

# 1 | Spline-Interpolation

Hier beziehe ich mich auf die Lecture Notes von Ruye Wang - im Speziellen auf seine Ausführungen über Splines in <http://fourier.eng.hmc.edu/e176/lectures/ch7/node6.html>.

Dieser Abschnitt ist in weiten Teilen nur eine Übersetzung seiner Ausführungen! Hinzugefügt von mir wurde der Tridiagonalmatrix-Algorithmus und die Implementation des kubischen Spline-Verfahrens in Javascript für *Geogebra*!

## Definition 1.1 Splinefunktion vom Grad $m$

Eine reelle Funktion  $f$  sei an  $(n + 1)$  Stellen bekannt:  $f(x_i) = y_i$ ,  $i \in \{0, 1, \dots, n\}$   
Wir bestimmen  $n$  Polynome  $P_i$  vom Grad  $m \ll n$ , die mit  $f$  an den Stützstellen übereinstimmen und dort glatt ineinander übergehen (Ableitungen stimmen überein):

$$S(x) = \begin{cases} P_1(x) & x_0 \leq x \leq x_1 \\ \vdots & \vdots \\ P_i(x) & x_{i-1} \leq x \leq x_i \\ \vdots & \vdots \\ P_n(x) & x_{n-1} \leq x \leq x_n \end{cases} \quad (1.1)$$

Es muss also gelten

$$(1) \quad P_i(x_i) = P_{i+1}(x_i) = f(x_i) = y_i, \quad i \in \{1, \dots, (n - 1)\} \quad (1.2)$$

$$(2) \quad P_1(x_0) = y_0 \quad \wedge \quad P_n(x_n) = y_n \quad (1.3)$$

$$(3) \quad P_i^{(k)}(x_i) = P_{i+1}^{(k)}(x_i), \quad i \in \{1, \dots, (n - 1)\}, \quad k \in \{0, \dots, k_m\} \quad (1.4)$$

Die maximale Übereinstimmung der Ableitungen  $k_m$  sollte dabei so groß wie möglich sein und hängt natürlich von  $m$  - dem Grad der Teilpolynome ab.

$S$  heißt dann die Splinefunktion von  $f$  vom Grad  $m$ .

# 1. Spline-Interpolation

## 1.1 Kubische Splines

Wir beschäftigen uns hier nur mit kubischen Splines - also  $m = 3$

Zuerst leiten wir die hier verwendete lineare Interpolationsformel zwischen 2 Punkten her:

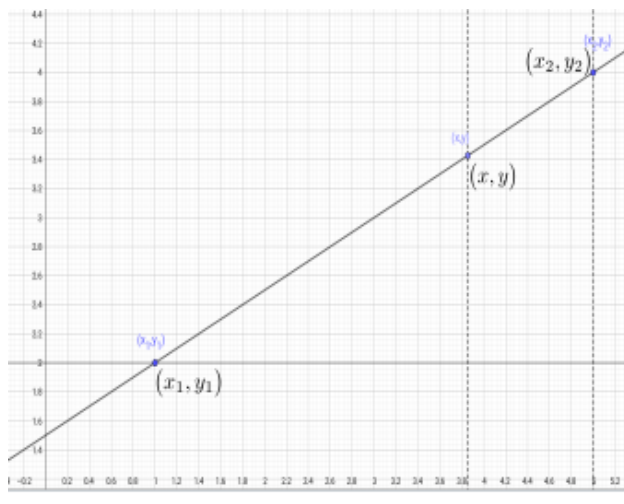


Abb.1 : Lineare Interpolation

$$\begin{aligned} \frac{y - y_1}{x - x_1} &= \frac{y_2 - y_1}{x_2 - x_1} \Rightarrow \\ y &= \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1 = \\ &= \frac{y_2 - y_1}{x_2 - x_1}x + \frac{y_2 - y_1}{x_2 - x_1}(-x_1) + y_1 \frac{x_2 - x_1}{x_2 - x_1} = \\ &= \frac{1}{\underbrace{x_2 - x_1}_{h_1}} = (y_2x - y_1x - x_1y_2 + y_1x_2) \Rightarrow \\ & \boxed{y = \frac{1}{h_1} \left( (x - x_1)y_2 + (x_2 - x)y_1 \right)} \quad (1.5) \end{aligned}$$

Wir suchen also für  $C_i(x) := P_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i$  die 4 Parameter  $a_i$ ,  $b_i$ ,  $c_i$  und  $d_i$ . Die Bedingungen 1.2 bis 1.4 werden jetzt zu

$$C_i(x_i) = y_i, \quad C_i(x_{i-1}) = y_{i-1}, \quad C_i'(x_i) = C_{i+1}'(x_i) \quad \text{und} \quad C_i''(x_i) = C_{i+1}''(x_i)$$

$C_i''(x) = 6a_i x + 2b_i$  ist dann eine lineare Funktion. Die unbekannt Krümmungen dieser Polynome seien

$$C_i''(x_{i-1}) =: M_{i-1} \quad \text{und} \quad C_i''(x_i) =: M_i$$

Mit 1.5 können die  $C_i''(x)$  geschrieben werden als

$$C_i''(x) = \frac{x_i - x}{h_i} M_{i-1} + \frac{x - x_{i-1}}{h_i} M_i \quad h_i := x_i - x_{i-1} \quad (1.6)$$

Wir integrieren 2 mal und erhalten

$$C_i(x) = \int \left( \int C_i''(x) dx \right) dx = \frac{(x_i - x)^3}{6h_i} M_{i-1} + \frac{(x - x_{i-1})^3}{6h_i} M_i + c_i x + d_i \quad (1.7)$$

Mit  $C_i(x_{i-1}) = y_{i-1}$ ,  $C_i(x_i) = y_i$  und  $h_i := x_i - x_{i-1}$  wird daraus

$$C_i(x_{i-1}) = \frac{h_i^2}{6} M_{i-1} + c_i x_{i-1} + d_i = y_{i-1}, \quad C_i(x_i) = \frac{h_i^2}{6} M_i + c_i x_i + d_i = y_i \quad (1.8)$$

Wir lösen diese beiden Gleichungen für  $c_i$  und  $d_i$  und erhalten

$$c_i = \frac{y_i - y_{i-1}}{h_i} - \frac{h_i}{6}(M_i - M_{i-1}) \quad (1.9)$$

$$d_i = \frac{x_i y_{i-1} - x_{i-1} y_i}{h_i} - \frac{h_i}{6}(x_i M_{i-1} - x_{i-1} M_i) \quad (1.10)$$

Wir setzen 1.9 und 1.10 wieder in 1.7 ein und fassen zusammen

$$\begin{aligned} C_i(x) &= \frac{(x_i - x)^3}{6h_i} M_{i-1} + \frac{(x - x_{i-1})^3}{6h_i} M_i + \left( \frac{y_i - y_{i-1}}{h_i} - \frac{h_i}{6}(M_i - M_{i-1}) \right) x \\ &\quad + \frac{x_i y_{i-1} - x_{i-1} y_i}{h_i} - \frac{h_i}{6}(x_i M_{i-1} - x_{i-1} M_i) \\ &= \frac{(x_i - x)^3}{6h_i} M_{i-1} + \frac{(x - x_{i-1})^3}{6h_i} M_i + \left( \frac{y_{i-1}}{h_i} - \frac{M_{i-1} h_i}{6} \right) (x_i - x) + \left( \frac{y_i}{h_i} - \frac{M_i h_i}{6} \right) (x - x_{i-1}) \end{aligned}$$

also haben wir jetzt einen Ausdruck für die  $C_i(x)$  in Abhängigkeit von den  $M_i$ :

$$\begin{aligned} C_i(x) &= \frac{(x_i - x)^3}{6h_i} M_{i-1} + \frac{(x - x_{i-1})^3}{6h_i} M_i + \left( \frac{y_{i-1}}{h_i} - \frac{M_{i-1} h_i}{6} \right) (x_i - x) + \\ &\quad + \left( \frac{y_i}{h_i} - \frac{M_i h_i}{6} \right) (x - x_{i-1}) \quad (1.11) \end{aligned}$$

Um in 1.11  $M_i$ ,  $i \in \{1, 2, \dots, (n-1)\}$  zu bestimmen, verwenden wir die Bedingung, dass an den "Schnittstellen" der  $C_i$  die Ableitungen übereinstimmen müssen -  $C'_i(x_i) = C'_{i+1}(x_i)$ .

Wir leiten 1.11 ab und erhalten

$$C'_i(x) = -\frac{(x_i - x)^2}{2h_i} M_{i-1} + \frac{(x - x_{i-1})^2}{2h_i} M_i - \frac{1}{h_i} \left( y_{i-1} - \frac{M_{i-1} h_i^2}{6} \right) + \frac{1}{h_i} \left( y_i - \frac{M_i h_i^2}{6} \right) \quad (1.12)$$

$$= -\frac{(x_i - x)^2}{2h_i} M_{i-1} + \frac{(x - x_{i-1})^2}{2h_i} M_i + \frac{y_i - y_{i-1}}{h_i} - \frac{h_i}{6}(M_i - M_{i-1}) \quad (1.13)$$

Mit der mittleren Steigung  $\bar{k}_f$  von  $f$  zwischen 2 Stützstellen  $x_{i-1}$  und  $x_i$

$$\bar{k}_f[x_{i-1}, x_i] := \frac{y_i - y_{i-1}}{h_i} \quad \text{ergibt sich dann}$$

$$\begin{aligned} C'_i(x_i) &= \frac{h_i}{3} M_i + \frac{y_i - y_{i-1}}{h_i} + \frac{h_i}{6} M_{i-1} = \frac{h_i}{6}(2M_i + M_{i-1}) + \bar{k}_f[x_{i-1}, x_i] \\ C'_i(x_{i-1}) &= -\frac{h_i}{3} M_{i-1} + \frac{y_i - y_{i-1}}{h_i} - \frac{h_i}{6} M_i = -\frac{h_i}{6}(2M_{i-1} - M_i) + \bar{k}_f[x_{i-1}, x_i] \quad (1.14) \end{aligned}$$

## 1. Spline-Interpolation

---

Wenn wir in 1.14  $i$  um 1 erhöhen, erhalten wir  $C'_{i+1}(x_i)$ :

$$C'_{i+1}(x_i) = -\frac{h_{i+1}}{3}M_i - \frac{h_{i+1}}{6}M_{i+1} + \bar{k}_f[x_i, x_{i+1}] \quad (1.15)$$

Durch Gleichsetzen mit 1.12 erhalten wir

$$\begin{aligned} \frac{h_i}{3}M_i + \bar{k}_f[x_{i-1}, x_i] + \frac{h_i}{6}M_{i-1} &= -\frac{h_{i+1}}{3}M_i + \bar{k}_f[x_i, x_{i+1}] - \frac{h_{i+1}}{6}M_{i+1} \Big| \cdot 6 \\ h_i M_{i-1} + 2(h_{i+1} + h_i)M_i + h_{i+1} M_{i+1} &= 6(\bar{k}_f[x_i, x_{i+1}] - \bar{k}_f[x_{i-1}, x_i]) \end{aligned} \quad (1.16)$$

Mit  $\frac{1}{x_{i+1} - x_{i-1}}(\bar{k}_f[x_i, x_{i+1}] - \bar{k}_f[x_i, x_{i-1}]) = \frac{\bar{k}_f[x_i, x_{i+1}] - \bar{k}_f[x_i, x_{i-1}]}{h_{i+1} + h_i} := \bar{c}_f[x_{i-1}, x_{i+1}]$

als mittlere Krümmung (Mittelwert der mittleren Steigungen) von  $f$  in  $[x_{i-1}, x_{i+1}]$  lässt sich 1.16 schreiben

$$\begin{aligned} h_i M_{i-1} + 2(h_{i+1} + h_i)M_i + h_{i+1} M_{i+1} &= 6 \bar{c}_f[x_{i-1}, x_{i+1}](h_{i+1} + h_i) \Big| : (h_{i+1} + h_i) \\ \frac{h_i}{h_{i+1} + h_i}M_{i-1} + 2M_i + \frac{h_{i+1}}{h_{i+1} + h_i}M_{i+1} &= \underbrace{6 \bar{c}_f[x_{i-1}, x_{i+1}]}_{d_i} \end{aligned}$$

Wir landen schließlich bei

$$\begin{aligned} \mu_i M_{i-1} + 2M_i + \lambda_i M_{i+1} &= d_i, \quad (i = 1, \dots, n-1) \\ \mu_i &= \frac{h_i}{h_{i+1} + h_i}, \quad \lambda_i = \frac{h_{i+1}}{h_{i+1} + h_i} = 1 - \mu_i, \quad d_i = 6 \bar{c}_f[x_{i-1}, x_{i+1}] \end{aligned} \quad (1.17)$$

Wir haben also ein Gleichungssystem mit  $(n-1)$  Gleichungen und  $(n+1)$  Variable  $M_0, M_1, \dots, M_n$ . Wir benötigen also 2 zusätzliche Bedingungen. Wir konzentrieren uns hier auf die "natürliche Randwertbedingung":  $M_0 = 0 \quad \wedge \quad M_n = 0$

Damit wird 1.17 zu folgendem tridiagonalen Gleichungssystem:

$$\begin{pmatrix} 2 & \lambda_1 & 0 & 0 & \dots & 0 \\ \mu_2 & 2 & \lambda_2 & 0 & \dots & 0 \\ 0 & \mu_3 & 2 & \lambda_3 & 0 & 0 \\ \vdots & 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & 0 & \mu_{n-2} & 2 & \lambda_{n-2} \\ 0 & \dots & 0 & 0 & \mu_{n-1} & 2 \end{pmatrix} \begin{pmatrix} M_1 \\ M_2 \\ \vdots \\ \vdots \\ M_{n-2} \\ M_{n-1} \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ \vdots \\ d_{n-2} \\ d_{n-1} \end{pmatrix} \quad (1.18)$$

Der Lösung dieses Problems widmen wir ein eigenes Unterkapitel!

## 1.2 Tridiagonal-Matrix Algorithmus(TDMA)

### Definition 1.2 Tridiagonales Gleichungssystem

Ein Gleichungssystem heißt *tridiagonal*, wenn es wie folgt strukturiert ist:

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 & \dots & 0 \\ a_2 & b_2 & c_2 & 0 & \dots & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & 0 \\ \vdots & 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & 0 & a_{N-1} & b_{N-1} & c_{N-1} \\ 0 & \dots & 0 & 0 & a_N & b_N \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ \vdots \\ d_{N-1} \\ d_N \end{pmatrix} \quad (1.19)$$

bzw.

$$\begin{aligned} a_i x_{i-1} + b_i x_i + c_i x_{i+1} &= d_i \\ a_i, b_i, c_i, d_i \in \mathbb{R}, i \in \{1, \dots, n\}, a_1 = 0 \wedge c_n = 0 \end{aligned} \quad (1.20)$$

Schauen wir uns so ein Gleichungssystem näher an. Die erste Gleichung lautet

$$b_1 x_1 + c_1 x_2 = d_1 \Rightarrow x_1 = P_1 x_2 + Q_1 \quad \text{mit } P_1 \left( = -\frac{c_1}{b_1} \right), Q_1 \left( = \frac{d_1}{b_1} \right) \in \mathbb{R}$$

Halten wir fest:  $x_1$  hängt linear von  $x_2$  ab:  $x_1 = x_1(x_2)_\ell$

Die zweite Gleichung lautet

$$a_2 x_1 + b_2 x_2 + c_2 x_3 = d_2 \Rightarrow a_2 x_1(x_2)_\ell + b_2 x_2 + c_2 x_3 = d_2 \Rightarrow x_2 = x_2(x_3)_\ell$$

Der letzte Schluss folgt aus der Abgeschlossenheit der linearen Funktionen. Wir können dies auch wieder schreiben als

$$x_2 = P_2 x_3 + Q_2$$

Allgemein können wir formulieren

$$\boxed{x_i = P_i x_{i+1} + Q_i \Leftrightarrow x_{i-1} = P_{i-1} x_i + Q_{i-1}} \quad (1.21)$$

Die letzte Gleichung lautet

$$a_N x_{N-1} + b_N x_N = d_N \quad \text{jetzt gilt aber } x_{N-1} = x_{N-1}(x_N)_\ell$$

damit ist diese Gleichung für  $x_N$  lösbar, mit Kenntnis von  $x_{N-1} = x_{N-1}(x_N)_\ell$  bekommen wir  $x_{N-1}$ , und so weiter bis  $x_1$ . Es gilt also die  $P_i$  bzw.  $Q_i$  zu bestimmen - dazu setzen wir den

## 1. Spline-Interpolation

---

rechten Teil von 1.21 in 1.20 ein:

$$\begin{aligned} a_i (P_{i-1} x_i + Q_{i-1}) + b_i x_i + c_i x_{i+1} &= d_i \quad \Rightarrow \\ x_i &= -\frac{c_i}{b_i + a_i P_{i-1}} x_{i+1} + \frac{d_i - a_i Q_{i-1}}{b_i + a_i P_{i-1}} \end{aligned} \quad (1.22)$$

Vergleichen wir 1.22 mit 1.21 (linker Teil) ergeben sich die Rekursionsformeln für  $P$  und  $Q$

$$P_i = -\frac{c_i}{b_i + a_i P_{i-1}} \quad Q_i = \frac{d_i - a_i Q_{i-1}}{b_i + a_i P_{i-1}} \quad (1.23)$$

Jetzt liegt der TDMA (**Thomas Algorithmus**) vor uns

- INPUT: Felder  $a$ ,  $b$ ,  $c$ , und  $d$
- $P_1 = -\frac{c_1}{b_1}$      $Q_1 = \frac{d_1}{b_1}$
- Berechnung der nächsten  $P_i$  und  $Q_i$  mit 1.23
- $x_N = Q_N$ , da  $P_N = 0$  wegen  $c_N = 0$  und eingesetzt in 1.21
- Berechnung der weiteren Lösungen mit 1.21
- OutPut: Lösungsvektor

```
1 function TDMA(a,b,c,d){
2 // fields a,b,c,d start at Index zero!!
3 // so the formula in the text above must be adapted
4 var P=[], Q=[], u=[], denom, n=a.length;
5
6 P[0]=0; Q[0]=0; // special case j=0 yields correct result
7 for (j=0;j<n; j++) {
8     denom = b[j]+a[j]*P[j];
9     P[j+1]= - c[j]/denom;
10    Q[j+1]= (d[j]-a[j]*Q[j])/denom;
11 }
12
13 u[n]=Q[n];
14 for (i=n-1; i >0; i--) u[i]=P[i]*u[i+1]+Q[i];
15 // u[0] is "undefined"; u[1] ... u[n]
16 return u
17 }
```

Änderungen im Code:  $a_i := \mu_i$ ,  $b_i := 2$ ,  $c_i := \lambda_i$ ,  $d_i := 6 \bar{c}_f[x_{i-1}, x_{i+1}]$  starten bei Index 0! Dadurch dekrementieren sich deren Indices in 1.23 um 1 (Zeilen 9 und 10):



$$P_i = -\frac{c_{i-1}}{b_{i-1} + a_{i-1} P_{i-1}} \quad Q_i = \frac{d_{i-1} - a_{i-1} Q_{i-1}}{b_{i-1} + a_{i-1} P_{i-1}}$$

### 1.3 Implementierung in *Geogebra (Javascript)*

`importPointList` in Zeile 4 erstellt aus der *Geogebra*-Punktliste einen Javascript-Array  
`buildSplineSum` in Zeile 10 erstellt aus den Funktionszweigen den *Geogebra*-Befehlsstring und  
 ist im Wesentlichen die Abbildung der Formeln 1.11 - anders als dort wurde die Indizierung  
 der  $C_i$  generell (für Javascript typisch) bei Null begonnen (siehe Zeile 28), auch das  $h$ -Feld  
 wird jetzt bei Index 0 gestartet (siehe Schleife Zeile 28)!

```

1  /***** MAIN *****/
const getX = 0, getY = 1;
3  var p_x=[], p_y=[], h=[], mu=[], lambda=[], M=[], cmdStr = "f_s(x)=";
var pL=importPointList();
5
p_x=getCoord(getX, pL); p_y=getCoord(getY, pL);
7 h=getDelta(p_x);
setMuLambda(h); // mu, lambda are set
9 M = TDMA(mu, lambda, getConstVec(h, p_y));
cmdStr += buildSplineSum(M, h, p_x, p_y);
11 ggbApplet.evalCommand(cmdStr);
/***** END-MAIN *****/
13
function importPointList(){
15   var pL = "+ggbApplet.getValueString("pointL");;
   pL=pL.split("=")[1];
17   pL=pL.replace(/{/g, "[");
   pL=pL.replace(/}/g, "]");
19   pL=pL.replace(/\\/g, "\\");
   pL=pL.replace(/\\)/g, "\\");
21   return eval(pL);
}
23
function buildSplineSum(M,h,p_x,p_y){
25   var splineStr="If(";
   var branches=[], N=[]; //M/6
27   for (var j=0; j<M.length; j++) N[j]=M[j]/6;
   for ( var k=0; k<M.length-1; k++) branches.push("+p_x[k]+ "<=x<="+p_x[k+1]+
   ", "+ buildFuncTerm(N, k,h,p_x,p_y));
29   return splineStr+branches.join(",")+";";
}
31 // a_s ... d_s are the coefficients in each spline-branch
function buildFuncTerm(N,i,h,p_x,p_y){
33   var a_s, b_s, c_s, d_s, f="(";
   a_s = N[i]/h[i];
35   b_s = N[i+1]/h[i];
   c_s = p_y[i]/h[i]-N[i]*h[i];
37   d_s = p_y[i+1]/h[i]-N[i+1]*h[i];
   f += a_s + ")*( "+ p_x[i+1] + "-x)^3 + (";
39   f += b_s + ")*(x- "+ p_x[i] + ")^3 + (";
   f += c_s + ")*( "+ p_x[i+1] + "-x) + (";
41   f += d_s + ")*(x- "+ p_x[i] + ")";
   return f;
43 }

```

## 1. Spline-Interpolation

---

Hier die Funktionen für den eigentlichen Algorithmus um 1.17 für  $M_1, \dots, M_{N-1}$  zu lösen:

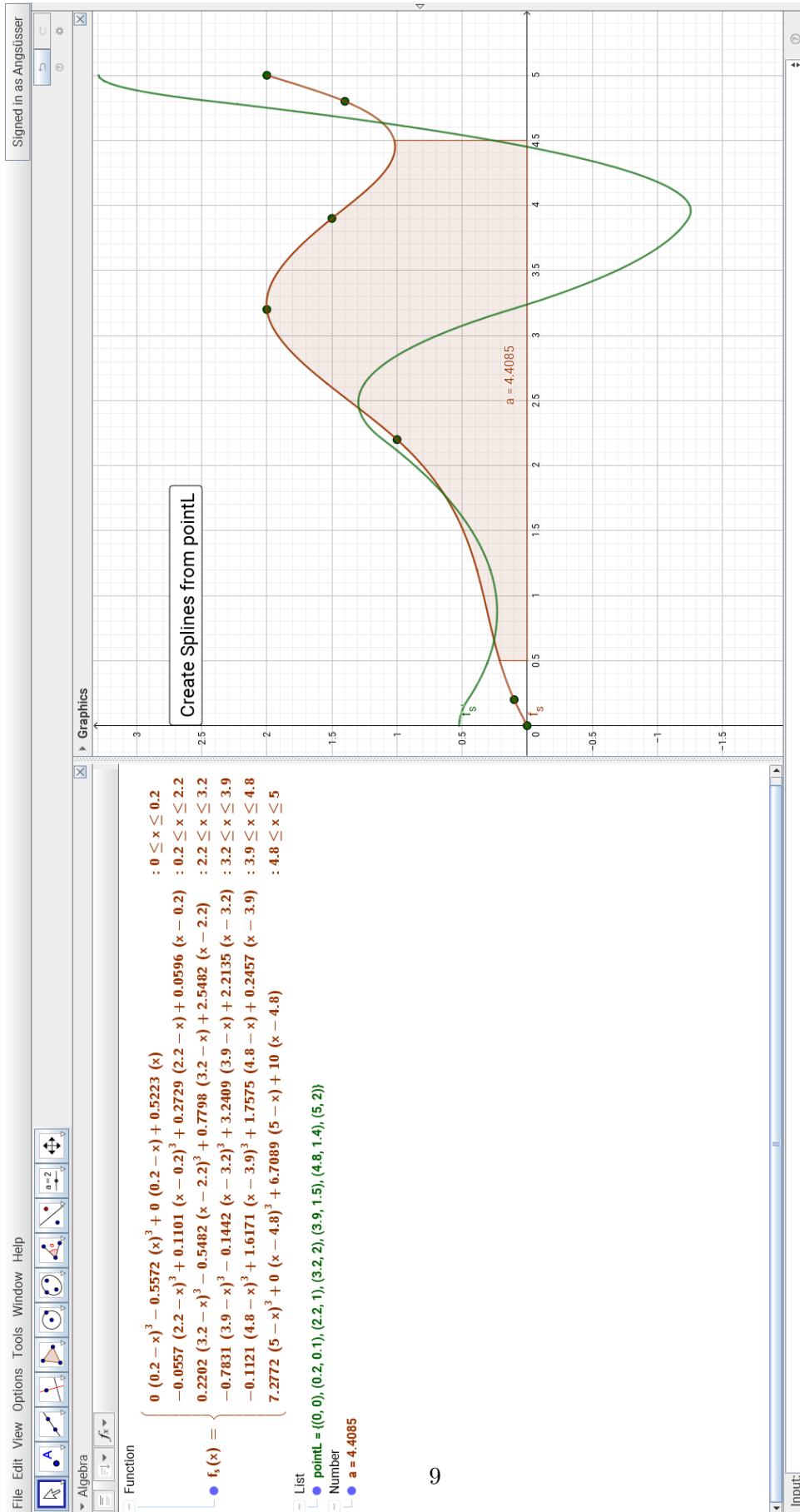
```
46 function getConstVec(hList, yList){
47     var m = [], k = [], n = yList.length - 1;
48     for (j=0; j<n; j++) m[j]=(yList[j+1]-yList[j])/hList[j];
49     for (j=0; j<n-1; j++) k[j]=6*(m[j+1] - m[j])/(hList[j+1] + hList[j]);
50     return k;
51 }
52
53 function getCoord(xOrY,pL){
54     var val=[];
55     for (i=0; i<pL.length; i++) val[i]=pL[i][xOrY];
56     return val;
57 }
58
59 function getDelta(list){
60     var r=[];
61     for (i=1; i<list.length; i++) r[i-1]=list[i]-list[i-1];
62     return r;
63 }
64
65 function setMuLambda(list){
66     var m=[], e=list.length-1;
67     for (var i=0; i < e; i++) {
68         mu[i]=list[i]/(list[i+1]+list[i]);
69         lambda[i]=1-mu[i];
70     }
71     mu[0]=0; lambda[e-1]=0; // first mu and last lambda are reset
72 }
73
74 // TriDiagonalMatrixAlgorithm
75 function TDMA(a, c, d){
76     var P=[], Q=[], u=[], denom, n=a.length;
77     // for (i = 0; i < n; i++) b[i]=2; //diagonal is set
78
79     P[0]=0; Q[0]=0; // special case i=0 yields correct result
80     for (j=0; j<n; j++) {
81         denom = 2+a[j]*P[j]; // b[j]=2
82         P[j+1]= - c[j]/denom;
83         Q[j+1]= (d[j]-a[j]*Q[j])/denom;
84     }
85     u[n]=Q[n];
86     for (i=n-1; i >0; i--) u[i]=P[i]*u[i+1]+Q[i];
87     u[0]=0; u[n+1]=0; //M[0]=0 and M[n]=0
88     return u
89 }
90 }
```

TDMA benötigt als Parameter keine Diagonalelemente und vor der Rückgabe wird der Lösungsvektor um die natürliche Lösung ergänzt.

`getConstVec` berechnet das 6-fache der mittleren Krümmungen.

Der Zweck aller anderer Funktionen ist hoffentlich selbsterklärend.





## 1. Spline-Interpolation

---

Auf der vorigen Seite sieht man die Ausgabe angewandt auf die Test-Punktliste `pointL = {(0, 0), (0.2, 0.1), (2.2, 1), (3.2, 2), (3.9, 1.5), (4.8, 1.4), (5, 2)}`. Das Javascript-Programm wird dabei in *Geogebra* hinter der Click-Methode des Buttons *Create Splines For pointL* “versteckt”.

Außerdem wurde das Integral  $\int_{0.5}^{4.5} f_s(x) dx = 4.4085$  mit `Integral(f_s, 0.5, 4.5)` und die Ableitungsfunktion  $f'_s(x)$  mit `Derivative(f_s)` berechnet. Beides wäre mit der “eingebauten” *Spline*-Funktion nicht möglich gewesen.

### 1.4 Gegencheck mit *wxMaxima* (native invert)

Wir halten uns an 1.18, statt *TDMA* benutzen wir allerdings einfach die Matrixinversion!

Keine Kovertierungswarnungen und höchstens 5 Ziffern Genauigkeit ausgeben

```
(%i2) ratprint:false$ fpprintprec: 5$
```

Wir kopieren die Punktliste aus Geogebra und passen die Syntax an

```
(%i3) pL:[[0, 0], [0.2, 0.1], [2.2, 1], [3.2,2],[3.9,1.5], [4.8, 1.4],[5,2]]$
```

Liste der x- und y-Werte

```
(%i4) xList:makelist(first(pL[i]),i,1,length(pL)); [0, 0.2, 2.2, 3.2, 3.9, 4.8, 5] (xList)
```

```
(%i5) yList:makelist(second(pL[i]),i,1,length(pL)); [0, 0.1, 1, 2, 1.5, 1.4, 2] (yList)
```

Start bei Index 0, sodass wir die Formeln aus dem Text verwenden können -  $p_x[0] \dots p_x[6]$

```
(%i6) p-x[i]:=xList[i+1]$ (%i7) p-y[i]:=yList[i+1]$
```

Die Stützstellen - hier wäre auch die auskommentierte Version denkbar

```
(%i8) /*xLimits:append([-inf],makelist(xList[i],i,2,length(xList)-1),[inf])*/  
xLimits:append([first(xList)],makelist(xList[i],i,2,length(xList)-1),[last(xList)]);  
[0, 0.2, 2.2, 3.2, 3.9, 4.8, 5] (xLimits)
```

Jetzt kommen die Zwischenergebnisse um den Konstantenvektor zu bestimmen

```
(%i9) h:makelist(xList[i+1]-xList[i],i,1,length(xList)-1); [0.2, 2.0, 1.0, 0.7, 0.9, 0.2] (h)
```

```
(%i10) hy:makelist(yList[i+1]-yList[i],i,1,length(yList)-1); [0.1, 0.9, 1, -0.5, -0.1, 0.6] (hy)
```

```
(%i11) hyOverH:makelist(hy[i]/h[i],i,1,length(h)); [0.5, 0.45, 1.0, -0.71429, -0.11111, 3.0] (hyOverH)
```

```
(%i12) deltaHyOverH:makelist(hyOverH[i+1] - hyOverH[i],i,1,length(hyOverH)-1);  
[-0.05, 0.55, -1.7143, 0.60317, 3.1111] (deltaHyOverH)
```

Der Konstantenvektor  $\vec{d}$  ist jetzt bestimmt - er hat 5 Komponenten für  $M_1 \dots M_5$

```
(%i13) dV:makelist(6*1/(h[i+1]+h[i])*deltaHyOverH[i],i,1,length(deltaHyOverH));  
[-0.13636, 1.1, -6.0504, 2.2619, 16.97] (dV)
```

Wie benötigen ihn als Spaltenvektor

```
(%i14) dVec:transpose(matrix(dV))$
```

$\vec{\mu}$  und  $\vec{\lambda}$  werden berechnet, von den 5 Komponenten werden jeweils 4 gebraucht

(%i15) `mu:makelist(h[i]/(h[i+1]+h[i]),i,1,length(h)-1);` [0.090909, 0.66667, 0.58824, 0.4375, 0.81818] (mu)

(%i16) `makelist(1-mu[i],i,1,length(pL)-2);` [0.90909, 0.33333, 0.41176, 0.5625, 0.18182] (%o16)

Die Gleichungsmatrix wird erstellt - siehe 1.18 im Text

(%i17) `setMatrix():=block([dim:length(pL)-2,A],  
A:diagmatrix(dim,2),  
for i thru dim do  
for j thru dim do block(  
if i-j=1 then A[i,j]:mu[i],  
if j-i=1 then A[i,j]:1-mu[i]  
),  
A  
)$`

(%i18) `A:setMatrix();` 
$$\begin{pmatrix} 2 & 0.90909 & 0 & 0 & 0 \\ 0.66667 & 2 & 0.33333 & 0 & 0 \\ 0 & 0.58824 & 2 & 0.41176 & 0 \\ 0 & 0 & 0.4375 & 2 & 0.5625 \\ 0 & 0 & 0 & 0.81818 & 2 \end{pmatrix} \quad (A)$$

Die Lösungen für  $M_1 \dots M_5$  - statt *TDMA* verwenden wir die  
(%i19) Matrixinversion  
`M:invert(A).dVec;` 
$$\begin{pmatrix} -0.66867 \\ 1.3211 \\ -3.2891 \\ -0.6056 \\ 8.7326 \end{pmatrix} \quad (M)$$

Wir ergänzen  $M_0$  und  $M_6$  und nennen dieses Feld  $m$ ; `length(pL) = 7`

(%i21) `m[0]:0$ m[length(pL)-1]:0$`

Zugriff auf  $M$  wird kodiert ( $M$  ist eine Matrix - für ein Feld wird nur die erste Komponente benötigt)

(%i22) `m[i]:=first(M[i])$`

(%i23) `makelist(m[i],i,1,length(M));` [-0.66867, 1.3211, -3.2891, -0.6056, 8.7326] (%o23)

Die Koeffizienten für die  $C_i$  werden erstellt:  $a_s, b_s, c_s, d_s$

(%i24) `a_s:makelist(m[i-1]/(6*h[i]),i,1,length(h));` [0.0, -0.055722, 0.22018, -0.78311, -0.11215, 7.2772] (a\_s)

(%i25) `b_s:makelist(m[i]/(6*h[i]),i,1,length(h));` [-0.55722, 0.11009, -0.54818, -0.14419, 1.6171, 0.0] (b\_s)

(%i26) `c_s:makelist(p_y[i-1]/h[i]-m[i-1]*h[i]/6,i,1,length(h));` [0, 0.2729, 0.78, 3.241, 1.76, 6.71] (c\_s)

(%i27) `d_s:makelist(p_y[i]/h[i]-m[i]*h[i]/6,i,1,length(h));` [0.5223, 0.05964, 2.5482, 2.2135, 0.2457, 10] (d\_s)

Der Term für die einzelnen  $C_i$  - `charfun(A)` ist die charakteristische Funktion (Indikatorfunktion)

$x \in A \rightarrow 1 \quad x \notin A \rightarrow 0$

(%i28) `C[i](x):=( a_s[i]*(p_x[i]-x)^3 + b_s[i]*(x-p_x[i-1])^3 + c_s[i]*(p_x[i]-x)+  
d_s[i]*(x-p_x[i-1]))*charfun("and" (x>xLimits[i],x<=xLimits[i+1]))$`

Für Vergleichszwecke lassen wir uns die Terme ausgeben

(%i29) `showSplineBranches(last):= for i thru last do display(C[i](x))$`

# 1. Spline-Interpolation

(%i30) showSplineBranches(length(xList)-1)\$

$$C_1(x) = (0.52229x - 0.55722x^3) \text{ charfun } (x>0 \text{ and } x\leq 0.2)$$

$$C_2(x) = (0.059644(x - 0.2) + 0.27289(2.2 - x) + 0.11009(x - 0.2)^3 - 0.055722(2.2 - x)^3) \text{ charfun } (x>0.2 \text{ and } x\leq 2.2)$$

$$C_3(x) = (2.5482(x - 2.2) + 0.77982(3.2 - x) - 0.54818(x - 2.2)^3 + 0.22018(3.2 - x)^3) \text{ charfun } (x>2.2 \text{ and } x\leq 3.2)$$

$$C_4(x) = (2.2135(x - 3.2) + 3.2409(3.9 - x) - 0.14419(x - 3.2)^3 - 0.78311(3.9 - x)^3) \text{ charfun } (x>3.2 \text{ and } x\leq 3.9)$$

$$C_5(x) = (0.24567(x - 3.9) + 1.7575(4.8 - x) + 1.6171(x - 3.9)^3 - 0.11215(4.8 - x)^3) \text{ charfun } (x>3.9 \text{ and } x\leq 4.8)$$

$$C_6(x) = (10.0(x - 4.8) + 6.7089(5 - x) + 7.2772(5 - x)^3) \text{ charfun } (x>4.8 \text{ and } x\leq 5)$$

The whole Spline ist the sum of alle spline-branches

(%i31) define(S(x),sum(C[i](x),i,1,length(xList)-1))\$

*wxMaxima* does not know the derivative of the characteristic function, so we must tell him - we declare a pattern match

(%i32) matchdeclare ([aa, bb], numberp, xx, symbolp)\$

The following pattern should be substituted by zero - no Dirac Delta-distribution is needed here

(%i33) tellsimp ('diff (charfun (xx > aa and xx <= bb), xx), 0)\$

Now the derivative can be calculated without any  $\frac{d}{dx} \text{charfunc}$  terms in the result

(%i34) define (S\_1(x), diff(S(x),x))\$

(%i35) plot2d([[discrete,pL],S(x),S\_1(x)],[x,first(xList),last(xList)],[y,-1.3,2.8],[style,[points,3,1,1],[lines,3,3,2],[lines,2,4,2]])\$

$\int_{0.5}^{4.5} f_s(x) dx$  wird numerisch berechnet - zum Vergleich mit *Geogebra*

(%i36) quad\_qag (S(x),x,0.5,4.5, 3, 'epsrel=5d-8); [4.4085, 1.212910<sup>-7</sup>, 465, 0] (%o36)

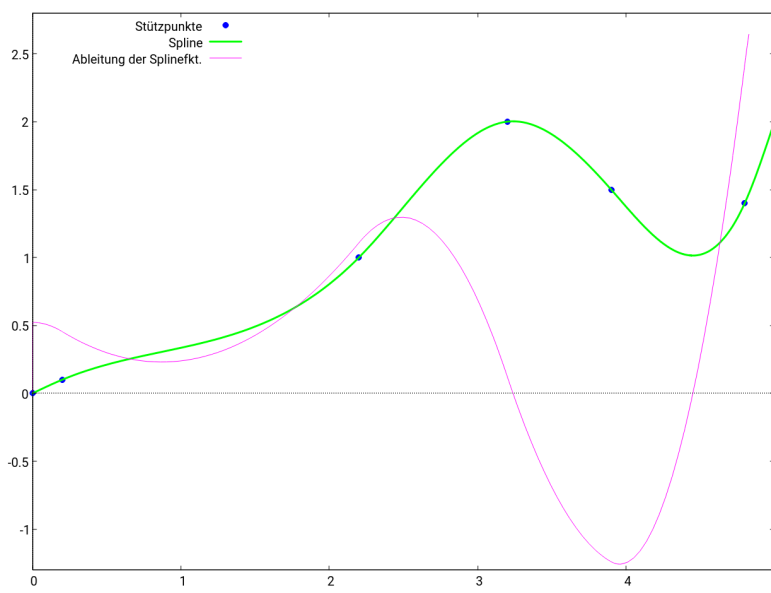
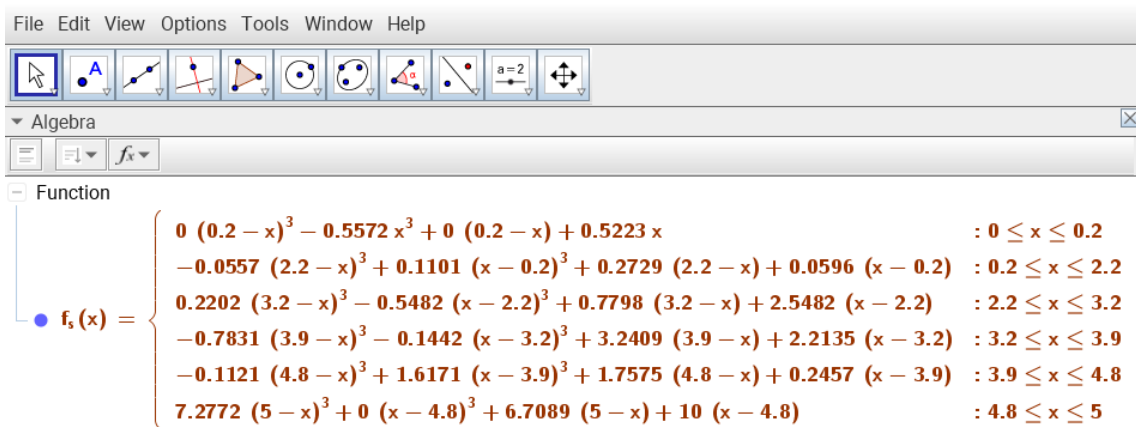


Abb.2 : Spline und seine Ableitung in *wxMaxima*-plot

Hier noch einmal die einzelnen Zweige der Spline-Funktion der beiden Programme zum Vergleich “herausgeschnitten”.

```
(%i33) showSplineBranches(length(xList)-1)$
C1(x)=(0.52229 x-0.55722 x3) charfun(x>0 ∧ x<=0.2)
C2(x)=(0.059644 (x-0.2)+0.27289 (2.2-x)+0.11009 (x-0.2)3-0.055722 (2.2-x)3) charfun(x>0.2 ∧ x<=2.2)
C3(x)=(2.5482 (x-2.2)+0.77982 (3.2-x)-0.54818 (x-2.2)3+0.22018 (3.2-x)3) charfun(x>2.2 ∧ x<=3.2)
C4(x)=(2.2135 (x-3.2)+3.2409 (3.9-x)-0.14419 (x-3.2)3-0.78311 (3.9-x)3) charfun(x>3.2 ∧ x<=3.9)
C5(x)=(0.24567 (x-3.9)+1.7575 (4.8-x)+1.6171 (x-3.9)3-0.11215 (4.8-x)3) charfun(x>3.9 ∧ x<=4.8)
C6(x)=(10.0 (x-4.8)+6.7089 (5-x)+7.2772 (5-x)3) charfun(x>4.8 ∧ x<=5)
```

Abb.3 : Spline Terme in *wxMaxima*Abb.4 : Spline Terme in *Geogebra*

Die Zweige stimmen haarscharf überein. Bei jedem Programm ist allerdings Handarbeit nötig: Bei *Geogebra* ist es meiner Ansicht etwas mehr - man könnte sich zwar den TDMA-Algorithmus sparen (*Geogebra* kann Matrizen invertieren), aber den gesamten Algorithmus mit bordeigenen Mitteln nachzubilden ist doch recht mühsam - wie wir im nächsten Abschnitt sehen werden. Da bietet eine ausgereifte Programmiersprache wie Javascript doch mehr Möglichkeiten. Dafür lässt sich ein in *Geogebra*-Script implementierte Lösung mit einem “Custom-Tool” verknüpfen und damit als eigene Datei/Icon leicht benutzen.

In *wxMaxima* könnte man noch mehr verkürzen, indem man das Package “interpol” verwendet:

```
load(interpol);
pL:[[0, 0], [0.2, 0.1], [2.2, 1], [3.2,2],[3.9,1.5], [4.8, 1.4],[5,2]]$
define(S(x),cspline(pL));
```

Hier wird dann die Indikator-Funktion *charfun2()* verwendet (sie gibt statt Null oder Eins *true* oder *false* zurück), dies erlaubt numerische Integration, aber für das Differenzieren braucht man ebenfalls händische Vereinfachung durch “pattern-matching” wie oben.

### 1.5 Implementierung in *Geogebra (native invert)*

Weitestgehend wurde hier derselbe Weg wie bei *wxMaxima* beschrieben - wobei das "human interface" mühsamer ist (wie weiter oben geschildert). Der Vollständigkeit halber sei aber auch dieser Weg hier gezeigt - vor allem für den reinen Anwender bleiben ja diese Mühen unsichtbar!

1. Am Beginn steht selbstverständlich die Punktliste:

```
pointL = {(0,0),(0.2,0.1),(2.2,1),(3.2,2),(3.9,1.5),(4.8,1.4),(5,2)}
```

2. Die Koordinatenlisten:

```
xList = Sequence(x(Element(pointL, i)), i, 1, Length(pointL))
```

```
yList = Sequence(y(Element(pointL, i)), i, 1, Length(pointL))
```

3. Jetzt die verschiedenen Differenzenlisten:

```
h = Sequence(Element(xList, i+1) - Element(xList, i), i, 1, Length(xList)-1)
```

```
hy = Sequence(Element(yList, i+1) - Element(yList, i), i, 1, Length(yList)-1)
```

```
hyOverH = Sequence(Element(hy, i) / Element(h, i), i, 1, Length(h))
```

```
deltaHyOverH = Sequence(Element(hyOverH, i+1) -  
-Element(hyOverH, i), i, 1, Length(hyOverH)-1)
```

4. Die Konstantenliste  $d$  wird ermittelt:

```
dV = Sequence(6/(Element(h, i+1) + Element(h, i))*
```

```
*Element(deltaHyOverH, i), i, 1, Length(deltaHyOverH))
```

5. Aus  $dV$  wird ein Spaltenvektor erzeugt (achten Sie auf die geschwungenen Klammern!):

```
dVec = Sequence({Element(dV, i)}, i, 1, Length(dV))
```

6. Nun wird  $\mu$  bestimmt ( $\lambda = 1 - \mu$ ; wir verzichten auf griechisch):

```
mu = Sequence(Element(h, i)/(Element(h, i+1)+Element(h, i)), i, 1, Length(h)-1)
```

7. Jetzt die Matrix (ein "furchtbarer" Befehl in Geogebra-script):

```
A = Sequence(Sequence(If(i == j, 2, i-j == 1, 1-Element(mu, j),
```

```
  j-i == 1, Element(mu, j), 0), i, 1, Length(mu)), j, 1, Length(mu))
```

Die innere Folge steht für die  $j$ -te Zeile der Matrix;  $i$  ist also der Spaltenindex; in der 1. Zeile ( $j = 1$ ) kann es kein  $mu$  geben ( $j - i \leq 0$ )

8. Wir invertieren  $A$  ( $B = A^{-1}$ ):

```
B = Invert(A)
```

9. Wir berechnen den Lösungs(spalten)vektor  $M_1 \dots M_5$ :

```
M = B * dVec
```

10.  $M_0 = 0$  und  $M_6 = 0$  wird hinzugefügt und Spaltenvektor auf *Liste m* konvertiert:

```
m = Sequence(If(i==0 || i==Length(M)+1, 0, Element(M, i, 1)), i, 0, Length(M)+1)
```

11. Die Koeffizienten von 1.11 werden als Liste von Listen (Matrix) berechnet (ein Monster):
- ```
coeffs = Sequence({Element(m,i)/(6*Element(h,i)),Element(m,i+1)/(6*Element(h,i)),
  Element(yList,i)/Element(h,i)-Element(m,i)*Element(h,i)/6,
  Element(yList,i+1)/Element(h,i)-Element(m,i+1)*Element(h,i)/6},
  i,1,Length(m)-1)
```

Man beachte das geschwungene Klammersymbol zum Erzeugen der inneren Liste (Zeile der Matrix). Die einzelnen Zeilen werden von *Sequence* erzeugt.

12. Noch ein Kraftakt und wir sind fast am Ziel: wir erstellen eine Liste der Spline-Zweige:

```
F = Sequence(If(Element(xList,i)<x<=Element(xList,i+1),
  Element(coeffs,i,1)*(Element(xList,i+1)-x)^3 +
  Element(coeffs,i,2)*(x-Element(xList,i))^3 +
  Element(coeffs,i,3)*(Element(xList,i+1)-x) +
  Element(coeffs,i,4)*(x-Element(xList,i)),0),i,1,Length(xList)-1)
```

Beachte das Kleinerzeichen beim *If*-Befehl - dadurch gehört die 1. Stützstelle nicht zur Spline-Funktion sondern zur konstanten Nullfunktion. Für ein Integral oder die Ableitungsfunktion hat dies - soweit ich weiß - keine Auswirkungen. Käme man auf die Idee das  $<$  durch  $\leq$  zu ersetzen (weil die Funktionswerte ohnehin an diesen Stellen übereinstimmen) passiert Folgendes:



Bei der anschließenden Summenbildung sind bei der zweiten, dritten bis zur letzten Stützstelle genau 2 Zweige zuständig, sodass sich der Funktionswert an diesen Stellen verdoppelt. Offenbar wird dies, wenn man einen Punkt auf die Spline-Funktion  $f_{sp}$  setzt und anschl. auf eine dieser Stützstellen verschiebt, springt dieser auf "wundersame Weise" auf den doppelten Funktionswert! Möchte man die erste Stützstelle unbedingt bei der Spline-Funktion dabei haben, müsste man die 1. Stützstelle dazugeben:

```
F_1 = Append(F,If(x==Element(xList,1),Element(yList,1),0)
```

13. So jetzt noch die eigentliche Spline-Funktion erstellen:  $f_{\{sp\}} = \text{Sum}(F)$

Sie verhält sich etwas anders als die Javascript-Version, weil sie außerhalb der Stützstellen den Wert 0 hat. Sollte das stören, kann man noch eine Indikator-Funktion (charakteristische Funktion) hinzufügen:

```
ind_f(x) = If(Element(xList, 1) < x Element(xList, Length(xList)), 1, 0)
g(x) = If(ind_f(x) > 0, f_{\{sp\}}(x))
```

Für das " $<$ " -Zeichen in der Indikator-Funktion siehe obige Zusatzbemerkung!

Der Vorteil dieses umständlichen Verfahrens gegenüber dem mit Javascript ist der, dass es auf eine Veränderung der Punktliste sofort wieder automatisch angewandt wird - während man bei der Javascript-Version erneut den Button anklicken muss!

Jetzt heißt es das Benutzer-Werkzeug (Custom Tool) erstellen. Da dies aber exemplarisch für die verschiedensten "Funktionen" gilt, ist dem ein eigener Abschnitt gewidmet.

## 1.6 Zerlegung in LU-Matrizen

Wir haben ein Gleichungssystem der Form  $A_T u = L \overbrace{U}^y u = f$  - wobei  $A_T$  unsere Tridiagonalmatrix ist.  $L$  ist eine untere und  $U$  eine obere (lower, upper) Dreiecksmatrix der folgenden Form:

$$\begin{pmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ 0 & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & a_n & b_n \end{pmatrix} = \begin{pmatrix} 1 & & & & \\ l_2 & 1 & & & 0 \\ & l_3 & 1 & & \\ 0 & & \ddots & \ddots & \\ & & & l_n & 1 \end{pmatrix} \begin{pmatrix} v_1 & c_1 & & & 0 \\ & v_2 & c_2 & & \\ & & \ddots & \ddots & \\ 0 & & & v_{n-1} & c_{n-1} \\ & & & & v_n \end{pmatrix}$$

■ Wir bestimmen die  $L$  und  $U$  durch Ausmultiplizieren und Vergleichen


$$\begin{aligned} b_1 &= v_1 & \Rightarrow & v_1 = b_1 \\ a_k &= l_k v_{k-1} & \Rightarrow & l_k = a_k / v_{k-1} \quad k = 2, \dots, n \\ b_k &= l_k c_{k-1} + v_k & \Rightarrow & v_k = b_k - l_k c_{k-1} = b_k - a_k (c_{k-1} / v_{k-1}) \end{aligned} \quad (1.24)$$

■ Wir bestimmen  $y$  mit  $Ly = f$

$$\begin{aligned} y_1 &= f_1 \\ y_k &= f_k - l_k y_{k-1} \quad k = 2, \dots, n \end{aligned} \quad (1.25)$$

■ Wir bestimmen  $u$  mit  $Uu = y$

$$\begin{aligned} u_n &= \frac{y_n}{v_n} \\ u_k &= \frac{y_k - c_k u_{k+1}}{v_k} = \underbrace{\frac{y_k}{v_k}}_{p_k} - \underbrace{\frac{c_k}{v_k}}_{q_k} u_{k+1} \quad k = (n-1), \dots, 1 \end{aligned} \quad (1.26)$$

 Die normale Inversion einer Matrix ist von den Multiplikationen ca.  $\mathcal{O}(n^3)$ , die obige  $LU$ -Zerlegung hat ca.  $\mathcal{O}(3n)$ . Bei größeren  $n$  wirkt sich das beachtlich aus!

Während man in *wxMaxima* obige Rekursionformeln mit indizierten Funktionen direkt implementieren kann (Beachte die verschiedenen Zuweisungszeichen!)

```
v[1]:b[1]$      1[k]:= a[k]/v[k-1]
```

ist dies in *Geogebra* etwas schwieriger



### 1.6.1 Rekursion in Geogebra

Eine Möglichkeit (sie geht auf eine Idee von *Michel Iroir* im Geogebra-Forum zurück) benutzt den Befehl *IterationList* und macht aus  $v_k$  einen "Punkt"  $(v, k)$  - also die  $x$ -Koordinate ist der Wert der Folge, die  $y$ -Koordinate ist der Index. Damit lässt sich mit

```
IterationList( <Expression>, <Variables>, <Start Values>, <Count> )
```

für Gleichungen 1.24 eine Liste erzeugen (beachte die "Punkteklammer" vor dem ersten *El...*

```
IterationList( (Element(b, y(A)+1)-Element(a, y(A)+1)*Element(c, y(A))/x(A),
               y(A)+1), A, {(Element(b, 1), 1)}, Length(c)-1)
```

Die erste "Iteration" (sie ist ja noch keine) liefert die Initialisierungswerte als Liste

```
{(b[1], 1)}
```

bei der ersten richtigen Iteration wird der Ausdruck ausgeführt und zur Liste hinzugefügt

```
{(b[2]-a[2]*c[1]/v[1], 2), (b[1], 1)}
```

die zweiten Iteration ergibt

```
{(b[3]-a[3]*c[2]/v[2], 3), (b[2]-a[2]*c[1]/v[1], 2), (b[1], 1)}
```

usw.

Wir müssen also  $(n - 1)$  Iterationen ausführen, um  $(v_n, n)$  zu erhalten, wobei  $n$  die Länge der Listen  $a$ ,  $b$  und  $c$  ist. Um die Folgenliste zu erhalten, brauchen wir von der Punktliste die  $x$ -Koordinaten, also lautet jetzt der vollständige Befehl für 1.24

```
x(IterationList( (Element(b, y(A)+1)-Element(a, y(A)+1)*Element(c, y(A))/x(A),
                 y(A)+1), A, {(Element(b, 1), 1)}, Length(c)-1))
```

Wenn es sich nicht um eine Rekursion handelt - wie z.B. bei  $l$ -Feld von 1.24 (rechts vom Gleichheitszeichen sind ja alle Elemente bekannt), dann bietet sich der *Zip*-Befehl von *Geogebra* an: `l=Join({{0}, Zip(a(k) / v(k - 1), k, 2...Length(a))})`



Beachte, dass das Feld  $l$  erst beim Index 2 beginnt, also fügen wir vorne eine "Dummy-Null" ein. Außerdem erlaubt *Geogebra* (Desktop-Version 5.0.470) beim *Zip*-Befehl eine abgekürzte Schreibweise für den *Element*-Befehl

```
a(k) statt Element(a, k)
```

## 1.7 Spline in *Geogebra* (L U-Zerlegung)

Um die Rechnungen etwas zu vereinfachen benutzen wir hier 2 *Custom Tools*:

1. `Delta(<Liste>)` --> `{l[2]-l[1], l[3]-l[2], ...l[n]-l[n-1]}`

2. `SuccSum(<Liste>)` --> `{l[2]-l[1], l[3]-l[2], ...l[n]-l[n-1]}`

*Delta(<Liste>)* bildet aus einer Liste von Zahlen eine Liste der Differenz der aufeinanderfolgenden Zahlen.

*SuccSum(<Liste>)*(successor sum) bildet aus einer Liste von Zahlen eine Liste der Summe der aufeinanderfolgenden Zahlen.

## 1. Spline-Interpolation

---

Implementiert wurden diese beiden Benutzer-Funktionen mit folgenden Befehlen:

```
a = {1, 2, 3, 4, 5}
b = Zip(a(k), k, 1...(Length(a) - 1))
c = Zip(a(k), k, 2...Length(a))
r = c - b --> Delta
s = b + c --> SuccSum
```

Wie das im einzelnen funktioniert wird in 1.8 näher erklärt.

Hier das Befehlslisting

```
1 pointL={ (0,0), (0.2,0.1), (2.2,1), (3.2,2), (3.9,1.5), (4.8,1.4), (5,2) }
2 xList = x(pointL)
3 yList = y(pointL)
4 // Einsatz der neuen Tools
5 h = Delta(xList)
6 hy = Delta(yList)
7 SuccSumH = SuccSum(h)
8 mu = Zip( h(k)/SuccSumH(k), k, 1...(Length(h)-1) )
9 lambda = 1 - mu
10 hyOverH = hy/h
11 deltaHyOverH = Delta(hyOverH)
12 f = Zip(6*deltaHyOverH(k)/SuccSumH(k), k, 1...Length(deltaHyOverH))
13 // Rekursion wie im Text oben beschrieben
14 v = x(IterationList((2-Element(mu, y(A) + 1)*Element(lambda, y(A))/x(A),
15 y(A) + 1), A, {(2, 1)}, Length(mu) - 1))
16 l = Join({{0}}, Zip( mu(k)/v(k-1), k, 2...Length(mu) ))
17 // Rekursion wie im Text oben beschrieben
18 yL = x(IterationList((Element(f, y(A) + 1) - Element(l, y(A) + 1) x(A),
19 y(A) + 1), A, {(Element(f, 1), 1)}, Length(f) - 1))
20 p = Zip( yL(k)/v(k), k, 1...Length(v))
21 q = Zip( lambda(k)/v(k), k, 1...Length(v))
22 M = Reverse(x(IterationList((Element(p, y(A)-1)-Element(q, y(A)-1)*x(A),
23 y(A)-1), A, {(Element(p, Length(p)), Length(p))}, Length(p)-1)))
24 m = Join({{0}}, M, {0})
25 h6 = 6*h
26 hOver6 = h/6
27 coeffs = Zip({m(k)/h6(k), m(k+1)/h6(k), yList(k)/h(k)-m(k)*hOver6(k),
28 yList(k+1)/h(k)-m(k+1)*hOver6(k)}, k, 1...(Length(m)-1))
29 //..... REST bleibt gleich .....
30
31
```

In Zeile 14 beginnt Implementierung von 1.24

In Zeile 18 beginnt Implementierung von 1.25

In Zeile 22 beginnt Implementierung von 1.26

Man erkennt, dass etwas andere Befehle verwendet wurden, wie bei der vorigen Version (native invert). Es führen eben viele Wege nach Rom!

Es bleibt nach dem Erstellen der Splinefunktion wieder ein Benutzerwerkzeug zu bauen:

*csplineLU*

## 1.8 Erstellen eines Custom Tool in *Geogebra*

Ein *Custom Tool* (Benutzerwerkzeug) ist eine Funktion, deren Argumente (Parameter) Geogebra-Objekte sind und als Output 1 Geogebra-Objekt liefert.

$$ct(Obj1, Obj2, \dots, Obj_n) \mapsto Obj_{target}$$

Der Name des Tools (= Funktion) (hier *ct*) kann während des Erstellungsprozesses frei gewählt werden. Außerdem ist es hilfreich (aber nicht notwendig) ein eigenes 32x32 Icon zur Verfügung zu haben. Für das Spline-Tool hab ich mir schnell eines mit Gimp erstellt:

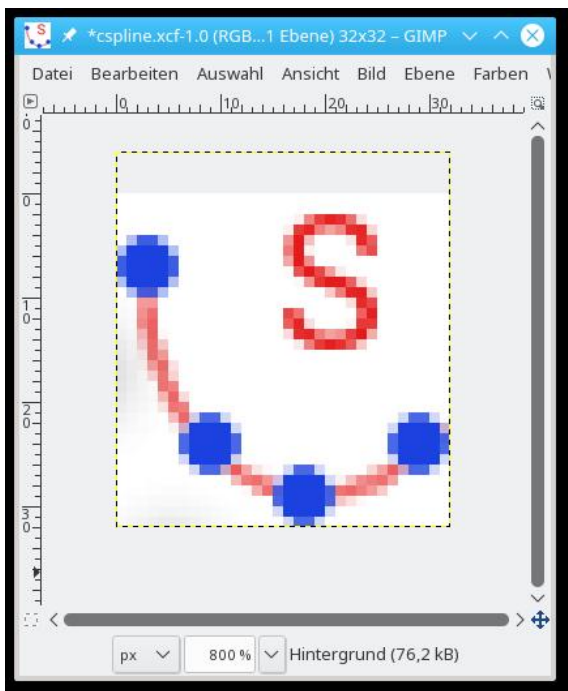


Abb.5 : Spline Icon für Geogebra-Menü

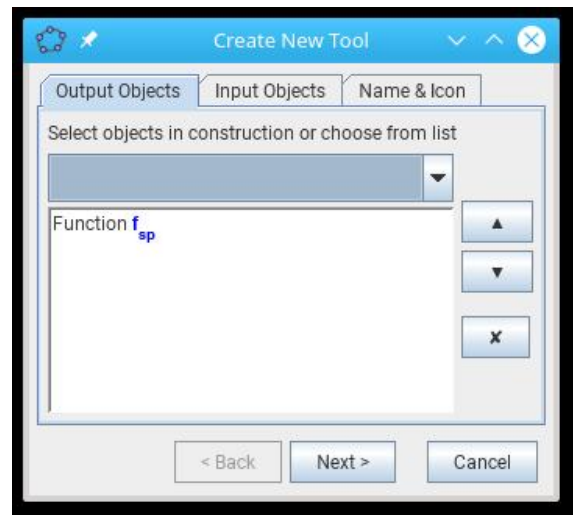


Abb.6 : Output des Custom Tool wird festgelegt

Damit *Geogebra* ein *Custom Tool* erzeugen kann, müssen alle Objekte (Parameter und Output) bereits existieren. Es wird als der Weg von den Parametern zum Output abgebildet!

Im *Tools*-Menü von *Geogebra* wird *Create New Tool* aufgerufen:

- Zuerst wird das Ergebnis(Output)-Objekt festgelegt (siehe Abb. 6)  $\rightarrow f_{sp}$
- Klicken von *Next* zeigt, dass *pointL* als Parameter (Input) benötigt wird
- Wieder klicken von *Next* bringt uns in den Reiter *name & icon*:  
Wir wählen den Namen des Werkzeugs und Befehl: **cspline** (für cubic spline) und wählen unser vorhin konstruiertes Icon aus!

## 1. Spline-Interpolation

Wenn die Auswahlbox *Show in Toolbar* angeklickt ist, sollte das Werkzeug mit unserem Icon im Werkzeugkasten zur Verfügung stehen - sollte man dies vergessen haben, kann man im *Tool*-Menü → *Customize Toolbar* das Icon in einer Ansicht(hier *General*) an einer bestimmten Stelle einfügen!

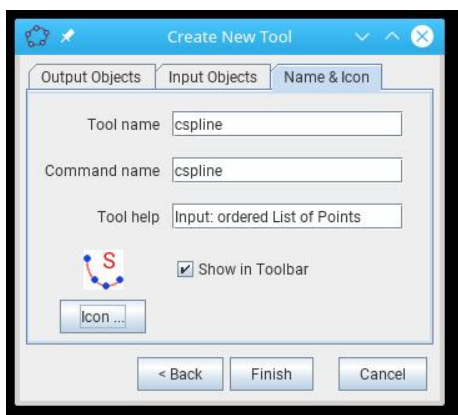


Abb.7 : Name,Befehl,Icon und Eingabetipp

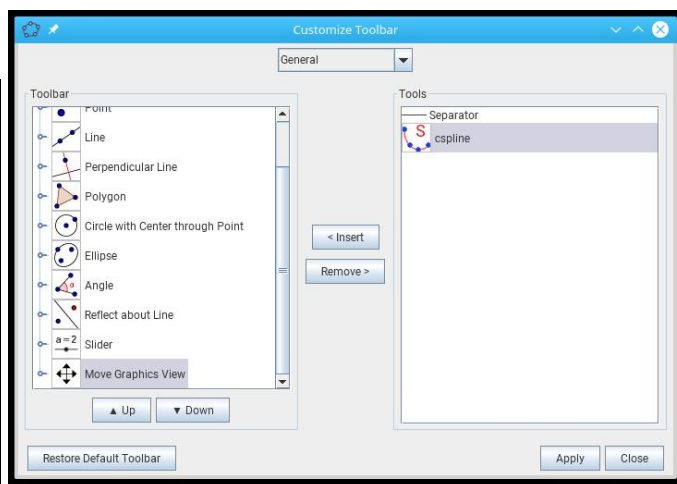


Abb.8 : Einfügen des Icons ins Menü

Nach dem Klicken von *Finish* kommt hoffentlich eine Erfolgsmeldung.

- Als nächstes sollte man das Werkzeug als eigenen File (\*.ggt, *geogebra tool*) speichern. Dies wird im *Tool*-Menü → *Manage Tools* mit *Save as* erledigt (ein aussagekräftiger Dateiname wäre nicht schlecht).
- Will man das Werkzeug verwenden, lädt man es wie mit dem *Datei*-Menü von *Geogebra*. Für das verwenden eines Werkzeugs **nicht(!)** *Manage Tools* → *Open* benutzen. Dies benutzt man, um den "Quellcode" (das Zustandekommen) eines Benutzerwerkzeugs nachzuvollziehen!
- Will man dieses Werkzeug in Zukunft beim Öffnen von *Geogebra* bereits zur Verfügung haben, gehen wir auf *Options* → *Save Settings*.

### 1.8.1 2 Verwendungsmöglichkeiten

1. Man klickt das Icon und anschließend die Punktliste
2. Man gibt in die Kommandozeile: `<FuncName> =cspline(<List of Points>)`

## 1.9 Theoretische Überlegungen

### 1.9.1 Thomas Algorithmus (TDMA)

Der TDMA 1.23 kann nur scheitern, wenn der Nenner  $b_i + a_i P_{i-1}$  verschwindet. Wenn wir uns die Matrix in 1.18 anschauen stellen wir fest:

1. Die Hauptdiagonale besteht aus der Zahl 2 ( $b_i$ )
2. Die Nebendiagonalen sind positiv und kleiner 1:  $0 < a_i < 1 \wedge 0 < c_i < 1$
3. Damit können wir eine vollständige Induktion starten:

$$\blacksquare |P_1| = \left| -\frac{c_1}{2} \right| < 1$$

- $$\blacksquare \text{ Falls } |P_{i-1}| < 1 \Rightarrow |P_i| < 1, \text{ weil der Nenner in 1.23 ist dann größer 1, bei einem Zähler der kleiner 1 ist.}$$

Es gilt also

$$\forall i \in \{1, \dots, n\} : |P_i| < 1 \Rightarrow b_i + a_i P_{i-1} > 0$$

### 1.9.2 LU-Faktorisierung

Der Algorithmus von 1.24 scheitert wenn  $v_k$  verschwindet.

$$\blacksquare v_1 = 2 > 1$$

- $$\blacksquare \text{ Wir zeigen auch hier: Wenn } v_{k-1} > 1 \text{ ist, dann ist es auch } v_k$$

$$v_k = 2 - \underbrace{a_k}_{< 1} \underbrace{\frac{c_{k-1}}{v_{k-1}}}_{< 1} > 1 \neq 0$$

Auch hier zeigt sich, dass bei unser Ausgangsmatrix der Algorithmus immer zum Erfolg führt!