h\n",argv[0]); return (1);
  }
  N=atoi(*++argv);
  x=atof(*++argv);
  h=atof(*++argv);
  if (N>=NMAX) {
    fprintf(stderr,"N should be smaller than %d.\n",NMAX);
    return(-1);
  }
  for (i=0;i<N;i++) for (j=0;j<N;j++) D[i][j]=0.0;
  Derivative(f,x,N,h,D);
  for (i=0;i<N;i++) {
    fprintf(stdout,"i: %2d   h/%3d :   ",i,(int)(pow(2.0,1.0*i)));
    for (j=0;j<=i;j++)
      fprintf(stdout,"%14.10f ",D[i][j]);
    fprintf(stdout,"\n");
  }
  return(0);
}
```

Function $f(x)$:

```
double f(double x) {
  return exp(-x*x);
}
```

Interpolation: numerical differentiation

- Results from the run:

```
progs> ./richardson 6 1.0 0.5
i:  0   h/  1 :    -0.6734015585
i:  1   h/  2 :    -0.7203428752  -0.7359899807
i:  2   h/  4 :    -0.7319209458  -0.7357803026  -0.7357663241
i:  3   h/  8 :    -0.7348004908  -0.7357603391  -0.7357590082  -0.7357588921
i:  4   h/ 16 :    -0.7355193541  -0.7357589753  -0.7357588843  -0.7357588824  -0.7357588823
i:  5   h/ 32 :    -0.7356990047  -0.7357588882  -0.7357588824  -0.7357588823  -0.7357588823  -0.7357588823
```

- Accurate value: $f(x) = e^{-x^2} \rightarrow f'(x) = -2xe^{-x^2} \rightarrow \quad f'(1) = -\frac{2}{e} = -0.73575888$

Interpolation: numerical differentiation

• Numerical derivatives can also be computed using the derivative if the interpolated polynomial:

$$f'(x) \approx P'(x)$$

- In practice the interpolating polynomial is determined only using a couple of points near the argument.
- For two nodes:

$$P_1(x) = f(x_0) + f[x_0, x_1](x - x_0)$$

$$\rightarrow \quad f'(x) \approx P'(x) = f[x_0, x_1] = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

- If $x_0 = x$ and $x_1 = x + h$ we get

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

- For $x_0 = x - h$ and $x_1 = x + h$

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \qquad \text{(central difference form)}$$

Interpolation: numerical differentiation

- Now consider three nodes:

$$P_2(x) = f(x_0) + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1)$$

- The derivative is

$$P_2'(x) = f[x_0, x_1] + f[x_0, x_1, x_2](2x - x_0 - x_1)$$

- This is the same as above but with the correction $f[x_0, x_1, x_2](2x - x_0 - x_1)$.

- The accuracy of the leading term $f[x_0, x_1]$ depends on the choice of x_0 and x_1.

- If $x = (x_0 + x_1)/2 \rightarrow 2x - x_0 - x_1 = 0$ and correction term is zero.
- This is the reason for the greater accuracy of the central difference form of derivative.

- In general we can analyze the error by using the error bound for the interpolating polynomial:

$$f(x) - P(x) = \frac{f^{(N+1)}(\xi(x))}{(N+1)!} \prod_{i=0}^{N} (x - x_i)$$

Interpolation: numerical differentiation

- Let's assume that P_N is the polynomial of the least degree that interpolates f at points $x_0, x_1, x_2, \ldots, x_N$.

- Now we can write the error bound as:

$$f(x) - P_N(x) = \frac{f^{(N+1)}(\xi)}{(N+1)!} \omega(x)$$

where $\omega(x) = \prod_{i=0}^{N} (x - x_i)$ and $\xi = \xi(x)$

- Differentiation gives

$$f'(x) - P_N'(x) = \frac{1}{(N+1)!} \omega(x) \frac{\mathrm{d}}{\mathrm{d}x} f^{(N+1)}(\xi) + \frac{1}{(N+1)!} f^{(N+1)}(\xi) \omega'(x)$$

- If we can set x equal to a node the expression is simplified because $\omega(x_i) = 0$

$$f'(x_i) = P_N'(x_i) + \frac{1}{(N+1)!} f^{(N+1)}(\xi) \omega'(x_i)$$

- Or if we can choose x such that $\omega'(x) = 0$:

$$f'(x) = P_N'(x) + \frac{1}{(N+1)!} \omega(x) \frac{\mathrm{d}}{\mathrm{d}x} f^{(N+1)}(\xi)$$

Interpolation: numerical differentiation

- Example: Derive the first-derivative formula using the interpolating polynomial $P_3(x)$ obtained with four nodes x_0, x_1, x_2, x_3. Use central difference in choosing the nodes.

  - Newton form of the polynomial

$$P_3(x) = f(x_0) + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + f[x_0, x_1, x_2, x_3](x - x_0)(x - x_1)(x - x_2)$$

  - And its derivative

$$P_3'(x) = f[x_0, x_1] + f[x_0, x_1, x_2]\{(x - x_0) + (x - x_1)\} + f[x_0, x_1, x_2, x_3]\{(x - x_1)(x - x_2) + (x - x_0)(x - x_2) + (x - x_0)(x - x_1)\}$$

Note that:

$$\omega'(x) = \sum_i \left[\prod_{j \neq i} (x - x_j) \right]$$

  - Formulas for the divided differences are obtained in a familiar way:

$$f[x_i] = f(x_i)$$

$$f[x_i, x_{i+1}] = \frac{f[x_{i+1}] - f[x_i]}{x_{i+1} - x_i}$$

$$f[x_i, x_{i+1}, x_{i+2}] = \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i}$$

Interpolation: numerical differentiation

- Now we choose the nodes using the central difference; i.e. x is in the middle of the nodes:

$$x_0 = x - h,\ x_1 = x + h,\ x_2 = x - 2h,\ x_3 = x + 2h$$

- This allows us to write down the divided differences:

$$f[x_0, x_1] = \frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{f(x+h) - f(x-h)}{2h}$$

$$f[x_1, x_2] = \frac{f(x_2) - f(x_1)}{x_2 - x_1} = \frac{f(x-2h) - f(x+h)}{-3h}$$

$$f[x_2, x_3] = \frac{f(x_3) - f(x_2)}{x_3 - x_2} = \frac{f(x+2h) - f(x-2h)}{4h}$$

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} = \frac{1}{-h}\left\{ \frac{f(x-2h) - f(x+h)}{-3h} - \frac{f(x+h) - f(x-h)}{2h} \right\}$$

$$= \frac{1}{6h^2}[2f(x-2h) - 3f(x-h) + f(x+h)]$$

$$f[x_1, x_2, x_3] = \frac{f[x_2, x_3] - f[x_1, x_2]}{x_3 - x_1} = \frac{1}{h}\left\{ \frac{f(x+2h) - f(x-2h)}{4h} - \frac{f(x-2h) - f(x+h)}{-3h} \right\}$$

$$= \frac{1}{12h^2}[3f(x+2h) - 4f(x+h) + f(x-2h)]$$

Interpolation: numerical differentiation

- And the last one:

$$f[x_0, x_1, x_2, x_3] = \frac{f[x_1, x_2, x_3] - f[x_0, x_1, x_2]}{x_3 - x_0}$$

$$= \frac{1}{3h}\left\{\frac{3f(x+2h) - 4f(x+h) + f(x-2h)}{12h^2} - \frac{2f(x-2h) - 3f(x-h) + f(x+h)}{6h^2}\right\}$$

$$= \frac{1}{12h^3}[f(x+2h) - 2f(x+h) + 2f(x-h) - f(x-2h)]$$

- The following two factors are still needed:

$$(x - x_0) + (x - x_1) = 2x - x_1 - x_0 = 0$$

$$(x - x_1)(x - x_2) + (x - x_0)(x - x_2) + (x - x_0)(x - x_1) = -h^2$$

- The derivative of the polynomial

$$P_3'(x) = f[x_0, x_1] + f[x_0, x_1, x_2]\{(x - x_0) + (x - x_1)\}$$

$$+ f[x_0, x_1, x_2, x_3]\{(x - x_1)(x - x_2) + (x - x_0)(x - x_2) + (x - x_0)(x - x_1)\}$$

$$= f[x_0, x_1] + f[x_0, x_1, x_2, x_3]\{(x - x_1)(x - x_2) + (x - x_0)(x - x_2) + (x - x_0)(x - x_1)\}$$

$$= \frac{f(x+h) - f(x-h)}{2h} - \frac{[f(x+2h) - 2f(x+h) + 2f(x-h) - f(x-2h)]}{12h}$$

Interpolation: numerical differentiation

- Approximation for the derivative is then

$$f'(x) \approx \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h}$$

- This can be used in the Richardson extrapolation to speed up the convergence.

- Example: $f(x) = \sin x$, $x = 1.2309594$, $h = 1$

```
D(n,0)     D(n,1)     D(n,2)     D(n,3)     D(n,4)
0.32347058
0.33265926 0.33572215
0.33329025 0.33350059 0.33335248
0.33333063 0.33334409 0.33333365 0.33333335
0.33333317 0.33333401 0.33333334 0.33333334 0.33333334
```

- For comparison when only the first term $f'(x) \approx [f(x+h) - f(x-h)]/(2h)$ is used we get the result

```
D(n,0)     D(n,1)     D(n,2)     D(n,3)     D(n,4)
0.28049033
0.31961703 0.33265926
0.32987195 0.33329025 0.33333232
0.33246596 0.33333063 0.33333332 0.33333333
0.33311636 0.33333317 0.33333333 0.33333334 0.33333334
```

Interpolation: Hermite interpolation

• The interpolating function goes through data points $(x_i, f(x_i))$, $i = 0, 1, \ldots, N$:

$$P(x_i) = f(x_i)$$

• We can generalize this by adding requirements that the interpolating function also reproduces derivatives:

$$P^{(m_i)}(x_i) = f^{(m_i)}(x_i) \ .$$

- The polynomial P is **osculating** to the function f.

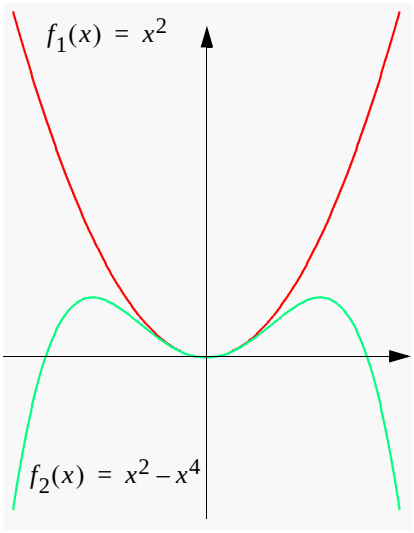
- We have $N+1$ points (x_i, y_i) and nonnegative integers $m_0, m_1, \ldots, m_N$.

- The polynomial fulfilling these conditions has a degree at most

$$M = \sum_{i=0}^{N} m_i + N \, .$$

- So our osculating polynomial interpolates the function f such that

$$\frac{\mathrm{d}^k}{\mathrm{d}x^k} P(x_i) = \frac{\mathrm{d}^k}{\mathrm{d}x^k} f(x_i) \quad \text{for each } i = 0, 1, \ldots, N \text{ and } k = 0, 1, \ldots, m_i \, .$$



$f_1^{(k)}(0) = f_2^{(k)}(0)$
for $k = 0, 1, 2$

Interpolation: Hermite interpolation

- Note:

1) When $N = 0$ we have the m_0th Taylor polynomial for f at x_0.

2) When $m_i = 0$ for $i = 0, \ldots, N$ we have the Nth order interpolating polynomial for f

- When $m_i = 1$ for $i = 0, \ldots, N$ the interpolating polynomials are called **Hermite polynomials**:

$$P(x_i) = f(x_i) \text{ and } P'(x_i) = f'(x_i) \, .$$

- One can derive the following form for the polynomial using the divided differences[1]

$$H_N(x) = Q_{0,0} + Q_{1,1}(x - x_0) + Q_{1,1}(x - x_0)^2 + Q_{3,3}(x - x_0)^2(x - x_1) + Q_{4,4}(x - x_0)^2(x - x_1)^2 + \ldots + Q_{2n+1,2n+1}(x - x_0)^2(x - x_1)^2(x - x_2)^2 \ldots (x - x_{n-1})^2(x - x_n)$$

- Coefficients $Q_{i,i}$ are obtained by the following algorithm:

1. For $i = 0, 1, \ldots, N$ do
   a) Set $z_{2i} = x_i$, $z_{2i+1} = x_i$, $Q_{2i,0} = f(x_i)$, $Q_{2i+1,0} = f(x_i)$, $Q_{2i+1,1} = f'(x_i)$
   b) If $i \neq 0$ set $Q_{2i,1} = \dfrac{Q_{2i,0} - Q_{2i-1,0}}{z_{2i} - z_{2i-1}}$

2. For $i = 2, 3, \ldots, 2N+1$, for $j = 2, 3, \ldots, i$ set $Q_{i,j} = \dfrac{Q_{i,j-1} - Q_{i-1,j-1}}{z_i - z_{i-j}}$

1. See e.g. Burden & Faires: *Numerical Analysis*, Ch. 3.5

Interpolation: Hermite interpolation

- An in Fortran it reads

```
program hermite
  implicit none
  integer,parameter :: &
              &rk=selected_real_kind(15,100)

  real(rk),allocatable :: xa(:),ya(:),&
                          &da(:),q(:,:),z(:)
  real(rk) ::  x,y,x1,x2,dx,p
  integer :: n,i,j,ix,ixmax,jmax,iargc
  character(len=80) :: argu
  call getarg(1,argu); read(argu,*) x1
  call getarg(2,argu); read(argu,*) x2
  call getarg(3,argu); read(argu,*) dx
  read(5,*) n
  allocate(xa(0:n-1),ya(0:n-1),da(0:n-1),&
           &q(0:2*n-1,0:2*n-1),z(0:2*n-1))
  do i=0,n-1
     read(5,*) xa(i),ya(i),da(i)
  end do
  n=n-1
  q=0.0
  z=0.0
  do i=0,n
     z(2*i)=xa(i)
     z(2*i+1)=xa(i)
     q(2*i,0)=ya(i)
     q(2*i+1,0)=ya(i)
     q(2*i+1,1)=da(i)
     if (i/=0) then
        q(2*i,1)=(q(2*i,0)-q(2*i-1,0))/&
                 &(z(2*i)-z(2*i-1))
     end if
  end do
  do i=2,2*n+1
     do j=2,i
        q(i,j)=(q(i,j-1)-q(i-1,j-1))/(z(i)-z(i-j))
     end do
  end do

  ! Use the coefficients to calculate P(x)
  ixmax=(x2-x1)/dx
  do ix=0,ixmax
     x=x1+dx*ix
     y=q(0,0)+q(1,1)*(x-xa(0))
     do i=2,2*n+1
        p=q(i,i)
        jmax=i/2-1
        if (ix==0) write(0,*) i,jmax
        do j=0,jmax
           p=p*(x-xa(j))**2
        end do
        if (mod(i,2)==1) p=p*(x-xa(jmax+1))
        y=y+p
     end do
     write(6,'(f10.4,g20.10)') x,y
  end do

  stop
end program hermite
```


Interpolation: Hermite interpolation

- Graphically: interpolating $f(x) = e^{-x^2}$:

- Hermite polynomial is often used in the form of cubic Hermite polynomial:

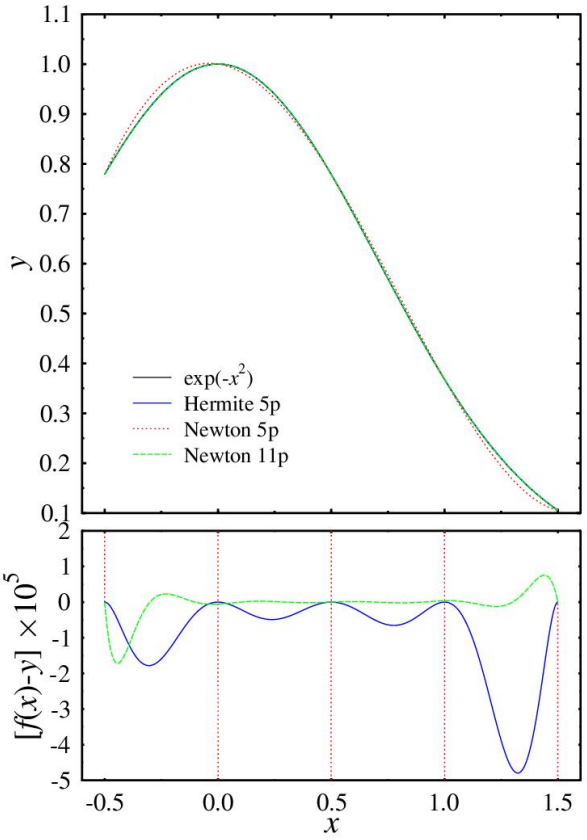
$$\begin{cases} P(a) = f(a) \\ P'(a) = f'(a) \\ P(b) = f(b) \\ P'(b) = f'(b) \end{cases}$$

- The divided difference formula for this is

$$H_3(x) = f(a) + (x-a)f'(a) - (x-a)^2 f[a,a,b] + (x-a)^2(x-b)f[a,a,b,b]$$

Note: The following definition applies to divided differences with repetition of the same argument n:

$$f[x_i, x_i, \ldots, x_i] = \frac{f^{(n-1)}(x_i)}{(n-1)!}$$



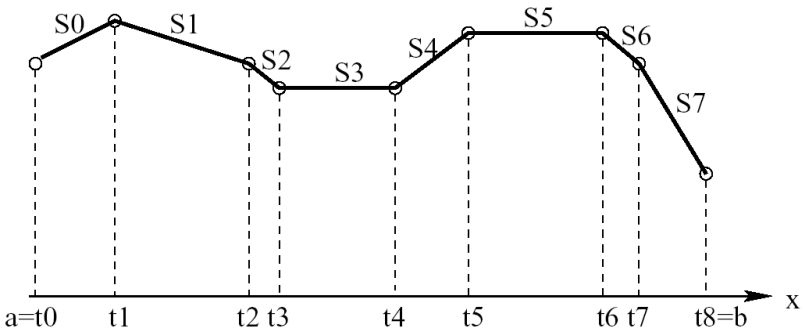
Interpolation: piecewise interpolation

• High degree polynomial → oscillations.

• Remedy: construct interpolating polynomials for subintervals of the data $\{x_i, f(y_i)\}$:

piecewise polynomials or **splines**

- The simplest spline is of the first order: linear interpolation



- In explicit form: $S(x) = \begin{cases} S_0(x) & x \in [t_0, t_1] \\ S_1(x) & x \in [t_1, t_2] \\ & \dots \\ S_{N-1}(x) & x \in [t_{N-1}, t_N] \end{cases}$

Interpolation: piecewise interpolation

- Where each piece of S is linear:

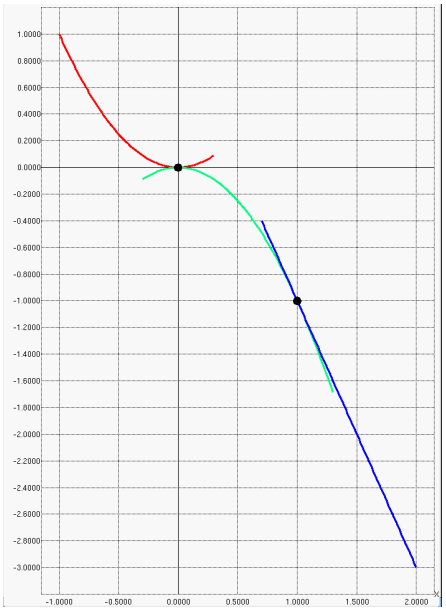
$$S_i(x) = a_i x - b_i$$

- In general a spline S of degree k fullfills the following conditions

1. The domain of S is interval $[a, b]$
2. S, S', S'', $S^{(3)}$, …, $S^{(k-1)}$ are all continuous on $[a, b]$.
3. There are points x_i (knots of S) such that $a = x_0 < x_1 < \dots < x_n = b$

and that S is

a polynomial of degree at most k on each subinterval $[x_i, x_{i+1}]$.



- A quadratic spline $Q(x)$ is a continuously differentiable piecewise quadratic function.

- Example:

$$Q(x) = \begin{cases} x^2 & x \le 0 \\ -x^2 & 0 \le x \le 1 \\ 1 - 2x & x \ge 1 \end{cases}$$

Interpolation: piecewise interpolation

- Ensure that the function $Q(x)$ really is a quadratic spline. Check all knots:

$$\lim_{x \to 0^-} Q(x) = \lim_{x \to 0^-} x^2 = 0 \qquad \lim_{x \to 0^+} Q(x) = \lim_{x \to 0^+} -x^2 = 0$$

$$\lim_{x \to 1^-} Q(x) = \lim_{x \to 1^-} -x^2 = -1 \qquad \lim_{x \to 1^+} Q(x) = \lim_{x \to 1^+} (1-2x) = -1$$

$$\lim_{x \to 0^-} Q'(x) = \lim_{x \to 0^-} 2x = 0 \qquad \lim_{x \to 0^+} Q'(x) = \lim_{x \to 0^+} -2x = 0$$

$$\lim_{x \to 1^-} Q'(x) = \lim_{x \to 1^-} -2x = -2 \qquad \lim_{x \to 1^+} Q'(x) = \lim_{x \to 1^+} (-2) = -2$$

- Let's assume we have the data set $\{x_i, y_i\}$, $i = 0, 1, \ldots, N$ and we want to interpolate it using quadratic splines.

$\rightarrow$ we have to determine N functions $Q_i(x) = a_i x^2 + b_i x + c_i$, $i = 0, 1, \ldots, N-1$.

$\rightarrow$ we have to determine $3N$ coefficients

- On each subinterval $[x_i, x_{i+1}]$ Q must satisfy:

$$Q_i(x_i) = y_i \text{ and } Q_i(x_{i+1}) = y_{i+1}$$

$\rightarrow 2N$ equations

- Continuity of the first derivative gives

$$Q'_{i-1}(x_i) = Q'_i(x_i),\ i = 1, 2, \ldots, N-1$$

$\rightarrow N-1$ equations

- Only $3N-1$ equations. Additional conditions e.g. $Q'(x_0) = 0$.

Interpolation: piecewise interpolation

- So, we seek for a piecewise quadratic function

$$Q(x) = \begin{cases} Q_0(x) & x \in [x_0, x_1) \\ Q_1(x) & x \in [x_1, x_2) \\ \ldots & \\ Q_{N-1}(x) & x \in [x_{N-1}, x_N] \end{cases}$$

which is continuously differentiable on the interval $[x_0, x_N]$ and which interpolates the table: $Q(x_i) = y_i$

- Denote $z_i = Q'(x_i)$. We can write

$$Q_i(x) = \frac{z_{i+1} - z_i}{2(x_{i+1} - x_i)}(x - x_i)^2 + z_i(x - x_i) + y_i$$

- It easy to verify that

$$Q_i(x_i) = y_i$$

$$Q'_i(x_i) = z_i$$

$$Q'_i(x_{i+1}) = z_{i+1}$$

Interpolation: piecewise interpolation

- In order to Q to be continuous and interpolate the data table $\{x_i, y_i\}$ it is necessary and sufficient that

$$Q_i(x_{i+1}) = y_{i+1}$$

- This gives us

$$z_{i+1} = -z_i + 2\left(\frac{y_{i+1}-y_i}{x_{i+1}-x_i}\right), \quad 0 \le i \le N-1 .$$

- The vector $[z_0, z_1, \ldots, z_N]^T$ can now be constructed by starting from (an arbitrary) value of $z_0 = Q'(x_0)$

- Well, the recursion relation above is a group of linear equations (though a trivial one):

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \ldots & 0 \\ -1 & 1 & 0 & 0 & \ldots & 0 \\ 0 & -1 & 1 & 0 & \ldots & 0 \\ 0 & 0 & -1 & 1 & \ldots & 0 \\ & & \ldots & & & \\ 0 & 0 & \ldots & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ z_3 \\ \ldots \\ z_{N-1} \\ z_N \end{bmatrix} = \begin{bmatrix} Q'(x_0) \\ (y_1-y_0)/(x_1-x_0) \\ (y_2-y_1)/(x_2-x_1) \\ \ldots \\ (y_{N-1}-y_{N-2})/(x_{N-1}-x_{N-2}) \\ (y_N-y_{N-1})/(x_N-x_{N-1}) \end{bmatrix}$$

Interpolation: piecewise interpolation

- The program that does the interpolation:

```
program qspline

implicit none
integer,parameter :: rk=selected_real_kind(15,100)

  real(rk),allocatable :: xa(:),ya(:),z(:)
  real(rk) ::  x,y,x1,x2,dx,Qp0
  integer :: n,i,j,ix,ixmax
  character(len=80) :: argu

  if (iargc()/=4) then
     call getarg(0,argu)
     write(0,'(a,a,a)') &
            &'usage: ',trim(argu),' x1 x2 dx Qp0'
     stop
  end if

  call getarg(1,argu); read(argu,*) x1
  call getarg(2,argu); read(argu,*) x2
  call getarg(3,argu); read(argu,*) dx
  call getarg(4,argu); read(argu,*) Qp0

  read(5,*) n
  allocate(xa(0:n),ya(0:n),z(0:n))
  do i=0,n-1
     read(5,*) xa(i),ya(i)
  end do
  n=n-1
  z(0)=Qp0
  do i=0,n-1
    z(i+1)=-z(i)+2.0*(ya(i+1)&
           & -ya(i))/(xa(i+1)-xa(i))
  end do

  ixmax=(x2-x1)/dx

  do ix=0,ixmax
     x=x1+dx*ix
     if (x<xa(0).or.x>xa(n)) cycle
     do i=0,n-1
        if (x>=xa(i).and.x<xa(i+1)) exit
     end do
     y=(z(i+1)-z(i))/2/(xa(i+1)&
        &-xa(i))*(x-xa(i))**2+z(i)*(x-xa(i))+ya(i)
     write(6,*) x,y
  end do

  stop
end program qspline
```

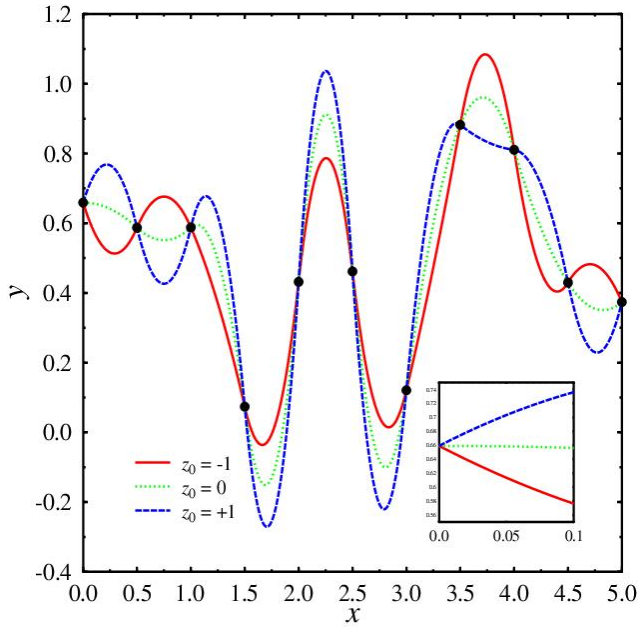
Interpolation: piecewise interpolation

- Random sets of knots created with gawk one-liner:

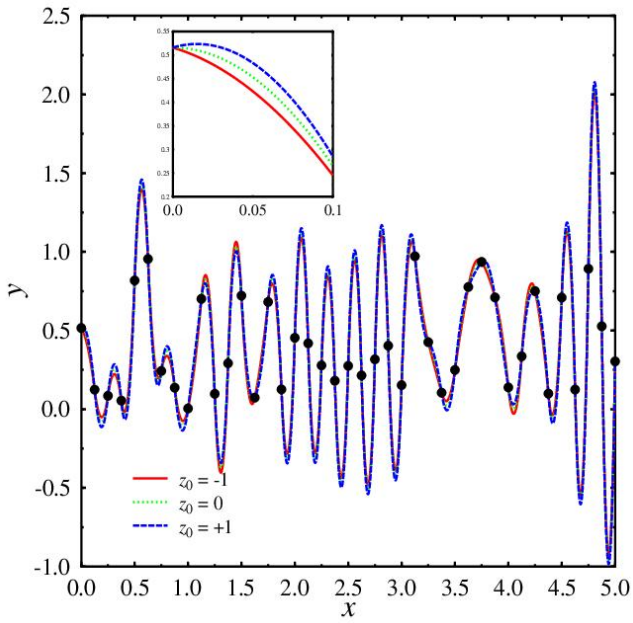
```
gawk 'BEGIN {srand(); for (x=0;x<=5.01;x+=0.5) print x,rand()}'
```

- Graphically:

11 knots



41 knots



Interpolation: piecewise interpolation

• Splines of degree 3 are **cubic splines** and they a probably the most commonly used splines.

- Cubic splines have more flexibility than quadratic splines → 'smoother' interpolant.

- In general for splines the following theorem applies:

Let $f(x) \in C^m[a,b]$ and $S_{2m-1}(x)$ be the interpolating spline of f. If $g(x)$ is an m times continuously differentiable function that fulfills the same interpolating conditions as S_{2m-1} then

$$\int_a^b \left|S^{(m)}_{2m-1}(x)\right|^2 dx \le \int_a^b \left|g^{(m)}(x)\right|^2 dx$$

- For cubic splines ($m = 2$) we can write this in the form

$$\int_a^b \left|S''_3(x)\right|^2 dx \le \int_a^b |g''(x)|^2 dx$$

- Now the curvature of a curve $y(x)$ is defined as

$$\kappa(x) = \frac{y''(x)}{\{1 + [y'(x)]^2\}^{3/2}}$$

- If we assume that $y'(x) \ll 1$ in $[a,b]$ we can write $\kappa(x) \approx y''(x)$

Interpolation: piecewise interpolation

- So, for cubic splines we can say that they are the interpolants that have — on the average — the smallest curvature.

- Another way to depict this is to imagine the spline as a solid wire that is forced to go through the knots and minimizes its bending energy.

- As usual we have the data set or knots

x	x_0	x_1	…	x_N
y	y_0	y_1	…	y_N

- We assume that knots are distinct and in ascending order: $x_0 < x_1 < \ldots < x_N$

- The interpolating function $S(x)$ consists of N cubic polynomials:

$$S(x) = \begin{cases} S_0(x) & x \in [x_0, x_1) \\ S_1(x) & x \in [x_1, x_2) \\ \ldots & \\ S_{N-1}(x) & x \in [x_{N-1}, x_N] \end{cases}$$

i.e. $S_i(x)$ interpolates the subinterval $[x_i, x_{i+1}]$

Interpolation: piecewise interpolation

- The **interpolation conditions** are

$$S_i(x_i) = y_i, \quad 0 \le i \le N$$

- The **continuity conditions** imposed only at the interior knots $x_1, x_2, \ldots, x_{N-1}$ are

$$\lim_{x \to x_i^-} S^{(k)}(x_i) = \lim_{x \to x_i^+} S^{(k)}(x_i), \quad k = 0, 1, 2$$

- Are these enough:

We have N 3rd order polynomials $\rightarrow 4N$ parameters.
Interpolation conditions $\rightarrow N+1$ equations.
Continuity conditions $\rightarrow 3(N-1)$ equations.
Total $4N-2$ equations.

$\rightarrow$ need two more equations.

- A common choice is

$$S''(x_0) = S''(x_N) = 0$$

- The resulting function is termed a **natural cubic spline**.

Interpolation: piecewise interpolation

- Example: Determine parameters a, b, c, d, e, f, g, and h so that $S(x)$ is a natural cubic spline, where

$$S(x) = \begin{cases} ax^3 + bx^2 + cx + d & x \in [-1, 0] \\ ex^3 + fx^2 + gx + h & x \in [0, 1] \end{cases}$$

and with interpolation conditions: $S(-1) = 1$, $S(0) = 2$, $S(1) = -1$.

- Let the two polynomials be $S_0(x)$ and $S_1(x)$.
- From interpolation conditions we get

$$S_0(0) = d = 2$$
$$S_1(0) = h = 2$$
$$S_0(-1) = -a + b - c = -1$$
$$S_1(1) = e + f + g = -3$$

- The 1st derivative of $S(x)$ is

$$S'(x) = \begin{cases} 3ax^2 + 2bx + c \\ 3ex^2 + 2fx + g \end{cases}$$

- Continuity of $S'(x)$ implies

$$S'(0) = c = g$$

- The 2nd derivative is

$$S''(x) = \begin{cases} 6ax + 2b \\ 6ex + 2f \end{cases}$$

Interpolation: piecewise interpolation

- From the continuity of $S''(x)$ we get

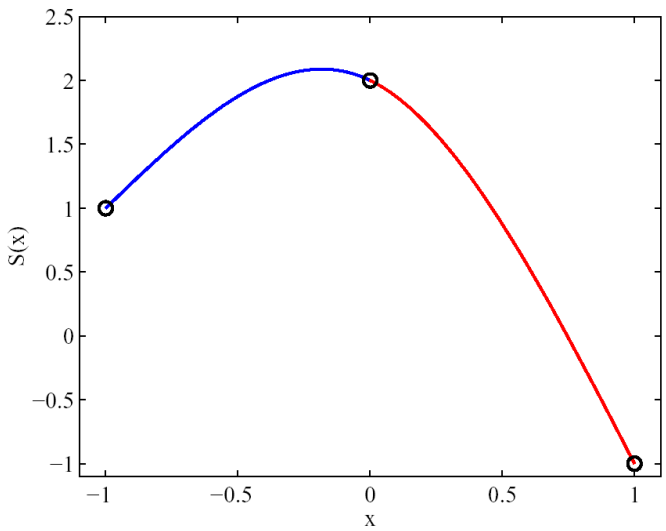
$$S''(0) = b = f$$

- And the two extra conditions

$$\begin{cases} S''(-1) = -6a + 2b = 0 \\ S''(1) = -6e + 2f = 0 \end{cases}$$

- From all these equations we get

$$a = -1, \quad b = -3, \quad c = -1, \quad d = 2, \quad e = 1, \quad f = -3, \quad g = -1, \quad h = 2$$



Interpolation: piecewise interpolation

- In what follows we will develop a general algorithm to determine the natural cubic spline.

- Based on the continuity of $S''(x)$ we can unambiguously define the following numbers

$$z_i \equiv S''(x_i),\ 0 \le i \le N$$

- The two extra conditions say that $z_0 = z_N = 0$ but the other values are still unknown.

- We know that on each subinterval $[x_i, x_{i+1}]$ the second derivative $S''(x)$ is linear, and takes the values z_i and z_{i+1} at the end points:

$$S_i''(x) = \frac{z_{i+1}}{h_i}(x - x_i) + \frac{z_i}{h_i}(x_{i+1} - x),$$

where we have defined $h_i = x_{i+1} - x_i,\ 0 \le i \le N-1$

- Integrating this twice we get

$$S_i'(x) = \frac{z_{i+1}}{2h_i}(x - x_i)^2 - \frac{z_i}{2h_i}(x_{i+1} - x)^2 + c$$

$$S_i(x) = \frac{z_{i+1}}{6h_i}(x - x_i)^3 + \frac{z_i}{6h_i}(x_{i+1} - x)^3 + cx + d$$

where c and d are integration constants.

Interpolation: piecewise interpolation

- By adjusting the constants we get a more convenient form of the spline

$$S_i(x) = \frac{z_{i+1}}{6h_i}(x - x_i)^3 + \frac{z_i}{6h_i}(x_{i+1} - x)^3 + C_i(x - x_i) + D_i(x_{i+1} - x)$$

- Constants C_i and D_i can be determined by applying the interpolation conditions:

$$S_i(x_i) = y_i\ ,\ S_i(x_{i+1}) = y_{i+1}$$

$$\rightarrow \quad S_i(x) = \frac{z_{i+1}}{6h_i}(x - x_i)^3 + \frac{z_i}{6h_i}(x_{i+1} - x)^3 + \left(\frac{y_{i+1}}{h_i} - \frac{h_i z_{i+1}}{6}\right)(x - x_i) + \left(\frac{y_i}{h_i} - \frac{h_i z_i}{6}\right)(x_{i+1} - x)$$

- We still have to find out the values of z_i.
- This is done by applying the remaining condition – namely the continuity of $S'(x)$:

- At the interior knots $x_i,\ 1 \le i \le N-1$, we must have

$$S'_{i-1}(x_i) = S'_i(x_i)$$

- From above we get the derivative:

$$S'_i(x) = \frac{z_{i+1}}{2h_i}(x - x_i)^2 - \frac{z_i}{2h_i}(x_{i+1} - x)^2 + \frac{y_{i+1}}{h_i} - \frac{h_i z_{i+1}}{6} - \frac{y_i}{h_i} + \frac{h_i z_i}{6}$$

Interpolation: piecewise interpolation

- And

$$S'_i(x_i) = -\frac{h_i}{6}z_{i+1} - \frac{h_i}{3}z_i + b_i, \qquad b_i = \frac{y_{i+1}-y_i}{h_i}$$

$$S'_{i-1}(x_i) = -\frac{h_{i-1}}{6}z_{i-1} - \frac{h_{i-1}}{3}z_i + b_{i-1}, \qquad b_{i-1} = \frac{y_i-y_{i-1}}{h_{i-1}}$$

- When we set these equal we get – after a little rearrangement

$$h_{i-1}z_{i-1} + 2(h_{i-1}-h_i)z_i + h_i z_{i+1} = 6(b_i - b_{i-1}), \qquad 1 \le i \le N-1$$

- By letting

$$\begin{cases} u_i = 2(h_{i-1}+h_i) \\ v_i = 6(b_i - b_{i-1}) \end{cases}$$

we get a tridiagonal system of equations for z_i:

$$\begin{cases} z_0 = 0 \\ h_{i-1}z_{i-1} + u_i z_i + h_i z_{i+1} = v_i \qquad 1 \le i \le N-1 \\ z_N = 0 \end{cases}$$

Interpolation: piecewise interpolation

- In matrix form this is

$$\begin{bmatrix} 1 & 0 & & & & & \\ h_0 & u_1 & h_1 & & & & \\ & h_1 & u_2 & h_2 & & & \\ & & \cdots & & & & \\ & & & h_{N-1} & u_{N-1} & h_{N-1} & \\ & & & & 0 & 1 & \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ \cdots \\ z_{N-1} \\ z_N \end{bmatrix} = \begin{bmatrix} 0 \\ v_1 \\ v_2 \\ \cdots \\ v_{N-1} \\ 0 \end{bmatrix}$$

- We can eliminate the first and the last equations and get finally the form:

$$\begin{bmatrix} u_1 & h_1 & & & \\ h_1 & u_2 & h_2 & & \\ & \cdots & \cdots & & \\ & & h_{N-3} & u_{N-2} & h_{N-2} \\ & & & h_{N-2} & u_{N-1} \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ \cdots \\ z_{N-2} \\ z_{N-1} \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ \cdots \\ v_{N-2} \\ v_{N-1} \end{bmatrix}$$

which is a symmetric tridiagonal system of order $N-1$.

Interpolation: piecewise interpolation

- This system can be readily solved by

the forward elimination:
$$\begin{cases} u_i \leftarrow u_i - \dfrac{h_{i-1}^2}{u_{i-1}} \\ v_i \leftarrow v_i - \dfrac{h_{i-1} v_{i-1}}{u_{i-1}} \end{cases} \qquad i = 2, 3, \ldots, N-1$$

and the back substitution:
$$\begin{cases} z_{N-1} \leftarrow \dfrac{v_{N-1}}{u_{N-1}} \\ z_i \leftarrow \dfrac{v_i - h_i z_{i+1}}{u_i} \end{cases} \qquad i = N-2, N-3, \ldots, 1$$

Interpolation: piecewise interpolation

- Finally we can write down the algorithm for the natural cubic spline for data set $\{x_i, y_i\}$, $i = 0, 1, \ldots, N$:

1. For $i = 0, 1, \ldots, N-1$ compute
$$\begin{cases} h_i = x_{i+1} - x_i \\ b_i = \dfrac{y_{i+1} - y_i}{h_i} \end{cases}$$
2. Set
$$\begin{cases} u_1 = 2(h_1 + h_0) \\ v_1 = 6(b_1 - b_0) \end{cases}$$
3. For $i = 2, 3, \ldots, N-1$ compute
$$\begin{cases} u_i = 2(h_i + h_{i-1}) - \dfrac{h_{i-1}^2}{u_{i-1}} \\ v_i = 6(b_i - b_{i-1}) - \dfrac{h_{i-1} v_{i-1}}{u_{i-1}} \end{cases}$$
4. Set
$$\begin{cases} z_N = 0 \\ z_0 = 0 \end{cases}$$
5. For $i = N-1, N-2, \ldots, 1$ compute
$$z_i = \frac{v_i - h_i z_{i+1}}{u_i}.$$

Interpolation: piecewise interpolation

- This algorithm might fail if $u_i = 0$ for some i.

  - It is clear that $u_1 > h_1 > 0$.
  - Now if $u_{i-1} > h_{i-1}$ then $u_i > h_i$ because

$$u_i = 2(h_i + h_{i-1}) - \frac{h_{i-1}^2}{u_{i-1}} > 2(h_i + h_{i-1}) - h_{i-1} > h_i$$

  - So, by induction: $u_i > 0$, $i = 1, 2, \ldots, N-1$.

  - Now we have the coefficients z_i of the spline

$$S_i(x) = \frac{z_{i+1}}{6h_i}(x - x_i)^3 + \frac{z_i}{6h_i}(x_{i+1} - x)^3 + \left(\frac{y_{i+1}}{h_i} - \frac{h_i z_{i+1}}{6}\right)(x - x_i) + \left(\frac{y_i}{h_i} - \frac{h_i z_i}{6}\right)(x_{i+1} - x)$$

  - However, this is not the best possible form to evaluate the spline.

  - In order to evaluate it using the nested multiplication we have to massage it into the form

$$S_i(x) = A_i + B_i(x - x_i) + C_i(x - x_i)^2 + D_i(x - x_i)^3$$

Interpolation: piecewise interpolation

  - Notice that the equation above is in the form of Taylor expansion of S_i around point x_i, Hence,

$$A_i = S_i(x_i), \quad B_i = S_i'(x_i), \quad C_i = \frac{1}{2}S_i''(x_i), \quad D_i = \frac{1}{6}S_i'''(x_i)$$

  - Therefore

$$A_i = y_i \text{ and } C_i = \frac{z_i}{2}.$$

  - The coefficient of x^3 in the form above is D_i and in the previous form $(z_{i+1} - z_i)/6h_i$

$$\rightarrow \quad D_i = \frac{z_{i+1} - z_i}{6h_i}$$

  - Finally the value $S_i'(x_i)$ is give as

$$B_i = -\frac{h_i z_{i+1}}{6} - \frac{h_i z_i}{3} + \frac{y_{i+1} - y_i}{h_i}$$

  - The nested form of $S_i(x)$ is then finally – after a few steps of algebra:

$$S_i(x) = y_i + (x - x_i)\left\{B_i + (x - x_i)\left[\frac{z_i}{2} + (x - x_i)\frac{z_{i+1} - z_i}{6h_i}\right]\right\}$$

Interpolation: piecewise interpolation

- Then to the implementation.

  - The following routine solves the coefficients z_i given the input parameters x_i, y_i, N.

```
void Spline3_Coeff(int n,double *x,double *y,double *z)
{
     int i;
     double *h, *b, *u, *v;
     h = (double *) malloc((size_t) (n*sizeof(double)));
     b = (double *) malloc((size_t) (n*sizeof(double)));
     u = (double *) malloc((size_t) (n*sizeof(double)));
     v = (double *) malloc((size_t) (n*sizeof(double)));
     for (i=0; i<n; i++){
          h[i] = x[i+1]-x[i];
          b[i] = (y[i+1]-y[i])/h[i];
     }
     u[1]= 2.0*(h[1]+h[0]);
     v[1]= 6.0*(b[1]-b[0]);
     for (i=2; i<n; i++){
          u[i]= 2.0*(h[i]+h[i-1])-h[i-1]*h[i-1]/u[i-1];
          v[i]= 6.0*(b[i]-b[i-1])-h[i-1]*v[i-1]/u[i-1];
     }
     z[n]=0;
     for (i=n-1; i>0; i--){
          z[i] = (v[i]-h[i]*z[i+1])/u[i];
     }
     z[0]=0;
     free(h); free(b); free(u); free(v);
}
```

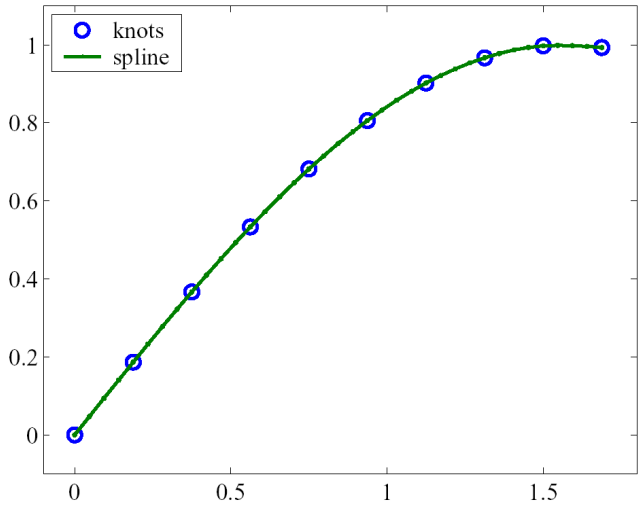

Interpolation: piecewise interpolation

- And the evaluation routine computes the spline value at `x0` given as an input argument.

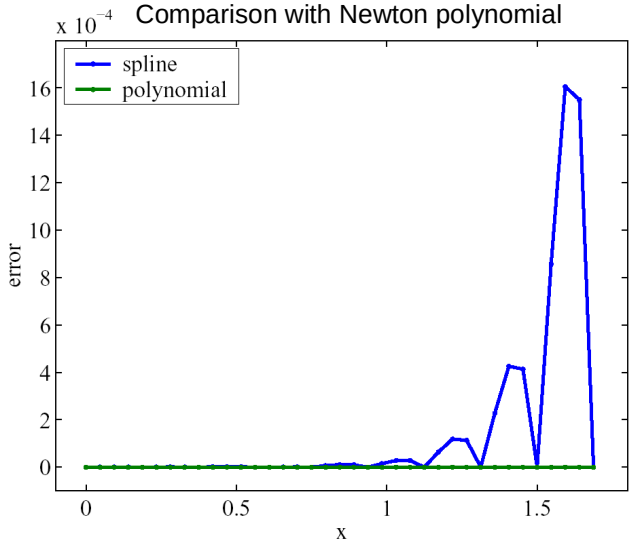
```
double Spline3_Eval(int n, double *x, double *y, double *z, double x0)
{
     int i;
     double h, tmp, result;
     for (i=n-1; i>=0; i--) {
          if (x0-x[i]>=0) break;
     }
     h = x[i+1]-x[i];
     tmp = 0.5*z[i] + (x0-x[i])*(z[i+1]-z[i])/(6.0*h);
     tmp = -(h/6.0)*(z[i+1]+2.0*z[i])+(y[i+1]-y[i])/h + (x0-x[i])*tmp;
     result = y[i] + (x0-x[i])*tmp;
     return(result);
}
```

Interpolation: piecewise interpolation

- Example: Function $\sin x$ at the interval $[0, 1.6875]$, with ten equidistant knots:



Comparison with Newton polynomial



Interpolation: piecewise interpolation

- Another example: The serpentine curve

$$y = \frac{x}{1/4 + x^2}$$

- And in parametric form

$$\begin{cases} x = \frac{1}{2}\tan\theta \\ y = \sin 2\theta \end{cases}$$

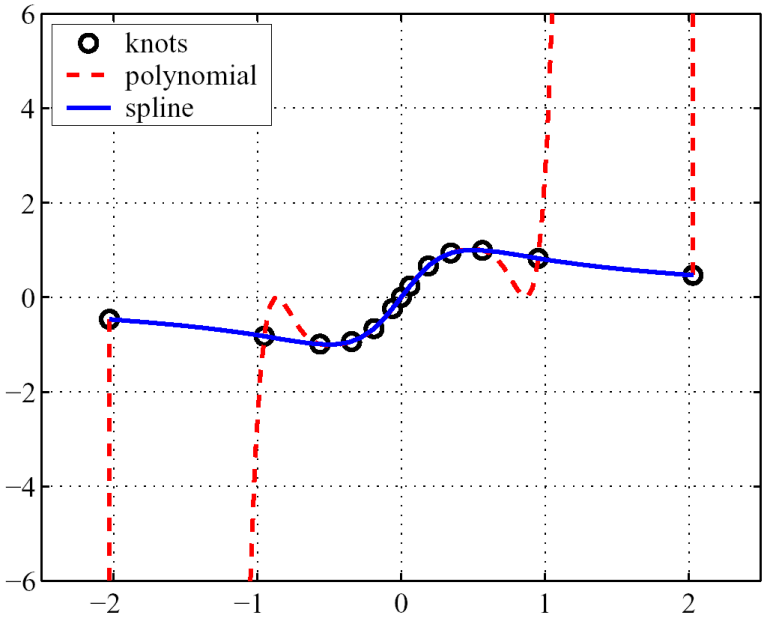
- Let's take knots evenly spaced in θ so that they are non-uniformly spaced in x axis:

$$\theta = n\frac{\pi}{12}, \quad n = -5, \ldots, 5$$

- We can interpolate the parametric form by constructing splines for both $x(\theta)$ and $y(\theta)$
- Note that the knots must be in ascending order.

Interpolation: piecewise interpolation

- Below are the results comparing the interpolation with a spline and with a polynomial.



Interpolation: Bézier curves

• One way of writing the interpolating polynomial is by using basis functions $b_i(x)$:

$$P(x) = \sum_{i=0}^{N} c_i b_i(x)$$

- Note that if we have one polynomial going through all data points it is unique and and does not depend on the basis functions used.

- As examples of basis functions we have used are monomials

$$b_i(x) = x^{i-1}, \quad i = 0, \ldots, N$$

and Lagrange's polynomials

$$b_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{N} \left(\frac{x - x_j}{x_i - x_j} \right), \quad i = 0, \ldots, N$$

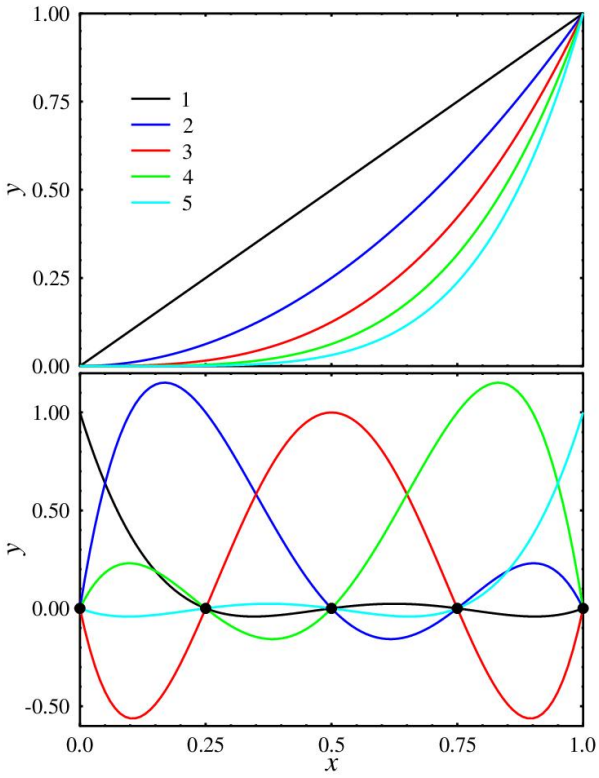
- Note that here the monomials do not depend on the data set but Lagrange's polynomials depend on the positions of the nodes $\{x_i\}$.

Interpolation: Bézier curves

- Graphically

Monomials

Lagrange



Interpolation: Bézier curves

• The Weierstrass approximation theorem:

For any continuous function f on a closed interval, say $[0, 1]$, and any $\varepsilon > 0$, there is a polynomial P such that for any $x \in [0, 1]$ $\quad |f(x) - P(x)| < \varepsilon$

- It was proven by Bernstein using polynomials $B_i^n(x)$:

$$\sum_{i=0}^{n} f\left(\frac{i}{n}\right) B_i^n(x) \to f(x) \ , \ \text{when } n \to \infty, \ x \in [0, 1]$$

where the **Bernstein polynomial** (BP) is

$$B_i^n(x) = \binom{n}{i}(1-x)^{n-i}x^i, \qquad \binom{n}{i} = \frac{n!}{i!(n-i)!}$$

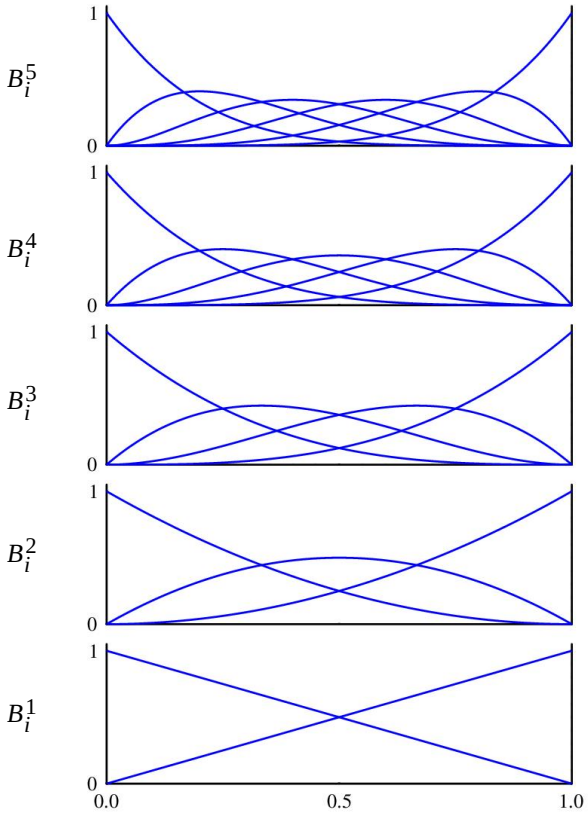
- These can be generalized to any interval $[a, b]$:

$$B_i^n(x) = \binom{n}{i}\frac{(b-x)^{n-i}(x-a)^i}{(b-a)^n}$$

- It is easy to show that the polynomial $B_i^n(x)$ has a maximum at the position $x = a + \frac{i}{n}(b-a)$

Interpolation: Bézier curves

- Below are examples of various BP's at the interval $[0, 1]$



$B_0^3(x) = (1-x)^3 \qquad B_1^3(x) = 3x(1-x)^2 \qquad B_2^3(x) = 3x^2(1-x)$

$B_3^3(x) = x^3$

$B_0^2(x) = (1-x)^2 \qquad B_1^2(x) = 2x(1-x) \qquad B_2^2(x) = x^2$

$B_0^1(x) = 1-x \qquad B_1^1(x) = x$

Interpolation: Bézier curves

- BP's are often used as a basis for interpolation of function $f(x)$:

$$P_N(x) = \sum_{i=0}^{N} p_i B_i^N(x)$$

where coefficients p_i are obtained from interpolation conditions:

$$P_N(x_i) = f(x_i) , \quad i = 0, 1, \ldots, N$$

- Why are BP's useful?

Coefficients p_i give a hint of how the interpolate P_N looks like.

- Let's associate with each p_i a **control point** (t_i, p_i) such that

$$t_i = a + \frac{i}{n}(b-a)$$

- Note that $(t_0, p_0) = (x_0, y_0)$ and $(t_N, p_N) = (x_N, y_N)$ but the control points between these limits do not necessarily correspond to the data set $\{x_i, y_i\}$, $i = 1, \ldots, N-1$.

Interpolation: Bézier curves

- In Fortran the interpolation is done as follows:

```
program bernstein
 implicit none
  integer,parameter ::  &
      rk=selected_real_kind(12,100)

  real(rk),allocatable :: b(:),p(:)
  real(rk) :: x,y,dx
  integer :: iargc,n,mx,i,m,k
  character(len=80) :: argu,file1,file2

  if (iargc()<5) then
     write(0,'(a,a)') 'usage: bernstein ',&
    'cpointfile interpfile dx n p0 p2 p3 .. pn'
     stop
  end if

  call getarg(1,file1)
  call getarg(2,file2)
  open(100,file=file1,status='new')
  open(200,file=file2,status='new')
  call getarg(3,argu); read(argu,*) dx
  call getarg(4,argu); read(argu,*) n
  allocate(b(0:n),p(0:n))
  do i=0,n
     call getarg(i+5,argu); read(argu,*) p(i)
     write(100,'(2g18.10)') &
       real(i,rk)/real(n,rk),p(i)
  end do
```

```
  mx=1.0/dx

  do m=0,mx
     x=dx*m
     y=0.0
     call bernstein_poly(n,x,b)
     do i=0,n
        write(i+10,'(2g18.10)') x,b(i)
        y=y+p(i)*b(i)
     end do
     write(200,'(2g18.10)') x,y
  end do

  close(100)
  close(200)

  stop

end program bernstein
```


Interpolation: Bézier curves

- Program uses the **polpak** library for calculating Bernstein polynomials.
  - The library is available in Fortran90, C and Matlab and can be downloaded from

    http://www.csit.fsu.edu/~burkardt/f_src/polpak/polpak.html

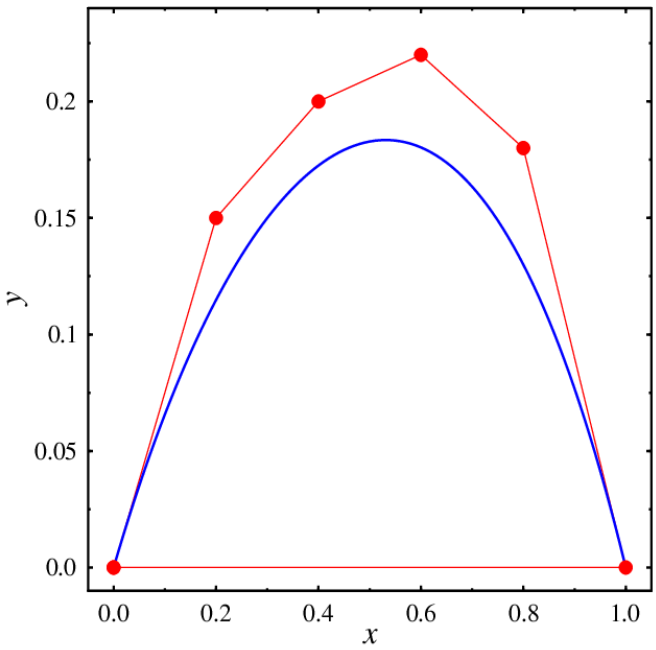
  - Compiling and linking a program using the library is easy (after you have created the library itself and assuming the library is in the current directory):

```
gfortran -o bezier bezier.f90 -L. -lpolpak
```

Interpolation: Bézier curves

- Now plot the control points $\{t_i, p_i\}$, $i = 0, \ldots, N$ and the interpolating polynomial $P_N(x)$

  - For example:



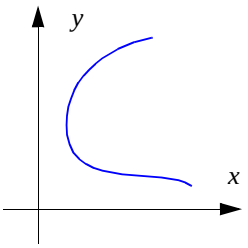
- This means that by specifying the control points we can easily construct a smooth curve that resembles a line going through those points.

- The more we have points the nearer the curve comes to the control points.

- Note that here we are no more talking about interpolation.
  - The polynomial does not go through the control points.

Interpolation: Bézier curves

- What if we have an arbitrary curve on the x, y plane (i.e. not a function $y = f(x)$)?



  - We can express the curve in the parametric form

$$\begin{cases} x = x(t) \\ y = y(t) \end{cases}, \qquad t \in [0, 1] \quad \text{(for example)}$$

  - If the set of control points is $\{x_i, y_i\}$ the interpolated curve is

$$\begin{cases} x = P_N^x(t) \\ y = P_N^y(t) \end{cases} \quad \text{where} \quad \begin{cases} P_N^x(t) = \sum_{i=0}^{N} x_i B_i^N(t) \\ P_N^y(t) = \sum_{i=0}^{N} y_i B_i^N(t) \end{cases}$$

  - This curve is called the **Bézier curve**

  - Note that $\{x_i, y_i\}$ are now the coefficients of the 'interpolating' polynomials.

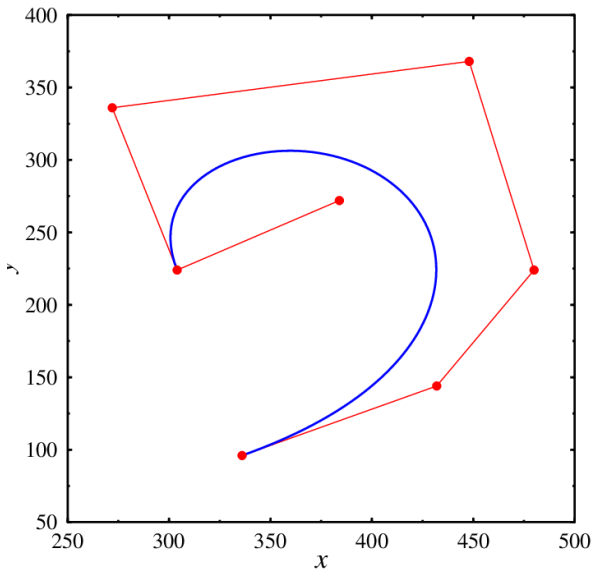
Interpolation: Bézier curves

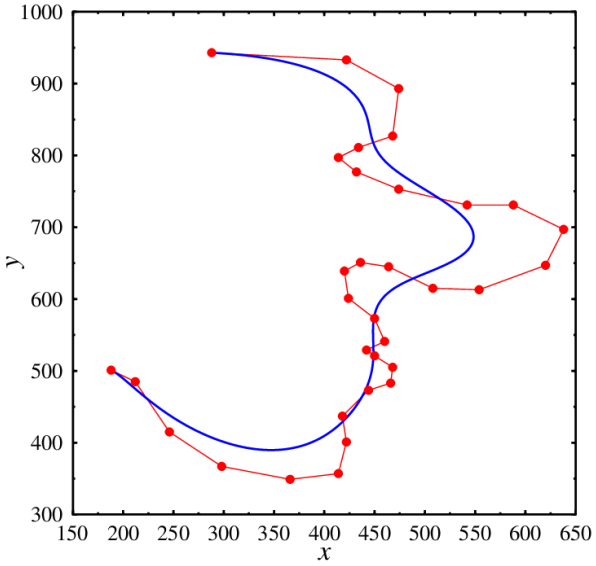
- The previous program only needs a little modification to compute the Bézier curve:

```
program bezier
  implicit none
  integer,parameter ::&
     & rk=selected_real_kind(12,100)

  real(rk),allocatable :: b(:),xa(:),ya(:)
  real(rk) :: x,y,t,dt
  integer :: iargc,n,mt,i,m,k
  character(len=80) :: argu,file1,file2

  if (iargc()/=2) then
     write(0,'(a)') 'usage: bezier n dt'
     stop
  end if

  call getarg(1,argu); read(argu,*) n
  call getarg(2,argu); read(argu,*) dt
  n=n-1
  allocate(b(0:n),xa(0:n),ya(0:n))
  do i=0,n
     read(5,*) xa(i),ya(i)
  end do
  mt=1.0/dt

  do m=0,mt
     t=dt*m
     x=0.0
     y=0.0
     call bernstein_poly(n,t,b)
     do i=0,n
        y=y+ya(i)*b(i)
     end do
     call bernstein_poly(n,t,b)
     do i=0,n
        x=x+xa(i)*b(i)
     end do
     write(6,'(2g18.10)') x,y
  end do

  stop

end program bezier
```


Interpolation: Bézier curves

- Two examples:





Interpolation: B-splines

• The basis functions $b_i(x)$ of the interpolate (note the slight change in notation: $N \to k$)

$$P(x) = \sum_{i=0}^{k} c_i b_i(x)$$

may also be splines itself: they are so called **B-splines** (B for basis).

- A B-spline of order k is denoted as $N_i^k(x)$ and it depends on the set of nodes x_i.

  For example in the case of **cubic B-splines** the functions $N_i^3(x)$ have two continuous derivatives and span four points around x_i.

- B-splines can be computed recursively

$$N_i^0(x) = \begin{cases} 1 & x_i \le x < x_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$N_i^k(x) = \left(\frac{x - x_i}{x_{i+k} - x_i}\right) N_i^{k-1}(x) + \left(\frac{x_{i+k+1} - x}{x_{i+k+1} - x_{i+1}}\right) N_{i+1}^{k-1}(x) \ , \quad \text{where } k = 1, 2, \ldots \text{ and } i = 0, \pm 1, \pm 2, \ldots$$

- B-splines can be used in a similar fashion as Bézier curves.
  - With B-splines one can model continuous curves with points where derivatives are discontinuous.

- For more information see Kahaner, Moler, Nash: *Numerical Methods and Software*, or Haataja *et al*., *Numeeriset menetelmät käytännössä*.

Interpolation: 2D

• Let the function depend on two variables: $y = f(x_1, x_2)$

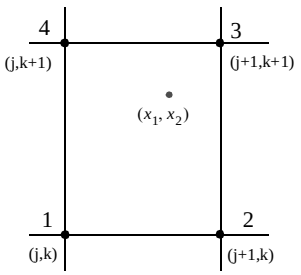
- The interpolation data is now a set of points in a rectangular grid $\{y_{ij}, x_1^{(i)}, x_2^{(j)}\}$, $i = 1, \ldots, m$, $j = 1, \ldots, n$

- Let's use the following notation:

$$\texttt{ya(j,k)} = y(\texttt{x1a(j)}, \texttt{x2a(k)})$$

- First we must find the right position in the grid where

$$\texttt{x1a(j)} \le x_1 \le \texttt{x1a(j+1)}$$
$$\texttt{x2a(k)} \le x_2 \le \texttt{x2a(k+1)}$$



- Also let the function values at the corners be

$$y_1 \equiv \texttt{ya(j,k)}$$
$$y_2 \equiv \texttt{ya(j+1,k)}$$
$$y_3 \equiv \texttt{ya(j+1.k+1)}$$
$$y_4 \equiv \texttt{ya(j,k+1)}$$

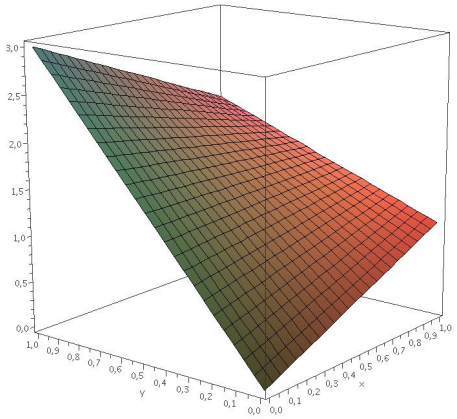
Interpolation: 2D

- The desired y value is obtained by using **bilinear interpolation**:

$$t = \frac{x_1 - \texttt{x1a(j)}}{\texttt{x1a(j+1)} - \texttt{x1a(j)}}$$

$$u = \frac{x_2 - \texttt{x2a(k)}}{\texttt{x2a(k+1)} - \texttt{x2a(k)}}$$

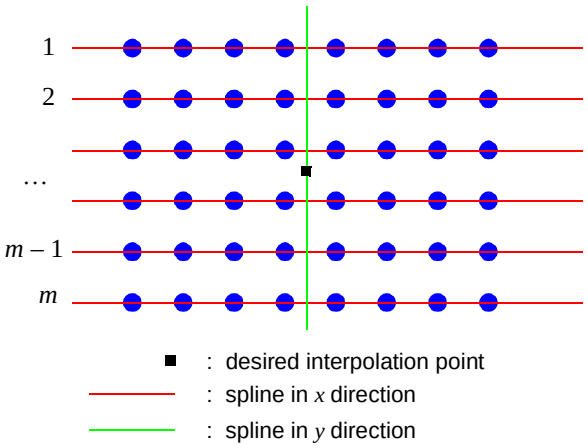
$$y(x_1, x_2) = (1-t)(1-u)y_1 + t(1-u)y_2 + tuy_3 + (1-t)uy_4$$



- For higher accuracy/smoothess one can generalize the spline interpolation:

  1. Construct m 1D interpolates in x direction.
  2. Based on these construct one 1D interpolate at the desired y value.

- For more details see e.g. Numerical Recipes paragraph 3.6



Interpolation: practical tools

• If you are using C then GSL has many interpolation routines:

Spline interpolation

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_spline.h>

int main (int argc, char **argv)
{
  int i,nk,np;
  double *x,*y,x1,x2,dx,xi,yi;
  gsl_interp_accel *acc;  //Index look-up acceleration
  gsl_spline *spline;
  if (argc!=4) {
    fprintf(stderr,"Usage: %s x1 x2 dx\n",argv[0]); return (1);
  }
  x1=atof(*++argv);a
  x2=atof(*++argv);
  dx=atof(*++argv);
  fscanf(stdin,"%d",&nk);
  x = (double *) malloc((size_t) (nk*sizeof(double)));
  y = (double *) malloc((size_t) (nk*sizeof(double)));
  for (i=0;i<nk;i++) fscanf(stdin,"%lg %lg",&x[i],&y[i]);

  acc = gsl_interp_accel_alloc();
  spline = gsl_spline_alloc(gsl_interp_cspline,nk);
  gsl_spline_init(spline,x,y,nk);

  np=(x2-x1)/dx;
  for (i=0;i<np;i++) {
    xi=x1+dx*i;
    yi=gsl_spline_eval(spline,xi,acc);
    printf("%g %g\n",xi,yi);
  }
  gsl_spline_free(spline);
  gsl_interp_accel_free(acc);
  return 0;
}
```

```
BEGIN {
  srand();
  x1=0;
  x2=100.0;
  dx=1.0;
  n=(x2-x1)/dx+1;
  print n;
  for (i=0;i<n;i++) {
    x=x1+dx*i;
    y=rand();
    print x,y;
    print x,y > "interpdata.dat";
  }
}
```

```
gsl> gcc gsl_spline.c -lgsl -lgslcblas
gsl> gawk -f makeinterpdata.awk |a.out -1 101 0.01 > interpdata.int
gsl> xgraph -P interpdata.*
```

Interpolation: practical tools

• For Fortran users the is e.g. the SLATEC package: http://www.netlib.org/slatec/

- At http://www.csit.fsu.edu/~burkardt/f_src/slatec/slatec.html you can find some documentation and test programs.

- Installed on punk.it.helsinki.fi at `/usr/local/lib` (linking: `gfortran prog.f90 -lslatec`)

- If you want to build the library yourself

1) Download the package: http://www.physics.helsinki.fi/courses/s/tl3/progs/slatecpackage.tgz

2) In files `f90split.csh` and `slatec.csh` change the appropriate string for the Fortran compiler:
`set FORT=gfortran` (or `f90`, ...)

3) Split the one large file to smaller pieces and build the library:
`./f90split.csh; ./slatec.csh`

• Matlab has the function `interp1` that does 1D interpolation using the method of choice (nearest, linear, spline, ...)

• Function `spline` computes the spline coefficients and they can be used to evaluate using function `ppval`:

```
pp=spline(x,y)
yi=ppval(pp,xi)
```


Interpolation: practical tools

- A simple example using SLATEC:

```
program spline_slatec
  implicit none
  integer,parameter :: rk=selected_real_kind(15,100)
  real(rk),allocatable :: xa(:),ya(:),w(:),t(:),bcoef(:)
  real(rk) ::  x,y,x1,x2,dx,fbcl,fbcr
  integer :: ibcl,ibcr,kntopt,ncoef,k,ideriv,inbv
  integer :: n,i,j,ix,ixmax
  character(len=80) :: argu
  real(rk),external :: dbvalu

  if (iargc()/=3) then
     call getarg(0,argu)
     write(0,'(a,a,a)') 'usage: ',trim(argu),' x1 x2 dx'
     stop
  end if
  call getarg(1,argu); read(argu,*) x1
  call getarg(2,argu); read(argu,*) x2
  call getarg(3,argu); read(argu,*) dx

  read(5,*) n
 allocate(xa(1:n),ya(1:n),w(1:5*(n+2)),t(1:n+4),bcoef(1:n+2))
  do i=1,n
     read(5,*) xa(i),ya(i)
  end do

  ibcl=2
  ibcr=2
  fbcl=0.0
  fbcr=0.0
  kntopt=2
  k=4
  w=0.0

  call dbint4(xa,ya,n,ibcl,ibcr,fbcl,fbcr,&
                      & kntopt,t,bcoef,ncoef,k,w)

  ixmax=(x2-x1)/dx
  ideriv=0
  inbv=1
  do ix=1,ixmax
     x=x1+dx*ix
     y=dbvalu(t,bcoef,n,k,ideriv,x,inbv,w)
     write(6,*) x,y
  end do

  stop
end program spline_slatec
```

```
spline> ifort spline_slatec.f90 -lslatec -L../../slatec/
spline>  a.out  1.0  8.0  0.01  <  sin_per_1+x2.dat  >
sin_per_1+x2.interp
```