

## ■ Basic definitions

Some necessary packages:

```
<< LinearAlgebra`MatrixManipulation`
<< Graphics`ImplicitPlot`
```

To suppress error messages for similar user defined functions:

```
Off[General::"spell1"]
Off[General::"spell"]
```

Since the knot vector can contain equal values, a division by zero can occur. In this case the quotient is set zero.

```
Unprotect[Power];
Power[0, -1] = 0;
Power[0., -1] = 0;
Protect[Power];
```

## ■ Definition of the basis functions

We merely use basis functions of degree 2 to approximate the ostracode outlines. We define the basis functions explicitly to accelerate the computation rate.

```
Bd[s_, j_, t_] := Which[t < s[j] || t > s[j + 3], 0,
  s[j] ≤ t ≤ s[j + 1] && s[j + 1] - s[j] > 0,  $\frac{(t - s[j])^2}{(-s[j] + s[j + 1])(-s[j] + s[j + 2])}$ ,
  s[j + 1] ≤ t ≤ s[j + 2] && s[j + 2] - s[j + 1] > 0,  $\frac{(t - s[j])(-t + s[j + 2])}{(-s[j] + s[j + 2])(-s[j + 1] + s[j + 2])}$  +
  ((t - s[j + 1])(-t + s[j + 3])) / ((-s[j + 1] + s[j + 2])(-s[j + 1] + s[j + 3])),
  s[j + 2] ≤ t ≤ s[j + 3] && s[j + 3] - s[j + 2] > 0,
  (-t + s[j + 3])^2 / ((-s[j + 1] + s[j + 3])(-s[j + 2] + s[j + 3]))]
```

## ■ Definition of the knot vector

The function  $so(k,n)$  provides a uniform knot vector, where the first  $k+1$  knots are 0, the last  $k+1$  knots 1 and the interior knots equally spaced.

```
so[k_, n_][j_Integer] := 0 /; 0 ≤ j ≤ k
so[k_, n_][j_Integer] := (j - k) / (n - k + 1) /; k < j < n + 1
so[k_, n_][j_Integer] := 1 /; n + 1 ≤ j ≤ n + k + 1
```

## ■ Definition of the B-spline curve

$B2OSpline(kp,t)$  computes an open B-spline curve of degree 2. The further routines plot the graph of the B-spline curve with its control points and its control polygon.

```

B2OSpline[kp_List, t_] :=
  Sum[kp[[j + 1]] * Bd[so[2, Length[kp] - 1], j, t], {j, 0, Length[kp] - 1}]

B2OSplinePlot[kp_, opts_: {}] :=
  ParametricPlot[B2OSpline[kp, t], {t, 0, 1}, Compiled → False, opts]

KontrollPolygonPlot[kp_, opts_: {}] :=
  ListPlot[kp, PlotJoined → True, PlotStyle → GrayLevel[0.8], opts];

KontrollRects[kp_] := Graphics[
  Map[{RGBColor[1, 1, 0], Rectangle[Offset[{-2, -2}, #], Offset[{2, 2}, #]]} &, kp]];

KontrollBox[p_, s_] := Line[{Offset[{-s, -s}, p], Offset[{s, -s}, p],
  Offset[{s, s}, p], Offset[{-s, s}, p], Offset[{-s, -s}, p]}];
KontrollBoxes[kp_] := Graphics[Map[KontrollBox[#, 2] &, kp]];

KontrollText[p_, i_] := Text[ToString[First[i - 1]], Offset[{0, 10}, p]];
KontrollTexts[kp_] := Graphics[MapIndexed[KontrollText, kp]];

KontrollPunktePlot[kp_, opts_: {}] := Show[
  {KontrollRects[kp], KontrollBoxes[kp], KontrollTexts[kp]}, PlotRange → All, opts];

```

## ■ Computation of the control points

To compute the control points we have to determine the parameters  $t_i$  first. This is done in the function  $params(dp)$  by a parameterization using the chord length, see section 2.4.1.

```

norm[{a_, b_}] := Sqrt[a^2 + b^2]
params[dp_] := Module[{paramlist}, paramlist =
  FoldList[Plus, 0., Map[norm, Table[dp[[i]] - dp[[i - 1]], {i, 2, Length[dp]}]]];
  paramlist / Last[paramlist]]

```

To get a better approximation to the point data we alter the parameter values  $t_i$  by adding a value  $\lambda_i$  as long as  $C(t_i) - Q_i$  is roughly perpendicular to the tangent  $C'(t_i)$ , see section 3.4.  $DBd(s,j,t)$  evaluates the derivate of the basis functions

```
DBd[s_, j_, t_] = D[Bd[s, j, t], t];
```

and  $DB2OSpline(kp,t)$  computes the derivate of the curve.

```

DB2OSpline[kp_List, t_] :=
  Sum[kp[[j + 1]] * DBd[so[2, Length[kp] - 1], j, t], {j, 0, Length[kp] - 1}];

```

The following functions determine the values  $\lambda_i$  and calculate new control points.

```

lambda[kp_, dp_, ti_, i_] :=
  Module[{b = B2OSpline[kp, ti[[i]]], d = DB2OSpline[kp, ti[[i]]]},
    ((dp[[i]] - b).d) / norm[d]^2];
t[kp_, dp_, ti_, i_] := ti[[i]] + lambda[kp, dp, ti, i];

```

$fitptiter(dp, anz, n)$  returns  $n$  control points defining a second degree open B-spline curve approximating the given dataset  $dp$  after adjusting the parameters  $anz$  times.

```
fitkpktiter[dp_, anz_, n_] :=
Module[{pariter}, pariter = pariterationanz[dp, anz, n]; kp[pariter, dp, n]]
```

## ■ Approximation to point data

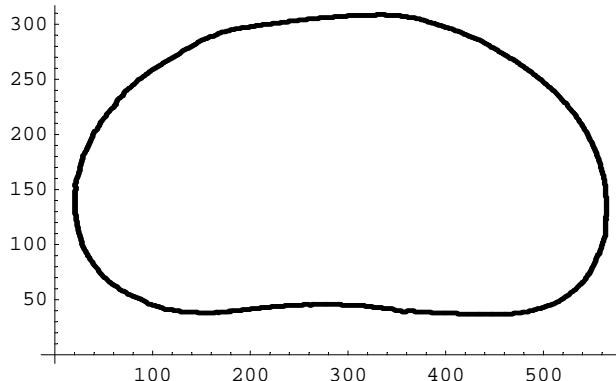
### ■ Importing the tps-data

```
datenpunktelesen[fileName_String] :=
Module[{stream, a, length, p, b}, stream = OpenRead[fileName];
a = Read[stream]; For[b = 1, b <= a, b++, Read[stream, {Number, Number}]];
Read[stream]; length = Read[stream]; p =
ReadList[stream, {Number, Number}, length, RecordLists → False]; Close[stream]; p]
```

```
SetDirectory["d:\Mathematik\Diplomarbeit\Arcine"]
```

```
d:\Mathematik\Diplomarbeit\Arcine
```

```
ListPlot[datenpunktelesen["Arc1.tps"]]
```



- Graphics -

### ■ Center of gravity and main axes of inertia

*schwerpkt(pnts)* and *tachsen(pnts)* calculate the center of gravity and the main axes of interia of the domain surrounded by the contour points, see section 3.3.

```
F1[Pgon_List] :=

$$\frac{1}{2} \sum_{i=1}^{\text{Length}[Pgon]-1} (\text{Pgon}[[i+1, 1]] - \text{Pgon}[[i, 1]]) * (\text{Pgon}[[i, 2]] + \text{Pgon}[[i+1, 2]]);$$


schwerpkt[pnts_List] :=
Module[{n, a, sx, sy, Pgon}, n = Length[pnts]; Pgon = Append[pnts, pnts[[1]]];
a = F1[Pgon]; sx = - $\frac{1}{6*a} \sum_{i=1}^n (\text{Pgon}[[i+1, 2]] - \text{Pgon}[[i, 2]]) *$ 
 $(\text{Pgon}[[i, 1]]^2 + \text{Pgon}[[i, 1]] * \text{Pgon}[[i+1, 1]] + \text{Pgon}[[i+1, 1]]^2);$ 
sy =  $\frac{1}{6*a} \sum_{i=1}^n (\text{Pgon}[[i+1, 1]] - \text{Pgon}[[i, 1]]) *$ 
 $(\text{Pgon}[[i, 2]]^2 + \text{Pgon}[[i, 2]] * \text{Pgon}[[i+1, 2]] + \text{Pgon}[[i+1, 2]]^2); \{sx, sy\}$ 
```

```

tachsen[pnts_List] := Module[{s, q, hta, n, a11, a12, a21, a22, Pgon, Ev},
  s = schwerpkt[pnts]; q = Map[# - s &, pnts]; Pgon = Append[q, q[[1]]];
  Pgon = Prepend[Pgon, Pgon[[Length[Pgon] - 1]]]; n = Length[Pgon];
  a11 = Sum[(Pgon[[i - 1, 2]]^3 + Pgon[[i - 1, 2]]^2 * Pgon[[i, 2]] +
    Pgon[[i - 1, 2]] * Pgon[[i, 2]]^2 - Pgon[[i, 2]]^2 * Pgon[[i + 1, 2]] -
    Pgon[[i, 2]] * Pgon[[i + 1, 2]]^2 - Pgon[[i + 1, 2]]^3) * Pgon[[i, 1]], {i, 2, n - 1}];
  a12 = Sum[(-1/2 * Pgon[[i - 1, 2]]^2 - Pgon[[i - 1, 2]] * Pgon[[i, 2]] +
    Pgon[[i, 2]] * Pgon[[i + 1, 2]] + 1/2 * Pgon[[i + 1, 2]]^2) * Pgon[[i, 1]]^2 +
    (Pgon[[i + 1, 2]]^2 - Pgon[[i, 2]]^2) * Pgon[[i, 1]] * Pgon[[i + 1, 1]], {i, 2, n - 1}];
  a21 = a12; a22 = Sum[(-Pgon[[i - 1, 1]]^3 - Pgon[[i - 1, 1]]^2 * Pgon[[i, 1]] -
    Pgon[[i - 1, 1]] * Pgon[[i, 1]]^2 + Pgon[[i, 1]]^2 * Pgon[[i + 1, 1]] +
    Pgon[[i, 1]] * Pgon[[i + 1, 1]]^2 + Pgon[[i + 1, 1]]^3) * Pgon[[i, 2]], {i, 2, n - 1}];
  hta = Eigenvectors[{{a11, a12}, {a21, a22}}];
  Ev = Eigenvalues[{{a11, a12}, {a21, a22}}];
  If[Ev[[1]] ≥ Ev[[2]], hta, {hta[[2]], hta[[1]]}]
  TraegheitsachsenPlot[s_, tha_List, fact_, opts_: {}] :=
  Show[Graphics[{RGBColor[0, 0, 1], Line[{s, s + fact * tha[[1]]}],
    Line[{s, s + fact * tha[[2]]}], Text["S", s - {20, 20}],
    Text["v", s + 1.2 * fact * tha[[1]]], Text["w", s + 1.2 * fact * tha[[2]]]}], opts]

```

## ■ Dividing of the contour into two halves

*halbieren(s,tha,pnts)* divides the contour into two halves where the dividing line is the main axis of inertia with minimum moment, see section 3.4.

```

schnittpunkt[pktA_, pktB_, pkts_, vctV_] := Module[{eqns, solution, lambda, mu},
  eqns = {pkts[[1]] + lambda * vctV[[1]] == pktA[[1]] + mu * (pktB[[1]] - pktA[[1]]),
    pkts[[2]] + lambda * vctV[[2]] == pktA[[2]] + mu * (pktB[[2]] - pktA[[2]])};
  pkts + lambda * vctV /. Solve[eqns, {lambda, mu}][[1]]];
halbieren[s_, tha_, pnts_List] :=
Module[{htha, paare, ol, ul, ur, or, olp, orp, ulp, urp, oh, uh, ls, rs},
(* The axes with the minor eigenvalue
corresponds with the axis with minimum moment of inertia *)
htha = tha[[2]];
(* A list of all pairs of consecutive points is generated *)
paare = Partition[pnts, 2, 1, 1];
(* This determines those pairs of
consecutive points lying on both sides of the dividing line *)
{ol, ul} = First[Select[paare, (#[[1]] - s).htha > 0. && (#[[2]] - s).htha <= 0. &]];
{ur, or} = First[Select[paare, (#[[1]] - s).htha < 0. && (#[[2]] - s).htha >= 0. &]];
olp = Position[pnts, ol][[1, 1]]; orp = Position[pnts, or][[1, 1]];
ulp = Position[pnts, ul][[1, 1]]; urp = Position[pnts, ur][[1, 1]];
(* Building the upper and lower contour *)
oh = If[orp < olp, Take[pnts, {orp, olp}],
  Join[Take[pnts, {orp, Length[pnts]}], Take[pnts, {1, olp}]]];
uh = If[ulp < urp, Take[pnts, {ulp, urp}],
  Join[Take[pnts, {ulp, Length[pnts]}], Take[pnts, {1, urp}]]];
(* This computes the intersection points of the contour
with the dividing line and adds these points to the contour *)
rs = schnittpunkt[or, ur, s, tha[[1]]]; ls = schnittpunkt[ol, ul, s, tha[[1]]];
oh = Prepend[Append[oh, ls], rs]; uh = Prepend[Append[uh, rs], ls]; {oh, uh}]

```

## ■ Translation into standardized position

To achieve reasonable methods of analyzing the structur of the ostrcoda outlines, the shapes need to be standardized. This can be done by a transformation of all contour points, so that the center of gravity is in the origin and the axis with the minimum moment of inertia is identical with the x-axis.

```

normallage[s_, tha_, pnts_List] :=
Module[{v, w}, v = tha[[1]]; w = tha[[2]]; Map[{v, w}.(# - s) &, pnts]]
normalhalbieren[pnts_List] :=
Module[{s, tha, halfs}, s = schwerpkt[pnts]; tha = tachsen[pnts];
halfs = halbieren[s, tha, pnts]; Map[normallage[s, tha, #] &, halfs]]

```

## ■ Approximation of the divided contours

*berechneAxy(tioben,tiunten,n)* evaluates a matrix A with  $a_{ij} = N_{j,p}(t_i)$  for both halves depending on the parameter values of the upper and lower half and the number of control points.

```
berechneAxy[tioben_, tiunten_, n_] := Module[
  {soben, sunten, matAx, matAy}, soben = Length[tioben]; sunten = Length[tiunten];
  matAx = AppendColumns[AppendRows[Array[Bd[so[2, n], #2 - 1, tiunten[[#1]]] &,
    {sunten, n + 1}], ZeroMatrix[sunten, n - 1]], AppendRows[
    Array[Bd[so[2, n], n, tioben[[#1]]] &, {soben, 1}], ZeroMatrix[soben, n - 1],
    Array[Bd[so[2, n], #2 - 1, tioben[[#1]]] &, {soben, n}]]]; matAy = matAx;
  Do[matAy[[i, 1]] = matAy[[i, n + 1]] = 0.0, {i, 1, soben + sunten}]; {matAx, matAy}]
```

We obtain a solution of the overdetermined system of equations using the pseudo-inverse matrix  $A^+$  of the matrix A (section 2.4.2). Solving  $P = A^+ Q$ , where Q are the given points of the contour in the dataset, leads us to an approximative solution for the control points P.

The following function *berechnekp(tioben,tiunten,dpopen,dpunten,n)* computes the control points on the basis of the parameter values and contour data for the two halves using of the pseudo-inverse matrix.

```
berechnekp[tioben_List, tiunten_List, dpopen_List, dpunten_List, n_] :=
Module[{dp, matAx, matAy, kp, kpopen, kpunten}, dp = AppendColumns[dpunten, dpopen];
{matAx, matAy} = berechneAxy[tioben, tiunten, n];
kp = Transpose[{PseudoInverse[matAx].First /@ dp, PseudoInverse[matAy].Last /@ dp}];
kpopen = Append[Take[kp, {n + 1, 2 n}], First[kp]];
kpunten = Take[kp, {1, n + 1}]; {kpopen, kpunten}]

BSplineHalfsPlot[{kpo_, kpu_}, opts_: {}] :=
Show[B2OSplinePlot[kpu, opts], B2OSplinePlot[kpo, opts]]
```

Launches the parameter correction and executes one step.

```
korrigiereti[kp_List, dp_List, ti_List] :=
Module[{tikorr}, tikorr = Table[t[fp, dp, ti, i], {i, 2, Length[ti] - 1}];
tikorr = Prepend[tikorr, 0.]; tikorr = Append[tikorr, 1.]; tikorr]
```

*approxhalfsiter(dp,n,k)* approximates the contour data *dp* by a B-spline curve with *n* control points. It adjusts the parameter with *k* iteration steps.

```
approxhalfsiter[dp_List, n_, k_] :=
Module[{dpopen, dpunten, kpopen, kpunten, tioben, tiunten, tiobenkorr, tiuntenkorr},
{dpopen, dpunten} = normalhalbieren[dp];
tioben = params[dpopen]; tiunten = params[dpunten];
Do[{{kpopen, kpunten}} = berechnekp[tioben, tiunten, dpopen, dpunten, n];
tiuntenkorr = korrigiereti[kpunten, dpunten, tiunten];
tiobenkorr = korrigiereti[kpopen, dpopen, tioben];
tioben = tiobenkorr; tiunten = tiuntenkorr;}, {k}]; {kpopen, kpunten} =
berechnekp[tioben, tiunten, dpopen, dpunten, n]; {kpopen, kpunten}]
```

## ■ Normalizing of the outer control points

*normalisierekp(kpoben,kpunten)* determines the variables *lam* and *b* of the equation *sglg(lam,b,x)* in a way, that the end control points shift to (-1000/0) and (1000/0) respectively and applies *sglg(lam,b,x)* to all control points.

```
sglg[lam_, b_, x_] := lam*x + b

normalisierekp[{kpoben_, kpunten_}] :=
Module[{loes, lam, bx, kpobentrans, kpuntentrans},
loes = Solve[{sglg[lam, {bx, 0}], kpunten[[1]]} == {-1, 0},
    sglg[lam, {bx, 0}], kpoben[[1]]} == {1, 0}], {lam, bx}]; lam = lam /. loes;
bx = bx /. loes; kpobentrans = Map[sglg[lam[[1]], {bx[[1]], 0}, #] &, kpoben];
kpuntentrans = Map[sglg[lam[[1]], {bx[[1]], 0}, #] &, kpunten];
{kpobentrans, kpuntentrans}]
```

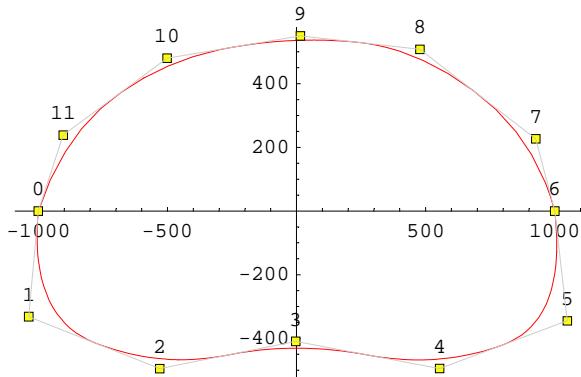
*krebsapproximieren(dp,n,k)* finally starts the routines for computing an approximating B-spline curve of the contour data. The user can select the number of control points and the number of iteration steps for the parameter correction.

```
krebsapproximieren[dp_, n_, k_] := normalisierekp[approxhalfsiter[dp, n, k]] * 1000
```

## ■ Graphical options and routines

```
Options[krebszeichnen] = {ControlPolygon -> Automatic, ControlPoints -> Automatic};
krebszeichnen[kp_, opts__?OptionQ] :=
Module[{plot1 = {}, plot2 = {}, plot3 = {}, plot2a = {}, plot4 = {}, plot5 = {}, optPlot,
    optShow, controlpolygon}, optPlot = FilterOptions[ParametricPlot, opts];
optShow = FilterOptions[Graphics, opts];
plot1 = BSplineHalfsPlot[kp, Flatten[{optPlot, DisplayFunction -> Identity}]];
controlpolygon = ControlPolygon /. Flatten[{opts, Options[krebszeichnen]}];
Which[controlpolygon === Automatic || controlpolygon === True,
    plot4 = KontrollPolygonPlot[kp[[1]], Flatten[{PlotStyle -> GrayLevel[0.6],
        DisplayFunction -> Identity, optPlot}]]]; plot5 = KontrollPolygonPlot[kp[[2]],
    Flatten[{PlotStyle -> GrayLevel[0.6], DisplayFunction -> Identity, optPlot}]]];
controlpoints = ControlPoints /. Flatten[{opts, Options[krebszeichnen]}];
Which[controlpoints === Automatic || controlpolygon === True,
    plot2 = KontrollPunktePlot[Join[Drop[kp[[2]], -1], Drop[kp[[1]], -1]],
        Flatten[{optPlot, DisplayFunction -> Identity}]]]; Show[plot1, plot2, plot4,
    plot5, Flatten[{optShow, PlotRange -> All, DisplayFunction -> $DisplayFunction}]]
krebseinlesenundzeichnen[file_, n_, k_, opts_] :=
Module[{dpall, dp, kp}, dpall = N[datenpunktelesen[file]];
kp = krebsapproximieren[dpall, n, k]; krebszeichnen[kp, opts]]
krebseinlesenundapproximieren[file_, n_, k_] := Module[{dpall, dp, kp},
dpall = N[datenpunktelesen[file]]; kp = krebsapproximieren[dpall, n, k]; kp]
```

```
krebseinlesenundzeichnen["Arc1.tps", 6, 6, PlotStyle→RGBColor[1, 0, 0]];
```



## ■ Area deviation

This module computes the cotangent values of the vertices of a polygon *pol*. The cotangent of the two end control points lying on the x-axis yields  $-\infty$  resp.  $\infty$ . Therefore, we set these cotangent values -1000 or 1000, respectively.

```
PolStg[Pol_] := Module[{T, t, s = Length[Pol]},
  T = Table[Pol[[t, 1]] / Pol[[t, 2]], {t, 1, s}]; T[[1]] = -1000; T[[s]] = 1000; T]
```

*PolygonFl(Liste)* evaluates the area of a polygon *Liste*.

```
PolygonFl[Liste_] := Module[{MPkt, A = 0, Pktlist = Liste},
  MPkt = (Pktlist[[1]] + Pktlist[[IntegerPart[Length[Pktlist] / 2 + 1]]]) / 2;
  AppendTo[Pktlist, Pktlist[[1]]];
  Do[A = A + Det[{{MPkt[[1]], MPkt[[2]], 1}, {Pktlist[[i, 1]], Pktlist[[i, 2]], 1},
    {Pktlist[[i + 1, 1]], Pktlist[[i + 1, 2]], 1}}] / 2,
    {i, 1, Length[Pktlist] - 1}]; Abs[A]]
```

The module *areadiff(kp1,kp2)* is the main routine of the calculation. It returns the area deviation of two superimposed polygons.

```
areadiff[kp1_, kp2_] :=
Module[{t, PS1, PS2, UntPkt, ObPkt = 2, Startpkt, x, y, xs, ys, Polygon,
c, Hilfsi = 1, Hilfsj = 1, F1 = 0, s = Length[kp1] - 1, A1, A2, B1, B2,
C1, C2, ParTest, p1, p2, q1, q2, PktStg, KonfTest = True, scale},

(* Computing of the cotangent values and conformity check *)
PS1 = PolStg[kp1];
Do[If[PS1[[i]] ≥ PS1[[i + 1]], KonfTest = False], {i, 1, s}];
PS2 = PolStg[kp2];
Do[If[PS2[[i]] ≥ PS2[[i + 1]], KonfTest = False], {i, 1, s}];
If[KonfTest == True,

(* Specifying the first point and the first segment of the polygon. *)
Startpkt = kp2[[1]];
Polygon = {Startpkt, kp1[[1]]};

(* The factor scale is required for the later proof of
numerically identical segments. It defines whether two segments
are regarded as ident and guarantees that the error does not
```

```

become larger then 1/1000 of the measured area deviation. *)
scale = Abs[Min[kp1[[1, 1]], kp2[[1, 1]]]] * 0.0002;

(* Detecting of applicable segments, which can intersect *)
Do[
  While[PS2[[ObPkt]] - PS1[[i + 1]] < 0, ObPkt++];
  UntPkt = ObPkt - 1;
  While[PS2[[UntPkt]] - PS1[[i]] > 0, UntPkt--];
  Do[

    (* Calculation of the coefficients of both lines *)
    A1 = kp1[[i, 2]] - kp1[[i + 1, 2]];
    A2 = kp2[[j, 2]] - kp2[[j + 1, 2]];
    B1 = kp1[[i + 1, 1]] - kp1[[i, 1]];
    B2 = kp2[[j + 1, 1]] - kp2[[j, 1]];
    C1 = kp1[[i + 1, 2]] * kp1[[i, 1]] - kp1[[i, 2]] * kp1[[i + 1, 1]];
    C2 = kp2[[j + 1, 2]] * kp2[[j, 1]] - kp2[[j, 2]] * kp2[[j + 1, 1]];

    (* Testing whether two lines are parallel *)
    ParTest = Det[{{A1, B1}, {A2, B2}}];
    If[Abs[ParTest] < 10-4,

      (* Testing whether two lines are ident *)
      If[Abs[C1 - C2] ≤ scale,
        If[Abs[B1] ≤ 0.01,
          p1 = kp1[[i, 2]];
          p2 = kp2[[j, 2]];
          q1 = kp1[[i + 1, 2]];
          q2 = kp2[[j + 1, 2]],
          p1 = kp1[[i, 1]];
          p2 = kp2[[j, 1]];
          q1 = kp1[[i + 1, 1]];
          q2 = kp2[[j + 1, 1]]];

        (* If the lines are ident,
        the first common point is set as point of intersection *)
        If[p1 < q1,
          If[p1 < p2, {xs, ys} = kp2[[j]], {xs, ys} = kp1[[i]]],
          If[p1 > p2, {xs, ys} = kp2[[j]], {xs, ys} = kp1[[i]]]];

      (* Generating a closed polygon and computing its area *)
      For[c = Hilfsi, c < i, AppendTo[Polygon, kp1[[c]]], c++];
      AppendTo[Polygon, {xs, ys}];
      For[c = j + 1, c > Hilfsj + 1, AppendTo[Polygon, kp2[[c]]], c--];
      F1 = F1 + PolygonF1[Polygon];
      Polygon = {{xs, ys}};
      Hilfsi = i;
      Hilfsj = j],]

      (* Computing the point of intersection *)
      xs = Det[{{B1, C1}, {B2, C2}}] / ParTest;
      ys = Det[{{C1, A1}, {C2, A2}}] / ParTest;

      (* Testing whether the
      achieved point of intersection lies within the segments *)
    ]
  ]
]

```

```

PktStg = If[Abs[ys] < 10-6, If[xs < 0, -1000, 1000], xs/ys];
If[PS1[[i]] ≤ PktStg &&
 PS1[[i + 1]] ≥ PktStg && PS2[[j]] ≤ PktStg && PS2[[j + 1]] ≥ PktStg,

    (* Generating a closed polygon and computing its area *)
    For[c = Hilfsi, c < i, AppendTo[Polygon, kp1[[c]]], c++];
    AppendTo[Polygon, {xs, ys}];
    For[c = j + 1, c > Hilfsj + 1, AppendTo[Polygon, kp2[[c]]], c--];
    F1 = F1 + PolygonFl[Polygon];
    Polygon = {{xs, ys}};
    Hilfsi = i;
    Hilfsj = j],
    {j, UntPkt, ObPkt - 1}],
{i, 1, s}];

    (* Generating the last closed polygon and computing its area *)
For[c = Hilfsi, c < s, AppendTo[Polygon, kp1[[c]]], c++];
AppendTo[Polygon, kp1[[s + 1]]];
AppendTo[Polygon, kp2[[s + 1]]];
For[c = s + 1, c > Hilfsj + 1, AppendTo[Polygon, kp2[[c]]], c--];
F1 = F1 + PolygonFl[Polygon],
F1 = -1000];
F1]

```

*Diff(s,kp1,kp2)* evaluates a B-spline curve with its control points, forms an approximating polygon using equally spaced parameter values and sends this polygon to the module *areadiff(kp1,kp2)*.

```

Diff[s_, kp1_, kp2_] :=
Module[{i, sp1, sp2, t}, sp1 = Table[B2OSpline[kp1, t], {t, 0, 1, 1/s}];
sp2 = Table[B2OSpline[kp2, t], {t, 0, 1, 1/s}]; areadiff[sp1, sp2]]

```

*Diffkrebs(s,krebs1,krebs2)* transforms the control points of both halves in such a way, that the cotangent values of the approximating polygons are in ascending order. This function also prints the error message if the conformity test fails.

```

Diffkrebs[s_, krebs1_, krebs2_] := Module[{krebs1polygon, krebs2polygon, d1, d2},
krebs1polygon = {Reverse[krebs1[[1]]], Map[{#[[1]], #[[2]]*(-1)} &, krebs1[[2]]]};
krebs2polygon = {Reverse[krebs2[[1]]], Map[{#[[1]], #[[2]]*(-1)} &, krebs2[[2]]]};
d1 = Diff[s, krebs1polygon[[1]], krebs2polygon[[1]]];
d2 = Diff[s, krebs1polygon[[2]], krebs2polygon[[2]]];
If[d1 < 0 || d2 < 0, "Daten nicht konform", (d1 + d2) / 1000]]

```

Importing of all tps-data of a subdirectory and launching of the approximating routines.

```

alledateien = FileNames["*.tps"];
filesapp = Map[krebseinlesenundapproximieren[#, 6, 6] &, alledateien];

```

*DiffMatrix* provides a matrix with the names of the tps-files in the first row/column and the area deviation in the concerning cell.

```

DiffMatrix =
Module[{M = Table[0, {Length[filesapp] + 1}, {Length[filesapp] + 1}], i, k},
  For[i = 2, i < Length[M], For[k = i + 1, k < Length[M],
    M[[k, i]] = Diffkrebs[50, filesapp[[k - 1]], filesapp[[i - 1]]]; k++; i++];
  For[i = 2, i < Length[M], For[k = i, k < Length[M], M[[i, k]] = M[[k, i]]; k++; i++];
  For[i = 2, i < Length[M], M[[i, 1]] = alledateien[[i - 1]];
  M[[1, i]] = alledateien[[i - 1]]; i++]; M];

MatrixForm[DiffMatrix]

0      Arc1.tps  Arc2.tps  Arc3.tps  Arc4.tps  Arc5.tps  Arc6.tps  Arc7.tps
Arc1.tps  0       45.6116  26.4441  31.558   26.0299  36.6316  24.0753
Arc2.tps  45.6116  0       62.6234  58.6141  52.9652  51.452   28.8497
Arc3.tps  26.4441  62.6234  0       14.3695  21.5412  36.6331  39.4138
Arc4.tps  31.558   58.6141  14.3695  0       19.8952  38.196   35.7911
Arc5.tps  26.0299  52.9652  21.5412  19.8952  0       37.5974  29.7787
Arc6.tps  36.6316  51.452   36.6331  38.196   37.5974  0       29.9872
Arc7.tps  24.0753  28.8497  39.4138  35.7911  29.7787  29.9872  0

```