In[2434]:=

```
(* The following is the Mathematica code that accompanies my paper
 "Separating Variables in Bivariate Polynomial Ideals: the Local Case". It
                 ⌊Variablen
 consists of two parts. The first part is the code Manuel wrote for the ISSAC
 paper "Separating Variables in Bivariate Polynomial Ideals". The second
                        ⌊Variablen
 is an implementation of the algorithms outlined in the recent paper. *)
```

In[2435]:=

```
(* first part... *)
```

In[2436]:=

```
(*started by MK 2019-12-30*)(**
 *Input:
  ⌊Eingabe
  *--ideal:a list of bivariate polynomials
   over QQ generating an ideal of dimension 0.*--x,
y:the variables with respect to which the polynomials are given*
  *Output:
 *--a list of generators of the the algebra of separated polynomials**)
Separate0D[ideal_List, x_, y_] :=
                ⌊Liste
 Module[{xpure, ypure, terms, rems, G, vars, a, mixed},
 ⌊Modul
  If[PolynomialGCD @@ ideal =!= 1, Throw["not zero dimensional"]];
  ⌊⋯  ⌊ggT von Polynomen                        ⌊wirf
  {xpure, ypure} =
   First[GroebnerBasis[ideal, {First[#]}, {Last[#]}]] & /@ {{x, y}, {y, x}};
   ⌊erstes⋯ ⌊Gröbnerbasis                  ⌊erstes Element  ⌊letztes Element
  If[xpure === 1, Return[{{1, 0}, {x, 0}, {0, 1}, {0, y}}]];
  ⌊wenn            ⌊gib zurück
  terms = Join[{0, #} & /@ Reverse[y^Range[0, Exponent[ypure, y] - 1]],
               ⌊verknüpfe         ⌊kehre um    ⌊Liste aufe⋯ ⌊Exponent
    {#, 0} & /@ Reverse[x^Range[0, Exponent[xpure, x] - 1]]];
                ⌊kehre um    ⌊Liste aufe⋯ ⌊Exponent
  vars = Array[a, Length[terms]]; G = GroebnerBasis[ideal, {x, y}];
         ⌊Array     ⌊Länge              ⌊Gröbnerbasis
  mixed =
   Transpose[terms].# & /@ NullSpace[Outer[Coefficient, Flatten[CoefficientList[
   ⌊transponiere              ⌊Nullraum   ⌊äußer⋯ ⌊Koeffizient  ⌊ebne ein  ⌊Liste der Koeffizienten
         (Last[PolynomialReduce[First[DeleteCases[{1, -1} * #, 0]], G, {x, y}]] & /@
          ⌊letzt⋯ ⌊reduziere Polynom   ⌊erstes⋯ ⌊lösche Fälle
            terms).vars, {x, y}]], vars]];
  Return[Join[mixed, {#, 0} & /@ (xpure x^Range[0, Exponent[xpure, x] - 1]),
  ⌊gib zur⋯ ⌊verknüpfe                        ⌊Liste aufe⋯ ⌊Exponent
    {0, #} & /@ (ypure y^Range[0, Exponent[ypure, y] - 1])]]]
                         ⌊Liste aufe⋯ ⌊Exponent


(**
 *Input:
  ⌊Eingabe
  *--A0:a list of generators of the algebra of
```

```
    separated polynomials of a bivariate zero-dimensional ideal,
as produced by the function Separate0D*--x,
y:the variables with respect to which the polynomials are given*
  *Output:
 *--a linear function which applied to a pair of
  polynomials produces a vector (a finite list of numbers)
  that is zero if and only if the input pair belongs to A0**)
MakeReductor[A0_, x_, y_] :=
  Module[{p, q, B, i, f}, p = First[Cases[A0, {p_, 0} ⧴ p]];
  ⌊Modul                                     ⌊erstes⋯ ⌊Fälle
   q = First[Cases[A0, {0, q_} ⧴ q]];
      ⌊erstes⋯ ⌊Fälle
   B = (# / Last[CoefficientList[First[#], x]]) & /@ Sort[DeleteCases[A0,
         ⌊letzt⋯ ⌊Liste der Koeffizienten   ⌊erstes Element          ⌊sorti⋯ ⌊lösche Fälle
       {_, 0} | {0, _}], (Exponent[First[#1], x] > Exponent[First[#2], x]) &];
                         ⌊Exponent   ⌊erstes Element      ⌊Exponent   ⌊erstes Element
   f[{u_, v_}] := Module[{u0, v0, t}, u0 = PolynomialRemainder[u, p, x];
                  ⌊Modul                  ⌊Rest von Polynomen
     v0 = PolynomialRemainder[v, q, y];
          ⌊Rest von Polynomen
     Do[{u0, v0} = Expand[{u0, v0} - Coefficient[u0, x, Exponent[First[b], x]] * b] +
     ⌊iteriere      ⌊multipliziere aus    ⌊Koeffizient            ⌊Exponent  ⌊erstes Element
        {t x ^ Exponent[First[b], x], 0}, {b, B}];
              ⌊Exponent   ⌊erstes Element
     Return[Join[DeleteCases[Most[CoefficientList[u0 + x ^ Exponent[p, x], x]], t],
     ⌊gib zur⋯ ⌊verk⋯ ⌊lösche Fälle ⌊alle ⋯ ⌊Liste der Koeffizienten  ⌊Exponent
       Most[CoefficientList[v0 + y ^ Exponent[q, y], y]]]];];
       ⌊alle ⋯ ⌊Liste der Koeffizienten   ⌊Exponent
   Return[f];];
   ⌊gib zurück


(**
  *Input:
   ⌊Eingabe
   *--poly:
   a univariate polynomial*--x:the variable in which the polynomial is stated*
       *Output:
      *--a positive integer n such that poly=const*Cyclotomic[n,x],
                                                     ⌊Kreisteilungspolynom
or-1 if no such n exists**)
CyclotomicQ[poly_, x_] := Module[{phin, bound, n}, phin = Exponent[poly, x];
                          ⌊Modul                           ⌊Exponent
   If[Coefficient[poly, x, phin] =!= 1,
   ⌊⋯ ⌊Koeffizient
    Return[CyclotomicQ[poly / Coefficient[poly, x, phin], x]]];
    ⌊gib zurück                ⌊Koeffizient
   If[Expand[poly - (x - 1)] === 0, Return[1]];
   ⌊⋯ ⌊multipliziere aus            ⌊gib zurück
   If[Coefficient[poly, x, 0] =!= 1, Return[-1]];
   ⌊⋯ ⌊Koeffizient                  ⌊gib zurück
   For[n = 0,
   ⌊For-Schleife
```

```
    n < 3 || n / (Exp[EulerGamma] Log[Log[n]] + 3 / Log[Log[n]]) ≤ phin, n += 1,
                 ⌊Ex··· ⌊eulersche Kon··· ⌊Lo··· ⌊Logarithmus    ⌊Lo··· ⌊Logarithmus
    If[EulerPhi[n] == phin && Expand[Cyclotomic[n, x] - poly] === 0, Return[n]]];
    ⌊··· ⌊eulersche Phi–Funktion    ⌊multipliz·· ⌊Kreisteilungspolynom          ⌊gib zurück
   Return[-1];];
   ⌊gib zurück


(**
 *Input:
  ⌊Eingabe
  *--f:a bivariate polynomial*--x,y:the two variables in which f is stated*
  *Output:
 *--a list of generators of the algebra of
  separated polynomials of the ideal generated by f**)
SeparatePrincipal[f_, x_, y_] :=
  Module[{h, t, n, alpha, index, a, b, i, j, sol, p, q, vars},
  ⌊Modul
   Which[Head[f] === List && Length[f] == 1,
   ⌊welches ⌊Kopf      ⌊Liste   ⌊Länge
    Return[SeparatePrincipal[First[f], x, y]], Expand[f] === 0, Return[{{1, 1}}],
    ⌊gib zurück                 ⌊erstes Element   ⌊multipliziere aus  ⌊gib zurück
    Expand[f] === 1, Return[{{1, 0}, {x, 0}, {0, 1}, {0, y}}], FreeQ[Expand[f], x],
    ⌊multipliziere aus ⌊gib zurück                              ⌊frei v··· ⌊multipliziere aus
    Return[Prepend[{0, y^# f} & /@ Range[0, Exponent[f, y] - 1], {1, 1}]],
    ⌊gib zur··· ⌊stelle voran            ⌊Liste aufe··· ⌊Exponent
    FreeQ[Expand[f], y],
    ⌊frei v··· ⌊multipliziere aus
    Return[Prepend[{x^# f, 0} & /@ Range[0, Exponent[f, x] - 1], {1, 1}]]];
    ⌊gib zur··· ⌊stelle voran            ⌊Liste aufe··· ⌊Exponent
   p = Exponent[f /. y → 0, x];
       ⌊Exponent
   q = Exponent[f /. x → 0, y];
       ⌊Exponent
   (*f contains x^p and y^q*)
   If[! IntegerQ[p] || ! IntegerQ[q], Return[{{1, 1}}]];
   ⌊wenn ⌊ganze Zahl?     ⌊ganze Zahl?    ⌊gib zurück
   (*f has x or y as factor*)n =
    Exponent[h = Expand[Last[CoefficientList[f /. {x → x t^q, y → y t^p}, t]]], x];
    ⌊Exponent      ⌊multipliz·· ⌊letzt··· ⌊Liste der Koeffizienten
   (*omega(x^i*y^j)=q*i+p*j*)If[Expand[(h /. y → 0) * (h /. x → 0)] === 0,
                               ⌊··· ⌊multipliziere aus
    Return[{{1, 1}}]];
    ⌊gib zurück
   (*nontrivial Newton polygon*)h = Expand[h / Coefficient[h, y, q]];
                                    ⌊multipliziere··· ⌊Koeffizient
   alpha = ToNumberField[Coefficient[h, x, p] ^ (1 / p)];
           ⌊als Zahlenfeld   ⌊Koeffizient
   (*normalize*)
   If[Length[Complement[Variables[h], {x, y}]] > 0, Return[{{1, 1}}]];
   ⌊··· ⌊Länge ⌊Komplement ⌊Variablen                  ⌊gib zurück
   (*no parameters beyond this point*)h = CyclotomicQ[#, x] & /@
     (MinimalPolynomial[Root[Function[x, First[#]], 1], x] &) /@ Select[
      ⌊Minimalpolynom    ⌊Nulls·· ⌊Funktion  ⌊erstes Element        ⌊wähle aus
```

```
      FactorList[h /. {x → x / alpha, y → 1}, Extension → alpha], ! FreeQ[#, x] &];
      Liste der Faktoren                           Erweiterung          frei von?
   If[MemberQ[h, -1], Return[{{1, 1}}]];
   ···  enthalten?          gib zurück
   sol = {Null, Null};
         Nulla··· Nullausdruck
   n = LCM @@ h + 1;
       kleinstes gemeinsames Vielfaches
   (*now n bounds the order of the roots of unity*)
   While[Length[sol] > 1, n -= 1;
   solange Länge
    vars = Join[Table[a[i], {i, 0, n}], Table[b[i], {i, 1, n * q / p}]];
           verk··· Tabelle                  Tabelle
    sol = Expand[Join[x ^ Range[0, n], y ^ Range[n * q / p]].#] & /@ NullSpace[
          multipliz·· verknüpfe Liste aufeinanderfo··· Liste aufeinanderfolgender Zah··· Nullraum
      Outer[Coefficient, Flatten[CoefficientList[PolynomialRemainder[vars.
      äußer··· Koeffizient   ebne ein   Liste der Koeffizienten   Rest von Polynomen
          Join[x ^ Range[0, n], y ^ Range[n * q / p]], f, x], {x, y}]], vars]];];
          verknüpfe Liste aufeinanderfo··· Liste aufeinanderfolgender Zahlen
   Return[Prepend[Expand[{# /. y → 0, (# /. y → 0) - #}] & /@ sol, {1, 1}]]];
   gib zur···  stelle voran multipliziere aus


 (**
  *Input:
   Eingabe
   *--ideal:a list of bivariate polynomials*--x,
 y:the two variables in which the ideal generators are stated*
   *Output:
  *--a list of generators of the algebra
   of separated polynomials of the input ideal**)
 Separate[ideal_, x_, y_] := Module[{id, A0, A1, f, d, a, t, g, G, Delta, S, s, p},
                                    Modul
   A1 = SeparatePrincipal[id[1] = PolynomialGCD @@ ideal, x, y];
                                  ggT von Polynomen
   A0 = Separate0D[id[0] = GroebnerBasis[Together[ideal / id[1]], {x, y}], x, y];
                           Gröbnerbasis  zusammen
   Which[id[1] === 1, Return[A0], id[0] === {1} || A1 === {{1, 1}},
   welches             gib zurück
    Return[A1], FreeQ[A1, y], p = First[GroebnerBasis[ideal, {x}, {y}]];
    gib zurück  frei von?         erstes··· Gröbnerbasis
    Return[{x ^ # p, 0} & /@ Range[0, Exponent[p, x] - 1]],
    gib zurück               Liste aufe··· Exponent
    FreeQ[A1, x], p = First[GroebnerBasis[ideal, {y}, {x}]];
    frei von?         erstes··· Gröbnerbasis
    Return[{0, y ^ # p} & /@ Range[0, Exponent[p, y] - 1]]];
    gib zurück               Liste aufe··· Exponent
   f = MakeReductor[A0, x, y];
   d = Length[f[{1, 1}]];
       Länge
   a = A1[[2]];
   G = {{1, 1}};
   g = 0;
```

```
Delta = {};
While[True, Which[Length[Delta] == 0, S = Range[d + 1], Length[Delta] == 1,
solange wahr welches Länge Liste aufeinande·· Länge
  S = Complement[Range[g * (d + 1)], g * Range[d + 1]], g ≠ 1,
  Komplement Liste aufeinanderfolgender ··· Liste aufeinanderfolgender Zahlen
  S = Select[Range[g * (d + 1)], Length[FrobeniusSolve[Delta, #, 1]] === 0 &],
  wähle aus Liste aufeinanderfolgen··· Länge löse Frobeniusgleichung
  True, S = Select[Range[FrobeniusNumber[Delta]],
  wahr wähle aus Liste a·· Frobenius-Zahl
    Length[FrobeniusSolve[Delta, #, 1]] === 0 &];];
    Länge löse Frobeniusgleichung
 If[Length[S] > d, S = Take[S, d + 1]];
 ··· Länge nimm
 If[Length[S] == 0, Break[]];
 ··· Länge beende Schleife
 p = NullSpace[Transpose[Table[f[a^s], {s, S}]]];
     Nullraum transponiere Tabelle
 If[Length[p] == 0, Break[], p = (# * LCM @@ (Denominator /@ #)) &[Last[p]]];
 ··· Länge beende Schleife kleinste·· Nenner letztes Element
 AppendTo[G, Expand[Sum[p[[i]] a^S[[i]], {i, 1, Length[S]}]]];
 hänge an bei multipliz·· summiere Länge
 AppendTo[Delta, Exponent[p.(t^S), t]];
 hänge an bei Exponent
 g = GCD[g, Last[Delta]];];
     größter·· letztes Element
Return[G];];
gib zurück

(*the functions below are test functions*)

CheckSeparate0D[ideal_List, x_, y_] :=
                       Liste
 Module[{A0, G, u, v, a, b, dx}, A0 = Separate0D[ideal, x, y];
 Modul
  G = GroebnerBasis[ideal, {x, y}];
      Gröbnerbasis
  If[! MatchQ[Last[PolynomialReduce[#, G, {x, y}]] & /@ (A0 /. {u_, v_} :> u - v),
  wenn übereins· letzt··· reduziere Polynom
     {0 ..}], Throw["incorrect!"];];];
              wirf

CheckSeparatePrincipal[f_, x_, y_] :=
 Module[{A1, dx, dy, a, b}, A1 = SeparatePrincipal[f, x, y];
 Modul
  If[
  wenn
   Length[A1] > 1 && ! FreeQ[Denominator[Together[(A1[[2, 1]] - A1[[2, 2]]) / f]], x | y],
   Länge frei v··· Nenner zusammen
   Throw["incorrect!"];];
   wirf
  dx = If[Length[A1] == 1, 10, Exponent[A1[[2, 1]], x] - 1];
       ··· Länge Exponent
  dy = If[Length[A1] == 1, 10, Exponent[A1[[2, 2]], y] - 1];
       ··· Länge Exponent
```

```
   If[Length[
   ⌊⋯ ⌊Länge
      DeleteCases[First[Solve[Flatten[CoefficientList[Last[PolynomialReduce[
      ⌊lösche Fälle ⌊erstes⋯ ⌊löse ⌊ebne ein ⌊Liste der Koeffizienten ⌊letzt⋯ ⌊reduziere Polynom
            Sum[a[i] x^i, {i, 0, dx}] + Sum[b[i] y^i, {i, 1, dy}], {f}, {x, y}]],
            ⌊summiere                   ⌊summiere
         {x, y}]] == 0]], _ → 0]] > 0, Throw["incomplete!"];];];
                                       ⌊wirf


CheckSeparate[ideal_, x_, y_] :=
  Module[{A, dx, dy, u, v, G, a, b, f}, A = Separate[ideal, x, y];
  ⌊Modul
   G = GroebnerBasis[ideal, {x, y}];
       ⌊Gröbnerbasis
   If[! MatchQ[Last[PolynomialReduce[#, G, {x, y}]] &/@ (A /. {u_, v_} :→ u - v),
   ⌊wenn ⌊übereins· ⌊letzt⋯ ⌊reduziere Polynom
      {0 ..}], Throw["incorrect!"];];
               ⌊wirf
   dx = If[Length[A] == 1, 10, Exponent[A⟦2, 1⟧, x] - 1];
        ⌊⋯ ⌊Länge              ⌊Exponent
   dy = If[Length[A] == 1, 10, Exponent[A⟦2, 2⟧, y] - 1];
        ⌊⋯ ⌊Länge              ⌊Exponent
   If[Length[
   ⌊⋯ ⌊Länge
      DeleteCases[First[Solve[Flatten[CoefficientList[Last[PolynomialReduce[
      ⌊lösche Fälle ⌊erstes⋯ ⌊löse ⌊ebne ein ⌊Liste der Koeffizienten ⌊letzt⋯ ⌊reduziere Polynom
            Sum[a[i] x^i, {i, 0, dx}] + Sum[b[i] y^i, {i, 1, dy}], {f}, {x, y}]],
            ⌊summiere                   ⌊summiere
         {x, y}]] == 0]], _ → 0]] > 0, Throw["incomplete!"];];];
                                       ⌊wirf
```

In[2444]:=

```
(* second part... *)
```

In[2445]:=

```
(* the following is an implementation of the algorithms outlined in
 "Separating Variables in Bivariate Polynomial Ideals: the Local Case" *)
               ⌊Variablen
```

In[2446]:=

```
(* takes a polynomial and the set of its variables,
and computes its support *)
Support[poly_, vars_] := Module[ {},
                         ⌊Modul
  If[ Together[poly] === 0, Return[ {} ]];
  ⌊w⋯ ⌊zusammen              ⌊gib zurück
  Exponent[#, vars] &/@ MonomialList[poly, vars]
  ⌊Exponent             ⌊Liste der Monome
 ]
```

In[2447]:=

```
(* takes a polynomial and the set of its variables,
and computes the vertices of its Newton polytope *)
NewtonPolytope[poly_, vars_] :=
  Module[ {OutsideQ, points, p, P, x, CornerQ, e},
  Modul
 OutsideQ[p_, P_] := Module[ {v},
                     Modul
  v = Array[x, Length[P]];
      Array     Länge
     ! Resolve[e[v,
      löse auf
   And @@ Join[Thread[p == v.P], Thread[v ≥ 0], {Plus @@ v == 1}]] /.
   und    verk… fädle auf            fädle auf         addiere
        e → Exists, Reals]];
            existiert  Menge reeller Zahlen
  points = Support[poly, vars];
  Select[points, OutsideQ[#, DeleteCases[points, #]] &]
  wähle aus                  lösche Fälle
 ];
```

In[2448]:=

```
(* takes the vertices of a polytope and computes its edges *)
GetEdges[polytope_] :=
  Module[ {InnerQ, vertex, edges, possibleEdges, vector, vectors, p, e, E,
  ⌊Modul                                                               ⌊Exponentialko
 x, ex},
 InnerQ[e_, E_] := Module[ {v = Array[x, Length[E]]},
                   ⌊Modul     ⌊Array   ⌊Länge    ⌊Exponentialkonstante E
  Resolve[ex[v, And @@ Join[Thread[e == v.E], Thread[v ≥ 0]]] /.
  ⌊löse auf     ⌊und    ⌊verk… ⌊fädle auf    ⌊Ex… ⌊fädle auf
   ex → Exists, Reals] ];
        ⌊existiert ⌊Menge reeller Zahlen
 edges = {};
 Do[
 ⌊iteriere
  possibleEdges = {vertex, #} & /@ Complement[polytope, {vertex}];
                                   ⌊Komplement
  Do[
  ⌊iteriere
   vector = edge[[2]] - edge[[1]];
   If[
   ⌊wenn
    InnerQ[vector,
     Complement[(#[[2]] - #[[1]]) & /@ possibleEdges, {vector}]],
     ⌊Komplement
    possibleEdges = Complement[possibleEdges, {edge}]
                    ⌊Komplement
    ]
   , {edge, possibleEdges}];

  edges = Join[edges, possibleEdges]
          ⌊verknüpfe
  , {vertex, polytope}];
 Return[Union[Sort /@ edges]]
 ⌊gib zur… ⌊Verein… ⌊sortiere
  ];
```

In[2449]:=

```
(* takes the vertices of a polygon and one of its edges,
and outputs an outward pointing normal for it *)
OuterNormal[polygon_, edge_] := Module[{v, w},
                                ⌊Modul
   v = edge[[2]] - edge[[1]];
   w = {-v[[2]], v[[1]]};
   If[Max[(w.#) & /@ (# - edge[[1]] & /@ polygon)] ≤ 0, Return[w], Return[-w]]
   ⌊… ⌊größtes Element                                  ⌊gib zurück ⌊gib zurück
  ]
```

In[2450]:=

```
(* some of the singularities of a separated multiple can be read off from
 some of the outward pointing normals of the edges of the Newton polygon,
the following function identifies them *)
validNormals[vectorSet_] := Module[{output, signVectors},
                                   Modul
  output = {};
  If[Length[vectorSet] == 1,
  ··· Länge
   If[Sign[vectorSet[[1]][[1]]] == -1, Return[-vectorSet]]];
   ··· Vorzeichen                     gib zurück
  signVectors = Sign[vectorSet];
                Vorzeichen

  output = Join[output,
                verknüpfe
    Select[vectorSet, Sign[#] == {1, 0} || Sign[#] == {1, 1} || Sign[#] == {1, -1} &]];
    wähle aus         Vorzeichen           Vorzeichen           Vorzeichen

  If[MemberQ[signVectors, {1, 1}],
  ··· enthalten?
   output =
    Join[output, Select[vectorSet, Sign[#] == {0, 1} || Sign[#] == {-1, 1} &]]];
    verknüpfe    wähle aus         Vorzeichen           Vorzeichen

  If[MemberQ[signVectors, {1, -1}],
  ··· enthalten?
   output =
    Join[output, Select[vectorSet, Sign[#] == {0, -1} || Sign[#] == {-1, -1} &]]];
    verknüpfe    wähle aus         Vorzeichen            Vorzeichen

  If[MemberQ[Sign /@ output, {-1, 1}] || MemberQ[Sign /@ output, {-1, -1}],
  ··· enthalten? Vorzeichen               enthalten? Vorzeichen
   output = Join[output, Select[vectorSet,
                 verknüpfe     wähle aus
      Sign[#] == {-1, 0} || Sign[#] == {-1, 1} || Sign[#] == {-1, -1} &]]];
      Vorzeichen            Vorzeichen            Vorzeichen
  Return[Union[output]]
  gib zur··· Vereinigung
 ]
```

In[2451]:=

```
(* takes a polynomial and the set of its variables,
and outputs its outward pointing normals *)
GetAllWeights[poly_, vars_] := Module[{polytope, edges},
                                      Modul
  polytope = NewtonPolytope[poly, vars];
  edges = GetEdges[polytope];
  OuterNormal[polytope, #] & /@ edges
 ]
```

In[2452]:=

```
(* takes a polynomial and the set of its variables,
and outputs its valid outward pointing normals *)
GetWeights[poly_, vars_] := Module[{polytope, edges},
                            ⌊Modul
  polytope = NewtonPolytope[poly, vars];
  edges = GetEdges[polytope];
  validNormals[OuterNormal[polytope, #] & /@ edges]
 ]
```

In[2453]:=

```
(* computes the weight of a polynomial with respect to a weight function *)
Weight[poly_, vars_, w_] := Module[{},
                            ⌊Modul
  Max[w.# & /@ Support[poly, vars]]
  ⌊größtes Element
 ]
```

In[2454]:=

```
(* computes the leading part of the transformation
 of a polynomial with respect to a weight function that
 matches a given pair of (potential) singularities *)
LeadingPart[poly_, vars_, singPair_] :=
 Module[{p, signVector, weightVectors, vector, weight, list},
 ⌊Modul
  p = poly;
  If[singPair[[1]] ≠ Infinity, p = p /. vars[[1]] → vars[[1]] + singPair[[1]]];
  ⌊wenn            ⌊Unendlichkeit
  If[singPair[[2]] ≠ Infinity, p = p /. vars[[2]] → vars[[2]] + singPair[[2]]];
  ⌊wenn            ⌊Unendlichkeit
  p = Expand[FullSimplify[p]];
     ⌊multipliz⋯ ⌊vereinfache vollständig
  signVector =
   {If[singPair[[1]] == Infinity, 1, -1], If[singPair[[2]] == Infinity, 1, -1]};
   ⌊wenn            ⌊Unendlichkeit      ⌊wenn            ⌊Unendlichkeit
  weightVectors = GetAllWeights[p, vars];

  vector = If[Length[weightVectors] == 1, weightVectors[[1]],
           ⌊⋯ ⌊Länge
    Select[weightVectors, Sign[#] == signVector &][[1]];
    ⌊wähle aus                ⌊Vorzeichen
  weight = Weight[p, vars, vector];

  list = MonomialList[p, vars];
         ⌊Liste der Monome
  Return[Plus @@ Select[list, Exponent[#, vars].vector == weight &]]
  ⌊gib zur⋯ ⌊addiere ⌊wähle aus   ⌊Exponent
 ]
```

In[2455]:=

```
(* computes the leading part of a polynomial
 with respect to a given weight function *)
LeadingPartW[poly_, vars_, w_] := Module[{weight, list},
                                  Modul
  weight = Weight[poly, vars, w];

  list = MonomialList[poly, vars];
          Liste der Monome
  Return[Plus @@ Select[list, Exponent[#, vars].w == weight &]]
  gib zur···  addiere  wähle aus   Exponent
 ]
```

In[2456]:=

```
(* computes a set of pairs of (potential) singularities that can be computed
 from the leading part of poly with respect to a given weight function *)
GetSing[poly_, vars_, weight_] := Module[{lp, sing, factor, output},
                                  Modul

  output = {};

  If[weight⟦1⟧ * weight⟦2⟧ ≠ 0,
  wenn
   output = {{If[weight⟦1⟧ > 0, Infinity, 0], If[weight⟦2⟧ > 0, Infinity, 0]}};
              wenn                Unendlichkeit   wenn                 Unendlichkeit
   Return[output]];
   gib zurück

  If[weight⟦1⟧ == 0,
  wenn
   lp = LeadingPartW[poly, vars, weight];
   factor = Times @@
            multipliziere
     (#⟦1⟧ & /@ Select[FactorList[lp], ! FreeQ[#, vars⟦1⟧] && #⟦1⟧ =!= vars⟦1⟧ &]);
                wähle aus Liste der Faktoren   frei von?
   sing = vars⟦1⟧ /. Solve[factor == 0, vars⟦1⟧];
                     löse
   Return[{#, If[weight⟦2⟧ > 0, Infinity, 0]} & /@ (FullSimplify /@ sing)]
   gib zurück  wenn               Unendlichkeit          vereinfache vollständig
  ];

  If[weight⟦2⟧ == 0,
  wenn
   lp = LeadingPartW[poly, vars, weight];
   factor = Times @@
            multipliziere
     (#⟦1⟧ & /@ Select[FactorList[lp], ! FreeQ[#, vars⟦2⟧] && #⟦1⟧ =!= vars⟦2⟧ &]);
                wähle aus Liste der Faktoren   frei von?
   sing = vars⟦2⟧ /. Solve[factor == 0, vars⟦2⟧];
                     löse
   Return[{If[weight⟦1⟧ > 0, Infinity, 0], #} & /@ (FullSimplify /@ sing)]
   gib zurück wenn              Unendlichkeit            vereinfache vollständig
  ];
 ]
```

In[2457]:=

```
(* takes a pair of (potential) singularities different from 0 and infinity,
performs a subsitution of variables, and looks for further singularities *)
GetFurtherSing[poly_, vars_, singPair_] :=
 Module[{p, polytope, edges, normals, weights, sings},
 Modul
  p = poly;
  If[singPair[[1]] ≠ Infinity, p = p /. vars[[1]] → vars[[1]] + singPair[[1]]];
  wenn                Unendlichkeit
  If[singPair[[2]] ≠ Infinity, p = p /. vars[[2]] → vars[[2]] + singPair[[2]]];
  wenn                Unendlichkeit
  p = Expand[FullSimplify[p]];
      multipliz·· vereinfache vollständig
  polytope = NewtonPolytope[p, vars];
  edges = GetEdges[polytope];
  normals = OuterNormal[polytope, #] & /@ edges;
  weights = validNormals[normals];
  If[singPair[[1]] ≠ Infinity && MemberQ[Sign[normals], {-1, 0}],
  wenn                Unendlichkeit   enthalten? Vorzeichen
   AppendTo[weights, {-1, 0}]];
   hänge an bei
  If[singPair[[2]] ≠ Infinity && MemberQ[Sign[normals], {0, -1}],
  wenn                Unendlichkeit   enthalten? Vorzeichen
   AppendTo[weights, {0, -1}]];
   hänge an bei

  sings = Flatten[GetSing[p, vars, #] & /@ weights, 1];
          ebne ein
  If[singPair[[1]] ≠ Infinity, sings = {singPair[[1]], 0} + # & /@ sings];
  wenn                Unendlichkeit
  If[singPair[[2]] ≠ Infinity, sings = {0, singPair[[2]]} + # & /@ sings];
  wenn                Unendlichkeit
  Return[FullSimplify /@ sings]
  gib zur··· vereinfache vollständig
 ]
```

In[2458]:=

```
(* function that computes the set of all pairs of potential singularities *)

getSing[poly_, vars_] := Module[{sing, output, outputNew,
                         Modul
   degX, degY, poly1, singF, singFnew, solF, singG, singGnew, solG},

  output = {};

  singF = {};
  singFnew = {Infinity};
             Unendlichkeit
  singG = {};
  singGnew = {};
```

```
degX = Exponent[poly, vars[[1]]];
degY = Exponent[poly, vars[[2]]];

While[True,
  (* finde Singularitäten von g *)
  Do[
   sing = {};
   If[s == Infinity,
    poly1 = Coefficient[poly, vars[[1]], degX],
    poly1 = Collect[Expand[poly /. vars[[1]] → s], vars[[2]], Simplify];
    (*poly1=Collect[Expand[poly/.vars[[1]]→s],vars[[2]],Simplify]*)
   ];
   If[Exponent[poly1, vars[[2]]] < degY,
    singGnew = Union[singGnew, {Infinity}];
    sing = {Infinity};
   ];

   If[Exponent[poly1, vars[[2]]] > 0,
    sing = Join[sing, Union[vars[[2]] /. Solve[poly1 == 0, vars[[2]]]]];
    singGnew = Union[singGnew, sing]
   ];
   output = Join[output, {s, #} & /@ Complement[sing, singG]];
   , {s, Complement[singFnew, singF]}];

  If[SubsetQ[singG, singGnew], Return[Union[output]]];
  singF = Union[singF, singFnew];
  singFnew = {};
  (* finde Singularitäten von f *)
  Do[
   sing = {};
   If[s == Infinity,
    poly1 = Coefficient[poly, vars[[2]], degY],
```

```
      poly1 = Collect[Expand[poly /. vars⟦2⟧ → s], vars⟦1⟧, Simplify];
              ⌊gruppiere·· ⌊multipliziere aus                 ⌊vereinfache
     ];

     If[Exponent[poly1, vars⟦1⟧] < degX,
     ⌊··· ⌊Exponent
      singFnew = Union[singFnew, {Infinity}]];
                 ⌊Vereinigung     ⌊Unendlichkeit
     If[Exponent[poly1, vars⟦1⟧] > 0,
     ⌊··· ⌊Exponent
      sing = Join[sing, Union[vars⟦1⟧ /. Solve[poly1 == 0, vars⟦1⟧]]]];
             ⌊verknüpfe ⌊Vereinigung      ⌊löse
      singFnew = Join[singFnew, sing];
                 ⌊verknüpfe
      output = Join[output, {#, s} & /@ Complement[sing, singF]];
               ⌊verknüpfe                ⌊Komplement
     ];
     , {s, Complement[singGnew, singG]}];
           ⌊Komplement
    If[SubsetQ[singF, singFnew], Return[Union[output]]];
    ⌊··· ⌊Teilmenge?             ⌊gib zur··· ⌊Vereinigung
    singG = Union[singG, singGnew];
            ⌊Vereinigung
    singGnew = {};
    (*Print[{singF,singG}]*)
      ⌊gib aus
   ]
  ]
 (*GetAllSing[poly_,vars_]:=
  Module[{polytope,edges,normals,weights,sings,singsUpdate},
  ⌊Modul
   polytope=NewtonPolytope[poly,vars];
   edges=GetEdges[polytope];
   normals=OuterNormal[polytope,#]&/@edges;
   weights=validNormals[normals];
   sings=Flatten[GetSing[poly,vars,#]&/@weights,1];
         ⌊ebne ein
   singsUpdate=sings;
   Do[
   ⌊iteriere
    If[pair≠{Infinity,Infinity},
    ⌊wenn   ⌊Unendlichk·· ⌊Unendlichkeit
     singsUpdate=Union[singsUpdate,GetFurtherSing[poly,vars,pair]]]
                 ⌊Vereinigung
    ,{pair,sings}];

   If[Union[sings]==Union[singsUpdate],Return[singsUpdate]];
   ⌊··· ⌊Vereinigung  ⌊Vereinigung       ⌊gib zurück

   While[sings≠singsUpdate,
   ⌊solange
```

```
   singsUpdate=singsUpdate;
   Do[
     If[pair≠{Infinity,Infinity},
       singsUpdate=Union[singsUpdate,GetFurtherSing[poly,vars,pair]]]
     ,{pair,sings}];

  ];

  Return[singsUpdate]
 ]*)
```

In[2459]:=

```
(* two functions, a projection, and a lift *)
projectionY[point_] := Module[{},
  Return[point[[2]]]
 ]
projectionX[point_] := Module[{},
  Return[point[[1]]]
 ]
```

In[2461]:=

```
liftY[poly_, vars_, coord_] := Module[{output, p},
  output = {};

  If[coord ≠ Infinity,
   p = poly /. vars[[2]] → coord;
   If[Exponent[p, vars[[1]]] < Exponent[poly, vars[[1]]],
    AppendTo[output, Infinity]],
   Join[output, vars[[1]] /. Solve[p == 0, vars[[1]]]]
  ];

  If[coord == Infinity,
   p = Coefficient[poly, vars[[2]], Exponent[poly, vars[[2]]]];
   If[Exponent[p, vars[[1]]] < Exponent[poly, vars[[1]]],
    output = {Infinity}];
```

```
      output = Join[output, vars[[1]] /. Solve[p == 0, vars[[1]]]]
               verknüpfe                 löse
    ];

    Return[{#, coord} & /@ output]
    gib zurück
  ]

liftX[poly_, vars_, coord_] := Module[{output, p},
                                Modul
    output = {};

    If[coord ≠ Infinity,
    wenn       Unendlichkeit
      p = poly /. vars[[1]] → coord;
      If[Exponent[p, vars[[2]]] < Exponent[poly, vars[[2]]],
      ⋯  Exponent                  Exponent
        AppendTo[output, Infinity]],
        hänge an bei     Unendlichkeit
      Join[output, vars[[2]] /. Solve[p == 0, vars[[2]]]]
      verknüpfe                 löse
    ];

    If[coord == Infinity,
    wenn        Unendlichkeit
      p = Coefficient[poly, vars[[1]], Exponent[poly, vars[[1]]]];
          Koeffizient                  Exponent
      If[Exponent[p, vars[[2]]] < Exponent[poly, vars[[2]]],
      ⋯  Exponent                  Exponent
        output = {Infinity}];
                  Unendlichkeit
      output = Join[output, vars[[2]] /. Solve[p == 0, vars[[2]]]]
               verknüpfe                 löse
    ];

    Return[{coord, #} & /@ output]
    gib zurück
  ]
```

In[2463]:=

```
projLiftY[poly_, vars_, point_] := Module[{},
                                   Modul
    Return[liftY[poly, vars, projectionY[point]]]
    gib zurück
  ]

projLiftX[poly_, vars_, point_] := Module[{},
                                   Modul
    Return[liftX[poly, vars, projectionX[point]]]
    gib zurück
  ]
```

In[2475]:=

```
(* function for computing the x-orbit of a point *)
```

```
getOrbit[poly_, vars_, point_] :=
 Module[{sing, output, outputNew, degX, degY, poly1,
 ⌊Modul
   singF, singFnew, solF, singG, singGnew, solG},

  output = {point};

  singG = {};
  singGnew = {point⟦2⟧};
  singF = {point⟦1⟧};
  singFnew = {};

  degX = Exponent[poly, vars⟦1⟧];
         ⌊Exponent
  degY = Exponent[poly, vars⟦2⟧];
         ⌊Exponent

  While[True,
  ⌊solange ⌊wahr
   (* find x-coordinates *)
   Do[
   ⌊iteriere
    sing = {};
    If[s == Infinity,
    ⌊wenn  ⌊Unendlichkeit
     poly1 = Coefficient[poly, vars⟦2⟧, degY],
             ⌊Koeffizient
     poly1 = Collect[Expand[poly /. vars⟦2⟧ → s], vars⟦1⟧, Simplify]
             ⌊gruppiere·· ⌊multipliziere aus                      ⌊vereinfache
    ];

    If[Exponent[poly1, vars⟦1⟧] < degX,
    ⌊···  ⌊Exponent
     singFnew = Union[singFnew, {Infinity}]];
                ⌊Vereinigung      ⌊Unendlichkeit
    If[Exponent[poly1, vars⟦1⟧] > 0,
    ⌊···  ⌊Exponent
     sing = Join[sing, Union[vars⟦1⟧ /. Solve[poly1 == 0, vars⟦1⟧]]]];
            ⌊verknüpfe  ⌊Vereinigung       ⌊löse
     singFnew = Join[singFnew, sing];
                ⌊verknüpfe
    ];
    output = Union[output, {#, s} & /@ Complement[singFnew, singF]];
             ⌊Vereinigung              ⌊Komplement
    , {s, Complement[singGnew, singG]}];
          ⌊Komplement

   singG = Union[singG, singGnew];
           ⌊Vereinigung

   (* find y-coordinates *)
```

```
  Do[
  iteriere
   sing = {};
   If[s == Infinity,
   wenn   Unendlichkeit
    poly1 = Coefficient[poly, vars⟦1⟧, degX],
            Koeffizient
    poly1 = Collect[Expand[poly /. vars⟦1⟧ → s], vars⟦2⟧, Simplify]
            gruppiere·· multipliziere aus                  vereinfache
   ];
   If[Exponent[poly1, vars⟦2⟧] < degY,
   ···  Exponent
    singGnew = Union[singGnew, {Infinity}];
               Vereinigung      Unendlichkeit
    sing = {Infinity};
            Unendlichkeit
   ];

   If[Exponent[poly1, vars⟦2⟧] > 0,
   ···  Exponent
    sing = Join[sing, Union[vars⟦2⟧ /. Solve[poly1 == 0, vars⟦2⟧]]];
           verknüpfe  Vereinigung         löse
    singGnew = Union[singGnew, sing]
               Vereinigung
   ];
   output = Join[output, {s, #} & /@ Complement[sing, singG]];
            verknüpfe                Komplement
   , {s, Complement[singFnew, singF]}];
         Komplement
  If[SubsetQ[singG, singGnew], Return[Union[output]]];
  ···  Teilmenge?              gib zur··· Vereinigung
  singF = Union[singF, singFnew];
          Vereinigung
  singFnew = {};
 ]
]
```

In[2476]:=

```
getSingMult[poly_, vars_] := Module[{p, polytope, edges, normals,
                             Modul
   weights, sings, singsUpdate, singsMult, pairMod, newSings},
  sings = getSing[poly, vars];
  singsMult = {};
  (* for each pair of singularities compute a leading part whose
   separated multiple gives information about its multiplicities *)
  Do[
  iteriere
   p = poly;
   If[pair[[1]] ≠ Infinity, p = p /. vars[[1]] → vars[[1]] + pair[[1]]];
   wenn        Unendlichkeit
   If[pair[[2]] ≠ Infinity, p = p /. vars[[2]] → vars[[2]] + pair[[2]]];
   wenn        Unendlichkeit
   p = Expand[FullSimplify[p]];
       multipliz·· vereinfache vollständig
   pairMod =
    {If[pair[[1]] ≠ Infinity, 0, Infinity], If[pair[[2]] ≠ Infinity, 0, Infinity]};
     wenn          Unendlichkeit    Unendlichkeit  wenn         Unendlichkeit     Unendlichkeit
   AppendTo[singsMult, {pair, LeadingPart[p, vars, pairMod]}];
   hänge an bei
   , {pair, sings}];
  Return[singsMult]
  gib zurück
 ]
```

In[2477]:=

```
(* modifies a homogeneous polynomial and separates the variables *)
nearSep[pairSing_, poly_, vars_] := Module[{v, p, d1, d2},
                                    Modul
  v = {If[pairSing[[1]] == Infinity, 1, -1], If[pairSing[[2]] == Infinity, 1, -1]};
       wenn             Unendlichkeit          wenn            Unendlichkeit
  p = Times @@ (#[[1]] ^ #[[2]]  & /@
      multipliziere
      Select[FactorList[poly], ! FreeQ[#, vars[[1]]] && ! FreeQ[#, vars[[2]]] &]);
      wähle aus Liste der Faktoren   frei von?                   frei von?
  If[v[[1]] < 0, d1 = Exponent[p, vars[[1]]];
  wenn                 Exponent
   p = Expand[p / vars[[1]] ^ d1];
       multipliziere aus
   p = p /. vars[[1]] → 1 / vars[[1]]];
  If[v[[2]] < 0, d2 = Exponent[p, vars[[2]]];
  wenn                 Exponent
   p = Expand[p / vars[[2]] ^ d2];
       multipliziere aus
   p = p /. vars[[2]] → 1 / vars[[2]]];
  p = Expand[FullSimplify[p]];
      multipliz·· vereinfache vollständig
  Return[Separate[{p}, vars[[1]], vars[[2]]]]
  gib zurück
 ]
```

In[2478]:=

```
(* takes a list of pairs of pairs of singularities and pairs of homogeneous
 polynomials (the leading parts of some polynomial and its substitutions),
and outputs their multiplicities *)
mult[pairSing_, vars_] := Module[{list, sep},
                          ⌊Modul
  list = {};
  Do[
 ⌊iteriere
   sep = nearSep[pair〚1〛, pair〚2〛, vars];
   If[Length[sep] == 1,
  ⌊⋯  ⌊Länge
    Throw["not separable, since a leading part is not separable"]];
   ⌊wirf
   AppendTo[list,
  ⌊hänge an bei
    {pair〚1〛, {Exponent[sep〚2〛〚1〛, vars〚1〛], Exponent[sep〚2〛〚2〛, vars〚2〛]}}];
               ⌊Exponent                           ⌊Exponent
   , {pair, pairSing}];

  Return[list]
 ⌊gib zurück

 ]
```

In[2479]:=

```
(* takes a list of pairs,
the first component of which is a pair of singularities,
the second of which is a pair of positive integers,
it indicates that the vector of the multiplicities
 of the singularities is a multiple of the given vector *)
linSystem[list_, m_, n_, k_] :=
 Module[{f, equations, vars1, vars2, sol, solution, sol1, sol2},
 Modul
  f[_[x__]] := x;
  equations = {};
  Do[
  iteriere
   AppendTo[equations, m[list⟦i⟧⟦1, 1⟧] - k[i] × list⟦i⟧⟦2, 1⟧];
   hänge an bei
   AppendTo[equations, n[list⟦i⟧⟦1, 2⟧] - k[i] × list⟦i⟧⟦2, 2⟧];
   hänge an bei
   , {i, Length[list]}];
         Länge
  vars1 = Select[Variables[equations], ! FreeQ[#, m] &];
          wähle aus Variablen                frei von?
  vars2 = Select[Variables[equations], ! FreeQ[#, n] &];
          wähle aus Variablen                frei von?
  solution = Solve[Thread[equations == 0]];
             löse  fädle auf
  sol1 = Riffle[f /@ vars1, First[vars1 /. solution]];
         füge wiederholt ein  erstes Element
  sol2 = Riffle[f /@ vars2, First[vars2 /. solution]];
         füge wiederholt ein  erstes Element
  Return[{sol1, sol2}]
  gib zurück
 ]
```

In[2480]:=

```
(* function, that makes an ansatz,
according to the singularities and multiplicities found,
and solves the linear system that results from comparing coefficients *)
separatedMultiple[poly_, vars_, singMult1_, singMult2_] :=
 Module[{f, fNumerator, fDenominator, g, gNumerator, gDenominator,
 Modul
   d, q, qNumerator, pair, tupel, infBoolean, vv, solution, output},
  fNumerator = 1;
  fDenominator = 1;
  gNumerator = 1;
  gDenominator = 1;
  pair = {False, 0};
          falsch
  Do[
  iteriere
   If[singMult1⟦i⟧ == Infinity, pair = {True, singMult1⟦i + 1⟧},
   wenn                Unendlichkeit        wahr
    fDenominator = fDenominator * (vars⟦1⟧ - singMult1⟦i⟧) ^ singMult1⟦i + 1⟧]
```

```
 , {i, 1, Length[singMult1], 2}];
          Länge

fNumerator = Sum[f[i] × vars⟦1⟧ ^ i,
             summiere
  {i, 0, Exponent[Expand[fDenominator], vars⟦1⟧] + pair⟦2⟧}];
         Exponent   multipliziere aus

pair = {False, 0};
        falsch

Do[
iteriere
 If[singMult2⟦i⟧ == Infinity, pair = {True, singMult2⟦i + 1⟧},
 wenn               Unendlichkeit      wahr
  gDenominator = gDenominator * (vars⟦2⟧ - singMult2⟦i⟧) ^ singMult2⟦i + 1⟧]
 , {i, 1, Length[singMult2], 2}];
          Länge

gNumerator = Sum[g[i] × vars⟦2⟧ ^ i,
             summiere
  {i, 0, Exponent[Expand[gDenominator], vars⟦2⟧] + pair⟦2⟧}];
         Exponent   multipliziere aus

d = Max[Exponent[fNumerator, vars⟦1⟧] + Exponent[Expand[gDenominator], vars⟦2⟧],
    grö⋯ Exponent                       Exponent  multipliziere aus
   Exponent[gNumerator, vars⟦2⟧] + Exponent[Expand[fDenominator], vars⟦2⟧]] -
   Exponent                        Exponent  multipliziere aus
  Min[Exponent[poly, vars⟦1⟧], Exponent[poly, vars⟦2⟧]]];
  kle⋯ Exponent                Exponent

qNumerator = Sum[q[i, j] × vars⟦1⟧ ^ i × vars⟦2⟧ ^ j, {i, 0, d}, {j, 0, d - i}];
             summiere

output = {fNumerator / fDenominator, gNumerator / gDenominator};


vv = Join[Select[Variables[fNumerator], FreeQ[#, vars⟦1⟧] &],
     verk⋯ wähle aus Variablen          frei von?
  Select[Variables[gNumerator], FreeQ[#, vars⟦2⟧] &],
  wähle aus Variablen          frei von?
  Select[Variables[qNumerator], FreeQ[#, vars⟦1⟧] && FreeQ[#, vars⟦2⟧] &]];
  wähle aus Variablen          frei von?            frei von?
solution = Solve[Thread[Union[Flatten[
           löse  fädle auf Verein⋯ ebne ein
      CoefficientList[Expand[qNumerator * poly - fNumerator * gDenominator +
      Liste der Koeffizienten multipliziere aus
         gNumerator * fDenominator], {vars⟦1⟧, vars⟦2⟧}]]] == 0], vv];

If[Length[solution] == 0, Return[{{1, 1}}]];
⋯  Länge                  gib zurück
solution = output /. solution;
```

```
Return[solution /. Thread[vv → 1]]
gib zurück              fädle auf
]
```

In[2481]:=

In[2482]:=

```
(*getSingMult[poly_,vars_]:=Module[{p,polytope,edges,
                                   Modul
 normals,weights,sings,singsUpdate,singsMult,pairMod,newSings},
 (*polytope=NewtonPolytope[poly,vars];
 edges=GetEdges[polytope];
 normals=OuterNormal[polytope,#]&/@edges;
 weights=validNormals[normals];
 sings=Flatten[GetSing[poly,vars,#]&/@weights,1];*)
       ebne ein
 sings=getSing[poly,vars];
 singsMult={};
 (* for each pair of singularities compute a leading part whose
  separated multiple gives information about its multiplicities *)
 Do[
 iteriere
  p=poly;
  If[pair⟦1⟧≠Infinity,p=p/.vars⟦1⟧→vars⟦1⟧+pair⟦1⟧];
  wenn        Unendlichkeit
  If[pair⟦2⟧≠Infinity,p=p/.vars⟦2⟧→vars⟦2⟧+pair⟦2⟧];
  wenn        Unendlichkeit
  p=Expand[FullSimplify[p]];
    multipliz·· vereinfache vollständig
  pairMod={If[pair⟦1⟧≠Infinity,0,Infinity],If[pair⟦2⟧≠Infinity,0,Infinity]};
          wenn          Unendlichkeit  Unendlichkeit wenn          Unendlichkeit  Unendlichkeit
  AppendTo[singsMult,{pair,LeadingPart[p,vars,pairMod]}];
  hänge an bei
  ,{pair,sings}];

 singsUpdate=sings;

 (* suche nach weiteren Singularitäten,
 iteriere dabei über alle Paare von Singularitäten *)

 Do[
 iteriere
  If[pair≠{Infinity,Infinity},
  wenn    Unendlichk·· Unendlichkeit
    newSings=GetFurtherSing[poly,vars,pair];
    Do[
    iteriere
     If[!MemberQ[sings,newPair],
     w··· enthalten?
       AppendTo[singsUpdate,newPair];
       hänge an bei
       p=poly;
```

```
        If[newPair⟦1⟧≠Infinity,p=p/.vars⟦1⟧→vars⟦1⟧+newPair⟦1⟧];
        wenn               Unendlichkeit
        If[newPair⟦2⟧≠Infinity,p=p/.vars⟦2⟧→vars⟦2⟧+newPair⟦2⟧];
        wenn               Unendlichkeit
        p=Expand[p];
          multipliziere aus
        pairMod={If[newPair⟦1⟧≠Infinity,0,Infinity],
                 wenn              Unendlichkeit  Unendlichkeit
          If[newPair⟦2⟧≠Infinity,0,Infinity]};
          wenn              Unendlichkeit  Unendlichkeit
        AppendTo[singsMult,{newPair,LeadingPart[p,vars,pairMod]}];
        hänge an bei
       ];
     ,{newPair,newSings}]];
  ,{pair,sings}];

sings=Union[sings];
      Vereinigung
singsUpdate=Union[singsUpdate];
            Vereinigung
singsMult=Union[singsMult];
          Vereinigung

Print["sings: ",sings];
gib aus
Print["singsUpdate: ",singsUpdate];
gib aus

If[sings==singsUpdate,Return[singsMult]];
wenn                  gib zurück

While[sings≠singsUpdate,
solange
 sings=singsUpdate;
 (* Print[sings]; *)
    gib aus
 Print[Length[sings]];
 gib aus Länge
 Do[
 iteriere
  If[pair≠{Infinity,Infinity},
  wenn      Unendlichk·· Unendlichkeit
    (* Print["pair: ",pair]; *)
       gib aus
    newSings=Union[GetFurtherSing[poly,vars,pair]];
             Vereinigung

    Do[
    iteriere
     If[!MemberQ[sings,newPair],
     w···  enthalten?
       Print[newPair];
       gib aus
```

```
        AppendTo[singsUpdate,newPair];
        p=poly;
        If[newPair[[1]]≠Infinity,p=p/.vars[[1]]→vars[[1]]+newPair[[1]]];
        If[newPair[[2]]≠Infinity,p=p/.vars[[2]]→vars[[2]]+newPair[[2]]];
        p=Expand[p];
        pairMod={If[newPair[[1]]≠Infinity,0,Infinity],
          If[newPair[[2]]≠Infinity,0,Infinity]};
        AppendTo[singsMult,{newPair,LeadingPart[p,vars,pairMod]}];
       ];
     ,{newPair,newSings}]];
   ,{pair,sings}];
  ];
  Return[singsMult]
 ]*)
```

In[2483]:=

```
(* put the functions together *)
```

In[2484]:=

```
nearSeparate[poly_, vars_] := Module[{weights, sV1, sV2, list,
   polytope, edges, list1, list2, list3, m, n, k, int, numbers, var},
  (* TODO:
   add a test that involves the shape of the Newton polygon of poly *)
  (* compute the pairs of singularities and the associated leading
   parts which give information about their multiplicities  *)
  list = FactorList[poly];
  If[Length[list] > 2 || (Length[list] == 2 && list[[2]][[2]] > 1),
   Return["poly is not irreducible"]];
  If[FreeQ[poly, vars[[1]]] || FreeQ[poly, vars[[2]]],
   Return["poly is univariate"]];
  weights = GetAllWeights[poly, vars];
  sV1 = Sign /@ weights;
  sV2 = Union[sV1];
  If[Length[sV1] ≠ Length[sV2], Return[{{1, 1}}]];

  list1 = Union[getSingMult[poly, vars]];
```

```
             Vereinigung
 (* compute the multiplicities by
  solving the separation problem for the polynomials *)
 list2 = mult[list1, vars];
 (* compute the 1-
  parameter family for the multiplicities of the singularities *)
 list3 = linSystem[list2, m, n, k];

 numbers = {};
 var = Variables[list3][[1]];
       Variablen
 Do[
 iteriere
  AppendTo[numbers, Denominator[Coefficient[list3[[1]][[i]], var]]]
  hänge an bei           Nenner           Koeffizient
  , {i, 2, Length[list3[[1]]], 2}];
           Länge
 Do[
 iteriere
  AppendTo[numbers, Denominator[Coefficient[list3[[2]][[i]], var]]]
  hänge an bei           Nenner           Koeffizient
  , {i, 2, Length[list3[[2]]], 2}];
           Länge
 list3 = list3 /. var → LCM @@ numbers;
                     kleinstes gemeinsames Vielfaches
 (* make an ansatz for a separated multiple *)
 Return[separatedMultiple[poly, vars, list3[[1]], list3[[2]]]]
 gib zurück
]
```