Technische Universität Ilmenau
Institut für Mathematik

Preprint No. M 00/10

MATLAB - Teil III; Komplexe LGS, Interpolation, Splines

Neundorf, Werner

2000

Impressum:
Hrsg.: Leiter des Instituts für Mathematik
Weimarer Straße 25
98693 Ilmenau
Tel.: +49 3677 69 3621
Fax: +49 3677 69 3270
http://www.tu-ilmenau.de/ifm/

ISSN xxxx-xxxx

ilmedia

Zusammenfassung

This is a tutorial on teaching and programming in MATLAB.
It is based on scripts and exercises in the course of numerical mathematics for students of the faculties Electrical Engineering and Information Technology and Computer Science and Automation after first term.
The part I contains basic aspects and elements of numerical linear algebra, especially methods for systems of linear equations. The second part gives some aspects about storage and import/export of data files, furthermore MATLAB-based algorithms and programming tools for special systems of linear equations, for eigenvalue problems and singular value decomposition, graphic aspects, nonlinear equations and systems of equations.
This third part concerns systems of complex linear equations, polynomial and spline interpolations.

Vorwort

MATLAB is an interactive, matrix-based system for scientific and engineering calculations. You can solve complex numerical problems without actually writing a program. The name MATLAB is an abbreviation for MATrix LABoratory.

A few words to those who are familiar with other programming languages.

- MATLAB is a user-friendly high-level programming language and important technical computing environment. It also includes a number of functional programming constructs, modeling, simulation and prototyping, application development and design.

- MATLAB has a rich environment of powerful toolboxes and data visualization. It offers programming structures like control flows, selections, decisions and m-files functions.

- Students can affordably use this powerful numeric computation, data analysis and visualization software in their undergraduate and graduate studies. The student edition encapsulates a wide range of disciplines.

- MATLAB is not strongly typed like C and Pascal. No declarations are required. It is more like Basic and Lisp in this respect. You can dynamically link C or FORTRAN subroutines. Some type checking is done at run time.

- MATLAB suitable for running numerically intensive programs with double-precision numerical calculations. On the other site there are, based on Maple V, many symbolic tools and symbolic functions to combine, simplify, differentiate, integrate, and solve algebraic and differential equations. The symbolic toolbox and the subroutines concept of MATLAB seems to be not so efficiently as is described in Maple.

- MATLAB is available for a number of environments: Sun/Apollo/VAXstation/HP workstations, VAX, MicroVAX, Gould, PC, Apple Macintosh, and several parallel machines.

The aim is to show how you can write simple instructions, commands or programs in MATLAB for doing numerical calculations, linear algebra, and programs for simplifying or transforming expressions, equations, mathematical formulas or arrays.

It is assumed that the reader is familiar with using MATLAB interactively. For beginners we propose the introductional tutorial, the so called Primer. The purpose of this Primer based on the version 3.5 is to help you begin to use MATLAB. They can best be used hands-on. You are encouraged to work at the computer as you read the Primer and freely experiment with examples.

This document is based on **MATLAB version 4.2c1**.
MATLAB development continues. New versions come out every one or two years which contain not only changes to the mathematical capabilities of MATLAB, but also changes to the programming language and user interface. The MATLAB 5.2 and 5.3 highlights you can find on the web site

`http://www.mathworks.com/products/matlab/highlights.shtml` .

We are pleased and somewhat surprised to see how quickly this movement is already happening. For this reason, I have applied constructs in the language that will be probable in the language in future versions of MATLAB.

You should liberally use the on-line help facility for more detailed information. After entering MATLAB the command `help` will display a list of functions for which on-line help is available. The command `help` *functionname* will give information about a specific function. You can preview some of the features of MATLAB by entering the command `demo`. Even better, access help from the menu.
In the bibliography there are given some useful sources of information and supplemental workbooks for MATLAB.

Inhaltsverzeichnis

1 Komplexe lineare Gleichungssysteme

Lösungsalgorithmen für reelle LGS und ihre Komplexität sind in [9] vorgestellt und diskutiert worden. An dieser Stelle soll noch eine kurze Behandlung des komplexen Falls erfolgen.
In MATLAB ist die direkte Lösung komplexer LGS $Ax = b,\ A = A(n,n)$, möglich. Natürlich kann man diese auch auf ihre reelle Form transformieren und dann erst lösen. Mit $A = A_1 + i\ A_2,\ b = b_1 + i\ b_2,\ x = x_1 + i\ x_2$ erhält man das reelle Gleichungssystem der Dimension $2n$

$$\begin{pmatrix} A_1 & -A_2 \\ A_2 & A_1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}.$$

Interessant ist ein Vergleich von Rechnungen im Komplexen und Reellen. Für die Komplexität und den Geschwindigkeitstest verwenden wir in MATLAB die zwei Varianten der Bestimmung der Anzahl von durchgeführten Gleitpunktoperationen mit `flops` (floating point operations) sowie der Zeitmessung mit `clock, etime`.
Zunächst ist von Interesse die Zählung von Gleitpunktoperationen bei den komplexen Grundoperationen. Wegen

$$\begin{aligned} (x_1 + i\ x_2) \pm (y_1 + i\ y_2) &= (x_1 \pm y_1) + i\ (x_2 \pm y_2) \\ (x_1 + i\ x_2) * (y_1 + i\ y_2) &= (x_1 * y_1 - x_2 * y_2) + i\ (x_1 * y_2 + x_2 * y_1) \end{aligned}$$

stellt man fest, daß die Addition/Subtraktion von 2 komplexen Zahlen 2 *flops*, die Multiplikation/Division 6 *flops* kostet. Sobald irgendein Operand komplex ist, werden die Operationen im Komplexen ausgeführt.

```
flops(0);
1+i

ans =
   1.0000 + 1.0000i

flops
ans =
     2

flops(0);
1*i

ans =
        0 + 1.0000i

flops
ans =
     6
```

Tabelle mit komplexen Ausdrücken und *flops*

Ausdruck	Ergebnis	Zählung der Operationen	*flops*
i	`0 + 1.0000i`		0
$1+i$	`1.0000 + 1.0000i`	2+ $(1+0)+i(0+1)$	2
$1*i$	`0 + 1.0000i`	4∗, 1-, 1+ $(1*0-0*1)+i(1*1+0*0)$	6
$(1+i)+i$	`1.0000 + 2.0000i`	2+, 2+ $(1+0)+i(0+1)=\alpha+i\beta$ $(\alpha+0)+i(\beta+1)$	4
$(1+i)-(3-i)$	`-2.0000 + 2.0000i`	2+, 2-, 2- $(1+0)+i(0+1)=\alpha+i\beta$ $(3-0)+i(0-1)=\gamma+i\delta$ $(\alpha-\gamma)+i(\beta-\delta)$	6
$1/i$	`0 - 1.0000i`	6/	6
$1/(2*i)$	`0 - 0.5000i`	6∗, 6/	12
$1/(1-2*i)$	`0.2000 + 0.4000i`	6∗, 2-, 6/	14
$(3+4*i)+(1-2*i)$	`4.0000 + 2.0000i`	6∗, 6∗, 2+, 2-, 2+	18
$(3+4/i)+(1-2*i)$	`4.0000 - 6.0000i`	6/, 6∗, 2+, 2-, 2+	18
$(3+4/i)*(1-2*i)$	`-5.0000 -10.0000i`	6/, 6∗, 2+, 2-, 6∗	22
$(3+4*i)*(1-2*i)$	`11.0000 -2.0000i`	6∗, 6∗, 2+, 2-, 6∗	22
$(3+4*i)/(1-2*i)$	`-1.0000 +2.0000i`	6∗, 6∗, 2+, 2-, 6/	22

1.1 Komplexität von Algorithmen zur Lösung von LGS

Wir vergleichen hier die Operationszahlen und Rechenzeiten von 5 Algorithmen wie in [9]. Die Rechnungen wurden auf einem PC im Netz mit Pentium II Prozessor 350MHz durchgeführt.

$Ax=b$, AB erweiterte Koeffizientenmatrix

$[L,U]=lu(A)$; $U\backslash(L\backslash b)$

$AT=gaussel(AB)$; x mit Rückwärtseinsetzen

$A\backslash b$

$rref(AB)$

$inv(A)*b$

Die verschiedenen Ergebnisse werden wieder so abgespeichert, um sie u.a. als vergleichendes Balkendiagramm darzustellen.

Die Komplexität im Reellen bei Dimension n für die Verfahren *lu*, *gaussel* und $A\backslash b$ hat die Größenordnung $\mathcal{K} = \mathcal{O}(\frac{2}{3}n^3)$. Beim Zugang über die inverse Matrix beträgt die Komplexität im Operationsmix $(+,*)$ $\mathcal{O}(\frac{n^3}{3} + n \cdot n^2 - \frac{n}{3})$ (n rechte Seiten), was auf die Größenordnung $\mathcal{O}(2n^3)$ führt.

Nun führen wir komplexe Rechnungen bei Dimension n und die zugehörigen reellen Rechnungen mit doppelter Dimension durch.

```
title('SPEEDTEST for SOLUTION of Ax=b for n = 20x20 .. 100x100 ')
clear n t s sn ns tn nt q tt ss t2 s2 q2

h = 1;    % Skalierung von Balkenabstand
na = 20;
nsw = 20;
ne = 100;
for n = na:nsw:ne
  A = rand(n)+n*eye(n)+i*rand(n);
  b = rand(n,1)+i*rand(n,1);
  AB = [A b];

  A2 = [real(A)  -imag(A); imag(A)  real(A)];
  b2 = [real(b); imag(b)];
  AB2 = [A2 b2];
  %
  t0 = clock;
  flops(0);
  [L U] = lu(A);
  U\(L\b);
  t(n) = etime(clock,t0);
  s(n) = flops;
  q(n) = 0;
  if (t(n)~=0), q(n) = s(n)./t(n); end;
  %
  t0 = clock;
  flops(0);
  [L2 U2] = lu(A2);
  U2\(L2\b2);
  t2(n) = etime(clock,t0);
  s2(n) = flops;
  q2(n) = 0;
  if (t2(n)~=0), q2(n) = s2(n)./t2(n); end;
  %
  % weitere Verfahren
  ......
end
```

Die Generierung des reellen LGS aus dem komplexen kostet zwar etwas Zeit, aber braucht keine *flops*.
Die Anzahl der *flops* ist im Komplexen ungefähr halb so groß.
Rechenzeitmäßig schneiden die komplexen Algorithmen noch besser ab.

Ausgewählte Ergebnisse

```
title('SPEEDTEST for SOLUTION of Ax=b for n = 20x20 .. 100x100')

Complex(n)
flops
           n          lu     gaussel         A\b        rref         inv
          20       24476       24373       35224      142617       77564
          40      182536      182723      224688      627614      564568
          60      602196      603073      696824     1632935     1845444
          80     1411456     1413423     1581608     3311517     4306168
         100     2738316     2741773     2998928     5862111     8322628

times
         n        lu   gaussel       A\b      rref       inv
   20.0000         0    0.0600         0    0.5500         0
   40.0000         0    0.5000         0    1.8700         0
   60.0000    0.0500    1.7100         0    4.2300         0
   80.0000    0.0600    4.1700    0.0600    7.8000    0.0500
  100.0000    0.0500    7.9100    0.0500   12.2000    0.1600


Real(2n)
flops
           n          lu     gaussel         A\b        rref         inv
          20       45238       46104       53554      135600      138134
          40      351258      354584      385072      758934     1064632
          60     1174078     1181464     1250626     2232994     3547566
          80     2769698     2782744     2906126     4939336     8354846
         100     5394118     5414424     5607756     9244228    16254656

times
         n        lu   gaussel       A\b      rref       inv
   20.0000         0    0.4900         0    1.4300         0
   40.0000         0    3.4600    0.0500    5.3800    0.0600
   60.0000         0   11.5900    0.0500   12.1400    0.1100
   80.0000    0.1100   30.3200    0.1100   22.1900    0.2800
  100.0000    0.1100   56.5800    0.1100   34.3800    0.2200
```

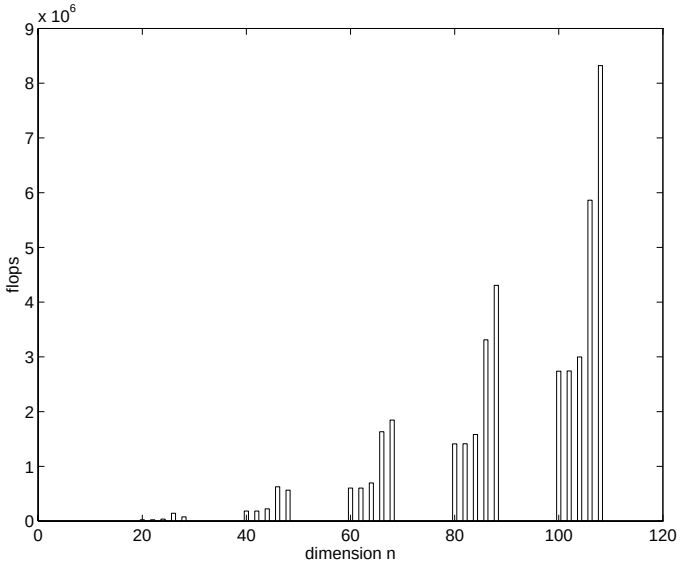
Berechnungen im Komplexen erweisen sich somit als günstiger.
Dazu machen wir folgende Gegenüberstellung der Operationsanzahl.

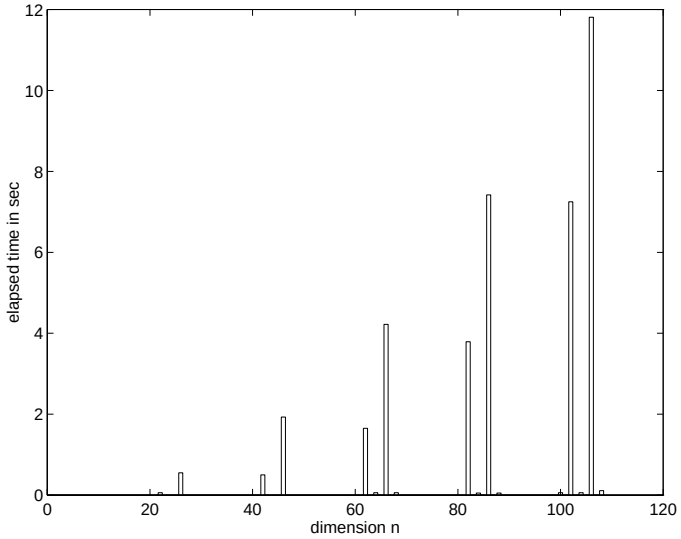
Rechnung	Methode	*flops*
Komplex (n)	LU	$6\frac{n^3}{3}\text{“}*\text{“} + 2\frac{n^3}{3}\text{“}+\text{“} = \frac{8n^3}{3}$
	$A^{-1}b$	$6n^3\text{“}*\text{“} + 2n^3\text{“}+\text{“} = 8n^3$
Reell $(N = 2n)$	LU	$\frac{N^3}{3}\text{“}*\text{“} + \frac{N^3}{3}\text{“}+\text{“} = \frac{16n^3}{3}$
	$A^{-1}b$	$2N^3 = 16n^3$

Einige graphische Auswertungen zur Komplexität im Komplexen bei $Ax = b$.

flops und *elapsed time in sec* in Abhängigkeit von n.

Balken v.l.n.r.: lu, $gaussel$, $A\backslash b$, $rref$, inv.





1.2 Komplexität von Algorithmen zur Bestimmung der Inversen

Auch hier vergleichen wir die Operationszahlen und Rechenzeiten von 4 Algorithmen. Die Rechnungen wurden auf einem PC mit Pentium II Prozessor 350MHz durchgeführt.

A^{-1}, AE um Einheitsmatrix I erweiterte Koeffizientenmatrix

$inv(A)$
$A\backslash I$
$[L,U] = lu(A);\ U\backslash(L\backslash I)$
$rref(AE)$

```
title('SPEEDTEST for INVERSION of MATRICES n = 20x20 .. 100x100 ')
clear n t s sn ns tn nt q tt ss t2 s2 q2
echo off

h = 1;
na = 20; nsw = 20; ne = 100;
for n=na:nsw:ne
  A = rand(n)+n*eye(n)+i*rand(n);
  I = eye(n);
  AE = [A I];

  A2 = [real(A)  -imag(A); imag(A)  real(A)];
  I2 = [I zeros(n,n); zeros(n,n) I];
  AE2 = [A2 I2];

  %
  t0 = clock;
  flops(0);
  inv(A);
  t(n) = etime(clock,t0);
  s(n) = flops;
  q(n) = 0;
  if (t(n)~=0), q(n) = s(n)./t(n); end;
  %
  t0 = clock;
  flops(0);
  inv(A2);
  t2(n) = etime(clock,t0);
  s2(n) = flops;
  q2(n) = 0;
  if (t2(n)~=0), q2(n) = s2(n)./t2(n); end;
  % weitere Verfahren
  ......
end
```

Die alleinige Invertierung der Matrix im Komplexen hat diesselbe Größenordnung der Komplexität wie die Bestimmung der Lösung eines LGS mittels $inv(A) * b$, also

$$\mathcal{K} = \mathcal{O}(8n^3) \quad (2n^3 \Rightarrow \{n^3*, n^3+\} \Rightarrow \{6n^3*, 2n^3+\} \Rightarrow\ 8n^3).$$

Die Kommandos $A\backslash I$ und lu zeigen sowohl für $flops$ als auch in der Rechenzeit ähnliches Verhalten, sind aber nicht so gut wie inv. Ihre Komplexität ist

$$\mathcal{O}\left(\frac{32}{3}n^3\right) \quad \left(\frac{8}{3}n^3 \Rightarrow \left\{\frac{4}{3}n^3*, \frac{4}{3}n^3+\right\} \Rightarrow \left\{6\frac{4}{3}n^3*, 2\frac{4}{3}n^3+\right\} \Rightarrow\ \frac{32}{3}n^3\right).$$

$rref$ ist im Vergleich die schlechteste Variante.
Die Anzahl der $flops$ ist im Komplexen auch hier ungefähr halb so groß.

Ausgewählte Ergebnisse

```
title('SPEEDTEST for INVERSION of MATRICES n = 20x20 .. 100x100 ')
clear n t s sn ns tn nt q tt ss t2 s2 q2
echo off

Complex(n)
flops
```

n	inv	A\I	lu	rref
20	74232	95136	89840	210625
40	551556	720560	700460	1169862
60	1816128	2388432	2343880	3421092
80	4254676	5613480	5532100	7530277
100	8246576	10903080	10777120	14048599

```
times
```

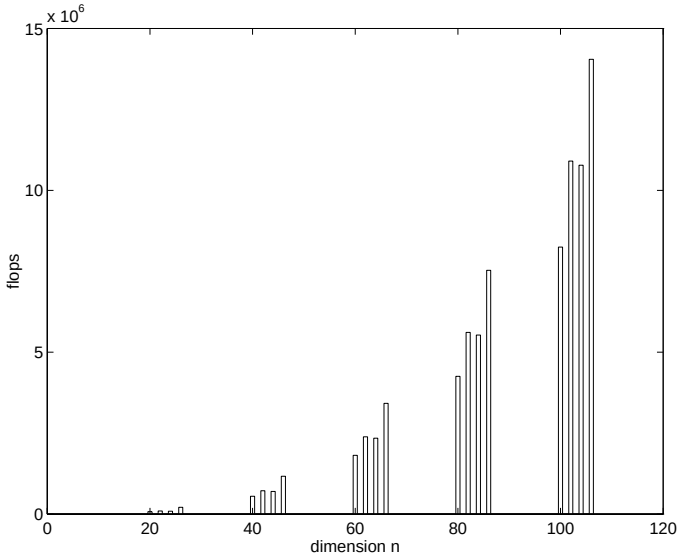
n	inv	A\I	lu	rref
20.0000	0	0	0	0.6600
40.0000	0	0.0500	0.0600	2.5800
60.0000	0	0.0500	0.0500	5.9900
80.0000	0.0500	0.0600	0.1700	11.0400
100.0000	0.1700	0.2200	0.2200	17.4100

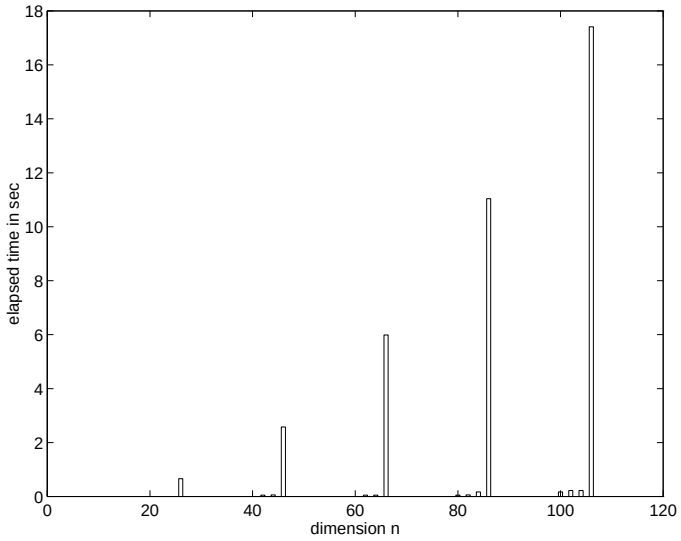
```
Real(2n)
flops
```

n	inv	A\I	lu	rref
20	134984	176848	176282	279172
40	1051860	1389984	1387742	1849980
60	3518720	4663504	4658402	5845912
80	8303816	11021660	11012262	13405526
100	16174806	21488110	21473322	25676272

```
times
         n        inv        A\I         lu       rref
   20.0000          0          0          0     2.0300
   40.0000          0          0          0     8.6200
   60.0000     0.1100     0.1700     0.1600    19.6100
   80.0000     0.1100     0.1600     0.2700    36.2000
  100.0000     0.4400     0.4900     0.8800    60.4200
```

Einige graphische Auswertungen zur Komplexität im Komplexen bei A^{-1}.
flops und *elapsed time in sec* in Abhängigkeit von n.
Balken v.l.n.r.: inv, $A\backslash I$, lu, $rref$.





2 Interpolation

2.1 Schemata für Polynome

- **Ermittlung der Normalform des Polynoms** $p_n(x) = \sum_{i=0}^{n} a_i x^{n-i}$

  aus der Newtonschen Darstellung

$$\begin{aligned} p_n(x) &= c_0 + c_1(x-x_0) + c_2(x-x_0)(x-x_1) + ... \\ &\quad + c_n(x-x_0)(x-x_1)\cdot ... \cdot (x-x_{n-1}) \\ &= c_0 + (x-x_0)[c_1 + (x-x_1)[c_2 + ... + (x-x_{n-2})[c_{n-1} + (x-x_{n-1})c_n]...]] \end{aligned}$$

  Anwendung des inversen Hornerschemas

	c_n					
$-x_{n-1}$	$a_0^{(0)} = c_n$	$a_1^{(0)} = c_{n-1}$ $-x_{n-1}a_0^{(0)}$				
$-x_{n-2}$	$a_0^{(1)}$	$a_1^{(1)}$ $-x_{n-2}a_0^{(1)}$	$a_2^{(1)} = c_{n-2}$ $-x_{n-2}a_1^{(1)}$			
$-x_{n-3}$	$a_0^{(2)}$	$a_1^{(2)}$	$a_2^{(2)}$			
	.....					
$-x_0$	$a_0^{(n-1)}$	$a_1^{(n-1)}$ $-x_0a_0^{(n-1)}$	$a_2^{(n-1)}$ $-x_0a_1^{(n-1)}$	... ...	$a_{n-1}^{(n-1)}$ $-x_0a_{n-2}^{(n-1)}$	$a_n^{(n-1)} = c_0$ $-x_0a_{n-1}^{(n-1)}$
	$a_0^{(n)}$	$a_1^{(n)}$	$a_2^{(n)}$	...	$a_{n-1}^{(n)}$	$a_n^{(n)}$
	a_0	a_1	a_2	...	a_{n-1}	a_n

  In den jeweiligen Spalten wird die Summe gebildet, d.h. für $k = 1, 2, ..., n$

$$a_0^{(k)} = a_0^{(k-1)}, \quad a_j^{(k)} = a_j^{(k-1)} + (-x_{n-k}\ a_{j-1}^{(k-1)}),\ j = 1, 2, ..., k.$$

  Die letzte Zeile und somit die Koeffizienten der Normalform ergeben sich zu $a_i = a_i^{(n)}$.

  Anwendung des Schemas auch in den speziellen Fällen

  - $c_0 = c_1 = ... = c_{n-1} = 0$,
  - $x_0 = x_1 = ... = x_{n-1}$,
  - $c_0 = c_1 = ... = c_{n-1} = 0,\ x_0 = x_1 = ... = x_{n-1}$.

TP-Prozedur für das allgemeine inverse Hornerschema

```
const nmax=101;                 {max. Dimension der Referenz}
type  float=extended;
      vektor=array[0..nmax] of float;

procedure AllgInvHS(n:integer; var x,c,a:vektor);
var i,j:integer;
    xh:float;
begin
 for i:=0 to n do a[i]:=c[n-i];
 for j:=n-1 downto 0 do
  begin
   xh:=-x[j];
   for i:=n-j downto 1 do a[i]:=a[i]+a[i-1]*xh;
  end;
end;
```

MATLAB-Funktion für das allgemeine inverse Hornerschema

```
% allinvhs.m

function a = allinvhs(n,x,c)

% Eingangsparameter
%  n        Umfang der Referenz, n+1 Punkte
%  x        Stuetzstellen
%  c        Koeffizienten des Polynoms in Newtonscher Form:
%  pn(x)=c[0]+c[1](x-x[0])+...
%            +c[n](x-x[0])(x-x[1])...(x-x[n-1])
% Ergebnisse
%  a        Koeffizienten des Polynoms in Normalform:
%  pn(x)=a[0]x^n+a[1]x^(n-1)+...+a[n]

for i=1:n+1
  a(i) = c(n+2-i);
end;
for j=n-1:-1:0
  xh = -x(j+1);
  for i=n-j:-1:1
    a(i+1) = a(i+1)+a(i)*xh;
  end;
end;
% Ende Funktion allinvhs
```

Beispiel

Referenz $(x_i, y_i),\ i = 0,1,2,3,4 = n,$

x_i	-1	0	1	2	3
y_i	-10	-17	-20	-25	10

Newtonsche Form des Polynoms

$$p_4(x) = -10-7(x+1)+2(x+1)(x-0)-1(x+1)(x-0)(x-1)+2(x+1)(x-0)(x-1)(x-2)$$

Normalform des Polynoms

$$p_4(x) = 2x^4 - 5x^3 + 0x^2 + 0x - 17 = 2x^4 - 5x^3 - 17$$

In der MATLAB-Toolbox `matlab\toolbox\symbolic` ist das m-File `poly2sym.m` für die symbolische Erzeugung der Normalform aus dem gegebenen Koeffizientenvektor.

```
a = [ 2 -5  0  0 -17 ]

a =
     2    -5     0     0   -17

poly2sym(a,'x')

ans =
     2*x^4-5*x^3-17
```

- **Einfaches Hornerschema** zur Berechnung von $p_n(x_0)$.

  Es entsteht durch sukzessives Ausklammern von x in der Normalform.

$$\begin{aligned} p_n(x) &= a_0x^n + a_1x^{n-1} + ... + a_{n-2}x^2 + a_{n-1}x + a_n \\ &= (a_0x^{n-1} + a_1x^{n-2} + ... + a_{n-2}x + a_{n-1})x + a_n \\ &= ([a_0x^{n-2} + a_1x^{n-3} + ... + a_{n-2}]x + a_{n-1})x + a_n \\ &= ... \\ &= ([...\{\underbrace{\underbrace{a_0}_{b_0}x + a_1}_{b_1}\}x + ... + a_{n-2}]x + a_{n-1})x + a_n \end{aligned}$$

b_{n-1} und b_n entsprechend.

	a_0	a_1	a_2	...	a_{n-1}	a_n
x_0		b_0x_0	b_1x_0	...	$b_{n-2}x_0$	$b_{n-1}x_0$
	b_0	b_1	b_2	...	b_{n-1}	$b_n = p_n(x_0)$

Die Größen b_i sind die Koeffizienten des Polynoms $p_{n-1}(x) = \sum_{i=0}^{n-1} c_i x^{n-1-i}$, das der Beziehung $p_n(x) = p_{n-1}(x)(x - x_0) + p_n(x_0)$ genügt.

Ist x_0 Nullstelle von $p_n(x)$, so bedeutet das Hornerschema die Abspaltung des Linearfaktors $(x - x_0)$. Das um den Grad 1 reduzierte Polynom $p_{n-1}(x)$ kann für eine weitere Nullstellenuntersuchung betrachtet werden. Der Prozeß der schrittweisen Abspaltung heißt auch *Deflation*.

- **Vollständiges Hornerschema**

  Sei $p_n(x)$ ein reelles Polynom n-ten Grades und x_0 ein gegebenes Argument.

  Berechnung von Funktionswert und aller Ableitungen.

  $p_n(x_0) = b_n^{(0)},\ p_n'(x_0) = b_{n-1}^{(1)},\ p_n''(x_0) = 2!\, b_{n-2}^{(2)}, ..., p_n^{(n)}(x_0) = n!\, b_0^{(n)}$.

  Es gilt $p_n^{(n)}(x_0) = n! a_0,\ \ p_n^{(k)}(x) = 0$ für $k > n$.

	a_0	a_1	a_2	... a_{n-3}	a_{n-2}	a_{n-1}	a_n
x_0		$b_0^{(0)} x_0$	$b_1^{(0)} x_0$	... $b_{n-4}^{(0)} x_0$	$b_{n-3}^{(0)} x_0$	$b_{n-2}^{(0)} x_0$	$b_{n-1}^{(0)} x_0$
	$b_0^{(0)}$	$b_1^{(0)}$	$b_2^{(0)}$	... $b_{n-3}^{(0)}$	$b_{n-2}^{(0)}$	$b_{n-1}^{(0)}$	$b_n^{(0)} = p_n(x_0)$
x_0		$b_0^{(1)} x_0$	$b_1^{(1)} x_0$	... $b_{n-4}^{(1)} x_0$	$b_{n-3}^{(1)} x_0$	$b_{n-2}^{(1)} x_0$	
	$b_0^{(1)}$	$b_1^{(1)}$	$b_2^{(1)}$	... $b_{n-3}^{(1)}$	$b_{n-2}^{(1)}$	$b_{n-1}^{(1)} = p_{n-1}(x_0)$	
x_0		$b_0^{(2)} x_0$	$b_1^{(2)} x_0$	... $b_{n-4}^{(2)} x_0$	$b_{n-3}^{(2)} x_0$		
	$b_0^{(2)}$	$b_1^{(2)}$	$b_2^{(2)}$	... $b_{n-3}^{(2)}$	$b_{n-2}^{(2)} = p_{n-2}(x_0)$		
	.....						
x_0	.....						
	$b_0^{(n-2)}$	$b_1^{(n-2)}$	$b_2^{(n-2)} = p_2(x_0)$				
x_0		$b_0^{(n-1)} x_0$					
	$b_0^{(n-1)}$	$b_1^{(n-1)} = p_1(x_0)$					
x_0							
	$b_0^{(n)} = p_0(x_0)$						

- **MATLAB-Kommandos für Polynomdarstellungen**

  Außer dem Kommando `poly2sym(a,'x')` sind natürlich auch individuelle Darstellungen der Newtonschen bzw. Normalform des Polynoms möglich. Dies erfolgt durch sukzessiven Aufbau als Zeichenketten.
  Wir setzen die Stützstellen x_i der Referenz und die Koeffizienten c_i des Newtonschen Interpolationspolynoms als gegeben voraus.

```
x = [ -1   0   1   2   3 ];   % x(1..5)
c = [-10 -7   2 -1   2 ];   % c(1..5)
neff = max(size(x))-1;
```

```
% Newtonsche Darstellung des Polynoms:
% pn(x)=c[0]+c[1](x-x[0])+...+c[n](x-x[0])(x-x[1])...(x-x[n-1])
% Achtung: Koeffizienten sind numerische Groessen

s = [];
% Umwandlung mit Festpunktformat 3.1
s1 = sprintf(' %3.1f',abs(c(1)));
if (c(1)<0)
  s1 = [' -',s1];
end;
s = [s,s1];
for k=2:neff+1
  s1 = sprintf(' %3.1f',abs(c(k)));
  if (c(k)<0)
    s1 = [' -',s1];
   else
    s1 = [' +',s1];
  end;
  s = [s,s1];
  s2 = [];
  for j=2:k
    s3 = sprintf('%3.1f',abs(x(j-1)));
    if (x(j-1)<0)
      s3 = ['+',s3];
     else
      s3 = ['-',s3];
    end;
    s2 = [s2,'(x',s3,')'];
  end;
  s = [s,s2];
end;

disp(['Newtonsche Form  pn(x) = ',s])
disp(' ')

Newtonsche Form  pn(x) =  - 10.0 - 7.0(x+1.0)
                                 + 2.0(x+1.0)(x-0.0)
                                 - 1.0(x+1.0)(x-0.0)(x-1.0)
                                 + 2.0(x+1.0)(x-0.0)(x-1.0)(x-2.0)
```

Will man die entstandene Zeichenkette für das Polynom durch Substitution (Kommando `subs`) der noch freien Variablen numerisch auswerten, muß man ev. ein genaueres Zahlenformat als 3.1 berücksichtigen sowie auf die Notation der arithmetischen Operatoren achten und hier das Zeichen der Multiplikation in der Zeile `s2 = [s2,'*(x',s3,')']` einfügen.

```
% Allgemeines inverses Hornerschema
a = allinvhs(neff,x,c)

a =
     2    -5     0     0   -17


% Polynomwertberechnung mittels  polyval
disp(['p4(4) = ',num2str(polyval(a,4))])

p4(4) = 175


% Darstellung als Polynom in Normalform:
% pn(x)=a[0]x^n+a[1]x^(n-1)+...+a[n]
% Achtung: Koeffizienten sind numerische Groessen

s = [];
for k=0:neff
  if k<neff
    s1 = sprintf(' %3.1fx^%1g',abs(a(k+1)),neff-k);
   else
    s1 = sprintf(' %3.1f',abs(a(k+1)));
  end;
  if (a(k+1)<0)
    s1 = [' -',s1];
   elseif k>0
    s1 = [' +',s1];
  end;
  s = [s,s1];
end;
disp(['Normalform  pn(x) = ',s])
disp(' ')

Normalform  pn(x) =  2.0x^4 - 5.0x^3 + 0.0x^2 + 0.0x^1 - 17.0
```

Auch hier ist ev. das Zahlenformat zu verändern und das Multiplikationszeichen noch zu ergänzen gemäß `s1 = sprintf(' %3.1f*x^%1g',abs(a(k+1)),neff-k);`

2.2 Polynominterpolation im $\mathbb{R}^1$

- Grundintervall I
  $I = [a, b] \subset \mathbb{R}$ mit $-\infty < a < b < \infty$ und (bekannte oder unbekannte) reelle Funktion $f : I \to \mathbb{R}$.

- Stützstellen x_i und Stützwerte y_i

$$R = \{ (x_i, y_i) \mid a \leq x_0 < x_1 < x_2 < \ldots < x_n \leq b \}.$$

  Referenz (Stützstellenfolge) mit $n+1$ paarweise verschiedenen *Stützstellen* und den $n+1$ zugehörigen *Stützwerten* $y_i = f(x_i),\ i = 0, 1, ..., n$.

- Interpolationspolynom

$$p_n(x) = a_0 x^n + a_1 x^{n-1} + ... + a_{n-1} x + a_n, \quad a_i \in \mathbb{R}.$$

  Algebraisches *Interpolationspolynom* vom Grade $\leq n$ mit den *Basisfunktionen* $x^n, x^{n-1}, ..., x, 1$ (monomiale Basis).

- Interpolationsforderung (Interpolationsbedingung)

$$p_n(x_i) = y_i = f(x_i), \quad i = 0, 1, ..., n.$$

- Interpolationsaufgabe
  Gesucht sind Koeffizienten $a_0, a_1, ..., a_n$, so daß die Interpolationsforderung erfüllt ist.

Bez. der **Existenz und Eindeutigkeit** gilt die Aussage, daß die Interpolationsaufgabe für beliebiges Intervall I und beliebige Referenzen $R \subset I$ *stets eindeutig lösbar* ist.

Beim Nachweis betrachtet man die Koeffizienten a_i als Lösung des linearen Gleichungssystems $Aa = y$

$$\begin{pmatrix} x_0^n & x_0^{n-1} & \cdots & x_0^2 & x_0 & 1 \\ x_1^n & x_1^{n-1} & \cdots & x_1^2 & x_1 & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ x_{n-1}^n & x_{n-1}^{n-1} & \cdots & x_{n-1}^2 & x_{n-1} & 1 \\ x_n^n & x_n^{n-1} & \cdots & x_n^2 & x_n & 1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \cdot \\ a_{n-1} \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \cdot \\ y_{n-1} \\ y_n \end{pmatrix}.$$

Seine Koeffizientenmatrix A ist regulär und führt auf die Haarsche Determinante

$$H = \begin{vmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{vmatrix} = \prod_{i,j=0,\ i>j}^{n} (x_i - x_j) \neq 0,$$

die wegen $x_i \neq x_j$ eine Vandermonde-Determinante ist.

Die Interpolationsaufgabe ist gleichzeitig ein Spezialfall der Approximation in Form der Ausgleichsrechnung mittels der Methode der kleinsten Quadrate. Die Behandlung des Abschnitts Approximation erfolgt im Skript MATLAB IV.
Dabei setzt man eine *Referenz* mit $N+1$ paarweise verschiedenen *Stützstellen* und den $N+1$ zugehörigen *Stützwerten* $y_j = f(x_j),\ j = 0(1)N$, voraus und nimmt das System der Basisfunktionen (Monome) $\{\varphi_i(x) = x^i,\ i = 0, 1, ..., n = N\}$. Das liefert die stets reguläre Matrix A.
Jedoch löst man i.a. beim Ausgleichsproblem für $N \geq n$ eine Minimierungsaufgabe mittels des sogenannten Normalgleichungssystems

$$A^T A a \ = \ A^T y$$

mit der symmetrischen und positiv definiten Koeffizientenmatrix $G = A^T A$. Folglich ist G regulär und damit die Lösung, d.h. das lokale Extremum eindeutig bestimmt.
Die Lösung des Normalgleichungssystems ist für $n \gg 1$ praktisch nicht zu empfehlen, da sich die Kondition der Koeffizientenmatrix $G = A^T A$ im Vergleich zur Kondition von A selber noch verschlechtert.

Das m-File für die Methode der kleinsten Quadrate mit Polynomen ist `polyfit`.
Seine einfachste gebräuchlichste Anwendung liefert die Koeffizienten a_i des Polynoms in seiner Normalform $p_n(x) = \sum_{i=0}^{n} a_i x^{n-i}$ und hat die folgende Anwendung.

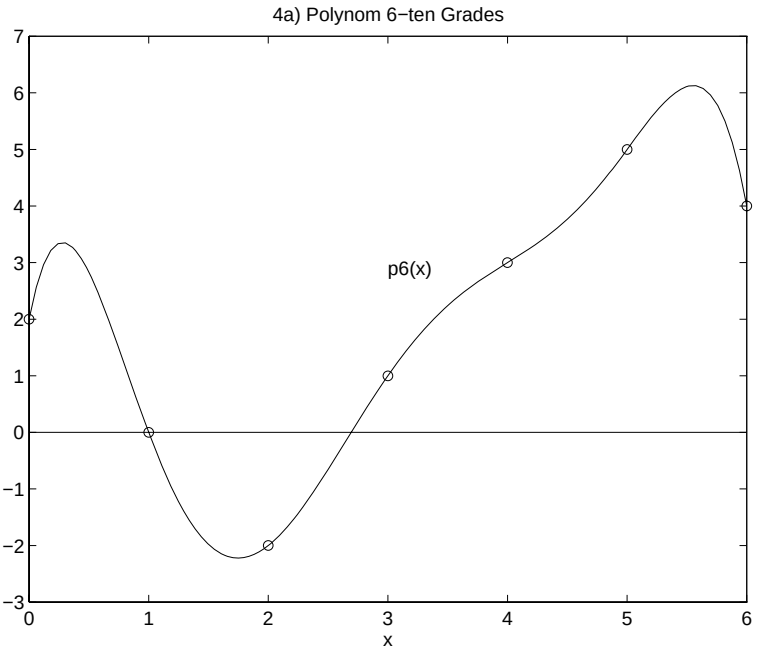
```
% Referenz
x = 0:6;
y = [ 2  0  -2  1  3  5  4 ];
n = max(size(x))-1;
disp('Normalform')
format long
a = polyfit(x,y,n)

% Effektiver Grad des Polynoms
format short
j = 1;
while (a(j)==0) & (j<n+1)
  j = j+1;
end;
neff = n+1-j

Normalform
a =
   Columns 1 through 4
   -0.04027777777778   0.75416666666663  -5.38194444444421  17.89583333333270
   Columns 5 through 7
  -26.07777777777720  10.85000000000035   1.99999999999943

neff =
```

```
6
```

Man bemerke die Ungenauigkeiten in den letzten Dezimalstellen bei der Berechnung der Koeffizienten aufgrund der schlechten Kondition des LGS.

Anschließend bieten sich Polynomwertberechnungen und graphische Darstellungen der Referenz und des Interpolationspolynoms an. Zu Bestimmung einzelner oder eines Vektors von Polynomwerten kann man das Kommando `polyval(koeff,arg)` verwenden.

```
% Kontrolle der Berechnung
polyval(a,0)

% beschriftete Graphik
xi = linspace(min(x),max(x),100);
pxi = polyval(a,xi);
plot(x,y,'o',x,0*y,'w',xi,pxi,':')
title(['Polynom ',sprintf('%1g',neff),'-ten Grades'])
xlabel('x')
text(3,2.9,['p',sprintf('%1g',neff),'(x)'])
```



Bemerkungen

1. Falls die Stützstellen x_i nahe beieinander liegen, ist H klein und die sogenannte Kondition der genannten Gleichungssysteme ist schlecht.
2. Wie kann $p_n(x)$ effektiv und numerisch stabil konstruiert werden?
   Dazu folgen einige Varianten.
3. Wie kann man eine Schätzung des Interpolationsfehlers
   $R_n(x) = f(x) - p_n(x)$ für alle $x \in I$ erhalten?
4. Wie läßt sich die Interpolationsaufgabe verallgemeinern?

Folgendes einfaches Beispiel illustriert die Probleme bei der Anwendung von `polyfit` und Lösung des Normalgleichungssystems für $n \gg 1$.

Die Referenz werde mit $x = 0(1)n$ und $y = x^3$ für wachsendes n erzeugt.

Der Ausgleich mit kubischem Polynom `polyfit(x,y,3)` liefert durchweg akzeptable Ergebnisse, auch wenn in einigen Fällen eine Warnung auftritt.

```
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = ...
```

Die Koeffizienten des kubischen Polynoms $p_3(x) = a_0x^3 + a_1x^2 + a_2x + a_3$ bewegen sich in den Grenzen

a_0 = 0.999 999 999 999 997 0 ... 1.000 000 000 000 002
$|a_{1,2,3}|$ = 1E-11 ... 1E-16

Beim Ausgleich mit Polynom n-ten Grades `polyfit(x,y,n)` werden die Ergebnisse mit wachsendem n erwartungsgemäß immer schlechter. Dabei wollen wir tabellarisch die Koeffizienten a_0, a_{n-3} (exakt =1) und a_n betrachten. Für weitere Polynomkoeffizienten gilt $|a_k| \approx |a_{n-3}|$ für $k = n-1, n-2$ und $k < n-3$ nahe $n-3$.

n	`polyfit(x,y,n)`			`polyfit(x,y,3)`
	$\|a_0\|$	a_{n-3}	$\|a_n\|$	a_0
3	1	1.000 000 000 000 000	5.9E-15	1.000 000 000 000 000
4	5.3E-16	0.999 999 999 999 995 6	3.8E-15	1.000 000 000 000 000
5	7.2E-18	0.999 999 999 999 999 0	6.6E-15	1.000 000 000 000 000
6	1.8E-17	0.999 999 999 999 985 0	2.5E-15	1.000 000 000 000 002
7	6.3E-17	0.999 999 999 999 892 6	2.8E-14	1.000 000 000 000 002
8	1.1E-17	0.999 999 999 999 869 6	7.5E-14	0.999 999 999 999 999 4
9	4.4E-18	1.000 000 000 001 557	4.0E-14	0.999 999 999 999 998 8
10	5.6E-19	1.000 000 000 000 613	5.0E-14	1.000 000 000 000 001
12	1.3E-19	0.999 999 999 961	3.2E-13	1.000 000 000 000 001
14	3.1E-21	0.999 999 999 966	4.2E-14	1.000 000 000 000 001
16	6.1E-23	0.999 999 999 464	5.7E-13	1.000 000 000 000 001
18	1.4E-24	0.999 999 993 176	4.2E-13	1.000 000 000 000 000
20	8.9E-26	1.000 000 118	1.1E-12	0.999 999 999 999 997 0
25	8.2E-32	0.999 999 769	8.1E-10	0.999 999 999 999 998 4
30	4.4E-39	1.000 004 994	3.2E-07	0.999 999 999 999 999 6
40	2.3E-56	0.999 927	7.2E-05	1.000 000 000 000 002
50	3.2E-73	1.005 240	9.9E-03	0.999 999 999 999 999 0
60	1.0E-92	1.009 757	3.3E-02	0.999 999 999 999 999 6

2.3 Lagrange-Interpolation

Lagrangesches Interpolationspolynom höchstens n-ten Grades

$$L_n(x) = \sum_{k=0}^{n} y_k \varphi_k(x), \quad \varphi_k(x) = \prod_{i=0,\ i\neq k}^{n} \frac{x - x_i}{x_k - x_i},$$

$\varphi_k(x) = L_n^{(k)}(x)$ Langrange-Polynome, Basispolynome, Knotenpunktpolynome.
Die Basispolynome genügen der Bedingung

$$\varphi_k(x_i) = \delta_{ik} = \begin{cases} 0 & \text{für } i \neq k \\ 1 & \text{für } i = k \end{cases}, \quad \delta_{ik} \text{ Kronecker-Symbol.}$$

Interpolationsfehler

$f(x)$ sei $(n+1)$-mal stetig differenzierbar auf dem Intervall $I = [a, b]$.
Das eindeutig bestimmte Interpolationspolynom vom Grade $\leq n$ zur Referenz R ist $L_n(x)$.
Dann existiert eine Stelle $\xi \in int(x_0, x_1, ..., x_n) = I_1 \subset I$, so daß für den Interpolationsfehler (Restglied der Interpolation) gilt

$$\begin{aligned} R_n(x) \;=\; f(x) - L_n(x) \;&=\; \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)(x - x_1) \cdot ... \cdot (x - x_n) \\ |R_n(x)| \;&\leq\; \frac{M_{n+1}}{(n+1)!}|\Pi_n(x)|, \\ &\qquad \Pi_n(x) = (x - x_0)(x - x_1) \cdot ... \cdot (x - x_n) \\ &\qquad M_{n+1} = \max_{x \in I_1} |f^{(n+1)}(x)| \end{aligned}$$

Die Lagrange-Interpolation ist wegen der Fehlerabschätzung und zahlreicher Eigenschaften der Basispolynome von Interesse.

(1) $1 = \sum_{k=0}^{n} \varphi_k(x) \leq \sum_{k=0}^{n} |\varphi_k(x)|.$

(2) $$\sum_{k=0}^{n} \varphi_k(0)\ x_k^j = \begin{cases} 1 & \text{für } j = 0 \\ 0 & \text{für } j = 1, 2, \ldots, n \\ (-1)^n\ x_0 \cdot \cdots \cdot x_n & \text{für } j = n+1. \end{cases}$$

(3) Die Lagrange-Polynome $\varphi_k(x)$ bilden eine Orthogonalsystem und damit eine Basis im Raum der Polynome höchstens n-ten Grades mit dem Skalarprodukt $(P, Q) = \sum_{i=0}^{n} P(x_i)Q(x_i)$, d.h. $(\varphi_k, \varphi_j) = \sum_{i=0}^{n} \varphi_k(x_i)\varphi_j(x_i) = \delta_{kj}$.

(4) Lagrange-Interpolationspolynom $L_n(x)$ für äquidistante Stützstellen.
$x_i = x_0 + ih,\ x = x_0 + th,\ h > 0.$

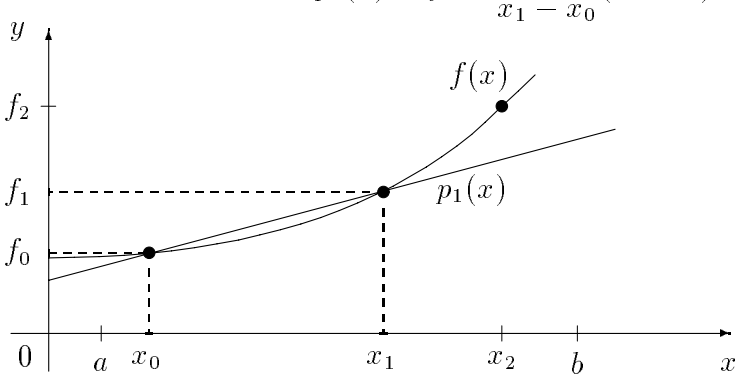
Auf Berechnungen in MATLAB soll hier nicht eingegangen werden.

2.4 Newton-Interpolation

Lineare und quadratische Interpolation

- Lineare Interpolation: 2 Stützstellen $x_0 < x_1$ mit $f_i = f(x_i)$.

$$p_1(x) = f_0 + \frac{f_1 - f_0}{x_1 - x_0}(x - x_0).$$



- Quadratische Interpolation: 3 Stützstellen $x_0 < x_1 < x_2$ mit $f_i = f(x_i)$.

$$p_2(x) = f_0 + \frac{f_1 - f_0}{x_1 - x_0}(x - x_0) + \frac{\frac{f_2 - f_1}{x_2 - x_1} - \frac{f_1 - f_0}{x_1 - x_0}}{x_2 - x_0}(x - x_0)(x - x_1).$$

Vorwärts- bzw. Rückwärtsdifferenzen

Wir notieren die Beziehungen für äquidistante Stützstellen.

$\Delta^0 f(x) = f(x)$	$\nabla^0 f(x) = f(x)$
$\Delta^1 f(x) = \Delta f(x) = f(x+h) - f(x)$	$\nabla^1 f(x) = \nabla f(x) = f(x) - f(x-h) = \Delta f(x-h)$
$\Delta^n f(x) = \Delta(\Delta^{n-1} f(x))$	$\nabla^n f(x) = \nabla(\nabla^{n-1} f(x))$.

Sei $x_i = x_{i-1} + h$, $f_i = f(x_i)$. Dann gilt $\nabla^k f_i = \Delta^k f_{i-k}$.

Dividierte Differenzen

$$f[x_1, x_0] = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

heißt *1. dividierte Differenz* von x_1 und x_0 bezüglich f.

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

heißt *2. dividierte Differenz* von x_2, x_1, x_0 bezüglich f.

Sei die *k-te dividierte Differenz* bereits definiert. Dann heißt

$$f[x_{k+1}, x_k, ..., x_1, x_0] = \frac{f[x_{k+1}, ..., x_2, x_1] - f[x_k, ..., x_1, x_0]}{x_{k+1} - x_0}$$

$(k+1)$−te dividierte Differenz von $x_{k+1}, ..., x_1, x_0$ bezüglich f.

Dividierte Differenzen sind symmetrisch in Bezug auf die Argumente.
Sie werden auch *Steigungen* genannt. Es gilt ebenfalls

$$c_{k+1} = f[x_{k+1}, x_k, ..., x_1, x_0] = \frac{f[x_{k+1}, x_{k-1}, ..., x_2, x_1, x_0] - f[x_k, ..., x_1, x_0]}{x_{k+1} - x_k}.$$

Schema der dividierten Differenzen ($n = 3$)

$$\begin{array}{cccc|ccccc}
 & & & x_0 & f_0 & & & & \\
 & & x_1 - x_0 & & & f[x_1, x_0] & & & \\
 & x_2 - x_0 & & x_1 & f_1 & & f[x_2, x_1, x_0] & & \\
x_3 - x_0 & & x_2 - x_1 & & & f[x_2, x_1] & & f[x_3, x_2, x_1, x_0] & \\
 & x_3 - x_1 & & x_2 & f_2 & & f[x_3, x_2, x_1] & & \\
 & & x_3 - x_2 & & & f[x_3, x_2] & & & \\
 & & & x_3 & f_3 & & & &
\end{array}$$

Folgerung für lineare und quadratische Interpolation

$$\begin{aligned}
p_1(x) &= f(x_0) + f[x_1, x_0](x - x_0), \\
p_2(x) &= f(x_0) + f[x_1, x_0](x - x_0) + f[x_2, x_1, x_0](x - x_0)(x - x_1).
\end{aligned}$$

Für dieses Schema der dividierten Differenzen erstellen wir in MATLAB ein m-File.
Neben den Koeffizienten des Newton-Interpolationspolynoms $N_n(x)$ berechnen wir auch den effektiven Grad des Polynoms.

```
% newtsdd.m
% Polynominterpolation nach Newton mit Schema der dividierten Differenzen

function [neff,c] = newtsdd(n,x,y)

% Eingangsparameter
%  n        Umfang der Referenz, n+1 Punkte
%  x        Stuetzstellen, paarweise verschieden
%  y        Stuetzwerte
% Ergebnisse
%  neff     effektiver Grad des Polynoms <=n
%  c        Koeffizienten des Polynoms in Newtonscher Form:
%  pn(x)=c[0]+c[1](x-x[0])+...+c[neff](x-x[0])(x-x[1])...(x-x[neff-1])

  t = y;
  c(1) = t(1);
  for j=1:n
    for i=1:n+1-j
      t(i) = (t(i+1)-t(i))/(x(j+i)-x(i));
```

```
    end;
    c(j+1) = t(1);
  end;
  j = n+1;
  while (c(j)==0) & (j>1)
    j = j-1;
  end;
  neff = j-1;
% Ende Funktion newtsdd
```

Bei Handrechnung ist die Referenzerweiterung einfach zu realisieren, indem zusätzliche Zeilen berechnet werden.
Geht man von den schon berechneten Koeffizienten $c_0, c_1, ..., c_n$ aus und fügt als neuen Punkt (x_{-1}, y_{-1}) im obigen Schema am Anfang hinzu, so berechnen sich die aktualisierten Koeffizienten $c_0, c_1, ..., c_n, c_{n+1}$ rekursiv gemäß

```
ch := c[0]; cn := y[-1]; c[0] := cn;
xh := x[-1];
for i:=1 to n+1 do
begin
  if i<n+1 then ch1 := c[i];
  cn := (ch-cn)/(x[i-1]-xh);
  c[i] := cn;
  ch := ch1;
end;
```

Dabei haben wir die Verwendung von Indizes möglichst minimal gehalten.
Ein leichte Modifikation ergibt

```
ch := c[0]; c[0] := y[-1];
xh := x[-1];
for i:=1 to n+1 do
begin
  if i<n+1 then ch1 := c[i];
  c[i] := (ch-c[i-1])/(x[i-1]-xh);
  ch := ch1;
end;
```

Newtonsche Interpolationsformel

1. Das algebraische Polynom höchstens n-ten Grades

$$N_n(x) = f(x_0) + \sum_{j=1}^{n} f[x_j, x_{j-1}, ..., x_0](x - x_0)(x - x_1) \cdot ... \cdot (x - x_{j-1})$$

$$= \sum_{k=0}^{n} \delta^k f_0 \; \omega_k(x), \quad \omega_k(x) = \prod_{i=0}^{k-1} (x - x_i),$$

heißt *Newtonsche Interpolationsformel.*

$\omega_k(x) = \Pi_{k-1}(x)$ sind die Newton-Polynome, Basispolynome oder Knotenpunktpolynome, $\delta^k f_0 = f[x_k, ..., x_1, x_0] = [x_k, ..., x_1, x_0]$ k-te dividierte Differenz, $\delta^0 f_i = f_i$.

2. $N_n(x)$ löst die Interpolationsaufgabe $N_n(x_i) = f(x_i), \quad i = 0, 1, ..., n,$ eindeutig und stellt damit das eindeutig bestimmte Interpolationspolynom dar.

3. Der *Interpolationsfehler* (Restglied der Interpolation) lautet

$$\begin{aligned} R_n(x) &= f(x) - N_n(x) \\ &= f[x, x_n, x_{n-1}, ..., x_1, x_0](x - x_0)(x - x_1) \cdot ... \cdot (x - x_n) \\ &= \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)(x - x_1) \cdot ... \cdot (x - x_n). \end{aligned}$$

mit $\xi \in (x_0, x_n)$, vorausgesetzt $f \in C^{n+1}([a, b])$.
Die Fehlerabschätzung ist wie bei der Lagrange-Formel.

Einfache Berechnung von Polynomwerten

Definiere $c_j = f[x_j, x_{j-1}, ..., x_1, x_0],\ j \geq 1,$ und $c_0 = f_0 = f(x_0) = f[x_0]$.

Dann gilt

$$\begin{aligned} N_n(x) &= c_0 + \sum_{j=1}^{n} c_j \prod_{i=0}^{j-1}(x - x_i) \\ &= c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \ldots + c_n(x - x_0)(x - x_1) \cdot ... \cdot (x - x_{n-1}) \\ &= c_0 + (x - x_0)\{c_1 + (x - x_1)[c_2 + \ldots + (x - x_{n-2})(c_{n-1} + (x - x_{n-1})c_n) \ldots]\}. \end{aligned}$$

Algorithmische Darstellung

$Nx := c[n];$
for $j := n - 1$ downto 0 do $\quad Nx := Nx * (x - x[j]) + c[j];$

Die **Interpolationsformel von Newton-Gregory** betrifft den Fall äquidistanter Stützstellen $x_i = x_0 + ih,\ i = 0, 1, ..., n$.
Für die Newton-Interpolation gelten dann die Beziehungen

$$\omega_k(x) = \prod_{i=0}^{k-1}(x - x_i) = h^k \prod_{i=0}^{k-1}(s - i), \quad x = x_0 + sh, \quad \omega_0(x) = 1,$$

$$c_k = \delta^k f_0 = f[x_0, x_1, ..., x_k] = \frac{\Delta^k f_0}{k! h^k}, \quad \Delta^k f_0 \ \ k\text{-te Differenz},$$

$$\begin{aligned} N_n(x) &= \sum_{k=0}^{n} c_k\ \omega_k(x), \quad x = x_0 + sh, \\ &= \sum_{k=0}^{n} \binom{n}{k} \Delta^k f_0. \end{aligned}$$

2.5 Beispiel für die Newton-Interpolation

Wir verwenden dabei die vorgestellten und weitere MATLAB-Routinen und -Kommandos für Polynome und Interpolation.

```
% Schema der dividierten Diff. -> Koeff. der Newton-Interpolationsform
  [neff,c] = newtsdd(n,x,y)

% Allgemeines inverses Hornerschema -> Koeff. der NF
  a = allinvhs(n,x,c)

% Polynomwertberechnung mittels Koeffizienten
  y  = polyval(a,x0)   % x0 Einzelargument
  yy = polyval(a,xx)   % xx Argumentvektor

% Konvertierung der Koeffizienten zu einer symbolischen NF des Polynoms
  spn = poly2sym(a)        % x ist Standardvariablenname
  spn = poly2sym(a,'x')

% Individuelle Kommandofolge zur Darstellung als Polynom
% bei gegebenen Koeffizienten (numerische Groessen) mit Zahlenformat

% ... der Newton-Interpolationsform
  sni='c[0]+c[1]*(x-x[0])+...+c[neff]*(x-x[0])*(x-x[1])*...*(x-x[neff-1])'

% ... der NF
  snf='a[0]*x^n+a[1]*x^(n-1)+...+a[n]'

% Polynomwertberechnung aus symbolischer Form des Polynoms
  numeric(subs(spn,x0,'x'))
  numeric(subs(sni,x0,'x'))
  numeric(subs(snf,x0,'x'))


% Koeffizienten der NF des Interpolationspolynoms
  p = polyfit(x,y,n)
% (Komplexe) Wurzeln des Polynoms mit Koeffizienten p
  r = roots(p)
% Bestimmung der Koeffizienten der NF aus den Wurzeln
  k = p(1).*poly(r)
% Real- bzw. Imaginaerteil der Koeffizienten
  rk = real(k)
  ik = imag(k)

% Koeffizienten der Ableitung der NF
  ps = polyder(p)
```

Aufgabenstellung

Die Ergebnisse sind numerisch und graphisch darzustellen.

1. Bestimme die Newtonsche Form und die Normalform des Interpolationspolynoms $N(x)$ höchstens 4.Grades zu folgender Referenz $\{(x_i, y_i),\ i = 0, 1, ..., 4\}$

x_i	-1	0	1	2	3
y_i	-10	-17	-20	-25	10

   Berechne die Größen $N(x),\ N'(x),\ N''(x)$ an der Stelle $x = 0.25$.

2. Wie verändert sich das Interpolationspolynom, wenn man noch den Punkt (4,175) hinzunimmt? Gebe seine Normalform an.

3. Gebe das Newtonsche Interpolationspolynom und seine Normalform höchstens 6.Grades an, das durch die Punkte $(x_i, y_i),\ i = 0, 1, ..., 4$, sowie durch $(x_5, y_5) = (-2, -65)$ und $(x_6, y_6) = (4, 55)$ verläuft.

4. Löse Teil 3 auch mit `polyfit`.

```
disp('1. Referenz, SDD, NIP und NF')
% Referenz
x = [ -1   0   1   2  3]
y = [-10 -17 -20 -25 10]
n = max(size(x))-1

% SSD
[neff,c] = newtsdd(n,x,y) % UP und m-File

% Newtonsche Darstellung des Polynoms:
% pn(x)=c[0]+c[1]*(x-x[0])+...+c[n]*(x-x[0])(x-x[1])...(x-x[n-1])
% Achtung: Koeffizienten sind numerische Groessen
s = [];
ss= [];
s1 = sprintf(' %3.1f',abs(c(1)));
ss1= sprintf(' %6.4f',abs(c(1)));
if (c(1)<0)
  s1 = [' -',s1];
  ss1= [' -',ss1];
end;
s = [s,s1];
ss= [ss,ss1];
for k=2:neff+1
  s1 = sprintf(' %3.1f',abs(c(k)));
  ss1= sprintf(' %6.4f',abs(c(k)));
```

```
  if (c(k)<0)
    s1 = [' -',s1];
    ss1= [' -',ss1];
   else
    s1 = [' +',s1];
    ss1= [' +',ss1];
  end;
  s = [s,s1];
  ss= [ss,ss1];
  s2 = [];
  ss2= [];
  for j=2:k
    s3 = sprintf('%3.1f',abs(x(j-1)));
    ss3= sprintf('%6.4f',abs(x(j-1)));
    if (x(j-1)<0)
      s3 = ['+',s3];
      ss3= ['+',ss3];
     else
      s3 = ['-',s3];
      ss3= ['-',ss3];
    end;
    s2 = [s2,'(x',s3,')'];
    ss2= [ss2,'*(x',ss3,')'];
  end;
  s = [s,s2];
  ss= [ss,ss2];
end;
sni=ss;
disp(['Newtonsche Form  pn(x) = ',s])
disp(['Newtonsche Form  pn(x) = ',ss])
disp(' ')


% Allgemeines inverses Hornerschema fuer Koeffizienten der NF
a = allinvhs(neff,x,c)

% Kontrolle des Polynomwertes bei x=4
disp(['p4(4) = ',num2str(polyval(a,4))])


% Darstellung als Polynom in Normalform:
% pn(x)=a[0]*x^n+a[1]*x^(n-1)+...+a[n]
% Achtung: Koeffizienten sind numerische Groessen
s = [];
ss= [];
```

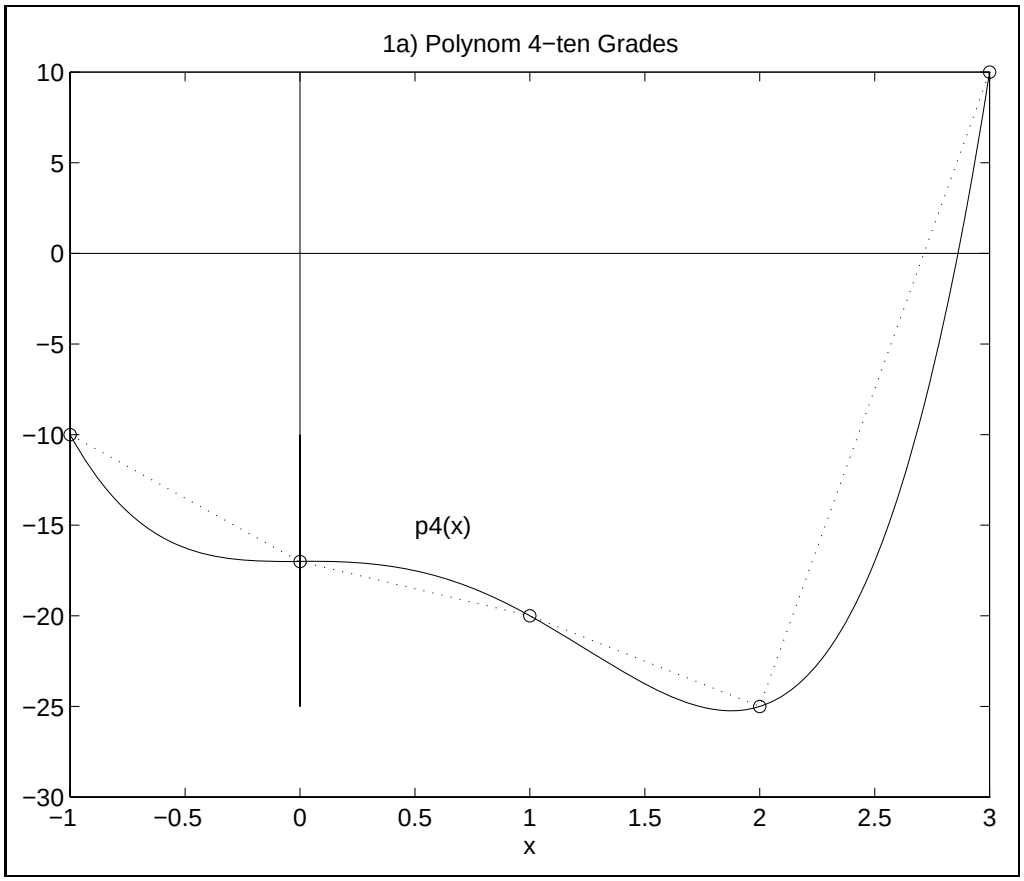
```
for k=0:neff
  if k<neff
    s1 = sprintf(' %3.1fx^%1g',abs(a(k+1)),neff-k);
    ss1= sprintf(' %6.4f*x^%1g',abs(a(k+1)),neff-k);
  else
    s1 = sprintf(' %3.1f',abs(a(k+1)));
    ss1= sprintf(' %6.4f',abs(a(k+1)));
  end;
  if (a(k+1)<0)
    s1 = [' -',s1];
    ss1= [' -',ss1];
   elseif k>0
    s1 = [' +',s1];
    ss1= [' +',ss1];
  end;
  s = [s,s1];
  ss= [ss,ss1];
end;
snf=ss;
disp(' ')
disp(['Normalform  pn(x) = ',s])
disp(['Normalform  pn(x) = ',ss])
disp(' ')

spn=poly2sym(a,'x');
disp(['Normalform mittels poly2sym   pn(x) = ',spn])
disp(' ')

% Berechnung von Funktions- und Ableitungswerten
disp('Polynomwert in x0')
x0=0.25;
format long
px0 = polyval(a,x0)
format short
disp(['pn_sni(x0) = ',num2str(numeric(subs(sni,x0,'x')))])
disp(['pn_snf(x0) = ',num2str(numeric(subs(snf,x0,'x')))])
disp(['pn_spn(x0) = ',num2str(numeric(subs(spn,x0,'x')))])

as = polyder(a)
disp('1.Ableitung in x0')
polyval(as,x0)
a2s = polyder(as)
disp('2.Ableitung in x0')
polyval(a2s,x0)
```

```
% Graphik
xi = linspace(min(x),max(x),100);
pxi = polyval(a,xi);
plot(x,y,'o',x,y,xi,pxi,':',x,0.*y,'w',0.*x,y,'w')
title(['1a) Polynom ',sprintf('%1g',neff),'-ten Grades'])
xlabel('x')
text(0.5,-15,['p',sprintf('%1g',neff),'(x)'])
print ser5gr01.ps -dps
```

Ergebnisse zu 1.

```
x =
    -1     0     1     2     3

y =
   -10   -17   -20   -25    10

n =
     4

neff =
     4

c =
   -10    -7     2    -1     2

Newtonsche Form  pn(x) =  - 10.0 - 7.0(x+1.0) + 2.0(x+1.0)(x-0.0)
                          - 1.0(x+1.0)(x-0.0)(x-1.0)
                          + 2.0(x+1.0)(x-0.0)(x-1.0)(x-2.0)

Newtonsche Form  pn(x) =  - 10.0000 - 7.0000*(x+1.0000)
                          + 2.0000*(x+1.0000)*(x-0.0000)
                          - 1.0000*(x+1.0000)*(x-0.0000)*(x-1.0000)
                          + 2.0000*(x+1.0000)*(x-0.0000)*(x-1.0000)*(x-2.0000)

a =
     2    -5     0     0   -17

p4(4) = 175

Normalform  pn(x) =  2.0x^4 - 5.0x^3 + 0.0x^2 + 0.0x^1 - 17.0
Normalform  pn(x) =  2.0000*x^4 - 5.0000*x^3 + 0.0000*x^2
                               + 0.0000*x^1 - 17.0000

Normalform mittels poly2sym   pn(x) = 2*x^4-5*x^3-17
```

```
Polynomwert in x0
px0 =
     -17.07031250000000
pn_sni(x0) = -17.07
pn_snf(x0) = -17.07
pn_spn(x0) = -17.07

as =
       8   -15     0     0
1.Ableitung in x0
ans =
     -0.8125

a2s =
      24   -30     0
2.Ableitung in x0
ans =
      -6
```




```
% 2. Hinzunahme eines Punktes
%    Nur Vektor c der Koeffizienten des NIP neu berechnen
disp('2. Hinzunahme von Punkt (4,175)')
```

```
xn = 4;
yn = 175;
xx = [xn,x]
yy = [yn,y]
n = max(size(xx))-1
ch = c(1);
c(1) = yn;
for i=1:n
  if i<n
    ch1 = c(i+1);
  end;
  c(i+1) = (ch-c(i))/(xx(i+1)-xn);
  ch = ch1;
end;
c
j = n+1;
while (c(j)==0) & (j>1)
  j = j-1;
end;
neff = j-1
a = allinvhs(neff,xx,c)
disp('Hinzunahme von Punkt (4,175) aendert nicht das NIP')
```

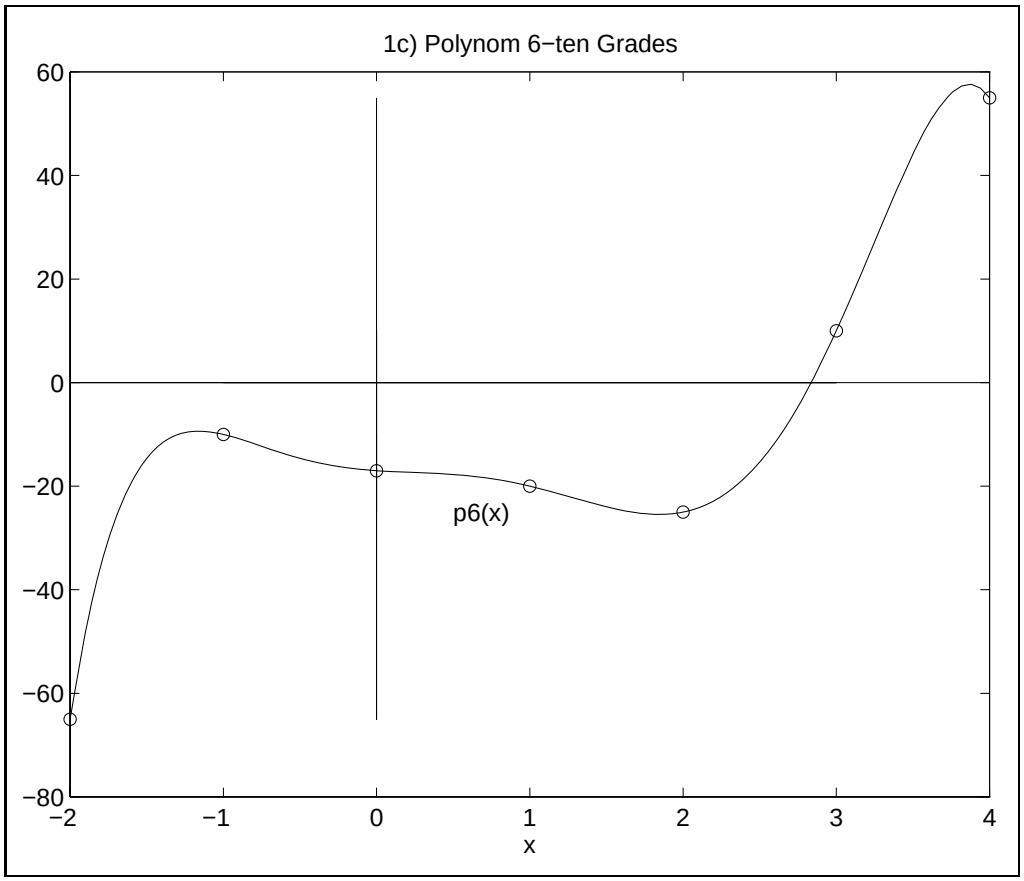
Ergebnisse zu 2.

```
2. Hinzunahme von Punkt (4,175)
xx =
       4    -1     0     1     2     3

yy =
     175   -10   -17   -20   -25    10

n =
     5

c =
   175    37    11     3     2     0

neff =
     4

a =
     2    -5     0     0   -17

Hinzunahme von Punkt (4,175) aendert nicht das NIP
```

```
% 3. NIP mit weiteren Punkten
disp('3. NIP mit 2 weiteren Punkten')
xc = [x, -2, 4]
yc = [y,-65,55]
nc = max(size(xc))-1

disp(' ')
disp('SDD und NIP')
[nceff,cc] = newtsdd(nc,xc,yc) % UP und m-File
ac = allinvhs(nceff,xc,cc)

% Graphik
xi = linspace(min(xc),max(xc),100);
pxi = polyval(ac,xi);
plot(xc,yc,'o',xi,pxi,':',xc,0.*yc,'w',0.*xc,yc,'w')
title(['1c) Polynom ',sprintf('%1g',nceff),'-ten Grades'])
xlabel('x')
text(0.5,-25,['p',sprintf('%1g',nceff),'(x)'])
print ser5gr02.ps -dps
```

Ergebnisse zu 3.

```
3. NIP mit 2 weiteren Punkten

xc =
     -1     0     1     2     3    -2     4

yc =
    -10   -17   -20   -25    10   -65    55

nc =
     6

SDD und NIP
nceff =
       6

cc =
   -10.0000   -7.0000    2.0000   -1.0000    2.0000    1.0000   -0.3333

ac =
    -0.3333    2.0000   -1.3333   -5.0000    3.6667   -2.0000  -17.0000
```




```
% 4. Vergleich von Teil 3 mit MATLAB(polyfit)
disp('4. MATLAB polyfit')
pc = polyfit(xc,yc,nc)
% Graphik wie in 3.
```

Ergebnisse zu 4.

```
4. MATLAB polyfit

pc =
    -0.3333    2.0000   -1.3333   -5.0000    3.6667   -2.0000  -17.0000
```

2.6 Fehler und Konvergenz

Für die Grundvarianten der Interpolation einer Funktion erhält man den schon erwähnten *Interpolationsfehler* $R_n(x) = f(x) - p_n(x)$.
Man hofft, daß mit wachsendem n bei hinreichend glatter Funktion das Interpolationspolynom $p_n(x)$ immer genauer wird, d.h.

$$\|f - p_n\|_\infty = \max_x |f(x) - p_n(x)| \to 0 \text{ für } n \to \infty.$$

Dabei sind natürlich einige Aspekte zu berücksichtigen.

- Glattheit der zu interpolierenden Funktion und Beschränkheit ihrer Ableitungen,
- Endlichkeit des Intervalls $[a,b]$ mit Stützstellen x_i,
- Verteilung der Stützstellen x_i verbunden mit der Kondition der Interpolation.

Beispiel 1

Gegeben sei aus der Wurzelfunktion die Referenz $(100,10),(121,11),(144,12)$.
Mit welcher Genauigkeit kann der Wert $\sqrt{115}$ mit der Lagrangeschen oder Newtonschen Interpolationsformel berechnet werden?

Werte: $n=2,\ a=100,\ b=144$ und
$f(x)=\sqrt{x},\ f\in C^\infty(\mathbb{R}^+),\ f'''(x)=\frac{3}{8}x^{-5/2}$.

$$\begin{aligned}
|R_2(115)| &= |f(115)-N_2(115)| \\
&\le \frac{\max\limits_{[100,144]}|f'''(x)|}{3!}|(15)(-6)(-29)| \\
&\le \frac{\max\limits_{[100,144]}x^{-5/2}}{16}\ 2610 \\
&\approx 1.63\cdot 10^{-3}.
\end{aligned}$$

Genauigkeit bis auf 2 Nachkommastellen ist möglich. Überprüfen wir den Sachverhalt.

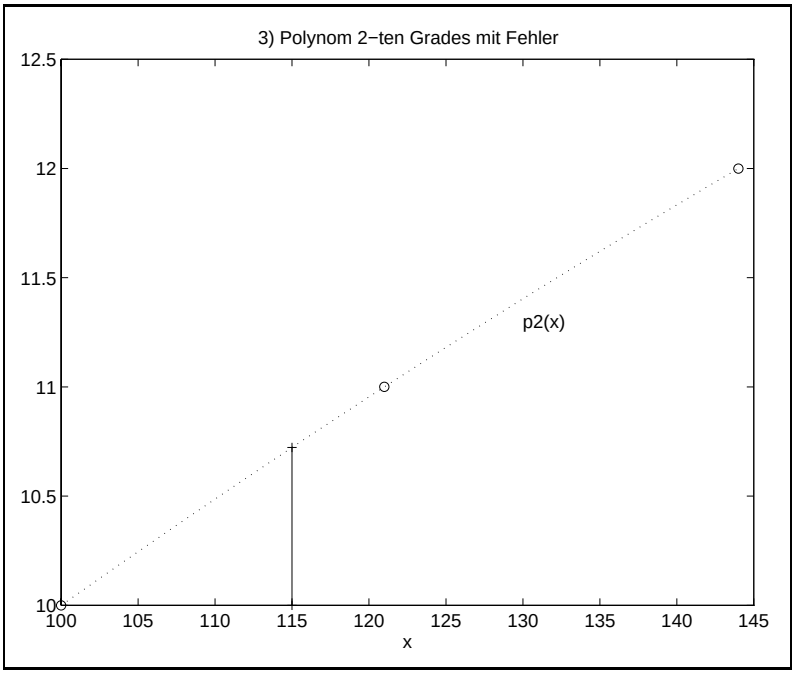
```
% Newtonsches Interpolationspolynom und Fehler
% Referenz
x = [ 100 121 144 ];
y = sqrt(x);
n = max(size(x))-1;
f = 'sqrt(x)';      % symbolische Definition
% f = sqrt(x);

disp(' ')
disp('Koeffizienten der Normalform')
format long
a = polyfit(x,y,n)
format short

x0 = 115
p_x0 = polyval(a,x0)
f_x0 = numeric(subs(f,x0,'x'))
% auch moeglich
% f_x0 = numeric(subs(f,115,'x'))
% f_x0 = numeric(subs(f,'115','x'))

% Graphik
xi = linspace(min(x),max(x),100);
pxi = polyval(a,xi);
```

```
plot(x,y,'o',x0,min(y),'+w',x0,p_x0,'r+',x0,f_x0,'w-',...
     xi,pxi,':',[x0 x0],[min(y) p_x0],'r')
title(['3) Polynom ',sprintf('%1g',n),'-ten Grades mit Fehler'])
xlabel('x')
text(130,11.3,['p',sprintf('%1g',n),'(x)'])
print ser5gr16.ps -dps

disp(['x0=115, |p(x0)-f(x0)| = ',num2str(abs(p_x0-f_x0))])
```

Ergebnisse

```
Koeffizienten der Normalform
a =
    -0.00009410878976   0.06841709015622   4.09937888198763

x0 =
     115

p_x0 =
     10.7228

f_x0 =
     10.7238

x0=115, |p(x0)-f(x0)| = 0.00105
```

Der wirkliche Fehler liegt unter der oben berechneten Schranke.

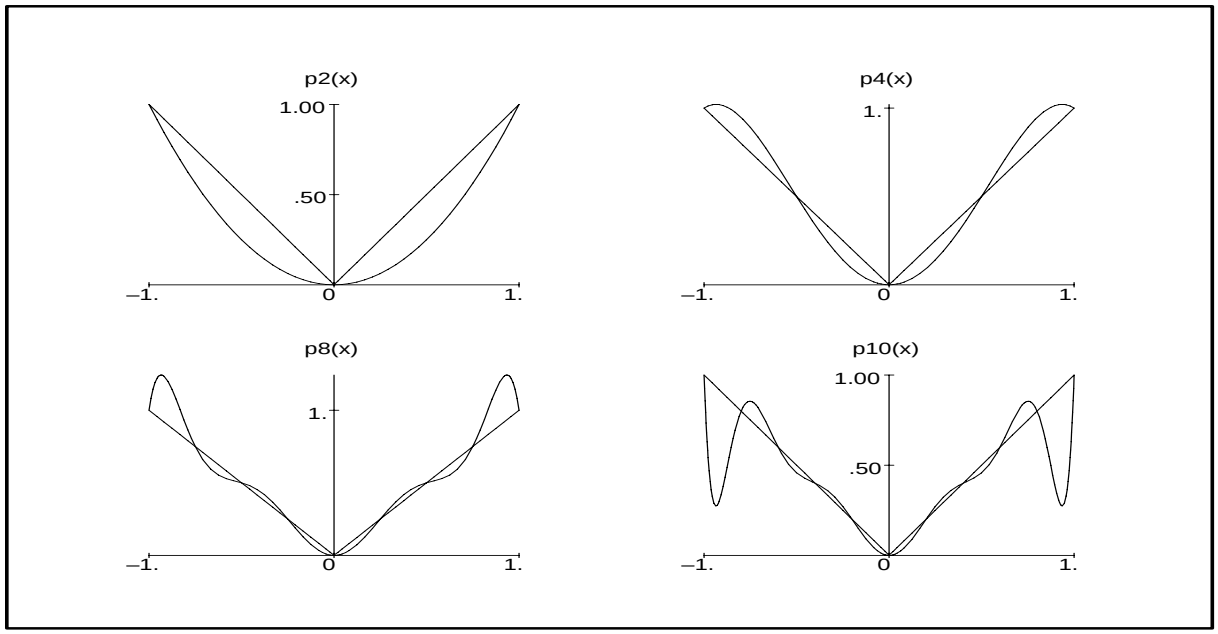


Beispiel 2 $f(x) = |x|,\ x \in [-1,1],\ x_0 = -1,\ x_n = 1,\ x_i$ äquidistant.

Interpolationspolynome

$$\begin{aligned}
n = 2 \quad & p_2(x) = x^2 \\
n = 4 \quad & p_4(x) = -\tfrac{4}{3}x^4 + \tfrac{7}{3}x^2 \\
n = 8 \quad & p_8(x) = -\tfrac{1024}{63}x^8 + \tfrac{1408}{45}x^6 - \tfrac{172}{9}x^4 + \tfrac{533}{105}x^2 \\
n = 10 \quad & p_{10}(x) = \tfrac{390625}{5184}x^{10} - \tfrac{1015625}{6048}x^8 + \tfrac{221875}{1728}x^6 - \tfrac{13375}{324}x^4 + \tfrac{1627}{252}x^2
\end{aligned}$$

Weiter gilt $|f(x) - p_{20}(x)| \approx 100$ für x nahe 1.



Ein Grund für das wachsende oszillierende Verhalten der Interpolationspolynome ist, daß die Funktion im Punkt 0 keine Ableitung besitzt. Das ist aber nicht die alleinige Ursache. Man bemerke, daß in der Mitte des Intervalls eine gute Näherung vorliegt, während zu den Rändern hin die Approximation immer schlechter wird. Dieser Sachverhalt wird *Rungescher* oder *Gibbscher Effekt* genannt.

Beispiel 3 $f(x) = \dfrac{1}{25x^2 + 1},\ x \in [-1,1],\ x_0 = -1,\ x_n = 1,\ x_i$ äquidistant.

Trotz Glattheit der Funktion tritt auch hier bei äquidistanten Stützstellen der Rungesche Effekt auf. Also ist eine andere Stützstellenverteilung zu wählen ($\rightarrow$ nicht äquidistant). Mit der linearen Transformation $5x \rightarrow x'$ erhalten wir die Funktion

$$f(x) = \frac{1}{x^2 + 1},\ x \in [-5,5].$$

Dann führen wir den Parameter $a > 0$ ein, um die Glockenkurve in der Mitte anzuheben und steiler zu machen.

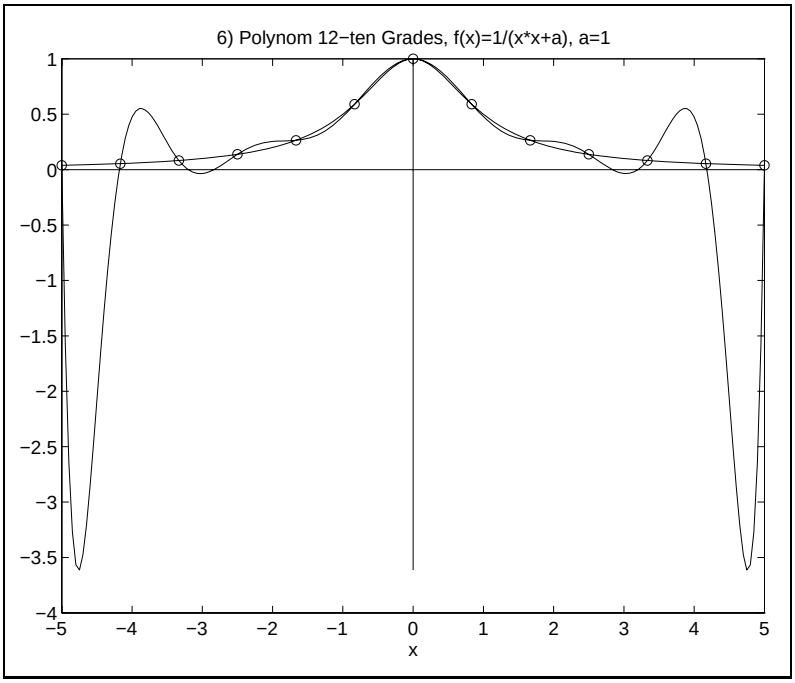
$$f(x) = \frac{1}{x^2 + a},\ x \in [-5,5].$$

Die $n+1$ Stützstellen x_i in $[-5,5]$ wählen wir äquidistant als $x_i = x_0+ih$, $h = (x_n-x_0)/n$. Da die Funktion gerade ist, treten im Interpolationspolynom nur geradzahlige Potenzen von x auf.

```
f = '1/(x*x+a)';  % symbolische Definition
ai = -5;
ei = 5;
% f = 1/(x*x+a)

% Referenz
a = 1;
n = 12;
h = (ei-ai)/n;
disp('Stuetzstellen x')
x = linspace(ai,ei,n+1)
y = [];
for i=0:n
  y = [y,numeric(subs(subs(f,ai+i*h,'x'),a,'a'))];
% f = 1/((ai+i*h)*(ai+i*h)+a);
% y = [y,f];
end;

disp(' ')
disp('Stuetzwerte y')
format long
disp(y)
format short

disp(' ')
disp('SDD, NIP und NF')
[neff,c] = newtsdd(n,x,y) % UP und m-File
ak = allinvhs(neff,x,c)

% Graphik
xi = linspace(min(x),max(x),200);
fxi = [];
for i=1:200
  fxi = [fxi,numeric(subs(subs(f,xi(i),'x'),a,'a'))];
end;
% fxi = 1./(xi.*xi+a);   % viel schneller als symbolisch
pxi = polyval(ak,xi);
plot(x,y,'o',xi,fxi,'y',xi,pxi,'g',x,0.*y,'w',... [0 0],[min(pxi) max(pxi)],'w')
title(['6) Polynom ',sprintf('%1g',neff),...
       '-ten Grades, f(x)=1/(x*x+a), a=',num2str(a)])
xlabel('x')
print ser5gr12.ps -dps
```

Ergebnisse

```
Stuetzstellen x
x =
   -5.0000   -4.1667   -3.3333   -2.5000   -1.6667   -0.8333         0
    0.8333    1.6667    2.5000    3.3333    4.1667    5.0000

Stuetzwerte y
   0.03846153846154   0.05446293494705   0.08256880733945   0.13793103448276
   0.26470588235294   0.59016393442623   1.00000000000000   0.59016393442623
   0.26470588235294   0.13793103448276   0.08256880733945   0.05446293494705
   0.03846153846154

SDD, NIP und NF
neff =
    12

c =
    0.0385    0.0192    0.0087    0.0044    0.0025    0.0011   -0.0016
    0.0001    0.0003   -0.0002    0.0001    0.0000    0.0000

ak =
   0.00000372685037   0.00000000000000  -0.00023924308895   0.00000000000000
   0.00563649022313   0.00000000000000  -0.06114389295888   0.00000000000000
   0.31795734970342   0.00000000000000  -0.78331342918523   0.00000000000000
   1.00000000000000
```

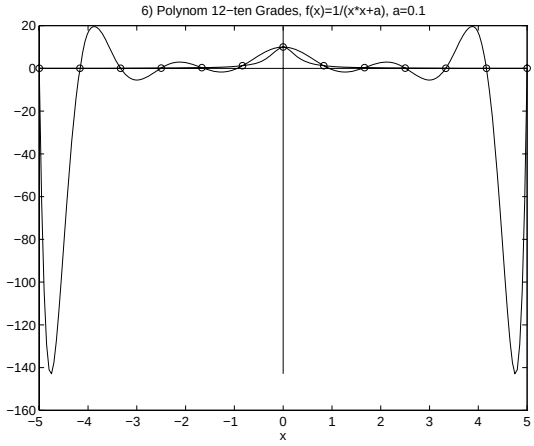


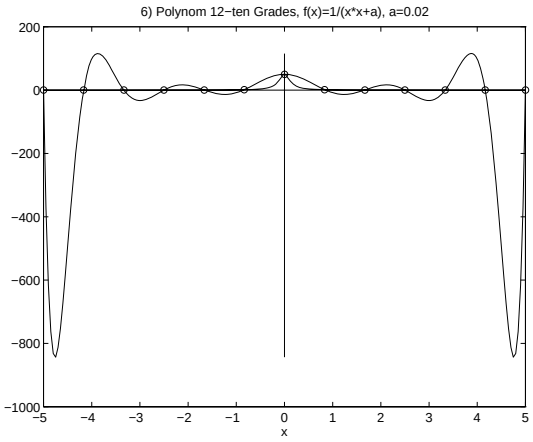
Analog kann man Rechnungen mit anderen Werten n und a durchführen.

Für die Parameterwerte $a = 0.1$ und $a = 0.02$ notieren wir nur die Koeffizienten der Normalform. Hier wird sich das oszillierende Verhalten des Interpolationspolynoms noch verstärken.

```
a = 0.1;
ak =
   0.00014018812706   0.00000000000000  -0.00887312961967   0.00000000000000
   0.20390860228830   0.00000000000000  -2.10834128947784   0.00000000000000
   9.87105494308666   0.00000000000000 -18.49180537250870   0.00000000000000
  10.00000000000000
```

```
a = 0.02;
ak =
    0.000824171181   0.000000000000    -0.052099523356   0.000000000000
    1.194611655581   0.000000000000   -12.299029992716   0.000000000000
   57.038775879196   0.000000000000  -104.051553204351   0.000000000000
   50.000000000000
```





2.7 Newton-Interpolation für äquidistante Stützstellen

Wir definieren dafür eine MATLAB Funktion, die die Newton-Interpolation sowie die Bestimmung der Normalform des Interpolationspolynoms enthält.

```
% newton_i.m
% Polynominterpolation nach Newton mit Schema der dividierten
% Differenzen fuer aequidistante Stuetzstellen und NF

function a = newton_i(n,x0,h,f)

% Eingangsparameter
%  n        Umfang der Referenz, n+1 Punkte
%  x0       erste Stuetzstelle
%  h        Abstand der Stuetzstellen
```

```
%  f        Stuetzwerte
% Ergebnisse
%  a        Koeffizienten des Polynoms in Normalform:
%  pn(x)=a[0]x^n+a[1]x^(n-1)+...+a[n]

% SDD
a = f;
for j=1:n
  for i=n+1:-1:j+1
    a(i) = (a(i)-a(i-1))/(j*h);
  end;
end;

% Inverses HS
for j=n-1:-1:0
  xh = -(x0+j*h);
  for i=j+1:n
    a(i) = a(i)+a(i+1)*xh;
  end;
end;

for i=1:fix((n+1)/2)
  ha = a(i);
  a(i) = a(n+2-i);
  a(n+2-i) = ha;
end;
% Ende Funktion newton_i
```

Es bleibt noch die Aufgabe, den effektiven Grad des Polynoms zu berechnen und eine kleine Anwendung zu demonstrieren.

```
x0 = -1;
h = 1;
n = 4;
x = [-1 0 1 2 3 ];
y = [ 1 0 1 4 9 ];

a = newton_i(n,x0,h,y)

% Effektiven Grad des Polynoms bestimmen
j = 1;
while (a(j)==0) & (j<n+1)
  j = j+1;
end
neff = n+1-j
```

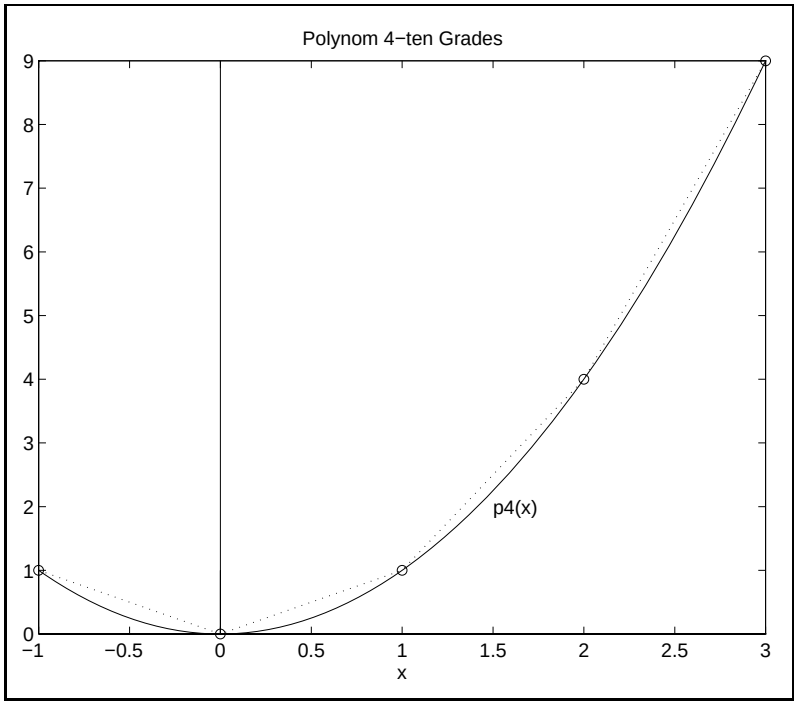
Es werden exakte Ergebnisse geliefert.

```
a    = 0     0     1     0     0
neff = 2
```

Nun stellen wir noch einen Vergleich mit der Funktion `polyfit` an.
Dabei berechnen wir zusätzlich Polynomwerte, Nullstellen und Ableitungen. Es wird deutlich, daß hier die numerischen Rechnungen nicht zu exakten Werten und in den komplexen Zahlenbereich führen und damit unerwünschte Nebeneffekte auftreten.

```
disp('Koeffizienten der NF')
format long e
p = polyfit(x,y,n)

p =
    1.451396787132872e-016  -5.671482051623341e-016   1.000000000000000e+000
    3.806736168603597e-016   2.369190090501221e-017

disp('(Komplexe) Wurzeln des Polynoms als Spaltenvektor')
r = roots(p)

r =
     1.953801366346874e+000 +8.300550981237336e+007i
     1.953801366346874e+000 -8.300550981237336e+007i
    -1.903368084301798e-016 +4.867432681097107e-009i
    -1.903368084301798e-016 -4.867432681097107e-009i

disp('Bestimmung der Koeffizienten aus den Wurzeln')
disp('(ev. komplexe Zahlen)')
kk = p(1).*poly(r)

kk =
     1.451396787132872e-016
    -5.671482051623338e-016
     9.999999999999994e-001
     3.806736168603594e-016
     2.369190090501220e-017 -8.922184596644220e-041i

disp('Realteil der Koeffizienten (falls Imaginaerteil = Null)')
rk = real(kk)

rk =
    1.451396787132872e-016  -5.671482051623338e-016   9.999999999999994e-001
    3.806736168603594e-016   2.369190090501220e-017
```

```
disp('Koeffizienten der Ableitung')
ps = polyder(p)

ps =
    5.805587148531489e-016  -1.701444615487002e-015   2.000000000000000e+000
    3.806736168603597e-016

format short
disp('Polynom-  und Ableitungswertberechnung')
y1 = polyval(p,1)
ys1 = polyval(ps,1)

y1 =
    1.0000

ys1 =
    2.0000

% Graphik
xi = linspace(min(x),max(x),100);
pi = polyval(p,xi);
plot(x,y,'o',x,y,':',xi,pi,'y-',x,0.*y,'w',0.*x,y,'w')
title(['Polynom ',sprintf('%1g',n),'-ten Grades'])
xlabel('x')
text(1.5,2,['p',sprintf('%1g',n),'(x)'])
print int2gr1.ps -dps
```



3 Hermite-Interpolation

Verallgemeinerte Polynominterpolation im $\mathbb{R}^1$

- Grundintervall I
  $I = [a,b] \subset \mathbb{R}$ mit $-\infty < a < b < \infty$ und (bekannte oder unbekannte) reelle Funktion $f \in C^\alpha(I),\ \alpha \geq 0$.

- Referenz
$$R = \{\ (x_i, y_i) \mid a \leq x_0 < x_1 < x_2 < \ldots < x_n \leq b\ \}.$$

- Interpolationspolynom vom Grade $\leq m$
$$p_m(x) = a_0 x^m + a_1 x^{m-1} + \ldots + a_{m-1}x + a_m, \quad a_i \in \mathbb{R}.$$

- Interpolationsforderung (Interpolationsbedingung)
$$p_m(x_0) = f(x_0),\ p_m'(x_0) = f'(x_0), \ldots,\ p_m^{(\alpha_0-1)}(x_0) = f^{(\alpha_0-1)}(x_0),$$
$$p_m(x_1) = f(x_1),\ p_m'(x_1) = f'(x_1), \ldots,\ p_m^{(\alpha_1-1)}(x_1) = f^{(\alpha_1-1)}(x_1),$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$p_m(x_n) = f(x_n),\ p_m'(x_n) = f'(x_n), \ldots,\ p_m^{(\alpha_n-1)}(x_n) = f^{(\alpha_n-1)}(x_n),$$
  wobei $\alpha_i \geq 1$ Vielfachheiten, $m = \sum\limits_{i=0}^{n} \alpha_i - 1$, $\alpha = \max\limits_{i=0(1)n} \alpha_i - 1$.

- Interpolationsaufgabe
  Gesucht sind Koeffizienten $a_0, a_1, \ldots, a_m$, so daß die Interpolationsforderung erfüllt ist.

Dividierte Differenzen mit mehrfachen Argumenten

Sei f hinreichend oft stetig differenzierbar auf I. Dann gilt

$$f[\underbrace{x_0, \ldots, x_0}_{k_0-mal}, \underbrace{x_1, \ldots, x_1}_{k_1-mal}, \ldots, \underbrace{x_n, \ldots, x_n}_{k_n-mal}] =$$
$$= \lim_{x_i^{(j)} \to x_i} f[x_0, x_0^{(1)}, \ldots, x_0^{(k_0-1)}, x_1, x_1^{(1)}, \ldots, x_1^{(k_1-1)}, \ldots, x_n, x_n^{(1)}, \ldots, x_n^{(k_n-1)}],$$

wobei alle $x_i^{(j)}$ voneinander verschieden sind.

Folgerung

Sei $f(x)$ k-mal stetig differenzierbar ($k \geq 1$) in einer Umgebung von x. Dann gilt

$$f[\underbrace{x, x, \ldots, x}_{(k+1)-mal}] = \frac{f^{(k)}(x)}{k!}.$$

Hermitesche Interpolationsformel

1. Das Polynom höchstens m-ten Grades

$$H_m(x) = f(x_0) + \sum_{j=1}^{m} f[x_j, x_{j-1}, ..., x_0](x - x_0)(x - x_1) \cdot ... \cdot (x - x_{j-1})$$

heißt *Hermitesche Interpolationsformel*. $H_m(x)$ löst die Interpolationsaufgabe eindeutig. x_j sind hier die Stützstellen nach Umnummerierung.

2. Der *Interpolationsfehler* (Restglied der Interpolation) lautet

$$\begin{aligned} R_m(x) &= f(x) - H_m(x) \\ &= \frac{f^{(m+1)}(\xi)}{(m+1)!}(x - x_0)(x - x_1) \cdot ... \cdot (x - x_m) \end{aligned}$$

mit $\xi \in I$, vorausgesetzt $f \in C^{m+1}(I)$.

3.1 Bestimmung des Hermite-Interpolationspolynoms

Gegeben seien zum Beispiel die Werte

$$\begin{array}{c|l} x_0 & f(x_0), \quad f'(x_0), \quad f''(x_0) \\ x_1 & f(x_1), \quad f'(x_1) \end{array}$$

Vielfachheiten: $\alpha_0 = 3,\ \alpha_1 = 2 \quad \Rightarrow \ m = 4$. Bestimmung von $H_4(x)$.

Entsprechend den Vielfachheiten ist das Ausgangsschema der dividierten Differenzen so zu ergänzen, daß die Paare (x_i, f_i) α_i-mal aufzunehmen sind, dazu entsprechend

$$f[x_i, x_i] = f'(x_i),\ f[x_i, x_i, x_i] = \frac{f''(x_i)}{2!}, ..., f[\underbrace{x_i, ..., x_i}_{\alpha_i - mal}] = \frac{f^{(\alpha_i - 1)}(x_i)}{(\alpha_i - 1)!}.$$

Ausgangsschema der dividierten Differenzen

$$\begin{array}{cc|l|ll} & x_0 & f_0 & & \\ & -- & & f[x_0, x_0] & \\ -- & x_0 & f_0 & & f[x_0, x_0, x_0] \\ & -- & & f[x_0, x_0] & \\ & x_0 & f_0 & & \\ \hline & x_1 & f_1 & & \\ & -- & & f[x_1, x_1] & \\ & x_1 & f_1 & & \end{array}$$

mit den speziellen dividierten Differenzen

$$f[x_0, x_0] = \frac{f'(x_0)}{1!},\ f[x_1, x_1] = \frac{f'(x_1)}{1!},\ f[x_0, x_0, x_0] = \frac{f''(x_0)}{2!}.$$

Erweitertes Schema der dividierten Differenzen

$$
\begin{array}{cccc|l}
 & & x_0 & & f_0 \\
 & -- & & & f[x_0,x_0] \\
 & & x_0 & & f_0 \quad f[x_0,x_0,x_0] \\
 & -- & & & f[x_0,x_0] \quad f[x_1,x_0,x_0,x_0] \\
x_1-x_0 & & x_0 & & f_0 \quad f[x_1,x_0,x_0] \quad f[x_1,x_1,x_0,x_0,x_0] \\
 & x_1-x_0 & & & f[x_1,x_0] \quad f[x_1,x_1,x_0,x_0] \\
 & & x_1 & & f_1 \quad f[x_1,x_1,x_0] \\
 & -- & & & f[x_1,x_1] \\
 & & x_1 & & f_1
\end{array}
$$

Interpolationsformel

$$
\begin{aligned}
H_4(x) = f_0 \;&+\; f[x_0,x_0](x-x_0) + f[x_0,x_0,x_0](x-x_0)^2 \\
&+\; f[x_1,x_0,x_0,x_0](x-x_0)^3 + f[x_1,x_1,x_0,x_0,x_0](x-x_0)^3(x-x_1).
\end{aligned}
$$

Liegen an den Stützstellen $x_k,\ k = 0,1,\ldots,n$, genau die Funktionswerte y_k und Ableitungswerte y_k' vor, kann man eine explizite Darstellung, des Hermiteschen Interpolationspolynoms angeben, ähnlich der Lagrange-Formel.

Gegeben sei die Referenz $R = \{(x_i, y_i, y_i'),\ i = 0,1,...,n\}$.

Man zeigt.

(1) Der Grad des Polynoms

$$
p(x) = \sum_{k=0}^{n} \frac{\Pi_k^2(x)}{\Pi_k^3(x_k)} \left\{ y_k \Pi_k(x_k) + (x - x_k)\left[y_k' \Pi_k(x_k) - 2 y_k \Pi_k'(x_k) \right] \right\}
$$

$$
\text{mit}\ \ \Pi_k(x) = \prod_{j=0, j\neq k}^{n} (x - x_j)\ \ \text{ist} \leq 2n+1.
$$

(2) $p(x_k) = y_k, \quad p'(x_k) = y_k'.$

Bei einer Stützstelle mit Funktionswert und Ableitungen bis zum Grad n erhält man das Interpolationspolynom

$$
p(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k,
$$

was nichts anderes als die Taylorreihe der Funktion ist.
Man spricht deshalb auch von der *Taylor-Interpolation.*

3.2 MATLAB Funktion für die Hermite-Interpolation

Wir betrachten die Hermite-Interpolation mit maximal 1.Ableitungen. Es ist sinvoll, einen Indikatorvektor für vorhandene Ableitungswerte zu definieren. Damit und mit der erweiterten Referenz wird das Schema der dividierten Differenzen aufgebaut. Insbesondere kann man die Funktion auch für die Newton-Interpolation verwenden.
Mögliche Parameter der Funktion sind der nachfolgenden Beschreibung zu entnehmen.

```
% hermsdd.m
% Polynominterpolation nach Hermite mit Schema der dividierten Differenzen
% mit vollstaendiger/unvollstaendiger Referenz

function [nmax,neff,c,xx] = hermsdd(n,x_ein,y_ein,ys_ein,voll)

% Eingangsparameter
%  n        Umfang der Referenz, n+1 Punkte
%  x_ein    Stuetzstellen, paarweise verschieden
%  y_ein    Stuetzwerte
%  ys_ein   Ableitungswerte
%  voll     Indikatorvektor fuer vorhandene Ableitungswerte y'
% Ergebnisse
%  nmax     maximal moeglicher Grad des Polynoms
%  neff     effektiver Grad des Polynoms <=nmax
%  c        Koeffizienten des Polynoms in Newtonscher Form:
%  pn(x)=c[0]+c[1](x-xx[0])+...
%            +c[neff](x-xx[0])(x-xx[1])...(x-xx[neff-1])
%  xx       Stuetzstellen, ev. mit Wiederholungen

  nmax = 0;
  for k=1:n+1
    nmax = nmax+1;
    xx(nmax) = x_ein(k);
    yy(nmax) = y_ein(k);
    if voll(k)==1
      t(nmax) = ys_ein(k);
      nmax = nmax+1;
      xx(nmax) = x_ein(k);
      yy(nmax) = y_ein(k);
    end;
    if k<n+1
      t(nmax) = (y_ein(k+1)-y_ein(k))/(x_ein(k+1)-x_ein(k));
    end;
  end;

  c(1) = yy(1);
  c(2) = t(1);
```

```
  for j=2:nmax-1
    for i=1:nmax-j
      t(i) = (t(i+1)-t(i))/(xx(j+i)-xx(i));
    end;
    % disp(t);
    c(j+1) = t(1);
  end;
  j = nmax;
  while (c(j)==0) & (j>1)
    j = j-1;
  end;
  nmax = nmax-1;
  neff = j-1;
% Ende Funktion hermsdd
```

3.3 Beispiele für die Hermite-Interpolation

Wir verwenden dabei die vorgestellten und weitere MATLAB-Routinen und -Kommandos für Polynome und Interpolation.
Bei der graphischen Darstellung der Daten der erweiterten Referenz werden die Ableitungen als Geradenstücken mit entsprechenden Anstiegen gezeichnet (vergleiche Richtungsfeld).

Beispiel 1

Referenz

x	-1	0	1	2	4
y	1	0	1	4	9
y'	1		1	-2	

Damit ist der Indikatorvektor für die Ableitungen $(1,0,1,1,0)$.

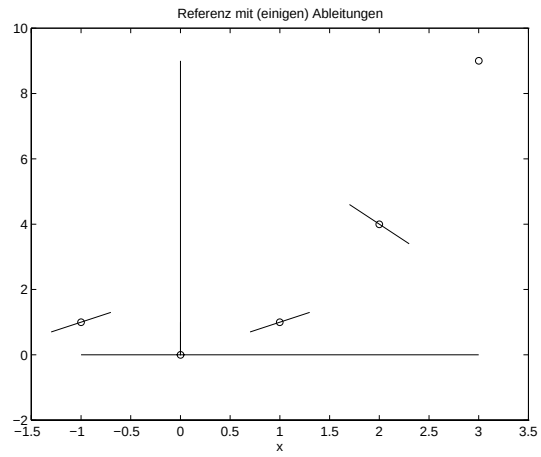
```
disp('Anwendung von UP:  hermsdd, allinvhs ')
% Referenz und Dimension
x = [-1 0 1 2 3];
y = [ 1 0 1 4 9];
ys = [1 0 1 -2 0];          % Ableitungswerte
indicate = [1 0 1 1 0];     % bei x(1),x(3),x(4)
n = max(size(x))-1;

[nmax,neff,c,xx] = hermsdd(n,x,y,ys,indicate)

% Darstellung als Polynom in Normalform:
% pn(x)=a[0]x^n+a[1]x^(n-1)+...+a[n]
format long e
a = allinvhs(neff,xx,c)
format short
```

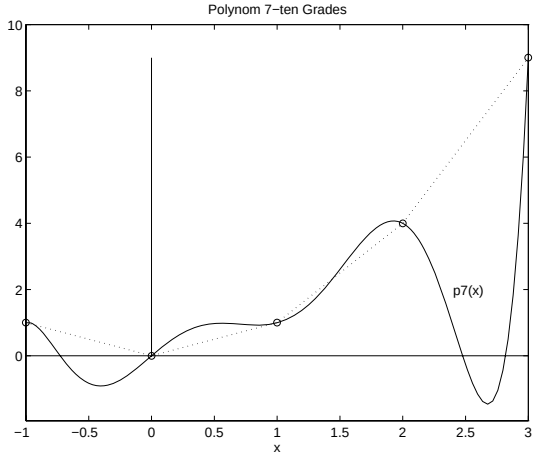
```
as = polyder(a)
disp('Kontrolle der Ableitungen in x')
polyval(as,x)

% Graphik
l = 0;
for k=1:n+1
  if indicate(k)==1
    l = l+1;
    plx(l,1:2) = [x(k)-0.3,x(k)+0.3];
    ply(l,1:2) = [y(k)-0.3*ys(k),y(k)+0.3*ys(k)];
  end;
end;
plot(x,y,'o',x,0.*y,'w',0.*x,y,'w')
title('Referenz mit (einigen) Ableitungen')
xlabel('x')
axis([-1.5 3.5 -2 10])
hold on
for i=1:l
  plot(plx(i, : ),ply(i, : ))
end;
hold off
print int3gr2.ps -dps

xi = linspace(min(x),max(x),100);
pxi = polyval(a,xi);
plot(x,y,'o',x,y,':',xi,pxi,'y-',x,0.*y,'w',0.*x,y,'w')
title(['Polynom ',sprintf('%1g',neff),'-ten Grades'])
xlabel('x')
axis([min(x) max(x)  min(pxi)-0.5 max(pxi)+1])
text(2.4,2,['p',sprintf('%1g',neff),'(x)'])
print int3gr3.ps -dps
```

Ergebnisse

```
nmax =
     7

neff =
     7

c =
    1.0000    1.0000   -2.0000    1.5000   -1.0000    0.5000   -0.6667
    0.4792
```

```
xx =
    -1    -1     0     1     1     2     2     3

a =
    4.791666666666666e-001  -2.583333333333333e+000   2.791666666666666e+000
    4.166666666666666e+000  -6.520833333333332e+000  -5.833333333333321e-001
    3.250000000000000e+000                        0

as =
    3.3542  -15.5000   13.9583   16.6667  -19.5625   -1.1667    3.2500

Kontrolle der Ableitungen in x

ans =
    1.0000    3.2500    1.0000   -2.0000   83.0000
```





Beispiel 2

Referenz

x	-1	0	2
y	7	6	22
y'	-1	0	56

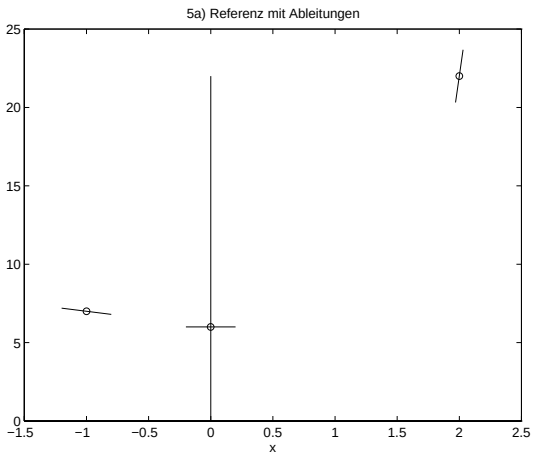
Damit liegen alle Werte der 1.Ableitung vor und der Indikatorvektor ist $(1, 1, 1)$.

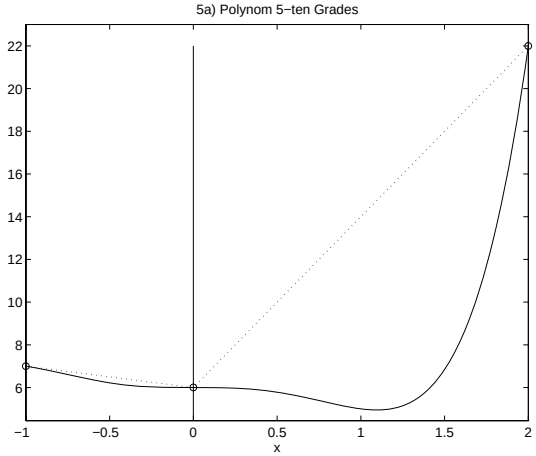
```
% Referenz
x =  [-1  0  2];
y =  [ 7  6 22];
ys = [-1  0 56];
indicate = [1  1  1];  % alle Ableitungen definiert
n = max(size(x))-1;

disp('SDD, HIP, Normalform')
[nmax,neff,c,xx] = hermsdd(n,x,y,ys,indicate)
```

```
% Koeffizienten des Polynoms in Normalform:
% pn(x)=a[0]x^n+a[1]x^(n-1)+...+a[n]
format long e
a = allinvhs(neff,xx,c)
format short

as = polyder(a)

disp('Kontrolle der Ableitungen in x')
polyval(as,x)

% Graphik
l = 0;
for k=1:n+1
  if indicate(k)==1
    l = l+1;
    if k<n+1    % steile Anstiege extra behandeln
      plx(l,1:2) = [x(k)-0.2,x(k)+0.2];
      ply(l,1:2) = [y(k)-0.2*ys(k),y(k)+0.2*ys(k)];
     else
      plx(l,1:2) = [x(k)-0.03,x(k)+0.03];
      ply(l,1:2) = [y(k)-0.03*ys(k),y(k)+0.03*ys(k)];
    end;
  end;
end;

plot(x,y,'o',x,0.*y,'w',[0 0],[0 max(y)],'w')
title('5a) Referenz mit Ableitungen')
xlabel('x')
hold on
for i=1:l
  plot(plx(i, : ),ply(i, : ))
end;
hold off
print ser5gr08.ps -dps

xi = linspace(min(x),max(x),100);
pxi = polyval(aa,xi);
plot(x,y,'o',x,y,':',xi,pxi,'y-',x,0.*y,'w',[0 0],[0 max(y)],'w')
title(['5a) Polynom ',sprintf('%1g',neff),'-ten Grades'])
xlabel('x')
axis([min(x) max(x)  min(pxi)-0.5 max(pxi)+1])
print ser5gr09.ps -dps
```

Ergebnisse

```
SDD, HIP, Normalform

nmax =
     5

neff =
     5

c =
     7    -1     0     1     0     1

xx =
    -1    -1     0     0     2     2

a =
     1     0    -2     0     0     6

as =
     5     0    -6     0     0

Kontrolle der Ableitungen in x

ans =
    -1     0    56
```





Beispiel 3

Referenz mit allen 1. und 2. Ableitungen

$$\begin{array}{rclcrclcrcl} f(0) & = & 1 & , & f'(0) & = & 0 & , & f''(0) & = & -8 \\ f(1) & = & 0 & , & f'(1) & = & 1 & , & f''(1) & = & 4 \end{array}$$

Das Ausgangstableau der dividierten Differenzen wird spaltenweise generiert.

Mit jeder neuen Spalte wird ein nächster Koeffizient c_i der Newtonschen Interpolationsformel erzeugt (Bezeichnung und Nummerierung wie in MATLAB).

k	$xx(1..6)$	Spalte 1 $yy(1..6)$	Spalte 2 $t(1..5)$	Spalte 3	Spalte 4 ...
1	x_1	$c_1 = y_1$	$c_2 = y_1'$	$c_3 = \frac{y_1''}{2}$	...
	x_1	y_1	y_1'	$\frac{t_3 - t_2}{x_2 - x_1}$	
	x_1	y_1	$\frac{y_2 - y_1}{x_2 - x_1}$	$\frac{t_4 - t_3}{x_2 - x_1}$	
2	x_2	y_2	y_2'	$\frac{y_2''}{2}$	
	x_2	y_2	y_2'		
	x_2	y_2			

```
% Referenz
x =  [ 0  1 ];
y =  [ 1  0 ];
ys  = [  0 1 ];
y2s = [ -8 4 ];      % alle Ableitungen definiert
n = max(size(x))-1;

disp('SDD, HIP, Normalform')

nmax = 0;
for k=1:n+1
  nmax = nmax+1;
  xx(nmax) = x(k);
  yy(nmax) = y(k);
  t(nmax)  = ys(k);
  nmax = nmax+1;
  xx(nmax) = x(k);
  yy(nmax) = y(k);
  t(nmax)  = ys(k);
  nmax = nmax+1;
  xx(nmax) = x(k);
  yy(nmax) = y(k);
  if k<n+1
    t(nmax) = (y(k+1)-y(k))/(x(k+1)-x(k));
  end;
end;
```

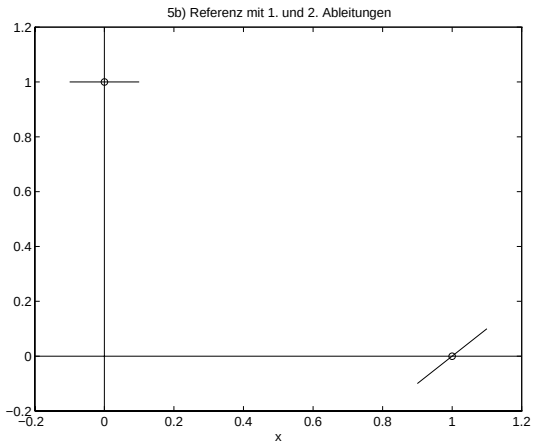
```
c(1) = yy(1);
c(2) = t(1);
nmax = nmax-1;   % nmax=5

% SDD ergaenzen mit 2.Ableitungen, 3.Spalte
l = 1;
for k=1:nmax-1
  if rem(k,3)==1
    t(k) = y2s(l)/2;     % 2.Ableitung beruecksichtigen
    l = l+1;
   else
    t(k) = (t(k+1)-t(k))/(xx(k+2)-xx(k));
  end;
end;
c(3) = t(1);

% SDD, Spalten 4..nmax+1
for k=nmax-2:-1:1
  for j=1:k
    t(j) = (t(j+1)-t(j))/(xx(j+nmax+1-k)-xx(j));
  end;
  c(nmax+2-k) = t(1);
end;

disp('Koeffizienten des NIP')
disp(c)

% Effektiven Grad des Polynoms bestimmen
j = nmax+1;
while (c(j)==0) & (j>1)
  j = j-1;
end
neff = j-1

% Koeffizienten des Polynoms in Normalform:
% pn(x)=a[0]x^n+a[1]x^(n-1)+...+a[n]

disp('Koeffizienten der NF')
a = allinvhs(neff,xx,c)

disp('Koeffizienten der Ableitung der NF');
as = polyder(a)
disp('Kontrolle der 1.Ableitungen in x')
polyval(as,x)
```

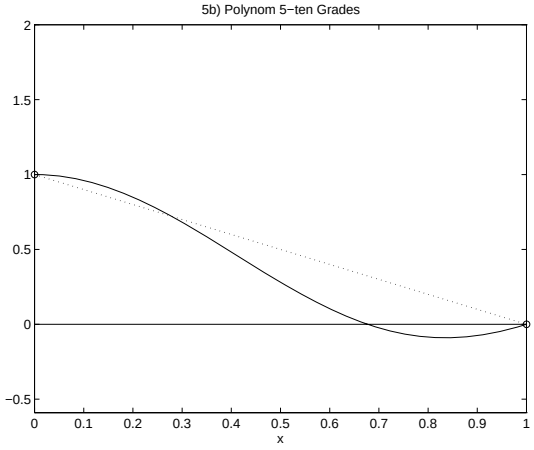
```
disp('Koeffizienten der 2.Ableitung der NF');
a2s = polyder(as)
disp('Kontrolle der 2.Ableitungen in x')
polyval(a2s,x)

% Graphik
for k=1:n+1
  plx(k,1:2) = [x(k)-0.1,x(k)+0.1];
  ply(k,1:2) = [y(k)-0.1*ys(k),y(k)+0.1*ys(k)];
end;
xh = [-0.2 1.2];
plot(x,y,'o',xh,0.*xh,'w',[0 0],xh,'w')
title('5b) Referenz mit 1. und 2. Ableitungen')
xlabel('x')
hold on
for i=1:n+1
  plot(plx(i, : ),ply(i, : ))
end;
hold off
print ser5gr10.ps -dps

xi = linspace(min(x),max(x),100);
pxi = polyval(aa,xi);
plot(x,y,'o',x,y,':',xi,pxi,'y-',x,0.*y,'w',[0 0],[0 max(y)],'w')
title(['5b) Polynom ',sprintf('%1g',neff),'-ten Grades'])
xlabel('x')
axis([min(x) max(x)  min(pxi)-0.5 max(pxi)+1])
print ser5gr11.ps -dps
```

Ergebnisse

```
SDD, HIP, Normalform
Koeffizienten des NIP
     1     0    -4     3     0    -3

neff =
     5

Koeffizienten der NF
a =
    -3     6     0    -4     0     1

Koeffizienten der Ableitung der NF
as =
   -15    24     0    -8     0
```

```
Kontrolle der 1.Ableitungen in x
ans =
     0     1

Koeffizienten der 2.Ableitung der NF
a2s =
   -60    72     0    -8

Kontrolle der 2.Ableitungen in x
ans =
    -8     4
```





4 Spline-Interpolation im $\mathbb{R}^1$

Das ist eine intervallweise Interpolation, wo man stückweise Polynome niedrigen Grades zu einer glatten Gesamtfunktion zusammensetzt.

- Grundintervall I
  $I = [a,b] \subset \mathbb{R}$ mit $-\infty < a < b < \infty$ und (bekannte oder unbekannte) reelle Funktion $f : I \to \mathbb{R}$.

- Stützstellen x_k, Stützwerte y_k und Schrittweite $h_k = x_{k+1} - x_k$

  $$R = \{ \ (x_k, y_k) \mid a \le x_0 < x_1 < x_2 < \ldots < x_n \le b \ \}.$$

  Referenz (Stützstellenfolge) mit $n+1$ paarweise verschiedenen *Stützstellen* und den $n+1$ zugehörigen *Stützwerten* $y_k = f(x_k),\ k = 0, 1, ..., n$.

- Splinefunktion = zusammengesetztes Polynom $s(x) = s(x, R)$ vom Grade m $(m \ge 1)$ mit

  (1) $s(x)$ ist ein Polynom vom Grade $\le m$ auf jedem der Teilintervalle, d.h.

  $$s(x) \in \mathcal{S}_m(R), \quad s(x) = s^{(k)}(x) = \alpha_{k0} + \alpha_{k1}x + ... + \alpha_{km}x^m, \quad x \in [x_k, x_{k+1}],$$

  (2) Glattheit $s(x) \in C^{m-1}(I)$.

- Interpolationsforderung (Interpolationsbedingung)

  (1) $s(x_k) = y_k = f(x_k), \quad k = 0, 1, ..., n.$

  (2) An den inneren Punkten $x_1, x_2, ..., x_{n-1}$ ist $s(x)$ stetig differenzierbar bis zur Ordnung $m-1$, d.h. für $k = 1, 2, ..., n-1$ gilt

  $$\begin{aligned} s^{(k-1)}(x_k) &= s^{(k)}(x_k), \\ s^{(k-1)}(x_k)' &= s^{(k)}(x_k)', \\ &\cdots\cdots\cdots \\ s^{(k-1)}(x_k)^{(m-1)} &= s^{(k)}(x_k)^{(m-1)}. \end{aligned}$$

- Interpolationsaufgabe
  Gesucht sind auf n Teilintervallen insgesamt $n(m+1)$ Koeffizienten α_{kl}, $k = 0, 1, ..., n-1,\ l = 0, 1, ...m$, so daß die Interpolationsforderung erfüllt ist.

Zunächst stellt man fest, daß man für die $n(m+1) = n + 1 + mn - 1$ Unbekannten nur $n + 1 + m(n-1) = n + 1 + mn - m$ Interpolationsbedingungen (1)+(2) zu Verfügung hat. Damit hat $s(x)$ die $m-1$ freien Parameter. Zwecks Eindeutigkeit sind diese durch zusätzliche Bedingungen zu binden.

$m = 1$: kein freier Parameter, eindeutige Lösung=Polygonzug,
$m = 2$: 1 freier Parameter, quadratischer Spline,
$m = 3$: 2 freie Parameter, kubischer Spline.

4.1 Einfache Typen von Splines

- **Linearer Spline**

  Für $x \in [x_k, x_{k+1}]$ sei $s^{(k)}(x) = a_k + b_k(x - x_k),\ k = 0(1)n-1.$

  $2n$ Bedingungen

$$\begin{aligned} s^{(k)}(x_k) &= f_k,\ k = 0(1)n-1, \quad s^{(n-1)}(x_n) = f_n, \\ s^{(k)}(x_k) &= s^{(k-1)}(x_k),\ k = 1(1)n-1. \end{aligned}$$

  Ergebnis: Newtonsche und Lagrangesche Form mit

$$\begin{aligned} s^{(k)}(x) &= f_k + \frac{f_{k+1} - f_k}{x_{k+1} - x_k}(x - x_k), \\ &= \frac{x_{k+1} - x}{x_{k+1} - x_k} f_k + \frac{x - x_k}{x_{k+1} - x_k} f_{k+1},\ k = 0(1)n-1. \end{aligned}$$

- **Quadratische Splines**

  Für $x \in [x_k, x_{k+1}]$ sei $s^{(k)}(x) = a_k + b_k(x - x_k) + c_k(x - x_k)^2, \quad k = 0(1)n-1.$

  $3n$ Bedingungen

$$\begin{aligned} s^{(k)}(x_k) &= f_k,\ k = 0(1)n-1, \quad s^{(n-1)}(x_n) = f_n, \\ s^{(k)}(x_k) &= s^{(k-1)}(x_k), \quad k = 1(1)n-1, \\ s^{(k)}(x_k)' &= s^{(k-1)}(x_k)',\ k = 1(1)n-1, \\ s^{(0)}(x_0)' &= m_0 \quad (m_0 \text{ gegeben oder approximiert}). \end{aligned}$$

  Anstelle der letzten Bedingung $s^{(0)}(x_0)' = m_0$ sind auch andere möglich. Sie werden Endbedingungen genannt, falls sie am Ende des untersuchten Bereichs definiert werden.

  - $s^{(k)}(\bar{x}) = \bar{y}, \quad \bar{x}$ zusätzliche Stelle
  - $s^{(n-1)}(x_n)' = m_n$
  - $s^{(0)}(x_0)' = \ s^{(n-1)}(x_n)'$, Periodizität verbunden mit $f_0 = f_n$
  - $s^{(0)}(x_0)' = -s^{(n-1)}(x_n)'$, Antiperiodizität verbunden mit $f_0 = f_n$
  - $K(s) = \int_{x_0}^{x_n} \omega(x)[s''(x)]^2\, dx \ \to\ \min, \quad \omega(x) > 0$ Gewichtsfunktion

    Gesamtkrümmung der Kurve minimal

4.2 Kubische Splines

Definition **Natürliche kubische Splinefunktion**

Eine *natürliche kubische Splinefunktion* $s(x)$ mit den Stützstellen (Knoten) $x_0 < x_1 < \ldots < x_n$ ist eine reelle Funktion mit folgenden 3 Eigenschaften.

(a) $s(x)$ ist in jedem Teilintervall $[x_k, x_{k+1}]$, $k = 0(1)n-1$, ein Polynom höchstens 3.Grades.

(b) $s(x)$ ist in den Intervallen $(-\infty, x_0)$ und (x_n, ∞) ein Polynom 1.Grades.

(c) $s(x), s'(x), s''(x)$ sind stetig in $\mathbb{R}$ und $s(x)$ interpoliert $f(x)$ an den $n+1$ Stützstellen x_k.

Darstellung des Splines $s(x)$

Sei $s(x) = s^{(k)}(x)$ für $x \in [x_k, x_{k+1}]$ mit

$$s^{(k)}(x) = a_k + b_k(x - x_k) + c_k(x - x_k)^2 + d_k(x - x_k)^3, \; k = 0, 1, ..., n-1.$$

Bedingungen mit $f_k = f(x_k)$

$$\begin{array}{lllll}
(a) & s^{(k)}(x_k) & = & f_k, & k = 0(1)n, \\
(b) & s^{(k)}(x_k) & = & s^{(k-1)}(x_k), & k = 1(1)n, \\
 & s^{(k)}(x_k)' & = & s^{(k-1)}(x_k)', & k = 1(1)n, \\
 & s^{(k)}(x_k)'' & = & s^{(k-1)}(x_k)'', & k = 1(1)n, \\
(c) & s^{(0)}(x_0)'' & = & 0, & \text{(2 Zusatzbedingungen)} \\
 & s^{(n)}(x_n)'' & = & 0, &
\end{array}$$

mit der zusätzlichen Funktion auf $[x_n, \infty)$

$$s^{(n)}(x) = a_n + b_n(x - x_n) + c_n(x - x_n)^2.$$

Die Funktion $s^{(n)}(x)$ wurde künstlich hinzugefügt, ohne die Aufgabenstellung zu verändern, so daß die Anzahl der unbekannten Koeffizienten = Anzahl der Bedingungen $= 4n + 3$ beträgt.

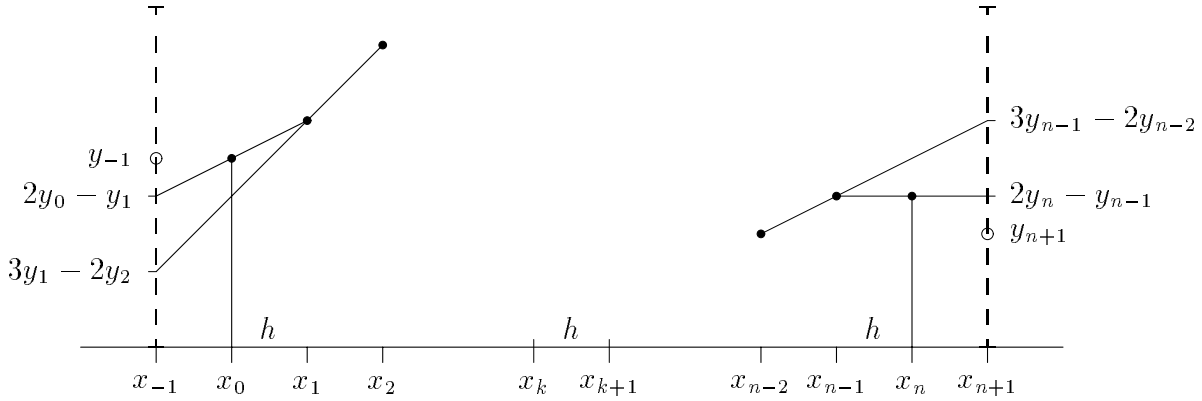
Weitere Typen kubischer Splines neben den natürlichen

- Eingespannte Splines (clamped spline)
  $s'(x_0) = m_0, \;\; s'(x_n) = m_n \;\; (m_0, m_n \text{ gegeben}).$
- Periodische Splines
  $s'(x_0) = s'(x_n), \;\; s''(x_0) = s''(x_n), \;$ wobei $f_0 = f_n$ sinnvoll ist.
- Splines mit not-a-knot Bedingung
  $s'''(x) = const$ ist auf $[x_0, x_1]$ und $[x_1, x_2]$ sowie $[x_{n-2}, x_{n-1}]$ und $[x_{n-1}, x_n]$ identisch. Bei x_1 und x_{n-1} haben wir also stetige 3.Ableitungen. Damit ist $s(x)$ ist auf $[x_0, x_1]$ und $[x_1, x_2]$ bzw. $[x_{n-2}, x_{n-1}]$ und $[x_{n-1}, x_n]$ identisch.
  x_1, x_{n-1} sind somit eigentlich "keine Knoten" mehr.

Eine besondere Form der Spline-Interpolation erhält man durch folgende Vorgehensweise. Voraussetzung sind dabei $n \geq 2$ sowie gleichabständige und aufsteigend geordnete Stützstellen $x_{i+1} = x_i + h$. Nun ergänzt man die Stützstellenfolge links um den Punkt $x_{-1} = x_0 - h$ und rechts um $x_{n+1} = x_n + h$. Die zugehörigen Stützwerte nimmt man aus einer einfachen Extrapolation der Anfangs- bzw. Endwerte.

$$y_{-1} = \frac{3}{2}(2y_0 - y_1) - \frac{1}{2}(3y_1 - 2y_2) = 3y_0 - 3y_1 + y_2,$$

$$y_{n+1} = \frac{3}{2}(2y_n - y_{n-1}) - \frac{1}{2}(3y_{n-1} - 2y_{n-2}) = 3y_n - 3y_{n-1} + y_{n-2}.$$



Dann berechnet man für $k = 0, 1, ..., n-1$ mit der lokalen Referenz aus 4 Punkten $\{(x_{k-1}, y_{k-1}), (x_k, y_k), (x_{k+1}, y_{k+1}), (x_{k+2}, y_{k+2})\}$ das kubische Interpolationspolynom $s_3^{(k)}(x)$, das für die Polynomwertberechnung nur im Intervall $[x_k, x_{k+1}]$ angewendet wird.

Diese Idee geht zurück auf R.G.Keys: *Cubic Convolution Interpolation for Digital Image Processing.* IEEE Trans. on Acoustics, Speech, and Signal Processing, Vol.29, No.6, Dec. 1981, pp. 1153-1160.

Beispiel

$n = 3,\ h = 1,\ \{(x_i, y_i)\} = \{(0,0), (1,1), (2,8), (3,27)\}$

$x_{-1} = -1,\ y_{-1} = 5,\ \ x_4 = 4,\ y_4 = 58$

$$s_3^{(0)}(x) = 3x^2 - 2x, \qquad x \in [0,1]$$

$$s_3^{(1)}(x) = x^3, \qquad x \in [1,2]$$

$$s_3^{(2)}(x) = 6x^2 - 11x + 6, \quad x \in [2,3]$$

Die Ergänzung der kubischen Referenz um die beiden äußeren Punkte bedeutet also nicht unbedingt ihre "kubische Fortsetzung".

Berechnung der Koeffizienten der natürlichen kubischen Splinefunktion

1. Rechte Seiten der Bestimmungsgleichungen
   Seien die Schrittweiten $h_k = x_{k+1} - x_k$ definiert.

$$e_k = 3\left(\frac{f_{k+1} - f_k}{h_k} - \frac{f_k - f_{k-1}}{h_{k-1}}\right), \quad k = 1(1)n-1.$$

2. Bestimmungsgleichungen für die c_k

$$h_{k-1}c_{k-1} + 2(h_{k-1} + h_k)c_k + h_k c_{k+1} = e_k, \quad k = 1(1)n-1,$$

   sowie $c_0 = c_n = 0$.

3. Restliche Splinekoeffizienten

$$\begin{aligned} a_k &= f_k, \quad k = 0(1)n, \\ b_k &= \frac{1}{h_k}(f_{k+1} - f_k) - \frac{1}{3}(2c_k + c_{k+1})h_k, \quad k = 0(1)n-1, \\ d_k &= \frac{1}{3h_k}(c_{k+1} - c_k), \quad k = 0(1)n-1. \end{aligned}$$

Algorithmus **(Lösung der Bestimmungsgleichungen)**

Dies ist ein Gleichungssystem mit diagonal dominanter Tridiagonalmatrix, das mit einer speziellen Variante des Gaußalgorithmus gelöst werden kann.

$$\begin{aligned} (1)\quad \gamma_0 &= 1,\ \gamma_1 = 2(h_0 + h_1),\ g_1 = e_1, \\ (2)\quad \gamma_k &= 2(h_{k-1} + h_k) - \frac{h_{k-1}^2}{\gamma_{k-1}}, \\ g_k &= e_k - \frac{h_{k-1}}{\gamma_{k-1}} g_{k-1},\ k = 2(1)n-1, \\ (3)\quad \gamma_n &= 1,\ g_n = 0, \\ (4)\quad c_n &= 0, \\ c_k &= \frac{1}{\gamma_k}(g_k - h_k c_{k+1}),\ k = n-1(-1)1, \\ c_0 &= 0. \end{aligned}$$

Herleitung der Beziehungen für $a_k,\ b_k,\ c_k,\ d_k$

- Einsetzen in Interpolationsbedingungen

$$\begin{aligned} (a)\quad a_k &= f_k, && k = 0(1)n, \\ (b1)\quad a_k &= a_{k-1} + b_{k-1}h_{k-1} + c_{k-1}h_{k-1}^2 + d_{k-1}h_{k-1}^3, && k = 1(1)n, \\ (b2)\quad b_k &= b_{k-1} + 2c_{k-1}h_{k-1} + 3d_{k-1}h_{k-1}^2, \\ (b3)\quad 2c_k &= 2c_{k-1} + 6d_{k-1}h_{k-1}, \\ (c)\quad c_0 &= 0, \\ c_n &= 0. \end{aligned}$$

- Umstellung

$$
\begin{array}{lll}
(b3) & d_k = \frac{1}{3h_k}(c_{k+1}-c_k), & k = 0(1)n-1, \\
 & \quad d_{k-1} \text{ in } (b2),\ (b1) \text{ einsetzen} & \\
(b2) & b_k = b_{k-1} + (c_k + c_{k-1})h_{k-1}, & k = 1(1)n, \\
(b1) & b_k = \frac{1}{h_k}(a_{k+1}-a_k) - \frac{1}{3}(2c_k + c_{k+1})h_k, & k = 0(1)n-1, \\
(c) & c_0 = c_n = 0. &
\end{array}
$$

- $(b1)$ in $(b2)$ einsetzen

$$\frac{1}{h_k}(a_{k+1}-a_k) - \frac{1}{3}(2c_k + c_{k+1})h_k = \frac{1}{h_{k-1}}(a_k - a_{k-1}) - \frac{1}{3}(2c_{k-1} + c_k)h_{k-1} + (c_k + c_{k-1})h_{k-1}$$

$$h_{k-1}(c_k + c_{k-1} - \tfrac{2}{3}c_{k-1} - \tfrac{1}{3}c_k) + \frac{h_k}{3}(2c_k + c_{k+1}) = \frac{1}{h_k}(a_{k+1}-a_k) - \frac{1}{h_{k-1}}(a_k - a_{k-1})$$

$$h_{k-1}c_{k-1} + 2(h_{k-1}+h_k)c_k + h_k c_{k+1} = \frac{3}{h_k}(f_{k+1}-f_k) - \frac{3}{h_{k-1}}(f_k - f_{k-1}) = e_k, \quad k = 1(1)n-1$$

$$c_0 = c_n = 0.$$

Lösung des Systems mit Tridiagonalmatrix $\Rightarrow$ c_k $\Rightarrow$ d_k, b_k.

Beispiel Natürlicher kubischer Spline

$n = 2$, $h_k = h = 0.5$,

x_k	0	0.5	1
y_k	1	-0.5	2

Bestimmung von c_k

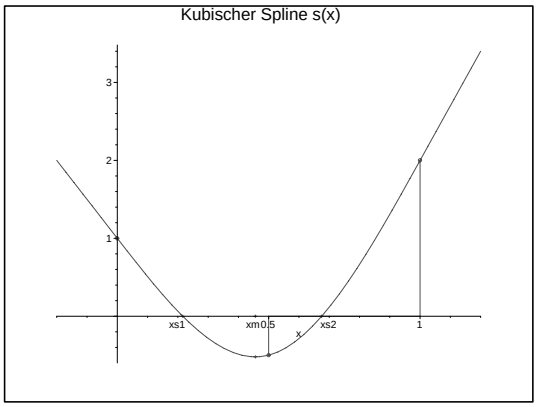
$c_0 = c_2 = 0$, $2(h_0+h_1)c_1 = e_1 = \frac{3}{h}(f_2 - 2f_1 + f_0)$, $a_k = f_k$, $\Rightarrow$ $c_1 = 12$

Bestimmung von b_k

$$
\begin{aligned}
b_0 &= 2(a_1 - a_0) - \tfrac{1}{3}(2c_0 + c_1)\tfrac{1}{2} = -5 \\
b_1 &= b_0 + (c_1 + c_0)\tfrac{1}{2} = 1 \\
b_2 &= b_1 + (c_2 + c_1)\tfrac{1}{2} = 7
\end{aligned}
$$

Bestimmung von d_k

$$
\begin{aligned}
d_0 &= \frac{1}{3\cdot\frac{1}{2}}(c_1 - c_0) = 8 \\
d_1 &= \frac{1}{3\cdot\frac{1}{2}}(c_2 - c_1) = -8
\end{aligned}
$$



$$x_1^* \approx 0.216,\ x_2^* \approx 0.675,\ x_{min} \approx 0.456,\ s(x_{min}) \approx -0.521.$$

Damit

$$
s(x) = \begin{cases}
1 - 5x, & x \in (-\infty, 0] \\
s^{(0)}(x) = 1 - 5x + 8x^3, & x \in [0, 0.5] \\
s^{(1)}(x) = -\frac{1}{2} + (x - \frac{1}{2}) + 12(x - \frac{1}{2})^2 - 8(x - \frac{1}{2})^3, & x \in [0.5, 1] \\
s^{(2)}(x) = 2 + 7(x - 1), & x \in [1, \infty)
\end{cases}
$$

Existenz und Eindeutigkeit der kubischen Splinefunktion

Zu paarweise verschiedenen Stützstellen x_k existiert stets **genau eine** natürliche kubische Splinefunktion.

Optimalität kubischer Splines

In der Klasse $C^2([x_0, x_n])$ der Funtionen $f(x)$, die die Interpolationsbedingung $f(x_k) = y_k,\ k = 0(1)n$, erfüllen, *minimiert* die natürliche kubische Splinefunktion $s(x)$ das Integral für die Gesamtkrümmung

$$\int_{x_0}^{x_n} [f''(x)]^2 dx.$$

MATLAB Funktion zur Berechnung der natürlichen kubischen Splines

$$s^{(k)}(x) = a_k + b_k(x - x_k) + c_k(x - x_k)^2 + d_k(x - x_k)^3, \quad k = 0, 1, ..., n - 1,$$

nach obigen Algorithmus. Man beachte dabei die Verschiebung des Index um 1.

```
% spline_k.m
% Spline-Interpolation
% Intervallweise natuerliche kubische Splines

function [a,b,c,d] = spline_k(n,x,f,ind)

% Eingangsparameter
%  n        Umfang der Referenz, n+1 Punkte, n Intervalle
%  x        Stuetzstellen x[1],...,x[n+1], paarweise verschieden
%  f        Stuetzwerte
%  ind      Ausgabe (1) des LGS fuer c, sonst 0

% Ergebnisse
%  a,b,c,d  Koeffizienten des kubischen Splinepolynoms
%  pk(x) = a[k]+b[k](x-x[k])+c[k](x-x[k])^2+d[k](x-x[k])^3
%          x[k]<=x<=x[k+1], k=1,2,...,n

  for k=1:n
    h(k) = x(k+1)-x(k);
    a(k) = f(k);
  end;
  e(1) = 0;
  for k=2:n
    e(k) = 3*((f(k+1)-f(k))/h(k)-(f(k)-f(k-1))/h(k-1));
  end;
```

```
  al(1) = 1;
  al(2) = 2*(h(1)+h(2));
  g(2) = e(2);
  for k=3:n
    al(k) = 2*(h(k-1)+h(k))-h(k-1)^2/al(k-1);
    g(k)  = e(k)-h(k-1)*g(k-1)/al(k-1);
  end;
  al(n+1) = 1;
  g(n+1)  = 0;

  c(n+1)  = 0;
  c(1) = 0;
  for k=n:-1:2
    c(k) = (g(k)-h(k)*c(k+1))/al(k);
  end;
  b(n+1)  = 0;
  d(n+1)  = 0;
  for k=1:n
    b(k) = (f(k+1)-f(k))/h(k)-(2*c(k)+c(k+1))*h(k)/3;
    d(k) = (c(k+1)-c(k))/(3*h(k));
  end;

  if ind==1
    ma = zeros(n-1);
    ma(1,1) = 2*(h(1)+h(2));
    ma(1,2) = h(2);
    for k=2:n-2
      ma(k,k-1) = h(k);
      ma(k,k) = 2*(h(k)+h(k+1));
      ma(k,k+1) = h(k+1);
    end;
    ma(n-1,n-2) = h(n-1);
    ma(n-1,n-1) = 2*(h(n-1)+h(n));
    disp('Gleichungssystem')
    disp('Tridiagonalmatrix fuer c')
    disp(ma)
    disp(' ')
    disp('Rechte Seite e')
    disp(e(2:n)')
  end;

% Ende Funktion spline_k
```

4.3 Einige MATLAB Funktionen für Splines

- `spline`

  Die kubische Spline-Interpolation mittels im Stützstellenbereich mit den Werten x_i benutzt diese Funktion in den Formen

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

  xi ist ein Argument bzw. ein Argumentvektor aus dem Stützstellenintervall, an dem die Werte der Splinefunktion berechnet werden.
  `pp = spline(x,y)` erzeugt die *pp*-Form (piecewise polynomial), also die Beschreibung des Splines, um diese z.B. für das Kommando `ppval` zu verwenden.

  Die kubische Spline-Interpolation wird mit der not-a-knot Bedingung durchgeführt, die schon beschrieben worden ist.

Einige Erläuterungen zu *pp*-Form

Die Darstellung und der Aufbau des Vektors *pp* ist abhängig vom Umfang der Referenz.
Wir beziehen uns auf die Nummerierung in MATLAB, also der Punktfolge
`(x[1],y[1]),(x[2],y[2]),...,(x[n+1],y[n+1]), n>=1`.

- 2 Stützstellen, $n = 1$

```
Lineare Interpolation
pp(1)=10, pp(2)=1,
pp(3)= 1 (Anzahl der Intervalle)
pp(4)=x[1], pp(5)=x[2]
pp(6)=2  (Anzahl der Koeffizienten
          der Geradengleichung k1*(x-x[1])+k2 )
pp(7)=k1, pp(8)=k2
```

- 3 Stützstellen, $n = 2$

```
Quadratische Interpolation
pp(1)=10, pp(2)=1,
pp(3)= 1 (Anzahl der Intervalle)
pp(4)=x[1], pp(5)=x[3]
pp(6)=3 (Anzahl der Koeffizienten
         der Parabelgleichung k1*(x-x[1])^2+k2*(x-x[1])+k3 )
pp(7)=k1, pp(8)=k2, pp(9)=k3
```

- 4 und mehr Stützstellen, $n \geq 3$

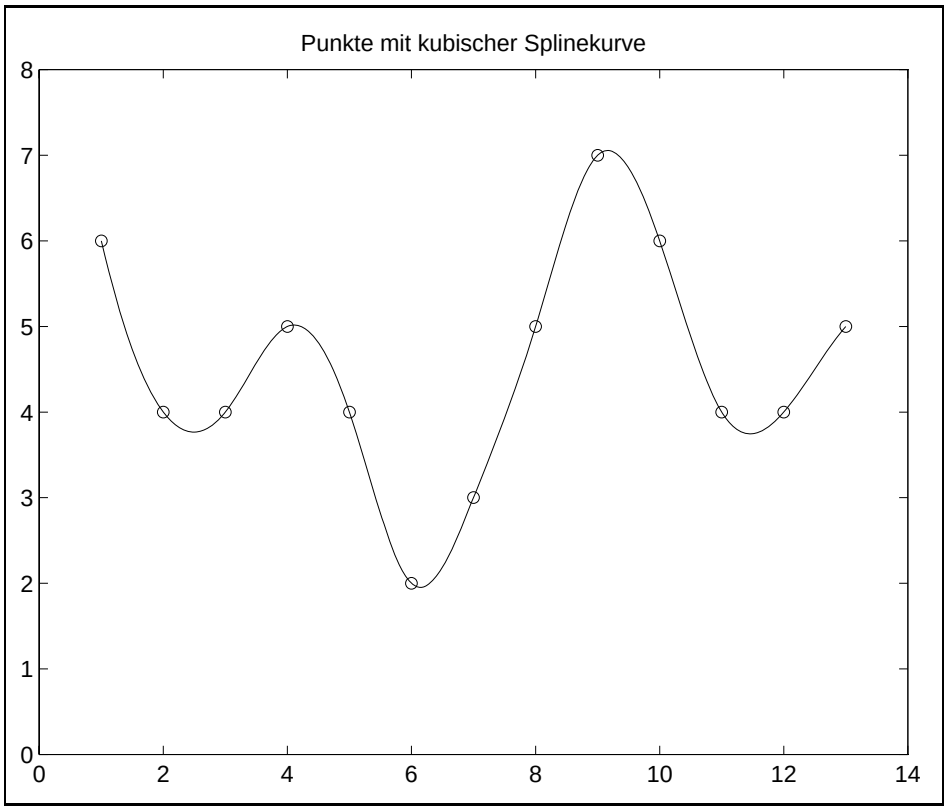
```
Kubische Spline-Interpolation mit not-a-knot Bedingung
pp(1)=10, pp(2)=1,
pp(3)= n (Anzahl der Intervalle)
pp(4)=x[1], pp(5)=x[2],..., pp(n+4)=x[n+1]
```

```
pp(n+5)=4 (Anzahl der Koeffizienten der kubischen Parabel
          pi(x)= ki1*(x-x[i])^3+ki2*(x-x[i])^2+ki3*(x-x[i])+ki4,
          x in [x[i],x[i+1]]  wie folgt geordnet )
pp( n+6)=k11, pp( n+7)=k21,..., pp(2n+5)=kn1
pp(2n+6)=k12, pp(2n+7)=k22,..., pp(3n+5)=kn2
pp(3n+6)=k13, pp(3n+7)=k23,..., pp(4n+5)=kn3
pp(4n+6)=k14, pp(4n+7)=k24,..., pp(5n+5)=kn4
```

Aus dem Vektor *pp* sind also die gewünschten Informationen herauszufiltern.

Anwendungen

```
disp('Spline-Interpolation mit MATLAB-Funktion spline')
% Referenz
x = [1:13]';
y = [6 4 4 5 4 2 3 5 7 6 4 4 5]';
xx= linspace(1,13,200);

% Kubischer Spline mit not-a-knot Bedingung
fs=spline(x,y,xx);      % keine Splinekoeffizienten
plot(x,y,'o',xx,fs,'y-',x,0.*y,'w')
title('Punkte mit kubischer Splinekurve')
print int5gr1.ps -dps
```



```
% Lineare Interpolation
n = 1;
xa = [1 2];
ya = [2 6];
pa = spline(xa,ya);
% Darstellung des linearen Splinepolynoms:
% p1(x)=k1*(x-x[1])+k2
s = [];
s1 = sprintf(' %3.1f',abs(pa(7)));
if (pa(7)<0)
  s1 = [' -',s1];
end;
s = [s,s1];
s3 = sprintf('%3.1f',abs(xa(1)));
if (xa(1)<0)
    s3 = ['+',s3];
  else
    s3 = ['-',s3];
end;
s2 = ['(x',s3,')'];
s = [s,s2];
s1 = sprintf(' %3.1f',pa(8));
if (pa(8)>=0)
   s1 = [' +',s1];
end;
s = [s,s1];
disp(['Linearer Spline auf [x(1),x(2)]   S1(x) = ',s])
disp('Berechnung von piecewise polynomial mittels ppval')
ppval(pa,1.5)
ppval(pa,[xa -1 3])

xxa = linspace(min(xa),max(xa),100);
fs1 = spline(xa,ya,xxa);      % keine Splinekoeffizienten
xxa1= min(xa):0.1:max(xa);
fs2 = ppval(pa,xxa1);
plot(xa,ya,'o',xxa,fs1,'y-',xxa1,fs2,'r+',xa,0.*ya,'w')
title('Punkte mit kubischer Splinekurve = linearer Spline')
print int5gr2.ps -dps
```

Ergebnisse

```
pa =
    10     1     1     1     2     2     4     2

Linearer Spline auf [x(1),x(2)]   S1(x) =  4.0(x-1.0) + 2.0
```

```
Berechnung von piecewise polynomial mittels ppval
ans =
     4

ans =
     2     6    -6    10


% Quadratische Interpolation
n = 2;
xb = [1 2 3];
yb = [2 6 1];
pb = spline(xb,yb)
% Darstellung des quadratischen Splinepolynoms:
% p2(x)=k1*(x-x[1])^2+k2*(x-x[1])+k3
% analog zu oben mit  pb(7), pb(8), pb(9)
disp(['Quadratischer Spline auf [x(1),x(3)]   S2(x) = ',s])
disp('Berechnung von piesewise polynomials  mittels "ppval"')
ppval(pb,1.5)
ppval(pb,[xb -1 4])

xxb = linspace(min(xb),max(xb),100);
fs1 = spline(xb,yb,xxb);      % keine Splinekoeffizienten
xxb1 = min(xb):0.1:max(xb);
fs2 = ppval(pb,xxb1);
plot(xb,yb,'o',xxb,fs1,'y-',xxb1,fs2,'r+',xb,0.*yb,'w')
title('Punkte mit kubischer Splinekurve = quadratischer Spline')
print int5gr3.ps -dps
```

Ergebnisse

```
pb =
   10.0000    1.0000    1.0000    1.0000    3.0000    3.0000   -4.5000
    8.5000    2.0000

Quadratischer Spline auf [x(1),x(3)]
    S2(x) =  - 4.5(x-1.0)^2 + 8.5(x-1.0) + 2.0

Berechnung von piecewise polynomial mittels ppval
ans =
    5.1250

ans =
     2     6     1   -33   -13
```

```
% Kubische Spline-Interpolation
n = 3; xc = [1 2 3 4]; yc = [2 6 1 1];
pc = spline(xc,yc)

% Darstellung des kubischen Splinepolynoms pi(x) auf [x[i],x[i+1]]
% ki1*(x-x[i])^3+ki2*(x-x[i])^2+ki3*(x-x[i])+ki4
for k=1:n
 s = [];
 s1 = sprintf(' %3.1f',abs(pc(n+5+k)));
 if (pc(n+5+k)<0)
  s1 = [' -',s1];
 end;
 s = [s,s1];
 s3 = sprintf('%3.1f',abs(xc(k)));
 if (xc(k)<0)
    s3 = ['+',s3];
  else
    s3 = ['-',s3];
 end;
 s2 = ['(x',s3,')'];
 s = [s,s2,'^3'];
 s1 = sprintf(' %3.1f',pc(2*n+5+k));
 if (pc(2*n+5+k)>=0)
   s1 = [' +',s1];
 end;
 s = [s,s1,s2,'^2'];
 s1 = sprintf(' %3.1f',pc(3*n+5+k));
 if (pc(3*n+5+k)>=0)
   s1 = [' +',s1];
 end;
 s = [s,s1,s2];
 s1 = sprintf(' %3.1f',pc(4*n+5+k));
 if (pc(4*n+5+k)>=0)
   s1 = [' +',s1];
 end;
 s = [s,s1];
 disp(['Kubischer Spline auf [x(',num2str(k),'),...
       x(',num2str(k+1),')]   S3(x) = ',s])
end;

% Eine einzige NF
k = 1;
ak = allinvhs(3,[xc(k) xc(k) xc(k) xc(k)],...
     [pc(4*n+5+k) pc(3*n+5+k) pc(2*n+5+k) pc(n+5+k)])
disp(['NF : ',poly2sym(ak,'x')])
```

```
disp('Berechnung von piecewise polynomial mittels ppval')
ppval(pc,1.5)
ppval(pc,[xc -1 5])

xxc = linspace(min(xc),max(xc),100);
fs1 = spline(xc,yc,xxc);       % keine Splinekoeffizienten
xxc1= min(xc):0.1:max(xc);
fs2 = ppval(pc,xxc1);
plot(xc,yc,'o',xxc,fs1,'y-',xxc1,fs2,'r+',xc,0.*yc,'w')
title('Punkte mit kubischer Splinekurve')
print int5gr4.ps -dps
```

Ergebnisse

Das kubische Splinepolynom hat auf dem Intervall [1,4] die einheitliche Darstellung

$$S_3(x) = \frac{7}{3}x^3 - \frac{37}{2}x^2 + \frac{259}{6}x - 25.$$

```
pc =
   10.0000    1.0000    3.0000    1.0000    2.0000    3.0000    4.0000
    4.0000    2.3333    2.3333    2.3333  -11.5000   -4.5000    2.5000
   13.1667   -2.8333   -4.8333    2.0000    6.0000    1.0000

Kubischer Spline auf [x(1),x(2)]
    S3(x) =  2.3(x-1.0)^3 -11.5(x-1.0)^2 + 13.2(x-1.0) + 2.0
Kubischer Spline auf [x(2),x(3)]
    S3(x) =  2.3(x-2.0)^3 -4.5(x-2.0)^2 -2.8(x-2.0) + 6.0
Kubischer Spline auf [x(3),x(4)]
    S3(x) =  2.3(x-3.0)^3 + 2.5(x-3.0)^2 -4.8(x-3.0) + 1.0

ak =
    2.3333  -18.5000   43.1667  -25.0000

NF : 1313549891316395/562949953421312*x^3
    -2603643534573569/140737488355328*x^2
    +3037584123669163/70368744177664*x
    -25

Berechnung von piecewise polynomial mittels ppval
ans =
     6

ans =
    2.0000    6.0000    1.0000    1.0000  -89.0000   20.0000
```

```
% Kubische Spline-Interpolation
n = 4;
xd = [1 2 3 4 5];
yd = [2 6 1 1 3];
pd = spline(xd,yd)
% Darstellung des kubischen Splinepolynoms pi auf [x[i],x[i+1]]
% ki1*(x-x[i])^3+ki2*(x-x[i])^2+ki3*(x-x[i])+ki4
for k=1:n
 ....
 s = [s,s1];
 disp(['Kubischer Spline auf [x(',num2str(k),'),...
        x(',num2str(k+1),')]   S3(x) = ',s])
 ak = allinvhs(3,[xd(k) xd(k) xd(k) xd(k)],...
                 [pd(4*n+5+k) pd(3*n+5+k) pd(2*n+5+k) pd(n+5+k)])
end;
disp('Berechnung von piecewise polynomial mittels ppval')
ppval(pd,1.5)
ppval(pd,[xd -1 5])

xxd = linspace(min(xd),max(xd),100);
fs1 = spline(xd,yd,xxd);      % keine Splinekoeffizienten
xxd1= min(xd):0.1:max(xd);
fs2 = ppval(pd,xxd1);
plot(xd,yd,'o',xxd,fs1,'y-',xxd1,fs2,'r+',xd,0.*yd,'w')
title('Punkte mit kubischer Splinekurve')
print int5gr5.ps -dps
```

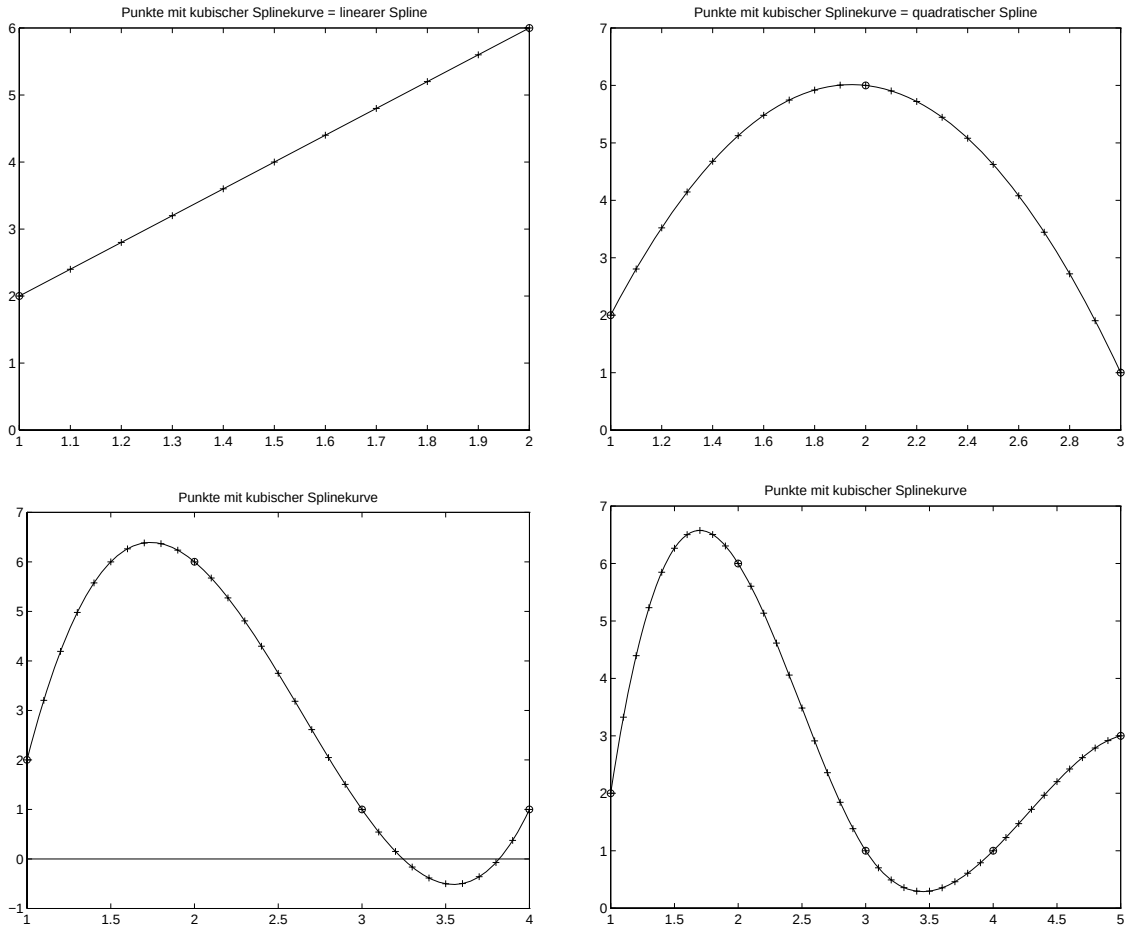
Ergebnisse

Die kubischen Splinepolynome im 1. und 2. bzw. 3. und 4.Intervall sind identisch.

```
pd =
   10.0000    1.0000    4.0000    1.0000    2.0000    3.0000    4.0000
    5.0000    4.0000    3.0417    3.0417   -1.2083   -1.2083  -13.6250
   -4.5000    4.6250    1.0000   14.5833   -3.5417   -3.4167    2.2083
    2.0000    6.0000    1.0000    1.0000

Kubischer Spline auf [x(1),x(2)]
 S3(x) =  3.0417(x-1.0000)^3-13.6250(x-1.0000)^2+14.5833(x-1.0000)+2.0000
ak =
    3.0417  -22.7500   50.9583  -29.2500

Kubischer Spline auf [x(2),x(3)]
 S3(x) =  3.0417(x-2.0000)^3-4.5000(x-2.0000)^2-3.5417(x-2.0000)+6.0000
ak =
    3.0417  -22.7500   50.9583  -29.2500
```

```
Kubischer Spline auf [x(3),x(4)]
 S3(x) =  -1.2083(x-3.0000)^3+4.6250(x-3.0000)^2-3.4167(x-3.0000)+1.0000
ak =
   -1.2083   15.5000  -63.7917   85.5000

Kubischer Spline auf [x(4),x(5)]
 S3(x) =  -1.2083(x-4.0000)^3+1.0000(x-4.0000)^2+2.2083(x-4.0000)+1.0000
ak =
   -1.2083   15.5000  -63.7917   85.5000

Berechnung von piecewise polynomial mittels ppval
ans =
    6.2656

ans =
    2.0000    6.0000    1.0000    1.0000    3.0000 -106.0000    3.0000
```

4 Graphiken zu $n = 1, 2, 3, 4$ (`int5gr2..5.ps`).



- `interp1`

  Die eindimensionale Interpolation im Stützstellenbereich mit monoton wachsenden Werten x_i benutzt diese Funktion in den Formen

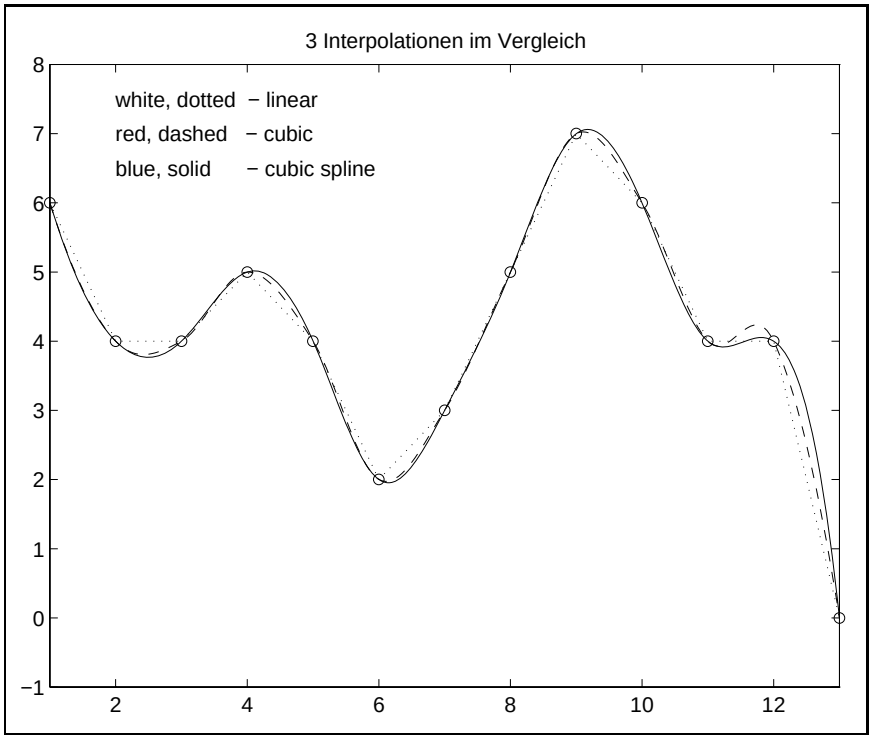
  ```
  yi = interp1(x,y,xi)
  yi = interp1(x,y,xi,'linear')
  yi = interp1(x,y,xi,'cubic')
  yi = interp1(x,y,xi,'spline')
  ```

  wobei die ersten beiden identisch sind. Die letzte Zeile kann auch mit
  `yi=spline(x,y,xi)` notiert werden.
  xi ist ein Argument bzw. ein Argumentvektor aus dem Stützstellenintervall, an dem die Werte der Spline-Funktion berechnet werden.
  Im Fall der Methode `cubic` müssen die Stützstellen äquidistant und ihre Anzahl ≥ 3 sein.

  Die Methode `linear` erzeugt einen Polygonzug zur Referenz.
  Die Methode `cubic` wurde oben als *Cubic Convolution Interpolation* erläutert und ist eher eine lokale kubische Interpolation mit Verschiebung. Sie benutzt das m-File `icubic`.
  Die Methode `spline` verwendet das gleichnamige m-File und ist die kubische Spline-Interpolation mit der not-a-knot Bedingung, wobei bei 2 Stützstellen linear bzw. bei 3 Stützstellen quadratisch interpoliert wird.

```
disp('Interpolation mit MATLAB Funktion interp1')

% Referenz
x = [1:13]';              % aequdistant und monoton
y = [6 4 4 5 4 2 3 5 7 6 4 4 0]';
plot(x,y,'o')
title('Datenwolke P(x,y) ')
xlabel('x')
ylabel('y')

h = min(x):0.05:max(x);
t = interp1(x,y,1.5)      % Argumente in [min(x),max(x)]
t1 = interp1(x,y,h,'linear');
plot(x,y,'o',h,t1,'y')
axis([min(x) max(x) min(y)-1 max(y)+1])
title('Lineare Interpolation (Polygonzug)')

t2 = interp1(x,y,h,'cubic');
plot(x,y,'o',h,t2,'r')
axis([min(x) max(x) min(y)-1 max(y)+1])
title('Kubische Interpolation (cubic convolution interpolation)')
```

```
xx = linspace(1,13,200);
fs=spline(x,y,xx);
t3 = interp1(x,y,h,'spline');
plot(x,y,'o',h,t3,'b')
axis([min(x) max(x) min(y)-1 max(y)+1])
title('Kubische Spline-Interpolation mit not-a-knot Bedingung')

% 3 Graphen im Vergleich
plot(x,y,'o',h,t1,'w:')
axis([min(x) max(x) min(y)-1 max(y)+1])
title('3 Interpolationen im Vergleich')
text(2,7.5,'white, dotted  - linear')
text(2,7.0,'red, dashed    - cubic')
text(2,6.5,'blue, solid      - cubic spline')
hold on
plot(h,t2,'r--')
plot(h,t3,'b-')
hold off
print int4gr1.ps -dps
```



- `ppval`

  Die eindimensionale Interpolation im Stützstellenbereich mit einer gegebenen Referenz $\{(x_i, y_i)\}$ liefert mittels `pp = spline(x,y)` die Beschreibung des kubischen Splines (piecewise polynomial).
  Daraus kann man nun bei gegebenen Argument bzw. Argumentvektor xi die zugehörigen Werte der Splinefunktion berechnen gemäß `yi = ppval(pp,xi)`.

4.4 Beispiele für die Spline-Interpolation

Wir verwenden dabei die vorgestellten und weitere MATLAB Routinen und Kommandos für Polynome und Interpolation.

```
% Kubischer Spline mit not-a-knot Bedingung
  y = spline(xi,yi,x)

% Beschreibung des kubischen Splines mit not-a-knot Bedingung
  pp = spline(xi,yi)

% Berechnung von Spline-Werten als piecewise polynomial
  y = ppval(pp,x)

% Natuerliche kubische Splines -> Koeffizienten intervallweise
% m-File
  [a b c d] = spline_k(n-1,xi,yi,ind)

% Polynomwertberechnung mittels Koeffizienten
  y0 = polyval(a,x0)   % x0 Einzelargument
  yy = polyval(a,xx)   % xx Argumentvektor

% Funktionswertberechnung aus symbolischer Form der Funktion
  f='sin(x)'
  numeric(subs(fs,xi,'x'))

% Koeffizienten der NF des Splinepolynoms auf Intervall
  sk = sym2poly(s)
```

Beispiel 1

Die Ergebnisse sind numerisch und graphisch darzustellen.
Gegeben sei die Referenz

$$\{(x_i, y_i) = (i\pi/n, f(x_i)),\ i = 0, 1, ..., n = 5, \quad f(x) = \sin x\}.$$

1. Interpoliere die Funktion $f(x)$ im Intervall $[0, \pi]$ bei dieser Referenz mittels kubischer Splines $S_3(x)$ unter Verwendung der Funktion `spline`.
   Gebe die Splinefunktion intervallweise an, auch in der Normalform.

2. Löse Teil 1 mittels der natürlichen kubischen Splinefunktion $S(x)$ unter Verwendung der Funktion `spline_k`.

3. Ermittle die Ableitungen $S'(x_i),\ i = 0, 1, ..., n$, und vergleiche diese mit den exakten Werten $f'(x_i)$.

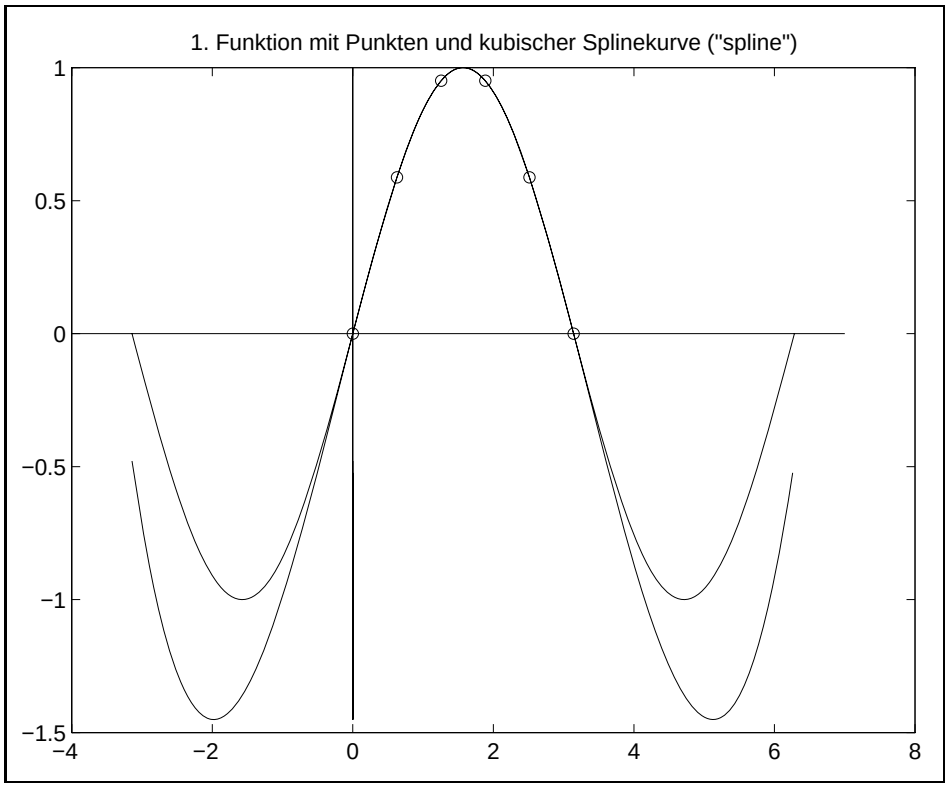
```
% 1. Referenz und kubische Splinefunktion mit y=sin(x)
n = 6;           % 6 Stuetzstellen, 5 Intervalle
xi = linspace(0,pi,n)
yi = sin(xi)
xxi = linspace(-pi,2*pi,200);
yyi = sin(xxi);
z=-pi:0.05:2*pi;

% Kubischer Spline mit not-a-knot Bedingung : spline
disp('Kubischer Spline mit not-a-knot Bedingung : spline')
pp = spline(xi,yi)

% Darstellung des kubischen Splinepolynoms pi(x) auf [x[i],x[i+1]]
% ki1*(x-x[i])^3+ki2*(x-x[i])^2+ki3*(x-x[i])+ki4
%
m = n-1;
for i=1:m
 s = [];
 s1 = sprintf(' %6.4f',abs(pp(m+5+i)));
 if (pp(m+5+i)<0)
  s1 = [' -',s1];
 end;
 s = [s,s1];
 s3 = sprintf('%6.4f',abs(xi(i)));
 if (xi(i)<0)
    s3 = ['+',s3];
  else
    s3 = ['-',s3];
 end;
 s2 = ['(x',s3,')'];
 s = [s,s2,'^3'];
 s1 = sprintf(' %6.4f',pp(2*m+5+i));
 if (pp(2*m+5+i)>=0)
   s1 = [' +',s1];
 end;
 s = [s,s1,s2,'^2'];
 s1 = sprintf(' %6.4f',pp(3*m+5+i));
 if (pp(3*m+5+i)>=0)
   s1 = [' +',s1];
 end;
 s = [s,s1,s2];
 s1 = sprintf(' %6.4f',pp(4*m+5+i));
 if (pp(4*m+5+i)>=0)
   s1 = [' +',s1];
 end;
```

```
 s = [s,s1];
 disp(['Kubischer Spline auf [x(',num2str(i),'), x(',num2str(i+1),']'])
 disp(['   S3(x) = ',s])
end;

[ pp(m+6:2*m+5)' pp(2*m+6:3*m+5)' pp(3*m+6:4*m+5)' pp(4*m+6:5*m+5)']

pz = ppval(pp,z);
plot(xi,yi,'o',z,pz,xxi,yyi,'g',[-5,10],[0,0],'w',0.*z,pz,'w')
title('1a) Funktion mit Punkten und kubischer Splinekurve ("spline")')
print ser6gr01.ps -dps
```

Ergebnisse zu 1.

Die NF der kubischen Splines auf dem 1. und 2. bzw. 4. und 5.Intervall sind identisch.

```
xi =
         0    0.6283    1.2566    1.8850    2.5133    3.1416

yi =
         0    0.5878    0.9511    0.9511    0.5878    0.0000

Kubischer Spline mit not-a-knot Bedingung : spline
pp =
   10.0000    1.0000    5.0000         0    0.6283    1.2566    1.8850
    2.5133    3.1416    4.0000   -0.1119   -0.1119    0.0000    0.1119
    0.1119   -0.0735   -0.2844   -0.4952   -0.4952   -0.2844    1.0258
    0.8010    0.3112   -0.3112   -0.8010         0    0.5878    0.9511
    0.9511    0.5878

Kubischer Spline auf [x(1), x(2]
   S3(x) =  -0.1119(x-0.0000)^3-0.0735(x-0.0000)^2+1.0258(x-0.0000)+0.0000
Kubischer Spline auf [x(2), x(3]
   S3(x) =  -0.1119(x-0.6283)^3-0.2844(x-0.6283)^2+0.8010(x-0.6283)+0.5878
Kubischer Spline auf [x(3), x(4]
   S3(x) =  -0.0000(x-1.2566)^3-0.4952(x-1.2566)^2+0.3112(x-1.2566)+0.9511
Kubischer Spline auf [x(4), x(5]
   S3(x) =   0.1119(x-1.8850)^3-0.4952(x-1.8850)^2-0.3112(x-1.8850)+0.9511
Kubischer Spline auf [x(5), x(6]
   S3(x) =   0.1119(x-2.5133)^3-0.2844(x-2.5133)^2-0.8010(x-2.5133)+0.5878

ans =
   -0.1119   -0.0735    1.0258         0
   -0.1119   -0.2844    0.8010    0.5878
    0.0000   -0.4952    0.3112    0.9511
    0.1119   -0.4952   -0.3112    0.9511
    0.1119   -0.2844   -0.8010    0.5878
```




```
% 2. Natuerliche kubische Splines : spline_k
%    pk(x) = a[k]+b[k]*(x-x[k])+c[k]*(x-x[k])^2+d[k]*(x-x[k])^3
%            x[k]<=x<=x[k+1], k=1,2,...,n-1

[a b c d] = spline_k(n-1,xi,yi,0);

for k=1:n-1
  interv(k,:) = ['[',sprintf('%6.4f',xi(k)),', ',sprintf('%6.4f',xi(k+1)),']'];
end;
disp('Natuerliche kubische Splines : spline_k')
disp(' ')
disp('[x[k],x[k+1]]      d[k]*(x-x[k])^3+c[k]*(x-x[k])^2+b[k]*(x-x[k])+a[k]')
disp('                   d[k]      c[k]      b[k]      a[k] ')
for k=1:n-1
  disp([interv(k,:),sprintf('   %7.4f',d(k)),...
                    sprintf('   %7.4f',c(k)),...
                    sprintf('   %7.4f',b(k)),...
                    sprintf('   %7.4f',a(k))])
end;

[ d(1:n-1)' c(1:n-1)' b(1:n-1)' a(1:n-1)']

Sk = 'd(k)*(x-x(k))^3+c(k)*(x-x(k))^2+b(k)*(x-x(k))+a(k)'
```

```
% Normalformen

disp('Normalformen, symbolisch')
S1 = expand(subs(subs(subs(subs(subs(Sk,xi(1),'x(k)'),...
        d(1),'d(k)'),c(1),'c(k)'),...
        b(1),'b(k)'),a(1),'a(k)'))
numeric(subs(S1,xi(1),'x'))   % Kontrolle
numeric(subs(S1,1,'x'))
S1k = sym2poly(S1);
Sm(1,1:4) = S1k;

S2 = expand(subs(subs(subs(subs(subs(Sk,xi(2),'x(k)'),...
        d(2),'d(k)'),c(2),'c(k)'),...
        b(2),'b(k)'),a(2),'a(k)'))
numeric(subs(S2,xi(2),'x'))
S2k = sym2poly(S2);
Sm(2,1:4) = S2k;

S3 = expand(subs(subs(subs(subs(subs(Sk,xi(3),'x(k)'),...
        d(3),'d(k)'),c(3),'c(k)'),...
        b(3),'b(k)'),a(3),'a(k)'))
numeric(subs(S3,xi(3),'x'))
S3k = sym2poly(S3);
Sm(3,1:4) = S3k;

S4 = expand(subs(subs(subs(subs(subs(Sk,xi(4),'x(k)'),...
        d(4),'d(k)'),c(4),'c(k)'),...
        b(4),'b(k)'),a(4),'a(k)'))
numeric(subs(S4,xi(4),'x'))
S4k = sym2poly(S4);
Sm(4,1:4) = S4k;

S5 = expand(subs(subs(subs(subs(subs(Sk,xi(5),'x(k)'),...
        d(5),'d(k)'),c(5),'c(k)'),...
        b(5),'b(k)'),a(5),'a(k)'))
numeric(subs(S5,xi(5),'x'))
numeric(subs(S5,xi(6),'x'))
S5k = sym2poly(S5);
Sm(5,1:4) = S5k;

disp('Normalformen intervallweise, numerisch')
xi
disp('      x^3       x^2       x^1       x^0')
disp([S1k; S2k; S3k; S4k; S5k])     % auch disp(Sm)
```

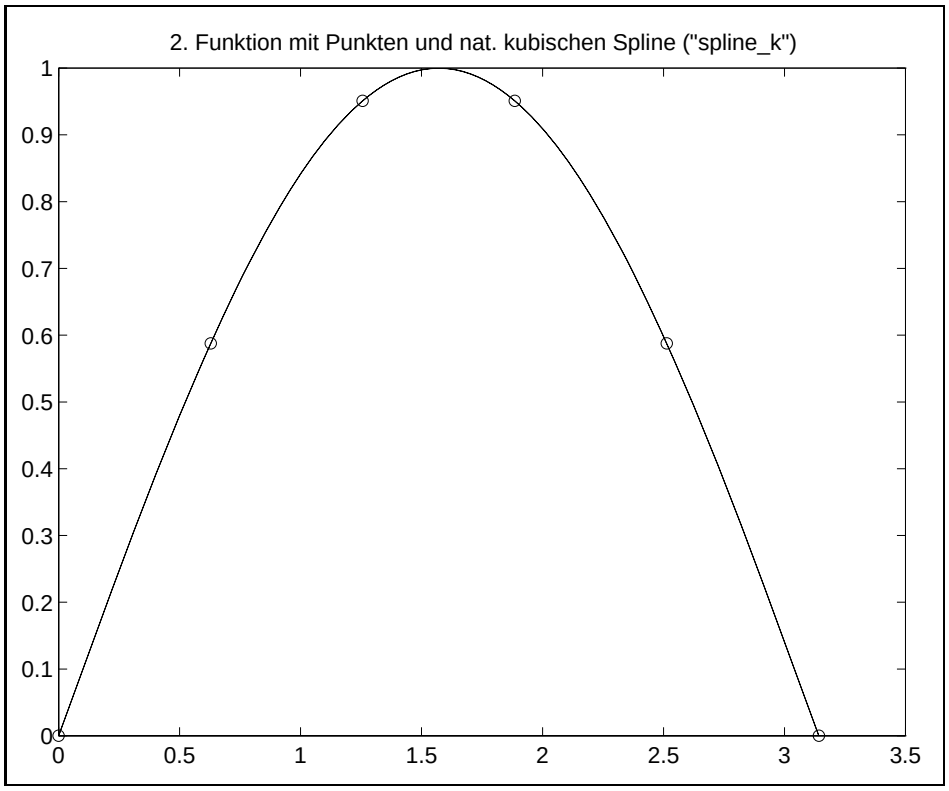
```
xp = [];
yp = [];
for k=1:n-1
  xh = linspace(xi(k),xi(k+1),20);
  yh = polyval(Sm(k,1:4),xh);
  xp = [xp xh];
  yp = [yp yh];
end;

% Graphik
xxi = linspace(0,pi,100);
yyi = sin(xxi);
plot(xi,yi,'o',xxi,yyi,'y',xp,yp,'b', xxi,0.*yyi,'w',0.*xxi,yyi,'w')
title('1a) Funktion mit Punkten und nat. kubischen Spline ("spline_k")')
print ser6gr02.ps -dps
```

Ergebnisse zu 2.

```
Natuerliche kubische Splines : spline_k

[x[k],x[k+1]]      d[k]*(x-x[k])^3+c[k]*(x-x[k])^2+b[k]*(x-x[k])+a[k]
                   d[k]      c[k]      b[k]      a[k]
[0.0000, 0.6283]   -0.1611    0.0000    0.9991    0.0000
[0.6283, 1.2566]   -0.0996   -0.3037    0.8083    0.5878
[1.2566, 1.8850]    0.0000   -0.4914    0.3087    0.9511
[1.8850, 2.5133]    0.0996   -0.4914   -0.3087    0.9511
[2.5133, 3.1416]    0.1611   -0.3037   -0.8083    0.5878

ans =
   -0.1611         0    0.9991         0
   -0.0996   -0.3037    0.8083    0.5878
    0.0000   -0.4914    0.3087    0.9511
    0.0996   -0.4914   -0.3087    0.9511
    0.1611   -0.3037   -0.8083    0.5878

Sk =
d(k)*(x-x(k))^3+c(k)*(x-x(k))^2+b(k)*(x-x(k))+a(k)

Normalformen, symbolisch
S1 =
-2902279292867599/18014398509481984*x^3+1124878168675093/1125899906842624*x

ans =
     0
ans =
    0.8380
```

```
S2 =
-1793707247837181/1801439850948l984*x^3+5381121743511543/9007199254740992*x^2*pi
-5381121743511543/45035996273704960*x*pi^2+1793707247837181/22517998136852480*pi^3
-683833447885397/2251799813685248*x^2+683833447885397/5629499534213120*x*pi
-683833447885397/5629499534213200*pi^2+910045555059519/1125899906842624*x
-910045555059519/5629499534213120*pi+2647149443198255/4503599627370496

ans =
    0.5878

S3 =
-2090579582008261/5070602400912917605986812821504*x^3
+6271738746024783/12676506002282294014967032053760*x^2*pi
-6271738746024783/31691265005705735037417580134400*x*pi^2
+2090579582008261/7922816251426433759354395033600O*pi^3
-1106465761322601/2251799813685248*x^2+1106465761322601/2814749767106560*x*pi
-1106465761322601/14073748835532800*pi^2+695212941443945/2251799813685248*x
-139042588288789/1125899906842624*pi+33462326346837/35184372088832

ans =
    0.9511

S4 =
896853623918595/9007199254740992*x^3-1614336523053471/9007199254740992*x^2*pi
+4843009569160413/45035996273704960*x*pi^2-4843009569160413/22517998136852480*pi^3
-1106465761322603/2251799813685248*x^2+3319397283967809/5629499534213120*x*pi
-9958191851903427/56294995342131200*pi^2-1390425882887889/4503599627370496*x
+4171277648663667/22517998136852480*pi+33462326346837/35184372088832

ans =
    0.9511

S5 =
1451139646433799/9007199254740992*x^3-4353418939301397/11258999068426240*x^2*pi
+4353418939301397/14073748835532800*x*pi^2-1451139646433799/17592186044416000*pi^3
-2735333791541587/9007199254740992*x^2+2735333791541587/5629499534213120*x*pi
-2735333791541587/14073748835532800*pi^2-910045555059519/1125899906842624*x
+910045555059519/1407374883553280*pi+165446840199891/281474976710656

ans =
    0.5878

ans =
  3.0000e-016

Normalformen intervallweise, numerisch
xi =
         0    0.6283    1.2566    1.8850    2.5133    3.1416
```

```
     x^3       x^2       x^1       x^0
  -0.1611         0    0.9991         0
  -0.0996   -0.1160    1.0720   -0.0153
   0.0000   -0.4914    1.5437   -0.2129
   0.0996   -1.0544    2.6050   -0.8797
   0.1611   -1.5184    3.7711   -1.8566
```



Graphisch ist der kleine Unterschied zwischen $f(x)$ und $S(x)$ nicht zu erkennen.
Die Differentiation des lokalen kubischen Splinepolynoms liefert

$$s^{(k)}(x)' = b_k + 2c_k(x - x_k) + 3d_k(x - x_k)^2,$$

womit die Berechnung der 1.Ableitung an beliebigen Stellen $x \in [x_k, x_{k+1}]$ erfolgen kann, insbesondere ist $s^{(k)}(x_k)' = b_k$.

```
% 3. Ableitungen, Vergleich
ysi = cos(xi)
f = 'sin(x)'
fs = diff(f)
ysi1 = numeric(subs(fs,xi,'x'));     % wie ysi
ss(1:n-1) = b(1:n-1);
ss(n) = 3*d(n-1)*(xi(n)-xi(n-1))^2+...
        2*c(n-1)*(xi(n)-xi(n-1))+b(n-1);
disp('    k        y'(x(k))  s'(x(k))  y'(x(k))-s'(x(k))')
disp([ [1:n]' ysi' ss' (ysi-ss)']) 
```

```
for k=1:n
  kv(k) = str2num(sprintf('%1g',k));
end;
```

Ergebnisse zu 3.

```
ysi =
    1.0000    0.8090    0.3090   -0.3090   -0.8090   -1.0000

f =
sin(x)

fs =
cos(x)

    k        y'(x(k))  s'(x(k))  y'(x(k))-s'(x(k))
    1.0000    1.0000    0.9991    0.0009
    2.0000    0.8090    0.8083    0.0007
    3.0000    0.3090    0.3087    0.0003
    4.0000   -0.3090   -0.3087   -0.0003
    5.0000   -0.8090   -0.8083   -0.0007
    6.0000   -1.0000   -0.9991   -0.0009
```

Beispiel 2

Lineare und natürliche kubische Splines mit `spline_k` zur Referenz

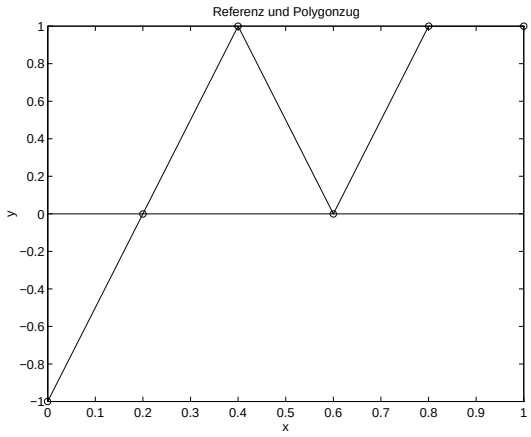
x_i	0	1	2	3	4	5
y_i	-1	0	1	0	1	1

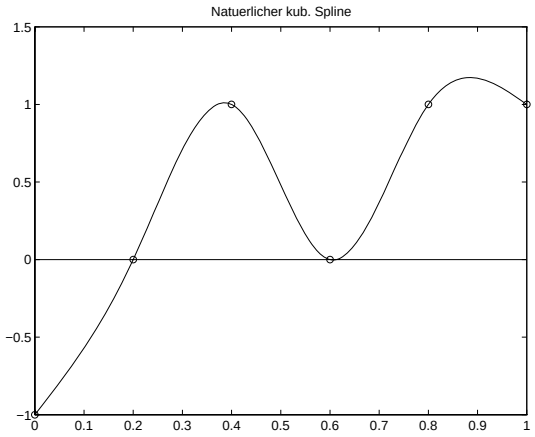
```
% Referenz
n = 6;
x = linspace(0,1,n)
y = [-1, 0,  1,  0,  1, 1 ]
plot(x,y,'o',x,y,[min(x),max(x)],[0,0],'w',...
     [0,0],[min(y),max(y)],'w')
title('Referenz und Polygonzug')
xlabel('x')
ylabel('y')
print ser6gr08.ps -dps

% UP-Aufruf mit Angabe des GS
[a b c d] = spline_k(n-1,x,y,1);    % Ausgabe des LGS

disp('Natuerliche kubische Splines : spline_k')
```

```
disp('Koeffizienten d,c,b,a fuer (x,y) intervallweise')
[ d(1:n-1)' c(1:n-1)' b(1:n-1)' a(1:n-1)']

xp = []; yp = [];
for k=1:n-1
  xh = linspace(x(k),x(k+1),20);
  Syk = d(k).*(xh-x(k)).^3+c(k).*(xh-x(k)).^2+b(k).*(xh-x(k))+a(k);
  xp = [xp xh];
  yp = [yp Syk];
end;

plot(x,y,'o',xp,yp,'y',[0,1],[0,0],'w',[0,0],[-1,1.5],'w')
title('Natuerlicher kub. Spline')
print ser6gr09.ps -dps
```

Ergebnisse

```
x =
         0    0.2000    0.4000    0.6000    0.8000    1.0000

y =
    -1     0     1     0     1     1

Gleichungssystem
Tridiagonalmatrix fuer c
    0.8000    0.2000         0         0
    0.2000    0.8000    0.2000         0
         0    0.2000    0.8000    0.2000
         0         0    0.2000    0.8000

Rechte Seite e
         0
  -30.0000
   30.0000
  -15.0000

Natuerliche kubische Splines : spline_k
Koeffizienten d,c,b,a fuer (x,y) intervallweise

ans =
   23.3254         0    4.0670   -1.0000
 -116.6268   13.9952    6.8660         0
  193.1818  -55.9809   -1.5311    1.0000
 -156.1005   59.9282   -0.7416         0
   56.2201  -33.7321    4.4976    1.0000
```





Beispiel 3

Vergleich von Interpolationspolynom, kubischen Splines mit not-a-knot Bedingung und natürlichen kubischen Splines zur Referenz

x_i	1	4	6	9
y_i	2	5	3	6

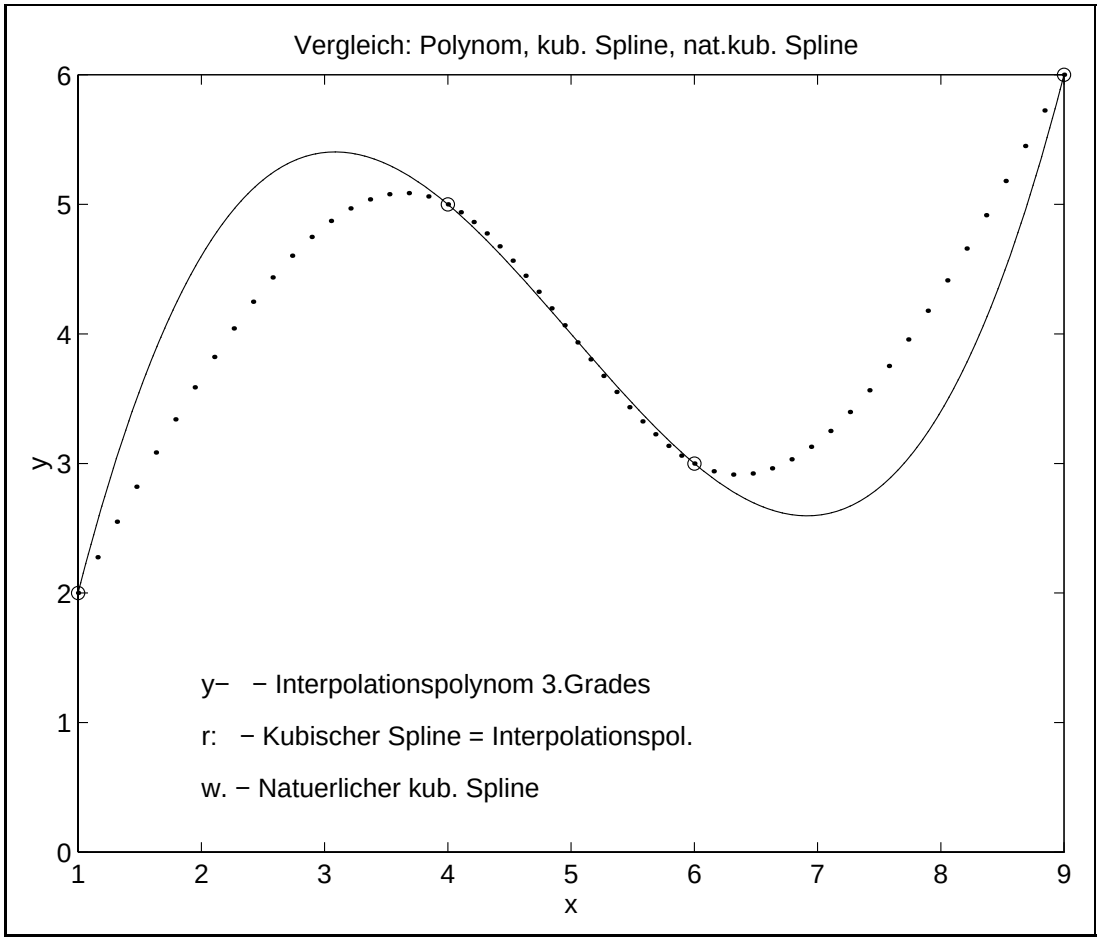
```
% Referenz
n = 4;
x = [1, 4, 6, 9]
y = [2, 5, 3, 6]
xx = linspace(min(x),max(x),100);

% Polynom 3.Grades
disp('Koeffizienten des Interpolationspolynoms')
p = polyfit(x,y,3)
py = polyval(p,xx);
plot(x,y,'o',xx,py,'y-',[min(x),max(x)],[0,0],'w')
title('Vergleich: Polynom, kub. Spline, nat.kub. Spline')
xlabel('x')
ylabel('y')
text(2,1.4,'y-   - Interpolationspolynom 3.Grades')
hold on

% Kubischer Spline : spline
pp = spline(x,y)

disp('Alle Koeffizienten (Zeile-Intervall-Polynom)')
[ pp(9:11)' pp(12:14)' pp(15:17)' pp(18:20)']
```

```
disp('Normalform im ersten Intervall')
Sk = 'd(k)*(x-x(k))^3+c(k)*(x-x(k))^2+b(k)*(x-x(k))+a(k)'
S1 = expand(subs(subs(subs(subs(subs(Sk,x(1),'x(k)'),...
        pp(9),'d(k)'),pp(12),'c(k)'),...
        pp(15),'b(k)'),pp(18),'a(k)'))

numeric(subs(S1,x(1),'x'))      % Kontrolle

disp('Koeffizienten des kubischen Splines')
S1k = sym2poly(S1)
disp('Kub.Spline = Interpolationspolynom 3.Grades')

ppy = spline(x,y,xx);      % analog ppy = ppval(pp,xx)
plot(xx,ppy,'r:')
text(2,0.9,'r:   - Kubischer Spline = Polynom')

% Natuerlicher kubischer Spline : spline_k
[a b c d] = spline_k(n-1,x,y,0);
disp('Natuerliche kubische Splines : spline_k')
disp('Koeffizienten intervallweise')

[ d(1:n-1)' c(1:n-1)' b(1:n-1)' a(1:n-1)']

SSk = 'd(k)*(x-x(k))^3+c(k)*(x-x(k))^2+b(k)*(x-x(k))+a(k)';

disp('Normalform im ersten Intervall')
SS1 = expand(subs(subs(subs(subs(subs(Sk,x(1),'x(k)'),...
        d(1),'d(k)'),c(1),'c(k)'),...
        b(1),'b(k)'),a(1),'a(k)'))

disp('Koeffizienten')
SS1k = sym2poly(SS1)
xp = [];
yp = [];
for k=1:n-1
  xh = linspace(x(k),x(k+1),20);
  Syk = d(k).*(xh-x(k)).^3+c(k).*(xh-x(k)).^2+b(k).*(xh-x(k))+a(k);
  xp = [xp xh];
  yp = [yp Syk];
end;

plot(xp,yp,'w.')
text(2,0.5,'w. - Natuerlicher kub. Spline')
print ser6gr10.ps -dps
hold off
```

Ergebnisse

```
x =
     1     4     6     9

y =
     2     5     3     6

Koeffizienten des Interpolationspolynoms
p =
    0.1000   -1.5000    6.4000   -3.0000

pp =
   10.0000    1.0000    3.0000    1.0000    4.0000    6.0000    9.0000
    4.0000    0.1000    0.1000    0.1000   -1.2000   -0.3000    0.3000
    3.7000   -0.8000   -0.8000    2.0000    5.0000    3.0000

Kubischer Spline auf [x(1), x(2]
   S3(x) =  0.1000(x-1.0000)^3-1.2000(x-1.0000)^2+3.7000(x-1.0000)+2.0000

Alle Koeffizienten (Zeile-Intervall-Polynom)
ans =
    0.1000   -1.2000    3.7000    2.0000
    0.1000   -0.3000   -0.8000    5.0000
    0.1000    0.3000   -0.8000    3.0000

Normalform im ersten Intervall
Sk =
d(k)*(x-x(k))^3+c(k)*(x-x(k))^2+b(k)*(x-x(k))+a(k)

S1 =
1801439850948199/1801439850948199*x^3-2702159776422298l/1801439850948199*x^2
+57646075230342352/90071992547409920*x-27021597764222979/90071992547409920

ans =
     2

Koeffizienten des kubischen Splines
S1k =
    0.1000   -1.5000    6.4000   -3.0000

Kub.Spline = Interpolationspolynom 3.Grades
```

```
Natuerliche kubische Splines : spline_k
Koeffizienten intervallweise

ans =
   -0.0833         0    1.7500    2.0000
    0.2500   -0.7500   -0.5000    5.0000
   -0.0833    0.7500   -0.5000    3.0000

Normalform im ersten Intervall
SS1 =
-1/12*x^3+1/4*x^2+3/2*x+1/3

Koeffizienten
SS1k =
   -0.0833    0.2500    1.5000    0.3333
```



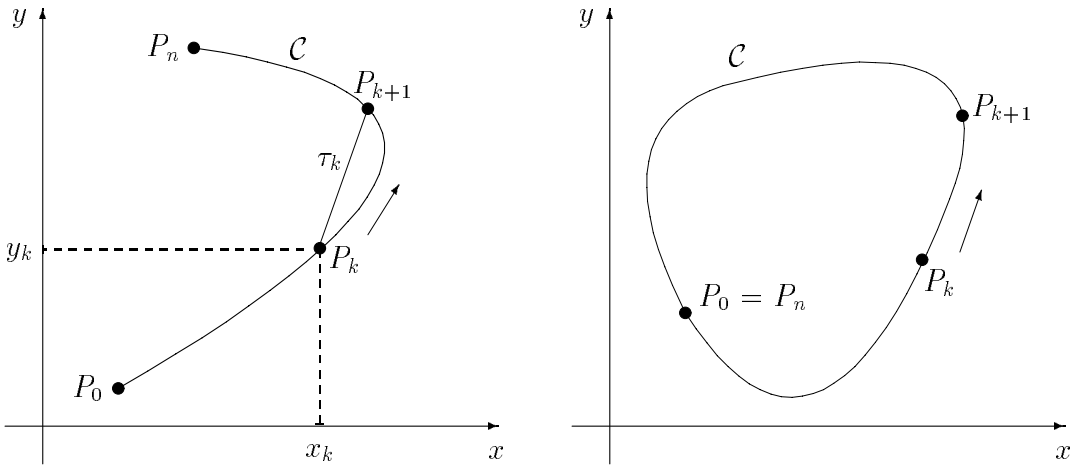
4.5 Parametrische Splines zur Kurvendarstellung

Ziel: Glatte Interpolation einer punktweise gegebenen Kurve in $\mathbb{R}^2$

$$\mathcal{C} = \{P_k \mid P_k = (x_k, y_k),\ k = 0(1)n\ \}.$$

Parameterdarstellung von $\mathcal{C}$

$$x = x(t),\ y = y(t),\quad a \le t \le b.$$



Algorithmus

1. Bestimmung der t-Werte t_k zu $x_k = x(t_k),\ y_k = y(t_k)$.
   Approximation der Bogenlänge zwischen $P_k = (x_k, y_k)$ und $P_{k+1} = (x_{k+1}, y_{k+1})$ durch die Verbindungsstrecke

$$\begin{aligned} \tau_k &= t_{k+1} - t_k = \sqrt{(x_{k+1} - x_k)^2 + (y_{k+1} - y_k)^2}, \\ t_{k+1} &= t_k + \tau_k,\ k = 0(1)n-1. \end{aligned}$$

2. Interpolation der 2 Funktionen $x(t),\ y(t)$ durch kubische Splinefunktionen $s_x(t),\ s_y(t)$ mit

$$s_x(t_k) = x_k,\quad s_y(t_k) = y_k,\ k = 0(1)n,$$

   und den Glattheitsforderungen für kubische Splines.

Bemerkungen

1. Approximation offener Kurven durch *kubische* oder *natürliche* Splines,
   Approximation geschlossener Kurven durch *periodische* Splines.

2. Approximation von Raumkurven mit Parameterdarstellung $(x(t), y(t), z(t))$, $a \le t \le b$, erfolgt analog mit Einbeziehung der Werte z_k.

Beispiel

Gegeben sei die ebene Kurve in Parameterdarstellung $x = x(t),\ y = y(t),\ t \in [0,1]$, durch die Punkte $(x_i, y_i),\ i = 0, 1, ..., 4$,

t_i	0	0.25	0.5	0.75	1
x_i	-1	0	1	0	1
y_i	0	1	0.5	0	-1

1. Interpoliere die Koordinatenfunktionen $x(t)$ und $y(t)$ durch Polynome maximal 4.Grades und stelle die Ergebnisse graphisch dar.

2. Löse Teil 1 mittels parametrischer kubischer Splines und `spline`.

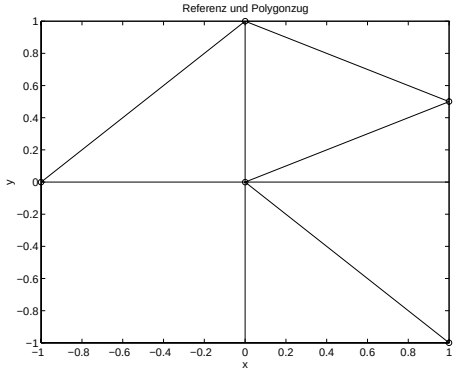
3. Löse Teil 1 mittels parametrischer natürlicher kubischer Splines.

```
% Interpolation einer ebenen Kurve in Parameterdarstellung
% Referenz
nn = 5;
ti = linspace(0,1,nn)
xi = [-1, 0,  1,  0,  1]
yi = [ 0, 1, 0.5, 0, -1]
plot(xi,yi,'o',xi,yi,[min(xi),max(xi)],[0,0],'w',...
     [0,0],[min(yi),max(yi)],'w')
title('Referenz und Polygonzug')
xlabel('x')
ylabel('y')
print ser6gr04.ps -dps

% 1. Interpolation mit Polynomen 4.Grades
disp('1. Interpolation mit Polynomen 4.Grades')
n = 4
px = polyfit(ti,xi,n)
py = polyfit(ti,yi,n)
tt = linspace(0,1,100);
pxt = polyval(px,tt);
pyt = polyval(py,tt);
plot(tt,pxt,'g',ti,xi,'go',...
     tt,pyt,'y',ti,yi,'y+',tt,0*pyt,'w')
title('1. Polynome px4(t), py4(t)')
xlabel('t')
print ser6gr05.ps -dps

plot(xi,yi,'o',pxt,pyt,'r-',[-2,2],[0,0],'w',[0,0],[-1.5,1.5],'w')
title('Parameterkurve P(t)=(px4(t),py4(t))')
print ser6gr13.ps -dps
```

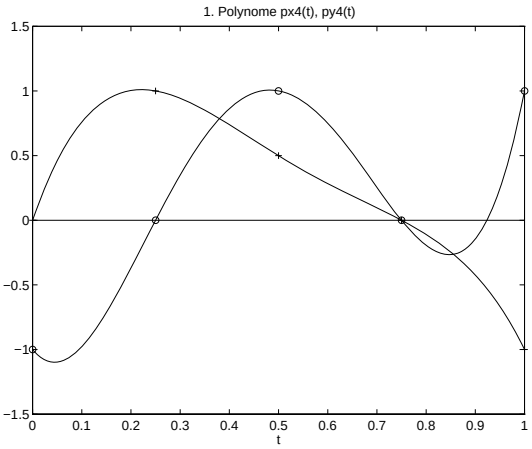
```
% 2. Parametrische kubische Splines mit spline
disp('2. Parametrische kubische Splines mit spline')
t = ti;
for k=2:nn
  t(k) = sqrt((xi(k)-xi(k-1))^2+(yi(k)-yi(k-1))^2)+t(k-1);
end;
t
tt = linspace(t(1),t(nn),100);
xt = spline(t,xi,tt);
yt = spline(t,yi,tt);
plot(xi,yi,'o',xt,yt,'y:',[-2,2],[0,0],'w',[0,0],[-1.5,1.5],'w')
title('2. Kubische Splinekurve mit "spline"')
print ser6gr06.ps -dps

plot(xi,yi,'o',xt,yt,'y:',pxt,pyt,'r-',...
     [-2,2],[0,0],'w',[0,0],[-1.5,1.5],'w')
title('Vergleich')
text(-1.9,-0.7,'r-  - Polynome')
text(-1.9,-1.0,'y:  - Kub. Spline')
hold on

% 3. Natuerliche kubische Splines mit spline_k
%    fuer beide Koordinaten
disp('3. Natuerliche kubische Splines mit spline_k')
[ax bx cx dx] = spline_k(nn-1,t,xi,0);
[ay by cy dy] = spline_k(nn-1,t,yi,0);

disp('Koeffizienten d,c,b,a fuer x(t) ')
[ dx(1:nn-1)' cx(1:nn-1)' bx(1:nn-1)' ax(1:nn-1)']

disp('Koeffizienten d,c,b,a fuer y(t) ')
[ dy(1:nn-1)' cy(1:nn-1)' by(1:nn-1)' ay(1:nn-1)']

xp = [];  yp = [];
for k=1:nn-1
  xh = linspace(t(k),t(k+1),20);
  Sxk = dx(k).*(xh-t(k)).^3+cx(k).*(xh-t(k)).^2+bx(k).*(xh-t(k))+ax(k);
  Syk = dy(k).*(xh-t(k)).^3+cy(k).*(xh-t(k)).^2+by(k).*(xh-t(k))+ay(k);
  xp = [xp Sxk];
  yp = [yp Syk];
end;
plot(xp,yp,'w.')
text(-1.9,-1.3,'w. - Nat. Spline')
print ser6gr07.ps -dps
hold off
```

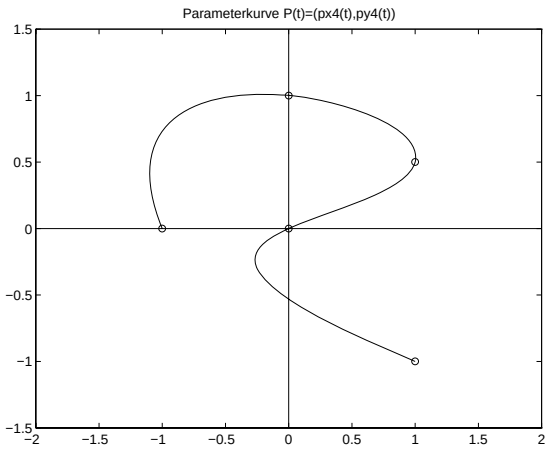
Ergebnisse

```
ti =
         0    0.2500    0.5000    0.7500    1.0000

xi =
    -1     0     1     0     1

yi =
         0    1.0000    0.5000         0   -1.0000
```

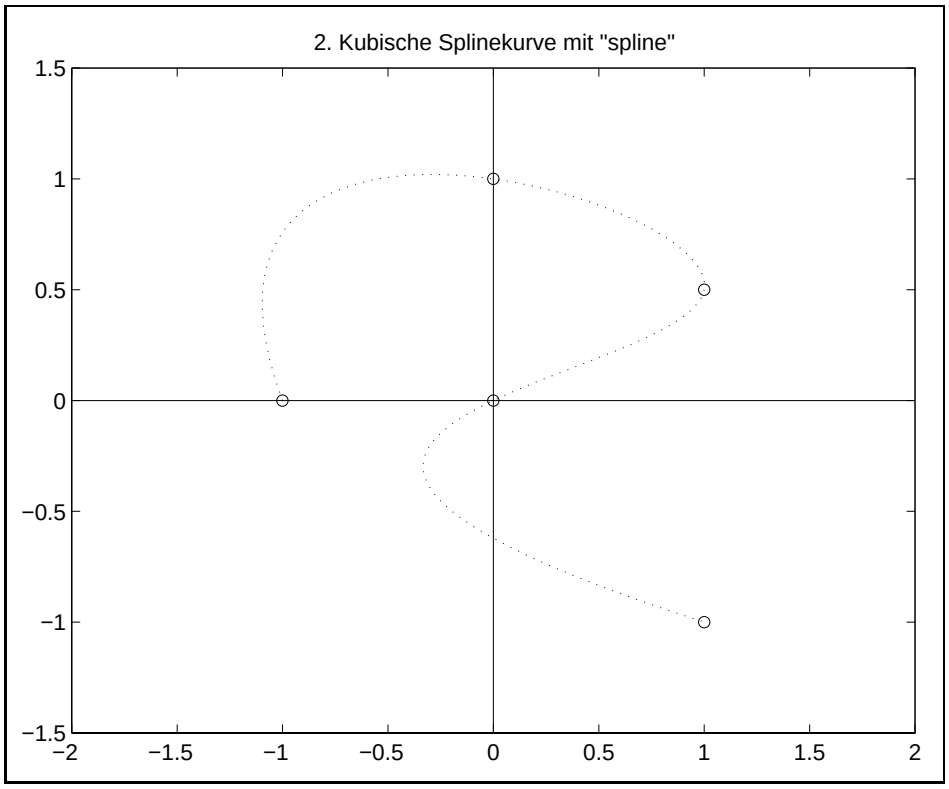



```
1. Interpolation mit Polynomen 4.Grades

n =
     4

px =
   64.0000 -117.3333   60.0000   -4.6667   -1.0000

py =
  -21.3333   48.0000  -38.6667   11.0000    0.0000
```

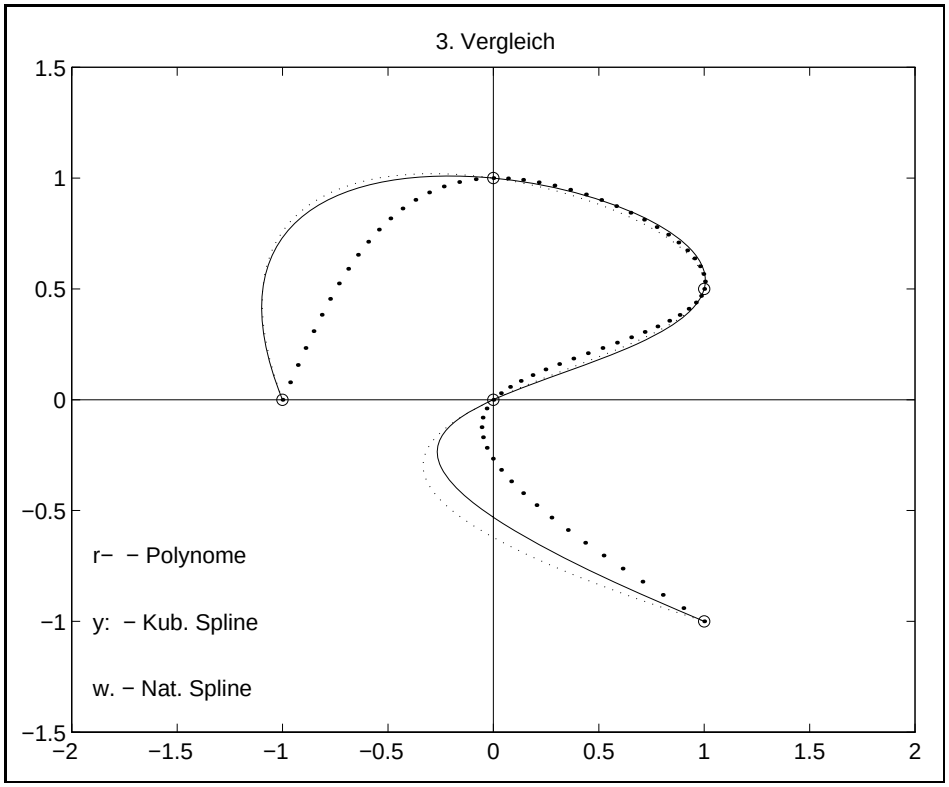




```
2. Parametrische kubische Splines mit spline

t =
         0    1.4142    2.5322    3.6503    5.0645
```




```
3. Natuerliche kubische Splines mit spline_k

Koeffizienten d,c,b,a fuer x(t)

ans =
    0.1118         0    0.4834   -1.0000
   -0.6324    0.4745    1.1544         0
    0.8822   -1.6467   -0.1561    1.0000
   -0.3093    1.3122   -0.5300         0

Koeffizienten d,c,b,a fuer y(t)

ans =
   -0.1734         0    1.0539         0
    0.2895   -0.7357    0.0134    1.0000
   -0.1316    0.2354   -0.5459    0.5000
    0.0485   -0.2059   -0.5130         0
```


3. Vergleich
r− − Polynome
y: − Kub. Spline
w. − Nat. Spline
1.5
1
0.5
0
−0.5
−1
−1.5
−2
−1.5
−1
−0.5
0
0.5
1
1.5
2

5 Anhang

Anhang A

Zusammenstellung von Adressen

1. World Wide Web (WWW) mit MATLAB Seiten

Die TEX Quelle sowie das PostScript file `primer35.ps` der 2.Edition des MATLAB Primers steht stehen mittels *ftp* auf `ftp.math.ufl.edu` im Verzeichnis `pub/matlab` zur Verfügung.

Die MathWorks Inc. und MathTools Ltd. entwickeln und vertreiben die Computersoftware MATLAB und andere Komponenten und sind natürlich mit ihrem ganzen Angebot auch im Internet zu finden.

http://www.mathworks.com/	The MathWorks Inc. home (auch Simulink)
http://www.mathworks.com/products/matlab/	MATLAB 5.3
http://www.mathworks.com/products/matlab/	MATLAB web server 1.0
http://www.mathworks.com/support/books/	MATLAB based books
http://www.mathtools.com/	MATLAB Toolboxen von MathTools Ltd. (auch MATCOM, MIDEVA)
http://krum.rz.uni-mannheim.de/cafgbench.html	Computer Algebra Benchmarks (RZ/Uni Mannheim)

2. Verzeichnisse mit MATLAB m-Files für Skripte und Funktionen

Zu den Kapiteln 1-4 des Skripts liegen die entsprechenden Files (meist m-Files) und diverse Daten- und Ergebnisfiles vor.

`*.m, *.ps, *.eps, *.mat`

Die Dateien sind zu finden im Novell-Netz PIVOT des Instituts für Mathematik bzw. auf der persönlichen Homepage im Internet.

\\PIVOT\SHARE Q:\NEUNDORF\STUD_M93\MATLAB3

Homepage Navigator → Publications → Computeralgebra → MATLAB3

e-mail: ***neundorf@mathematik.tu-ilmenau.de***

Homepage: ***http://imath.mathematik.tu-ilmenau.de/~neundorf/index_de.html***

Literatur

[1] Sigmon, K.: *MATLAB Primer, Second Edition.*
Department of Mathematics, University of Florida USA 1992.

[2] Sigmon, K.: *MATLAB Primer 5e.* CRC Press 1998.

[3] *MATLAB User's Guide and Reference Guide.*

[4] *MATLAB Release Notes 4.1. For Unix Workstations.*
The MathWorks Inc. Natick, Massachusetts USA 1993.

[5] *The Student Edition of MATLAB. Version 4. User's Guide.*
The MathWorks Inc. Prentice Hall, Englewood Cliffs New Jersey 1995.

[6] *The Student Edition of MATLAB 5.* Prentice Hall.

[7] Redfern, D.; Campbell, C.: *The MATLAB 5 Handbook.* Springer-Verlag 1998.

[8] Köckler, N.: *Numerische Algorithmen in Softwaressystemen: unter besonderer Berücksichtigung der NAG-Bibliothek.* B.G. Teubner Stuttgart 1990.

[9] Neundorf, W.: *MATLAB - Teil I: - Vektoren, Matrizen, lineare Gleichungssysteme.* Preprint M 20/99 IfMath der TU Ilmenau, Juli 1999.

[10] Neundorf, W.: *MATLAB - Teil II: - Speicheraspekte, spezielle LGS, SDV, EWP, Graphik, NLG, NLGS.* Preprint M 23/99 IfMath der TU Ilmenau, September 1999.

[11] Mathews, J.H.; Fink, K.D.: *Numerical Methods using MATLAB.* Prentice Hall London 1999.

[12] Chen, K.; Giblin, P.J.; Irving, A.: *Mathematical explorations with MATLAB.* Cambridge University Press 1999.

[13] Mohr, R.: *Numerische Methoden in der Technik: eine Lehrbuch mit MATLAB-Routinen.* Vieweg Braunschweig 1998.

[14] Golubitsky, M.; Dellnitz, M.: *Linear algebra and differential equations using MATLAB.* Brooks/Cole Pub. Co Pacific Grove 1999.

[15] Maeß, G.: *Vorlesungen über numerische Mathematik II.* Akademie-Verlag Berlin 1988.

Anschrift:

Dr. Werner Neundorf
Technische Universität Ilmenau, Institut für Mathematik
PF 10 0565
D - 98684 Ilmenau

e-mail : neundorf@mathematik.tu-ilmenau.de