MASTER'S DEGREE THESIS

Master of Science in Advanced Mathematics and Mathematical Engineering

Title: Effective computation of base points of two-dimensional ideals

Author: Guillem Blanco Fernández

Advisor: Maria Alberich Carramiñana and Josep Àlvarez Montaner

Department: Department of Applied Mathematics I

Academic year: 2014-2015

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

Universitat Politècnica de Catalunya
Facultat de Matemàtiques i Estadística

Treball Final de Màster

Effective computation of base points of two-dimensional ideals

Guillem Blanco Fernández

Maria Alberich Carramiñana and Josep Àlvarez Montaner

Departament de Matemàtica Aplicada I

Abstract

Keywords: Equisingularity, Base points, Puiseux series, Two-dimensional ideal

MSC2010: 14B05, 32S15, 14Q05, 13P05

This works focus on computational aspects of the theory of singularities of plane algebraic curves. We show how to use the Puiseux factorization of a curve, computed through the Newton-Puiseux algorithm, to study the equisingularity type of a curve. We present a novel version of the Newton-Puiseux algorithm that can compute all the Puiseux factorization of any arbitrary polynomial, removing the restriction of reduced inputs. Next, we introduce the theory of infinitely near points and the concept of base points of an ideal. Finally, we develop a novel algorithm that, using our novel version of the Newton-Puiseux algorithm, computes the weighted cluster of base points of any two dimensional ideal from any set of generators.

Contents

List of Figures

List of Algorithms

Introduction

The study of singularities of algebraic and analytic varieties is an old and still very interesting field of research. Amongst all singularities, those of plane curves are the most studied ones, and there is a well-established theory for its analysis and classification, mostly due to Noether, Zariski and Enriques.

In this work, we will follow the geometrical approach of Enriques' infinitely near points, developed and synthesized by Casas in his book 'Singularities of Plane Curves' [4]. As explained in [4], Puiseux series and the Puiseux factorization of the algebraic equation of a curve provide a unique insight about the singularities of plane curves. Therefore, we can use an algebraic object, the Puiseux series, to completely determine the equisingularity class of a curve, which is equivalent to its topological class.

Furthermore, Puiseux series can be, in some cases, computed algorithmically given the equation of a curve. This make them even more suitable to study singularities from the computational point of view. However, the algorithm for computing the Puiseux factorization of an algebraic equation, the Newton-Puiseux algorithm, is only known to work for reduced inputs, i.e. not containing multiple factors.

The weighted cluster of base points of an ideal is well studied in [4], however it is not known any method to compute it algorithmically from the equations of the generators of the ideal. One of the major issues when computing the weighted cluster of base points is that one cannot reduce the generators of the ideals and hence apply to them the known techniques to study their equisingularity type. Results for some special cases are known [1].

The goal of this work is then double. First, we present a novel version of the Newton-Puiseux algorithm to compute all the Puiseux series and their algebraic multiplicities. Thus, our new version of the Newton-Puiseux algorithm is no longer restricted to reduced inputs. We can then use all this information to determine the equisingularity class of any arbitrary algebraic equation. Second, we present a novel algorithm to compute the weighted cluster of base points, extending the results in [1]. This algorithm relies on the fact that we can compute all the Puiseux series and their algebraic multiplicities given any arbitrary algebraic equation.

This work achieves the goals of a Master's Thesis. We have developed skills to study plane curve singularities through the use of Puiseux series and weighted cluster of infinitely near points. On the other hand, some results and algorithms from chapter 1 are novel. Chapter 2 contains a new algorithm for computing the equisingularity type of a curve from the equisingularity types of its branches. Finally, most of the results in chapter 3 are either new or generalizations of previous ones in the literature.

This memory is structured in three chapters. In the first one we introduce Puiseux series and the traditional Newton-Puiseux algorithm to compute the Puiseux factorization. After that, we show how to extend the traditional Newton-Puiseux factorization to compute also the algebraic multiplicities of each Puiseux series.

The second chapter is devoted to the study of the theory of infinitely near points. We introduce all the results that will be necessary to compute the equisingularity type of a curve. We see how we can encode the equisingularity class of a curve in a finite combinatorial object, the Enriques diagram, and how we can use the Puiseux factorization to compute the Enriques diagram of curve. With a new algorithm, we will merge all the results in [4] and compute the equisingularity type of any curve.

In the last chapter, we first develop the theory of base points of an ideal and then we extend the results in [1] to the case of two-dimensional ideals. These generalized results plus two novel propositions will allow us to present a completely novel algorithm to compute the weighted cluster of base points of and ideal. This algorithm will require a modified versions of the new Newton-Puiseux algorithm in order to make the computations practical.

We will provide detailed explanations of all the algorithms, including the already known ones in [4] and, of course, the new ones. We have also implemented and tested all these algorithms in Macaulay2 [7], a software system devoted to supporting research in algebraic geometry and commutative algebra. Our work will, very likely, become a new publicly available package inside the Macaulay2 framework. The code of all the algorithms implemented in Macaulay2 has been included in the appendices.

Last but not least, I would like to end this introduction thanking my advisors, Maria Alberich and Josep Àlvarez, because this work would not have been possible without their constant advice, patience and intuition.

Chapter 1

Puiseux series

This first chapter is devoted to the study of Puiseux series and its relation with y-roots of series in $\mathbb{C}[[x, y]]$. This study starts with the basic definitions of fractionary power series. Next, we will see how Puiseux series are important to understand y-roots of a bivariate formal power series in $\mathbb{C}[[x, y]]$. Explicit computations of y-roots of polynomials in $\mathbb{C}[x, y]$ will be possible thanks to the Newton-Puiseux algorithm. All this will allow us to state and understand the Puiseux factorization theorem. Because the traditional Newton-Puiseux algorithm only works with reduced polynomials, we present a novel modification of the Newton-Puiseux algorithm that work with any arbitrary polynomial and can compute the algebraic multiplicities of the Puiseux series. Finally, we discuss the specific implementation issues of this algorithm in the mathematical software Macaulay2.

1.1 Basic definitions

Given the ring of univariate formal power series $\mathbb{C}[[x]]$ one can construct its field of fractions, that will be denoted by $\mathbb{C}((x))$. This corresponds to the set of formal Laurent series since its elements can be written as $\sum_{i=d}^{\infty} a_i z^i, d \in \mathbb{Z}, a_i \in \mathbb{C}$. Formally, one can set $\mathbb{C}((x^{1/n}))$, for $n \in \mathbb{N}$. Then, the field of fractionary power series

$$\mathbb{C}\langle\langle x\rangle\rangle := \Big\{ \sum_{i \geq r} a_i x^{i/n} \mid r \in \mathbb{Z}, n \in \mathbb{N} \Big\}$$

can be constructed as the direct limit of the system $\{\mathbb{C}((x^{1/n})), x^{1/n} \mapsto x^{1/n'} \mid n' = dn\}$. The exact details of this construction can be found in [4, page 17]. In particular, this means that two elements of $\mathbb{C}\langle\langle x\rangle\rangle$ can be added or multiplied in a suitable $\mathbb{C}((x^{1/n}))$ since any series belongs to $\mathbb{C}((x^{1/n'}))$ for any $n' \in (n)$.

Given a fractionary series

$$s = \sum_{i \geq r} a_i x^{i/n} \in \mathbb{C}\langle\langle x \rangle\rangle \tag{1.1}$$

we can define, as for regular series, the *order* in x of s to be either $o_x(s) = \infty$ if $s = 0$ or

$$o_x(s) = \frac{\min\{i \mid a_i \neq 0\}}{n}$$

otherwise. Fractionary power series with positive order, that is $o_x(s) > 0$, will be called *Puiseux series* and will be our main object of study in this chapter.

After simplifying all the exponents appearing in s we can assume that n and $\gcd\{i \mid a_i \neq 0\}$ have no common factor. Then we say that n is the *polydromy order* of s, usually denoted by $\nu(s)$.

Fix $n \in \mathbb{N}$ and consider $\mathbb{C}((x^{1/n}))$ as an extension of $\mathbb{C}((x))$. For each n-th root of unity $\varepsilon \in \mathbb{C}$ the substitution $\varepsilon x^{1/n} \mapsto x^{1/n}$ induces an automorphism σ_ϵ, $\varepsilon^n = 1$ of $\mathbb{C}((x^{1/n}))$ over $\mathbb{C}((x))$. If s is as in equation (1.1) above, then

$$\sigma_\varepsilon(s) = \sum_{i \geq r} \varepsilon^i a_i x^{i/n}$$

from which is easy to see that the conjugate do not depend on the field $\mathbb{C}((x^{1/n}))$. It is also clear that all conjugates have the same order in x and the same polydromy order. Finally, it is not difficult to see that the number of conjugates of a given series s is $\nu(s)$ and that a series belongs to $\mathbb{C}((x))$ if and only if $\sigma_\varepsilon(s) = s$ for all $\nu(s)$-th roots of unity ε.

1.2 Solving $f(x, y) = 0$

The main object of study in this section will be the complex bivariate formal power series, i.e. $\mathbb{C}[[x, y]]$. All the results below could also be applied to convergent series, i.e. $\mathbb{C}\{x, y\}$, or to bivariate polynomial in $\mathbb{C}[x, y]$. In the sequel, we will work with elements in $\mathbb{C}[x, y]$ when discussing algorithmic details as the inputs of any algorithm must be finite.

Let $f \in \mathbb{C}[[x, y]]$. In this section we are interested in solving the equation $f(x, y) = 0$. We will focus on solving the equation $f(x, y) = 0$ with respect to y since solving for x is analogous. This means that we need to find some sort of function of x, $y(x)$, such that $f(x, y(x)) = 0$. Trying to find a general function accomplishing $f(x, y(x))$ is hard. Thus, we will restrict ourself to series in x.

Obviously, the elements of the form $ux^d, d \in \mathbb{N}$, $u \in \mathbb{C}[[x, y]]$ invertible, have no y-roots. If one considers $f \in \mathbb{C}[[x]][y]$, then the y-roots of f can be seen as

roots of a certain polynomial and elements of the form ux^d are coefficients. In the sequel, we will work in a neighbourhood of the origin $O = (0, 0)$ and assume that $f(0, 0) = 0$, otherwise, we can apply a change of coordinates.

If we restrict ourselves to $\mathbb{C}\{x, y\}$, their elements are well-defined functions. If the origin is a regular point the inverse function theorem states that you can find a function $y(x)$ in a neighbourhood of O such that $f(x, y(x)) = 0$. Furthermore, since $f(x, y)$ is analytic so it is $y(x)$ and hence,

$$y(x) = a_1x + a_2x^2 + \cdots + a_mx^m + \cdots .$$

On the other hand, if the point is singular, the inverse function theorem does not apply. For instance, taking the polynomial $f = y^2 - x^3 \in \mathbb{C}[x, y]$ it is easy to see that $y(x) = x^{2/3}$ does not belong to $\mathbb{C}[[x]]$ but belongs to $\mathbb{C}\langle\langle x\rangle\rangle$. Thus, fractionary power series, and in particular Puiseux series, play a fundamental role in this problem.

The following lemmas show the close relationship between y-roots and divisors of $f \in \mathbb{C}[[x, y]]$. All the following results have an algebraic nature so they are not restricted to $\mathbb{C}\{x, y\}$ and can be applied to $\mathbb{C}[[x, y]]$. The lemmas are stated without proof, a proof for each one of them can be found in [4, page 19]

Lemma 1.2.1 ([4, 1.2.2])**.** *A Puiseux series $s \in \mathbb{C}[[x^{1/n}]]$ is a y-root of $f \in \mathbb{C}[[x, y]]$ if and only if $y - s$ divides f in $\mathbb{C}[[x^{1/n}, y]]$.*

Lemma 1.2.2 ([4, 1.2.3])**.** *If s is a y-root of $f \in \mathbb{C}[[x, y]]$, then all conjugates of s are y-roots of f too.*

Take a Puiseux series $s \in \mathbb{C}[[x^{1/n}]]$ and write $g_s = \prod_{i=1}^{\nu}(y - \sigma_{\varepsilon_i}(s))$. Then $g_s \in \mathbb{C}[[x]][y]$ as all its coefficients are invariant by conjugation.

Lemma 1.2.3 ([4, 1.2.4])**.** *A Puiseux series $s \in \mathbb{C}[[x^{1/n}]]$ is a y-root of $f \in \mathbb{C}[[x, y]]$ if and only if g_s divides f in $\mathbb{C}[[x, y]]$.*

Lemma 1.2.4 ([4, 1.2.5])**.** *The series g_s is irreducible in $\mathbb{C}[[x, y]]$.*

One can notice the similarity between the y-roots and algebraic roots in number fields. Before describing a method to find y-roots of any element in $\mathbb{C}[[x, y]]$ we need to introduce the Newton polygon.

1.3 Newton polygon

Take $\pi = \mathbb{R}^{+2}$ a plane with an orthogonal system of coordinates α, β. Let $f = \sum_{\alpha,\beta\geq 0} A_{\alpha,\beta}x^\alpha y^\beta$ belonging to $\mathbb{C}[[x, y]]$. For each $(\alpha, \beta) \in \mathbb{N}^2$ with $A_{\alpha,\beta} \neq 0$, we

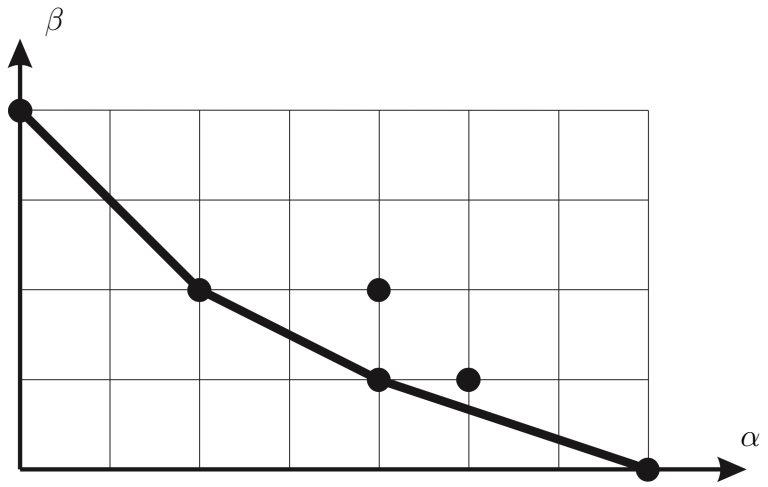

Figure 1.1: $\Delta(f)$ and $\mathbf{N}(f)$ for $f = y^4 - x^2y^2 - 2x^4y^2 + x^4y + x^5y + x^7$. From [4, page 16].

plot on π the point (α, β). This way we obtain a discrete set of of points with non-negative integral coordinates

$$\Delta(f) = \{(\alpha, \beta) \in \mathbb{N}^2 \mid A_{\alpha,\beta} \neq 0\}$$

called the *Newton diagram*.

We are interested in the lower left part of the convex hull of the Newton diagram. This set can be expressed as the convex hull of $\Delta'(f) = \Delta(f) + (\mathbb{R}^+)^2$ minus the two half lines parallel to the axis. This polygonal line (possibly reduced to a single vertex) is called *Newton polygon* of f and it is denoted by $\mathbf{N}(f)$.

We will always consider the vertices and the sides of the Newton polygon ordered from left to right. Thus, if the vertices of the Newton polygon are $P_i = (\alpha_i, \beta_i), i = 0, \ldots, k$ then $\alpha_i \leq \alpha_{i+1}$, being P_0 the beginning and P_k the end of the polygon. The *height* $h(\mathbf{N})$ and the *width* $w(\mathbf{N})$ are, respectively, the maximal ordinate and the maximal abscissa of its vertices, that is, $h(\mathbf{N}) = \beta_0$ and $w(\mathbf{N}) = \alpha_k$. Moreover, the side with endpoints $(\alpha_i, \beta_i), (\alpha_{i+1}, \beta_{i+1})$ will be denoted by Γ_i and $h(\Gamma_i) = \beta_i - \beta_{i+1}, w(\Gamma_i) = \alpha_i - \alpha_{i+1}$ will be its *height* and *width*, respectively.

The following proposition summarizes several properties of the Newton polygon that are easy to check and that will be important later on.

Proposition 1.3.1 ([4, 1.1])**.** *Let $f, g \in \mathbb{C}[[x, y]]$ and $\mathbf{N}(f), \mathbf{N}(g)$ their Newton polygons, then:*

1. *$\mathbf{N}(f)$ begins (resp. ends) on the β-axis (resp. α-axis) if and only if f has no factor x (resp. factor y).*

2. $\mathbf{N}(f)$ *of* f *is reduced to a single vertex if and only if* $f = ux^\alpha y^\beta$, *where* u *is an invertible series.*

3. *If* $u \in \mathbb{C}[[x, y]]$ *is invertible, then* $\mathbf{N}(f) = \mathbf{N}(uf)$.

4. *If* x *(resp.* y*) does not divide* f, *then the height of* $\mathbf{N}(f)$ *is* $o_y(f(0, y))$ *(resp.* $o_x(f(x, 0))$*).*

5. *Height and width of Newton polygons are additive, i.e.,*

$$h(\mathbf{N}(fg) = h(\mathbf{N}(f_1)) + h(\mathbf{N}(f_2))$$
$$w(\mathbf{N}(fg) = w(\mathbf{N}(f_1)) + w(\mathbf{N}(f_2)).$$

In algorithm 1 we detail an efficient algorithm to compute the Newton polygon. This is a modified version of Andrew's monotone convex hull algorithm [2].

Algorithm 1 Newton polygon

Require: A polynomial $f(x, y) = \sum_{\alpha,\beta} A_{\alpha,\beta} x^\alpha y^\beta \in R[x, y]$ with R a ring.
Ensure: The Newton polygon of f: $\{(\alpha_1, \beta_1), (\alpha_1, \alpha_2), \ldots, (\alpha_k, \beta_k)\}$.

```
1:  function NewtonPolygon(f)
2:      E ← {(α, β) ∈ ℕ² | f(x, y) = Σ_{α,β} A_{α,β} x^α y^β} ∪ {(∞, 0)}
3:      E ← Sort(E)                          ▷ E = {(α_1, β_1), ..., (α_n, β_n)}
                                             ▷ α_i < α_{i+1}, β_i < β_{i+1} if α_i = α_{i+1}
4:      N(f) ← ∅
5:      for i ← 1, ..., |E| do               ▷ |E| = n
6:          j ← |N(f)|
7:          α ← (α_j − α_{j−1})(β_i − β_{j−1}) − (β_j − β_{j−1})(α_i − α_{j−1})
8:          while |N(f)| ≥ 2 and α ≤ 0 do
9:              N(f) ← N(f) \ {(α_j, β_j)}   ▷ (α_j, β_j) ∈ N(f)
10:         end while
11:         N(f) ← N(f) ∪ {(α_i, β_i)}       ▷ (α_i, β_i) ∈ E
12:     end for
13:     return N(f)
14: end function
```

Andrew's convex hull algorithm computes the upper and the lower convex hulls separately. Hence, by adding a single point at infinity in the x-axis and only computing the lower convex hull we can compute the Newton polygon more efficiently. Andrew's algorithm has a time complexity of $O(n \log n)$, where n is the number of non-zero monomials in the input polynomial. Because our input points are natural numbers the Newton polygon can be computed in $O(n)$ steps using a specialized sorting algorithm for integer values in algorithm 1. The reader

can check that algorithm 1 returns the points in the order required by the Newton polygon.

1.4 Searching for y-roots

Let us say that we want to determine the y roots of the following series:

$$f(x,y) = \sum_{\alpha,\beta} A_{\alpha,\beta} x^{\alpha} y^{\beta} \in \mathbb{C}[[x,y]], \tag{1.2}$$

by means of an inductive procedure, we start by testing solutions of the form

$$s(x) = ax^{m/n} + \cdots$$

where $m, n \in \mathbb{Z}$, such that $\gcd(m,n) = 1$ and $a \in \mathbb{C}$ is non-zero. Substituting in equation (1.2) we can see that the initial terms of $f(x, s(x))$ are

$$\sum_{\alpha,\beta \geq 0} A_{\alpha,\beta} a^{\alpha} x^{\alpha+\beta m/n}. \tag{1.3}$$

This last series can be also written as

$$\sum_{k} \left(\sum_{n\alpha+m\beta=k} A_{\alpha,\beta} a^{\beta} \right) x^{k/n}.$$

From here it is clear that the pair (α, β) giving rise to a term of degree k/n in equation (1.3) are the points of $\Delta(f)$ lying on the line $n\alpha + m\beta = k$. Consider now only one of those lines l with the minimal possible k: the terms of lowest degree in equation (1.3) are then,

$$\sum_{(\alpha,\beta) \in l \cap \Delta(f)} A_{\alpha,\beta} a^{\beta} x^{\alpha+\beta m/n}.$$

We can now consider two cases:

Case (a): There is no side on $\mathbf{N}(f)$ with slope $-n/m$. Then there is just a single point (α_0, β_0), a vertex of $\mathbf{N}(f)$, on l and then $f(x, s(x))$ has a single term of minimal degree, namely $A_{\alpha_0,\beta_0} a_0^{\beta} x^{k/n}$, which cannot be cancelled by other terms. Thus $f(x, s(x)) \neq 0$ and such an y-root is not possible.

Case (b): There is a side of $\mathbf{N}(f)$, say Γ, with slope $-n/m$ and hence on l. The terms of lowest degree in $f(x, s(x))$ are then

$$\left(\sum_{(\alpha,\beta) \in \Gamma} A_{\alpha,\beta} a^{\beta} \right) x^{k/n}$$

Assuming that (α_1, β_1) and (α_0, β_0) are the first and last end of Γ the former expression may be written in the form

$$a^{\beta_0}\left(\sum_{(\alpha,\beta)\in\Gamma} A_{\alpha,\beta}a^{\beta-\beta_0}\right)x^{k/n} = a^{\beta_0}F_\Gamma(a)x^{k/n}$$

where

$$F_\Gamma(Z) = \sum_{(\alpha,\beta)\in\Gamma} A_{\alpha,\beta}Z^{\beta-\beta_0} \in \mathbb{C}[Z]$$

is a polynomial with non-zero constant term and degree equal to the height $\beta_1 - \beta_0$ of Γ. The polynomial $F_\Gamma(Z)$ (resp. $F_\Gamma(Z) = 0$) is called the *polynomial (resp. equation) associated with* Γ.

We have found then necessary conditions for a monomial to be the initial term of a solution. The Newton-Puiseux algorithm will iterate this process to find all the terms of the y-roots.

Remark 1.4.1. We have been assuming that $a \neq 0$ which may not be the case if f has 0 as y-root. This is easy to check and will occur if and only if y is a factor of f, that is, if and only if $\mathbf{N}(f)$ ens above the α-axis, i.e., $\beta_k > 0$.

Remark 1.4.2. It is not difficult to see that if the slope of Γ is $-n/m$, with $\gcd(n, m) = 1$, and $n > 0$ then $F_\Gamma \in \mathbb{C}[z^n]$. Thus, if a is a root also is εa, with $\varepsilon^n = 1$.

1.5 Newton-Puiseux semi-algorithm

In this section we will describe the inductive step of a semi-algorithm to compute one of the y-roots of a general series in $\mathbb{C}[[x, y]]$. This method is a semi-algorithm, and not a proper algorithm, because the input is potentially infinite series and also because the number of steps performed is not necessarily finite.

As always, let f be a bivariate formal power series as in equation (1.2). We will start assuming that $h(\mathbf{N}(f)) > 0$, otherwise it has no y-root by 1.3.1. Keeping the same notation as in the initial step in the previous section:

Step (i): Each step begins with a series $f_i(x_i, y_i)$. The induction hypothesis is that $h(\mathbf{N}(f_i)) > 0$ so either $\mathbf{N}(f_i)$ ends above the α-axis or it has at least one side:

(*i*.a) If $\mathbf{N}(f)$ ends above the α-axis, take $s^{(i)} = 0$ and stop.

(*i*.b) Otherwise, choose a side Γ of $\mathbf{N}(f)$ and a root a of F_Γ. Assume that Γ has equation $n_i\alpha + m_i\beta = k_i, \gcd(m_i, n_i) = 1$. Perform the following change of variables:

$$x_i = x_{i+1}^{n_i} \tag{1.4}$$
$$y_i = x_{i+1}^{m_i}(a_i + y_{i+1}). \tag{1.5}$$

Then we have

$$f_i(x_{i+1}, y_{i+1}) = x_{i+1}^{k_{i+1}} \left(\sum_{n_i\alpha + m_i\beta \geq k} A_{\alpha,\beta} x_{i+1}^{n_i\alpha + m_i\alpha - k_{i+1}} (a_i + y_{i+1})^\beta \right)$$

and we define $f_{i+1} = x_{i+1}^{-k_{k+1}} f_i \in \mathbb{C}[[x_{i+1}, y_{i+1}]]$. Finally take as y_i-root of f_i the element

$$s^{(i)}(x_i) = x_i^{m/n}(a_i + s^{(i+1)}(x_{i+1}))$$

where s^{i+1} is a y_{i+1}-root of f_{i+1} and has to be determined by the next iteration.

Starting with $f_0 = f$ the result, once all the change of variables are reverted, is:

$$\begin{aligned} s = s^{(0)} &= x_0^{m_0/n_0}(a_0 + x_1^{m_1/n_1}(a_1 + \cdots + x_i^{m_i/n_i}(a_i + s^{(i+1)}) \cdots)) \\ &= x^{m/n}(a_0 + x^{m_1/nn_1}(a_1 + \cdots + x^{m_i/n\cdots n_i}(a_i + s^{(i+1)}) \cdots)). \end{aligned}$$

The fact that if one start with $h(\mathbf{N}(f)) > 0$ then $h(\mathbf{N}(f_i)) > 0$ for any i, is proved in lemma 1.4.1 of [4].

Finally, it is not hard to prove that the Puiseux series and y-roots can be obtained using this iterated procedure.

As a consequence of all these facts we can obtain a constructive proof of the Puiseux theorem:

Theorem 1.5.1 (Puiseux, [4, 1.5.4])**.** *If $f \in \mathbb{C}[[x, y]]$ and $h(\boldsymbol{N}(f)) > 0$, then there is a Puiseux series s which is a y-root of f, namely $f(x, s(x)) = 0$*

Corollary 1.5.2 ([4, 1.5.6])**.** *Any $f \in \mathbb{C}[[x, y]]$ has a unique decomposition of the form*

$$f(x, y) = ux^r \prod_{i=1}^{l} \prod_{j=1}^{n_i} (y - \sigma_{\varepsilon_j}(s_i))^{\alpha_i}$$

with $r, \alpha_1, \ldots, \alpha_l \in \mathbb{Z}$, $u \in \mathbb{C}[[x, y]]$ invertible and $h(\mathbf{N}(f)) = \alpha_1\nu(s_1) + \cdots + \alpha_l\nu(s_l) = \alpha_1 n_1 + \cdots + \alpha_l n_l$.

1.6 Separation of y-roots

The first thing that one have to do to turn the previous semi-algorithm into an algorithm that ends in a finite number of steps is to restrict the input to polynomials in $\mathbb{C}[x, y]$ instead of general elements of $\mathbb{C}[[x, y]]$.

The other problem is that, even if the inputs are in $\mathbb{C}[x, y]$, the previous method can, potentially, run indefinitely. Usually, one only needs to find a partial sum of each of the series that is the partial sum of no other y-root. In such a situation we will say that a root has been completely *separated* from the rest.

A sufficient condition for this fact can be given in the case that the input polynomial is reduced, that is, all its factors have multiplicity one. The condition is that $h(\mathbf{N}(f_i)) = 1$, for some $i \in \mathbb{N}$. Following the notation in theorem 1.5.1, we known that $h(\mathbf{N}(f_i)) = \alpha_1 n_{i_1} + \cdots + \alpha_{i_l} n_{i_l}$.

Lemma 1.6.1 ([4, 1.6.3])**.** *For any $i > 0$, the multiplicity of s as y-root of f equals the multiplicity of $s^{(i)}$ as y_i-root of f_i.*

By lemma 1.6.1, if the input f is reduced so is f_i for any i and hence, $\alpha_i = 1, i = 1, \ldots, l$, and the height can only be one if $l = 1$ and $n_{i_1} = 1$. Hence, the y-root does not share the i-th non-zero term with any other y-root.

However, it is not clear that $h(\mathbf{N}(f_i)) = 1$ will happen at all. Indeed,

Lemma 1.6.2 ([4, 1.5.1])**.** *There exists an integer i_0 such that $n_i = 1$ if $i > i_0$.*

Finally, a detailed description of the Newton-Puiseux algorithm is given in algorithm 2.

As noted in remark 1.4.2, one could obtain all the y-roots directly from the Newton-Puiseux algorithm, even the conjugated ones. Usually, it suffices to find a root for each conjugacy class. As stated in [4, page 42], it is not difficult to see that by taking always one of the conjugated roots of $F_\Gamma(Z)$, one obtains one and only one Puiseux series for each conjugacy class of y-roots. For simplicity, in line 16 of algorithm 2 one takes the root of $F_\Gamma(Z)$ corresponding to $\varepsilon = 1$.

Line 3 of algorithm 2 is the responsible for the finiteness of the algorithm, as noted previously. Line 7 deals case (i.a), which is not mutually exclusive with case (i.b). Finally, the case of an element of $\mathbb{C}[x, y]$ with no y-roots, i.e. elements of the form ux^r, is handled automatically by algorithm 2 since the loop in line 10 will not be executed and the algorithm will return the empty set, as one should expect.

Remark 1.6.1. The separability condition presented in this chapter works because of theorem 1.5.1. This means that, in principle, it applies only to inputs in $\mathbb{C}[[x, y]]$ or $\mathbb{C}\{\{x, y\}\}$. As we have seen, in order to make our algorithm practical we need to restrict the inputs to $\mathbb{C}[x, y]$. Being reduced in $\mathbb{C}[x, y]$ trivially implies being

Algorithm 2 Newton-Puiseux algorithm (reduced)

Require: A reduced bivariate polynomial $f(x,y) = \sum_{\alpha,\beta} A_{\alpha,\beta} x^\alpha y^\beta \in \mathbb{C}[x,y]$ with Puiseux factorization $f(x,y) = ux \prod_{i=1}^{l} \prod_{j=1}^{n_i} (y - \sigma_{\varepsilon_j}(s_i))$.

Ensure: The shortest series $\bar{s}_1, \ldots, \bar{s}_l \in \mathbb{C}\langle\langle x \rangle\rangle$ approximating $s_1, \ldots s_l$, and such that $\bar{s}_i \neq \bar{s}_j$, $i \neq j$.

function NewtonPuiseuxReduced(f)
  $\mathbf{N}(f) \leftarrow$ NewtonPolygon(f) ▷ $\mathbf{N}(f) = \{(\alpha_0, \beta_0), \ldots, (\alpha_k, \beta_k)\}$
  if $\beta_0 = 1$ **then** ▷ $h(\mathbf{N}(f)) = \beta_0$
    return $\{0\}$
  end if
  $S \leftarrow \emptyset$
  if $\beta_k > 0$ **then**
    $S \leftarrow \{0\}$
  end if
  for $(\alpha_i, \beta_i), (\alpha_{i+1}, \beta_{i+1}) \in \mathbf{N}(f)$ **do**
    $n \leftarrow \beta_i - \beta_{i+1}$
    $m \leftarrow \alpha_{i+1} - \alpha_i$
    $k \leftarrow \beta_i \alpha_{i+1} - \alpha_i \beta_{i+1}$
    $\Gamma \leftarrow nx + my - k$ ▷ $\Gamma \in \mathbb{Z}[x,y]$
    $F_\Gamma \leftarrow \sum_{(\alpha,\beta)\in\Gamma} A_{\alpha,\beta} Z^{\beta-\beta_0}$ ▷ $F_\Gamma \in \mathbb{C}[Z]$
    for $a \in \{F_\Gamma(z^n) = 0 \mid z \in \mathbb{C}\}$ **do**
      $\overline{f} \leftarrow x^{-k} f(x^n, x^m(a+y))$
      $\overline{S} \leftarrow$ NewtonPuiseuxReduced($\overline{f}$)
      $S \leftarrow S \cup \{x^{m/n}(a + \bar{s}(x^{1/n})) \mid \bar{s} \in \overline{S}\}$
    end for
  end for
  return S
end function

reduced in $\mathbb{C}[x,y]$. Fortunately, the converse is also valid but far from trivial. The result is true because an algebraic variety is analytically unramified [5].

Example 1. The output of algorithm 2 for the following reduced polynomial

$$\begin{aligned} f = {} & x^{17}y^{11} - x^{17}y^{10} + 4x^{18}y^8 + 2x^{19}y^5 + 4x^{10}y^{12} - x^{21} - 4x^{10}y^{11} + 16x^{11}y^9 + \\ & 8x^{12}y^6 - x^6y^{11} + x^6y^{10} + 2x^3y^{13} - 4x^{14}y - 4x^7y^8 - 2x^3y^{12} - y^{15} + \\ & 8x^4y^{10} + y^{14} - 2x^8y^5 - 4xy^{12} + 4x^2y^9 + x^{10} - 2x^7y^2 + x^4y^4 \end{aligned}$$

are the following Puiseux series:

$$s_1 = x^{\frac{3}{2}} + x^{\frac{17}{4}} + \cdots, \qquad s_2 = x^{\frac{2}{5}} + \frac{4}{25}x^{\frac{1}{2}} + \cdots$$

with polydromy orders $\nu(s_1) = 4$ and $\nu(s_2) = 10$. The series are separated after the first terms is computed, but lemma 1.6.2 does not occur until the second iteration. This fact, later on, will turn out to be very useful (see remark 2.8.1). By theorem 1.5.1 the polynomial f has then two factors $f_1, f_2 \in \mathbb{C}[[x]][y]$. In fact, $f_1, f_2 \in \mathbb{C}[x,y]$ and they are precisely:

$$f_1 = -x^{17} - 4x^{10}y + x^6 - 2x^3y^2 + y^4, \quad f_2 = -y^{11} + y^{10} - 4xy^8 - 2x^2y^5 + x^4.$$

1.7 A general algorithm

The algorithm 2 only works for reduced polynomials. However, for our final goal we need the exact Puiseux decomposition of any given polynomial; that is, the separated partial sums of the Puiseux series and the multiplicity of each series.

We have developed a new algorithm that extends algorithm 2 and computes the multiplicity of each Puiseux series. The first step in this new algorithm consists in computing the *square-free decomposition* of the input polynomial.

Definition 1.7.1. Let f an element of a unique factorization domain R such that

$$f = f_1^{\alpha_1} f_2^{\alpha_2} \cdots f_n^{\alpha_n}$$

with $f_i \in R$ irreducible, $\alpha_i \in \mathbb{N}$. The *square-free decomposition* of f is

$$f = g_1 g_2^2 g_3^3 \cdots a_n^n$$

with $g_i \in R$ reduced and $n = \max\{\alpha_1, \alpha_2, \cdots, \alpha_n\}$.

Note that g_i could be equal to 1 for some i. The standard way to compute the square-free factorization of a polynomial $f \in k[x]$ with k a field of characteristic

Algorithm 3 Yun's algorithm

Require: A polynomial $f \in k[x]$ with $\text{char}(k) = 0$.
Ensure: $(g_1, 1), (g_2, 2), \ldots, (g_n, n), g_i \in k[x]$ reduced such that $f = g_1 g_2^2 \cdots g_n^n$.

1: **function** SQUAREFREEFACTORIZATION(f)
2: $A \leftarrow \emptyset$
3: $f_0 \leftarrow f$
4: $g_0 \leftarrow f'$
5: $i \leftarrow 0$
6: **while** $\deg f \neq 0$ **do**
7: $g_i \leftarrow \gcd(f_i, g_i)$
8: $f_{i+1} \leftarrow f_i / g_i$
9: $g_{i+1} \leftarrow g_i / g_i - f'_{i+1}$
10: $A \leftarrow A \cup \{(g_i, i)\}$
11: $i \leftarrow i + 1$
12: **end while**
13: **return** $A \setminus \{(g_0, 0)\}$
14: **end function**

0, is the Yun's algorithm [10]. A detailed description of Yun's algorithm can be found in algorithm 3.

Algorithm 3 can also be used to compute the square-free factorization of a polynomial in $k[x, y]$ by making the identification $k[x, y] \cong k[x][y]$. This is possible because the greatest common divisor in line 7 can also be computed in $k[x, y]$ as this ring is still a UFD. Also, the divisions in lines 8 and 9 are exact and hence, they don't present any problem.

The only problem is that this algorithm ignores any factor that is a constant in $k[x][y]$, i.e., any element of $k[x]$. However, this is not an issue for us as any element in $\mathbb{C}[x]$ is either a unit in $\mathbb{C}[[x]][y]$ or a power of x and hence, it does not have y-roots. If we were interested in extracting the x factor, we could use the Newton polygon and proposition 1.3.1 to do so.

In order to overcome the problem with reduced inputs one may be tempted to apply algorithm 2 to each reduced part of the square-free factorization of a polynomial. However it is important to remark that this solution does not work as it is not possible to know if the series obtained from one of the reduced factors will be completely separated from the series coming from any other reduced factor. Algorithm 4 extracts both the Puiseux series and their multiplicities using the square-free factorization of the input polynomial.

Algorithm 4 makes a subtle but important distinction in the output between the polynomial $x \in \mathbb{C}[x]$ and the Puiseux series $x \in \mathbb{C}\langle\langle x \rangle\rangle$ of polydromy order 1.

Algorithm 4 Newton-Puiseux algorithm

Require: A bivariate polynomial $f(x,y) = \sum_{\alpha,\beta} A_{\alpha,\beta} x^\alpha y^\beta \in \mathbb{C}[x,y]$ with Puiseux factorization $f(x,y) = ux^r \prod_{i=1}^{l} \prod_{j=1}^{n_i} (y - \sigma_{\varepsilon_j}(s_i))^{\alpha_i}$.

Ensure: $(x,r), (\bar{s}_1, \alpha_i), \ldots, (\bar{s}_l, \alpha_l) \in \mathbb{C}\langle\langle x \rangle\rangle \times \mathbb{N}$, the shortest series approximating $s_1, \ldots s_l$ and their multiplicities such that $\bar{s}_i \neq \bar{s}_j$, $i \neq j$.

function NewtonPuiseux(f)
  $S \leftarrow \emptyset$
  $\mathbf{N}(f) \leftarrow$ NewtonPolygon(f) ▷ $\mathbf{N}(f) = \{(\alpha_0, \beta_0), \ldots, (\alpha_k, \beta_k)\}$
  if $\alpha_0 > 0$ **then**
    $S \leftarrow \{(x, \alpha_0)\}$ ▷ $x \in \mathbb{C}[x]$, $r := \alpha_0$
  end if
  $\widetilde{f} \leftarrow f / \gcd(f, f_y)$ ▷ $f_y(x,y) := \frac{d}{dy} f(x,y)$
  $L \leftarrow$ SquareFreeFactorization(f)
  return $S \cup$ NewtonPuiseuxLoop($\widetilde{f}, L$)
end function
function NewtonPuiseuxLoop(f, L)
  $N \leftarrow \{$NewtonPolygon(g) $\mid (g, \alpha) \in L\}$
  $L \leftarrow \{(g, \alpha) \in L \mid h(\mathbf{N}(g)) \neq 0, \mathbf{N}(g) \in N\}$
  $\mathbf{N}(f) \leftarrow$ NewtonPolygon(f) ▷ $\mathbf{N}(f) = \{(\alpha_0, \beta_0), \ldots, (\alpha_k, \beta_k)\}$
  if $\beta_0 = 1$ **then** ▷ $h(\mathbf{N}(f)) = \beta_0$
    return $\{(0, \alpha)\}$ ▷ $L = \{(g, \alpha) \mid g \in \mathbb{C}[x][y], \alpha \in \mathbb{N}\}$
  end if
  $S \leftarrow \emptyset$
  if $\beta_k > 0$ **then**
    $S \leftarrow \{(0, \{\alpha \in \mathbb{N} \mid \beta_k > 0,\ (\alpha_k, \beta_k) \in \mathbf{N}(g) \in N, (g, \alpha) \in L\})\}$
  end if
  for $(\alpha_i, \beta_i), (\alpha_{i+1}, \beta_{i+1}) \in \mathbf{N}(f)$ **do**
    $n \leftarrow \beta_i - \beta_{i+1}$
    $m \leftarrow \alpha_{i+1} - \alpha_i$
    $k \leftarrow \beta_i \alpha_{i+1} - \alpha_i \beta_{i+1}$
    $\Gamma \leftarrow nx + my - k$ ▷ $\Gamma \in \mathbb{Z}[x,y]$
    $F_\Gamma \leftarrow \sum_{(\alpha,\beta) \in \Gamma} A_{\alpha,\beta} Z^{\beta - \beta_0}$ ▷ $F_\Gamma \in \mathbb{C}[Z]$
    for $a \in \{F_\Gamma(z^n) = 0 \mid z \in \mathbb{C}\}$ **do**
      $\overline{f} \leftarrow x^{-k} f(x^n, x^m(a + y))$
      $\overline{L} \leftarrow \{(x^{-k} g(x^n, x^m(a+y)), \alpha) \mid (g(x,y), \alpha) \in L\}$
      $\overline{S} \leftarrow$ NewtonPuiseuxLoop($\overline{f}, \overline{L}$)
      $S \leftarrow S \cup \{(x^{m/n}(a + \bar{s}(x^{1/n})), \alpha) \mid (\bar{s}, \alpha) \in \overline{S}\}$
    end for
  end for
  return S
end function

This is because x as a factor of f does not have a Puiseux series and has to be treated separately; on the other hand, $f = y - x$ has Puiseux series $x \in \mathbb{C}\langle\langle x \rangle\rangle$.

NewtonPuiseux is the entry point of the algorithm. In line 5 it extracts the x factor using the Newton polygon of the input. In line 7 it computes the reduced part of the input polynomial so we can apply the separability condition.

NewtonPuiseuxLoop is basically a modified version of algorithm 2. Let's see how this new function differs form the original NewtonPuiseuxReduced function and why it works:

- Given the input polynomial f, f_i was recursively computed from f_{i-1} and it has as first term of its Puiseux expansions the i-th term of some of the Puiseux expansions of f.

- In the first iteration L_0 contains the square-free decomposition of f. Following the notations in definition 1.7.1 and in algorithm 4, define L_i inductively from L_{i-1} in the following way

$$\begin{aligned} L_0 &= \{(g_1, 1), (g_2, 2) \ldots, (g_n, n) \mid f = g_1 g_2^2 \cdots g_n^n\} \\ L_i &= \{(g_j^{(i)}, j) \mid g_i = x^{-k} g_j^{(i-1)}(x^n, x^m(a+y)), \\ &\qquad h(\mathbf{N}(g_j^{(i)})) \neq 0,\ (g_j^{(i-1)}, j) \in L_{i-1}\}. \end{aligned} \tag{1.6}$$

The following new proposition shows that the sets L_i contains the square-free decomposition of each f_i in algorithm 4.

Proposition 1.7.2. *Let $f \in \mathbb{C}[x, y]$ and let $f = g_1 g_2^2 \cdots g_n^n$ be its square-free decomposition. Then, following the notation in algorithm 4, the set L_i in equation* (1.6) *contains the square-free decomposition of f_i for any $i \in \mathbb{N}$.*

Proof. It suffices to prove this for $i = 1$ and apply induction. Compute f_1 and apply the same transformation, i.e. equation (1.4) and divide by x^k, to the reduced factors of f in L. We known from lemma 1.6.3 in [4] that this transformation cannot create new factors in f_1 or increase the multiplicity of the existent ones. However, some of the g_i could become units after the transformation. If g_{i1} is a unit, from proposition 1.3.1 we known that $h(\mathbf{N}(g_{i1})) = 0$ and hence, the result is proved. □

In algorithm 4, line 13 removes the possible units generated in the previous iteration and line 30 applies a new transformation to L_i. Line 20 select the terms from L_i that fall under the case (i.a) and therefore have 0 as an y-root. This is done by looking at the Newton polygon of each square-free term, just as we did in algorithm 2. Finally, the stopping condition in line 16 can return the algebraic multiplicity of the y-root that has been separated from the rest because at this

point L can only contain one square-free factor, the square-free factor the y-root belongs to. This is true because the height is one and because, by definition, different square-free factors in L cannot share the same y-root.

1.8 Implementation details

We have implemented all the above algorithms using the Macaulay2 computational algebra system [7]. This section explains the most important aspects of our implementation and how it compares against other implementations in other mathematical libraries such as Singular or Maple.

When dealing with polynomial roots and computers, as in the case of the Newton-Puiseux algorithm, one has to make a choice between computing the roots numerically or working with them symbolically. Each method has its own advantages and disadvantages: numerical methods are very fast but inaccurate, symbolical methods are exact but expensive to compute.

For the implementation of our Newton-Puiseux algorithm we have chosen to work with floating point values and compute the polynomials roots using numerical methods. The Newton-Puiseux algorithm could potentially compute many polynomials roots for a single input and this process is done recursively, i.e. the roots in an iteration depend on the roots computed in the previous iteration. This means that a symbolic method that starts in $\mathbb{Q}$ will have to work in arbitrary field extensions of $\mathbb{Q}$. Algorithms that work with polynomials over arbitrary field extensions of $\mathbb{Q}$ are only practical with relatively small inputs and this would limit the size of the possible inputs for our algorithm. Another problem is that these algorithms are not implemented in Macaulay2 yet.

We will see in the section 2.11 that, as long as the precision used for the computations is enough two separate numerically the roots of each side F_Γ, possible small errors in the computation of the roots are not an issue for our purpose. Furthermore, the Macaulay2 software has the option to work with floating point numbers of arbitrary precision. This means that we can specify how many bits of precision we want in our computations. Although there is no rule of thumb to select the best number of precision bits, we have observed that the running time of our program is not very sensible to the number of bits specified. If the user does not specify any particular value, our program defaults to 300 bits.

We have tried to take advantage of all the packages that Macaulay2 include, however this has not been always possible. In the computation of the Newton Polygon we tried to use the Polyhedra package in Macaulay2. This solution turned to be very slow since the algorithm to compute convex hull in the Polyhedra package of Macaulay2 is only designed to work with small inputs as it has a $O(n^2)$ complexity. This is the reason why we had to develop algorithm 1.

Macaulay2 has no built-in functions to compute the roots of a polynomials numerically. Therefore, in order to compute the roots of the polynomials in algorithm 4, we had to use some routines that the NumericalAlgebraicGeometry package in Macaulay2 provides. Although this solution works reasonably well for small inputs, it starts to fails for some univariate polynomials of degree 17 and 19. This became a major issue as our algorithm could potentially work with polynomials much bigger than that. For that reason, we had to modify the core of Macaulay2 to provide a built-in function to compute roots of polynomials using a much more robust function from the PARI library [9].

Another problem we have found is that Macaulay provide limited support for polynomial rings over the complex numbers. In particular, by the time this thesis was written, Macaulay2 has no support for polynomials divisions over $\mathbb{C}$. This basically means that the inputs of our program have to be polynomials over $\mathbb{Q}[x, y]$ and the computation of the square-free factorization and the reduced part of the input only works over $\mathbb{Q}[x, y]$. As long as the input is in $\mathbb{Q}[x, y]$ this does not change the output of the algorithm since the results of all the algorithms are the same regardless if the input is considered to be in $\mathbb{Q}[x, y]$ or $\mathbb{C}[x, y]$.

By the time this work was done, we are aware of two other mathematical softwares that can compute Puiseux series for reduced polynomials, the Singular computational algebra system [6] and Maple™ [8]. Both packages use symbolic methods to compute the roots of the polynomials.

The Macaulay2 code of all the algorithms described in this chapter can be found in Appendix B. The code necessary for manipulating Puiseux series in Macaulay2, can be found in Appendix A. We will end this section showing an example of the usage and the output of our program running in Macaulay2 version 1.8.2.

Example 2. Using the same polynomials as in example 1 but now adding multiplicities greater than one:

```
i1: f1 = y^4 - 2*x^3*y^2 + x^6 - 4*x^10*y - x^17;

i2: f2 = x^4 - 2*x^2*y^5 - 4*x*y^8 + y^10 - y^11;

i3: puiseuxExpansion(f1^2*f2^3)

        2       1         1          3    17        1
        -       -        --          -    --        -
        5       2        10          2     4        4
o3 = {(x  + .16x  + O ( x   ), 3), (x  + x   + O ( x  ), 2)}
```

We can see how our program return both the Puiseux series and the multiplicities.

Chapter 2

Infinitely near points

The goal of this chapter is to make a study of the singular points in plane algebraic curves and see how to compute the necessary information to determine the equisingularity type of a singularity. We first make a brief introduction to the theory of infinitely near points and the resolution of singularities. After that, we introduce the notion of a weighted cluster and how weighted clusters can be represented with Enriques diagrams. Finally, we show how the Enriques theorem can be used to compute proximity matrices from Puiseux series. We show a new algorithms to compute the equisingularity type of a curve from the algorithms developed in the previous chapter and the already existing ones in [4]. Through this chapter we basically follow [4] for all the theoretical results.

2.1 Germs of curves

Fix a point O on a smooth analytic surface S. We will denote by $\mathcal{O}_{S,O}$ (or just $\mathcal{O}$ if S and O are clear from the context) the ring of holomorphic functions in a neighbourhood of the point O. Similarly, we will denote by $\mathfrak{m}_{S,O}$ (or just $\mathfrak{m}$) its maximal ideal, which consist of the functions f such that $f(O) = 0$.

If x, y are local coordinates at the point O, the representation of holomorphic functions by convergent series leads to the identification: $\mathcal{O} \cong \mathbb{C}\{x, y\}$. Furthermore, any pair of local coordinates, in particular x, y, will be a pair of generators of $\mathfrak{m}$.

Given an analytic curve ξ lying on S and going through O it can be written locally around O as the zero locus of $f \in \mathbb{C}\{x, y\}$ and we just say $\xi : f = 0$. The *germ* of a curve ξ at point O is an equivalence class of all curves defined in a neighbourhood of O, modulo the equivalence relation of having the same restriction to an open neighbourhood of O. The same construction can be done for holomorphic functions giving rise to germs of holomorphic functions. If ξ is a

curve on S, we will denote ξ_O its germ at O, being ξ_O the empty set if ξ does not go through O.

The concept of a germ of curve is important as there is a one to one correspondence between germs of curves at O and non-zero principal ideals of $\mathcal{O}_{S,O}$: two equations f, g generate the same germ if and only if f/g is an invertible element of $\mathcal{O}_{S,O}$.

Direct from the Puiseux theorem applied to any equation of a germ ξ:

Proposition 2.1.1 ([4, 2.1.1]). *If $\xi : f = 0$ is a germ of a curve with origin at $\mathcal{O}$, then it has a uniquely determined decomposition as a sum of irreducible germs: if $f = f_1^{\alpha_1} \cdots f_r^{\alpha_r}$, the equations f_i being irreducible, then $\xi = \alpha_1\gamma_1 + \cdots + \alpha_r\gamma_r$, where each γ_i is the irreducible germ $f_i = 0$.*

The germs γ_i are also called *branches* of ξ. The positive integer α_i is called the multiplicity of γ_i as a component of ξ. A component γ_i is called multiple if and only if $\alpha_i > 1$. Finally, a germ ξ is said to be reduced if and only if it has no multiple components, that is, $\xi = \gamma_1 + \cdots + \gamma_r$ with all γ_i irreducible and $\gamma_i \neq \gamma_j$ if $i \neq j$. Given a germ ξ there is just and just one reduced germ ξ_{red} with the same branches as ξ: if $\xi = \alpha_1 f_1 + \cdots + \alpha_r f_r$ with $\alpha_i > 0$, γ_i irreducible for $i = 1, \ldots, r$ and $\gamma_i \neq \gamma_j$ for $i \neq j$, then $\xi_{red} = \gamma_1 + \cdots + \gamma_r$.

2.2 Infinitely near points

Let $\pi : \bar{S} \to S$ be the blowing-up of a point O in a smooth analytic surface S. The precise construction of the blowing-up can be found in [4, ch. 3]. Take ξ a curve in an open subset $W \ni O$ of S: we will denote by $\bar{\xi}$ the pull-back of ξ by π, i.e. $\bar{\xi} = \pi^*(\xi)$, and it will be called the *total transform* of ξ.

Lemma 2.2.1 ([4, 3.2.1]). *The total transform of a curve ξ has the form*

$$\bar{\xi} = \tilde{\xi} + e_O(\xi)E$$

where $\tilde{\xi}$ is a curve in $\pi^{-1}(W)$ with finitely many intersections with E.

The integer $e_O(\xi)$ is the multiplicity of ξ at the point O. The *strict transform* of ξ is the component $\tilde{\xi}$. This is a curve defined on $\pi^{-1}(W)$ and which does not contain any copy of the exceptional divisor E.

The exceptional divisor E of blowing up a point O in a surface S will be called the *first (infinitesimal) neighbourhood* of O in S. Since blowing up a point on a smooth surface give rise to another smooth surface, we can iterate the process: if $i > 0$, the points in the *i-th (infinitesimal) neighbourhood* of O (on S) will be the

points in the first (infinitesimal) neighbourhood of some point in the $(i-1)$-th (infinitesimal) neighbourhood of O. The points which are in the i-th neighbourhood of O, for some $i > 0$ are called points *infinitely* near to O. Sometimes the points on the original surface S are called *ordinary points*.

An essential notion regarding the relative position of infinitely near points is the notion of proximity. Let two points p, q be equal or infinitely near to O. The point q is said to be *proximate* to p (and denoted $q \to p$) if it belongs to the exceptional divisor E^p of blowing up p. This can happen for two reasons: q is an ordinary point of E^p, that is $\pi^p(q) = p$, where π^p is the blowing up of p; or, q is an infinitely near point to E^p that still belongs to one of the successive strict transforms of E^p.

Because of the fact that the total transforms of the exceptional divisors (which are smooth curves isomorphic to the projective line) either do not meet or meet transversally at a single point and no three have a common point (see [4, 3.5.5]), if p is infinitely near to O, then p is proximate to just one or two points equal or infinitely near to O. A point p infinitely near to O is called *free* if and only if it is proximate to just one point equal or infinitely near to O. Otherwise, if p is proximate to exactly two points, it is called a *satellite* point.

2.3 Resolution of singularities

Given a point p infinitely near to O, the process of blowing up a curve ξ can be iterated to get a curve on S_p denoted $\tilde{\xi}_p$ (resp. $\bar{\xi}_p$) and called the *strict* (resp. *total*) *transform* of ξ at p. The curve ξ will go through p, or p will belong to ξ, if and only if the curve $\bar{\xi}_p$ is not empty. We will denote by $\mathcal{N}_O(\xi)$ the set of points equal or infinitely near to O that lie on ξ. The reader may notice that $\mathcal{N}_O(\xi) = \mathcal{N}_O(\xi_{red})$.

We define the *multiplicity* of ξ at p as being the multiplicity of $\tilde{\xi}_p$ and it will be written $e_p(\xi)$. In particular ξ goes through p if and only if $e_p(\xi) > 0$. As usual, if $e_p(\xi) > 1$ (resp. $e_p(\xi) = 1$), p is called a *multiple* (resp. *simple*) (infinitely near) point of ξ. Sometimes these multiplicities are called *effective* to distinguish them from virtual multiplicities defined in section section 2.5.

The most important result regarding these concepts is that one can transform a reduced compact singular curve into a smooth one by mean of a finite sequence of blowing-ups.

Theorem 2.3.1 ([4, 3.7.1])**.** *A reduced curve contains finitely many multiple infinitely near points to a given ordinary point on the curve.*

However, for the purpose of classifying singularities knowing only the multiple infinitely near points is not enough. Given a curve ξ an ordinary or infinitely near point p on ξ will be called a *singular point* of ξ if and only if either

- p is multiple on ξ,
- p is a satellite point,
- p precedes a satellite point on ξ.

Notice that there may be simple and free points on a germ ξ preceding satellite points still on ξ. The easiest example being the first neighborhood of the origin on $\xi : y^2 - x^3 = 0$. An ordinary point is singular if and only if it is multiple (see [4, 3.6.2]). Hence, this new notion of singular points extends the usual one to infinitely near points.

Proposition 2.3.2 ([4, 3.7.7])**.** *A germ of a curve contains at most finitely many satellite points.*

Thus, thanks to 2.3.1 and 2.3.2 the number of singular points is finite. Define $\mathcal{S}_O(\xi)$ as the subset of $\mathcal{N}_O(\xi)$ containing the first non-singular points in each branch of ξ and all the points preceding them.

Definition 2.3.3. We will say that two curves germs of a curves ξ and ζ are said to be *equisingular* if and only if there exists a bijection $\varphi : \mathcal{S}(\xi_{red}) \longrightarrow \mathcal{S}(\zeta_{red})$ such that φ and φ^{-1} preserve the natural ordering and the proximities of infinitely near points and the multiplicities of branches are the same.

2.4 Clusters of points

Let K be a finite non-empty set of points equal or infinitely near to O, and assume that for all $p \in K$ all the points preceding p are contained in K. Such a set will be called a *cluster* of points infinitely near to O and O the origin of K.

As the reader may notice, clusters can be used to represent desingularizations of curves at a given point p. It is extremely useful to associate to each cluster a graphical representation in the form of a tree-shaped diagram as we can see in figure 2.1.

However, this kind of diagram does not encode the proximity relations between points, only the natural ordering. This is why another kind of diagrams, called *Enriques diagram*, is used to represent infinitely near points. These diagrams are also a tree, like the one described before, but the edges are drawn in two ways, curved or straight, according to the following rules:

- If q is free and proximate to p, the edge joining p and q is a smooth curve which, if $p = O$, has the same tangent at p as the edge ending at p.

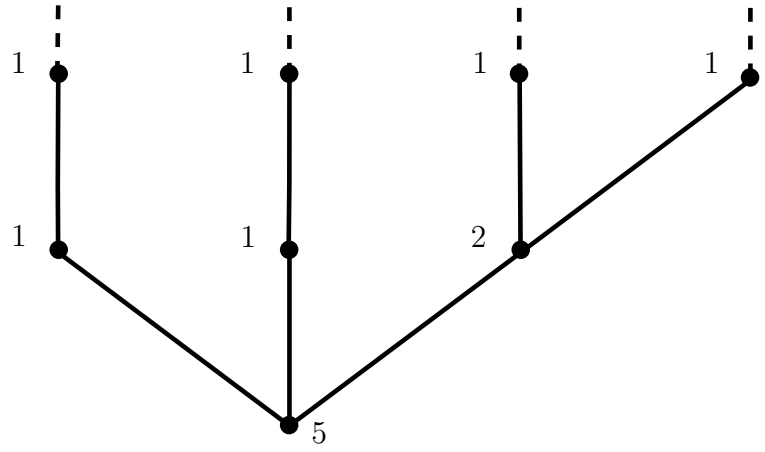

Figure 2.1: The tree of $f = xy(x-y)(x^3-y^2)$.

- If points p and q, q in the first neighbourhood of p, have been represented, the rest of the points proximate to p in the successive neighbourhoods of q (and the corresponding edges) are represented on a straight half-line starting at q and orthogonal to the edge joining it with p. To avoid self-intersections in the diagram, such half-lines are drawn alternatively to the right and to the left of the preceding one.

In particular, the Enriques diagram of $\mathcal{S}(\xi)$ will be called the *Enriques diagram* of ξ and is a representation of the equisingularity type of ξ at O.

Example 3. The Enriques' diagram in the right hand-size of figure 2.2 represents the cluster of infinitely near points and the multiplicities of the infinitely near points of the curve defined by the equation $f = xy(x-y)(x^3-y^2)$. It can be seen how the curve is composed by four irreducible branches; usually, the last free point with multiplicity one in each of the branches is omitted. However, for the sake of clarity we will include them. It is importance to notice the difference between figure 2.2 and figure 2.1 and the presence of a satellite point in the second neighbourhood of the origin.

The notion of proximity leads to the following statement that describe the multiplicity of an ordinary or infinitely near point in terms of the multiplicities of tis proximate points.

Theorem 2.4.1 (Proximity equalities, [4, 3.5.3])**.** *For any ordinary or infinitely near point p on a curve ξ,*

$$e_p(\xi) = \sum_{q \to p} e_q(\xi). \tag{2.1}$$

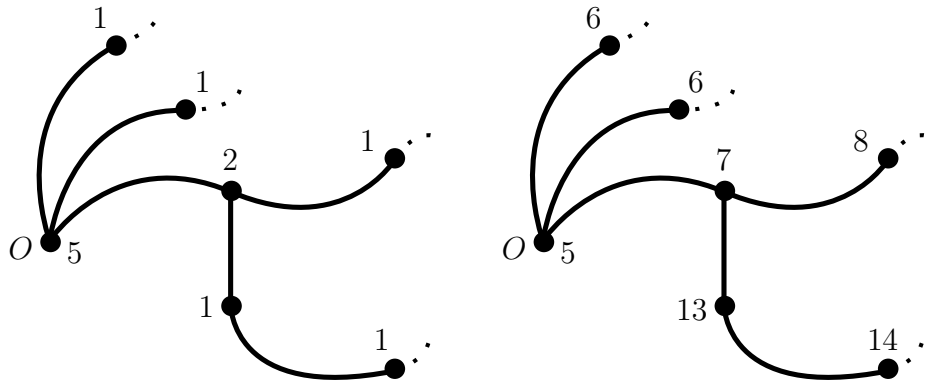

Figure 2.2: Enriques diagrams of $xy(x-y)(x^3-y^2)$ representing the weighted cluster defined with multiplicities on the left side and with values on the right side, see proposition 2.5.5.

Thus, if we assume that the branches are reduced, given the Enriques diagram of a desingularization one can recover the multiplicities of the infinitely near points using the formula above.

2.5 Virtual multiplicities

A pair $\mathcal{K} = (K, \nu)$, where K is a cluster and $\nu : K \longrightarrow \mathbb{Z}$ is an arbitrary map, is called a *weighted cluster*. The integer $\nu_p = \nu(p)$, for $p \in K$, is called the *virtual multiplicity* at p of $\mathcal{K}$.

To a weighted cluster $\mathcal{K} = (K, \nu)$ we associate another map $\bar{\nu} : K \longrightarrow \mathbb{Z}$ defined recursively as $\bar{\nu}_O = \nu_O$ and $\bar{\nu}_p = \nu_p + \sum_{p \to q} \bar{\nu}_q$ for $p \in K - \{O\}$. The integer $\bar{\nu} = \bar{\nu}(p)$, for $p \in K$, is called the *virtual value* at p of $\mathcal{K}$. Conversely, the virtual multiplicities can be obtained recursively from the virtual values of a cluster.

In this section we will work with virtual multiplicities. We say that a curve ξ *goes through the point* O *in a cluster with virtual multiplicity* ν_O if and only if $e_O(\xi) \geq \nu_O$.

Definition 2.5.1. Assume that a ξ curve goes through the point O with virtual multiplicity ν_O. Then, the *virtual transform* of ξ relative to the virtual multiplicity ν_O is

$$\hat{\xi} = \tilde{\xi} + (e_O(\xi) - \nu_O)E \tag{2.2}$$

where $\tilde{\xi}$ and E denote, respectively, the strict transform of ξ and the exceptional divisor of blowing up O.

Notice that also,

$$\hat{\xi} = \bar{\xi} - \nu_O E$$

so that on may understand the virtual transform $\tilde{\xi}$ as being obtained from ξ like the strict transform, but formally considering O as a point of ξ of multiplicity ν_O. In particular we have $\hat{\xi} = \tilde{\xi}$ if and only if $e_O(\xi) = \nu_O$.

Assume now that $\mathcal{K} = (K, \nu)$ is a weighted cluster which has origin at O. Denote by $p_i, i = 1, \ldots, s$ the points of $\mathcal{K}$ in the first neighbourhood of O.

For each $i = 1, \ldots, s$, denote by K_i the cluster with origin at p_i that contains p_i and all the points infinitely near to it in K. For $i = 1, \ldots, s$, the restriction to K_i is a system ν_i of virtual multiplicities for K_i. It gives rise to a weighted cluster $\mathcal{K}_i = (K_i, \nu_i)$.

Definition 2.5.2. If $\mathcal{K} = (K, \nu)$ is a weighted cluster with more than one point, we say that a curve ξ *goes through* $\mathcal{K}$ if and only if

1. ξ goes through O with virtual multiplicity ν_O and,
2. the virtual transform of ξ relative to the virtual multiplicity ν_O goes through K_i, for $i = 1, \ldots, s$.

If a curve ξ goes through a weighted cluster $\mathcal{K}$ with (effective) multiplicities $(e_p(\xi))$ equal to the virtual ones $(\nu(p))$ and all the singular points of ξ are inside K, i.e. $\mathcal{S}_O(\xi) \subset K$, then we will say that ξ goes *sharply* through $\mathcal{K}$.

The next condition is useful to characterize weighted cluster for which there are curves going with effective multiplicities equal to the virtual ones.

Definition 2.5.3 (Proximity Inequality)**.** A *consistent cluster* is a weighted cluster $\mathcal{K} = (K, \nu)$ such that for all $p \in K$,

$$\nu_p \geq \sum_{q \to p} \nu_q. \tag{2.3}$$

This basically means that the cluster has to fulfill, at least, the proximity equalities in 2.4.1. The empty sum being zero by definition, the reader may notice that if $\mathcal{K} = (K, \nu)$ is consistent, then $\nu_p \geq 0$ for all $p \in K$.

It is clear from the last section that any weighted cluster can be associated with an Enriques diagram. We can also represent a cluster K with a matrix.

Definition 2.5.4. The *proximity matrix* of K is

$$P_K = I - M$$

where I denotes the $|K| \times |K|$ unit matrix and M is the matrix defined by

$$m_p^q = \begin{cases} 1, & \text{if } p \text{ is proximate to } q \\ 0, & \text{otherwise} \end{cases}$$

The dimension of the proximity matrix represents the number of points in the cluster. By encoding the virtual multiplicities as a column vector ν_K, we can encode a weighted cluster using a pair (P_K, ν_K). The proximity matrix is lower triangular and has ones in the diagonal, hence it is invertible over $\mathbb{Z}$.

Example 4. The proximity matrix of the cluster represented in figure 3 together with its vectors of multiplicities and values is

$$P_K = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & -1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 1 \end{pmatrix}, \quad e_K = \begin{pmatrix} 5 \\ 2 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \quad v_K = \begin{pmatrix} 5 \\ 7 \\ 13 \\ 6 \\ 6 \\ 8 \\ 14 \end{pmatrix}$$

However, the proximity matrix is not unique since the order of the points is not unique; any partial order that maintains the relative position of infinitely near points is equally valid.

Alternatively one can assign to each infinitely near point a *value* instead of a multiplicity. The value of an infinitely near point p in the curve ξ, denoted $v_p(\xi)$, is defined as the multiplicity of the exceptional divisor in the total transform of ξ at p. As an example, the left-hand size picture of figure 2.2 shows an Enriques diagram with values instead of multiplicities.

The relation between the multiplicities and the values is explained in the next proposition.

Proposition 2.5.5 ([4, 4.5.1])**.** *Given a cluster K with origin O, for any germ of curve at ξ at O,*

$$e_K(\xi) = P_K v_K(\xi)$$

If $\nu : K \to \mathbb{Z}$ is system of virtual multiplicities, as we already said, we will denote by $\bar{\nu} : K \to \mathbb{Z}$ the system of virtual values associated with it, where $\bar{\nu} = P_K^{-1}\nu$. Virtual values are useful as they allow us to state the condition of a curve ξ going through a weighted cluster more easily:

$$v_p(\xi) \geq \bar{\nu}_p \qquad \forall p \in K. \tag{2.4}$$

2.6 Characteristic exponents

So far we have introduced Puiseux series and infinitely near points. In the remaining sections we will see how to use the Puiseux factorization of an irreducible curve γ to obtain its Enriques diagram or, equivalently, its proximity matrix.

The Puiseux series of an irreducible germ provide a set of numerical equisingularity invariants that determine its equisingularity class. Let $s = \sum_{j>0} a_j x^{j/n}$ be a Puiseux series and assume that its polydromy order is n. We define a set of rational numbers $m_1/n, \ldots, m_k/n$, the *characteristic exponents* of s, in the following way: m_1/n is the first non-integral exponent that effectively appears in s, and, for each i, m_i/n is the first exponent effectively appearing in s that cannot be reduced to the minimal common denominator of $m_1/n, \ldots, m_{i-1}/n$. In other words, we have

$$m_1 = \min\{j \mid a_j \neq 0 \text{ and } j \notin (n)\},$$

and, inductively, provided that $n^{i-1} = \gcd(n, m_1, \ldots, m_{i-1}) \neq 1$,

$$m_i = \min\{j \mid a_j \neq 0 \text{ and } j \notin (n^{i-1})\}.$$

Since n is the polydromy order of s, it is clear that we will eventually reach an integer k for which $n^k = 1$. Notice also that the set of characteristic exponents of an integral powers series ($n = 1$) is the empty set.

After the definition of characteristic exponents, the series s may be written, in the form

$$s = \sum_{\substack{j\in(n)\\ 1\leq j<m_1}} a_j x^{j/n} + \sum_{\substack{j\in(n^1)\\ m_1\leq j<m_2}} a_j x^{j/n} + \cdots + \sum_{\substack{j\in(n^{k-1})\\ m_{k-1}\leq j<m_k}} a_j x^{j/n} + \sum_{j\geq m_k} a_j x^{j/n}. \quad (2.5)$$

In the sequel, we will refer to the terms, exponents or coefficients in

$$\sum_{\substack{j\in(n^i)\\ m_i\leq j<m_{i+1}}} a_j x^{j/n}$$

as *the terms, exponents or coefficients depending* on the i-th characteristic exponent. Furthermore, the terms, coefficients or exponents not associated with any characteristic exponents will be called *the tail terms, coefficient or exponents*, respectively.

Assume that local analytic coordinates x, y are fixed and let γ be an irreducible germ of curve at O. The characteristic exponents of all Puiseux series of γ relative to the coordinates x, y being the same, they will be called the *characteristic exponents* of γ relative to the coordinates x, y. We will see later that they do not depend on the coordinates x, y, as long as the second axis is non-tangent to γ, and hence they constitute an equisingularity invariant.

2.7 Enriques' theorem

Given an irreducible germ, the set of terms associated to a characteristic exponents have associated with them a finite sequence of consecutive free and satellite points

on the germ. In this section we will determine the multiplicities of the infinitely near points on an irreducible germ γ, as well as their proximity relations, from one of the Puiseux series of γ.

Assume that γ is an irreducible germ with origin at O, that is s is one of the Puiseux series relative to fixed local coordinates x, y and that s has polydromy order n and characteristic exponents $\{m_1/n, \ldots, m_k/n\}$.

Set $m_0 = 0$ and $n_i = \gcd(n, m_1, \ldots, m_i)$ so that, in particular $n^0 = n$ and $n^k = 1$. For each $i = 1, \ldots, k$, perform the successive Euclidean divisions leading to $n^i = \gcd(n^{i-1}, m_i) = n^i_{r(i)}$,

$$\begin{aligned} m_i - m_{i-1} &= h^i_0 n^{i-1} + n^i_1 \\ n^{i-1} &= h^i_1 n^i_1 + n^i_2 \\ &\cdot \\ &\cdot \\ &\cdot \\ n^i_{r(i)-2} &= h^i_{r(i)-1} n^i_{r(i)-1} + n^i_{r(i)} \\ n^i_{r(i)-1} &= h^i_{r(i)} n^i_{r(i)} \end{aligned}$$

and notice that $r(i) \geq 1$ and $h^i_j > 0$ for $j = 1, \ldots, r(i)$. Then we have:

Theorem 2.7.1 (Enriques, [4, 5.5.1])**.** *There are on the irreducible germ γ in successive neighbourhoods, corresponding to the i-th characteristic exponent of s,*

$$\begin{gathered} h^i_0 \text{ points with multiplicity } n^i_0 : p^i_{0,1}, \ldots, p^i_{0,h^i_0}, \\ h^i_1 \text{ points with multiplicity } n^i_1 : p^i_{1,1}, \ldots, p^i_{1,h^i_1}, \\ \cdots \\ h^i_{r(i)-1} \text{ points with multiplicity } n^i_{r(i)-1} : p^i_{r(i)-1,1}, \ldots, p^i_{r(i)-1,h^i_{r(i)-1}} \\ h^i_{r(i)} \text{ points with multiplicity } n^i_{r(i)} : p^i_{r(i),1}, \ldots, p^i r(i), h^i_{r(i)}. \end{gathered}$$

The first of these points (i.e., $p^i_{0,1}$ if $h^i_0 \neq 0$ or $p^i_{1,1}$ if $h^i_0 = 0$) is the origin O if $i = 1$ or a free point in the first neighbourhood of $p^{i-1}_{r(i-1),h^{i-1}_{r(i-1)}}$ if $i > 1$. Furthermore, all point after $p^k_{r(k),h^k_{r(k)}}$ are simple and free. The above points are related by proximity in the following way:

- *Exclude first the case $i = 1$ and $m_1/n < 1$. Then all the points*

$$p^i_{0,1}, \ldots, p^i_{0,h^i_0}, p^i_{1,1},$$

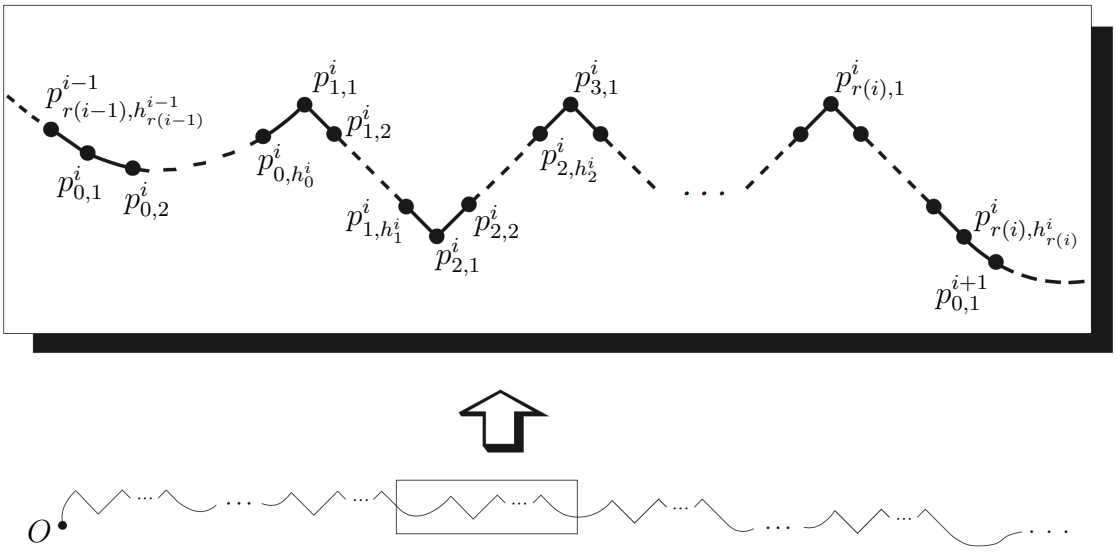

Figure 2.3: The Enriques diagram of an arbitrary irreducible germ as described in theorem 2.7.1. From [4, page 175].

$i = 1, \ldots, k$ but $p^1_{0,1} = O$ are free points. The remaining ones are satellite point, precisely, for $j = 1, \ldots, r(i) - 1$,

$$p^i_{j,1}, \ldots, p^i_{j,h^i_j}, p^i_{j+1,1}$$

are proximate to $p^i_{j-1,h^i_{j-1}}$, and

$$p^i_{r(i),1}, \ldots, p^i_{r(i),h^i_r(i)}$$

are proximate to $p^i_{r(i)-1,h^i_{r(i)-1}}$

- *In the case $i = 1$ and $m_1/n < 1$, we have $h^1_0 = 0$, then $p^1_{1,1} = O$,*

$$p^1_{1,2}, \ldots, p^1_{1,h^1_1}$$

and also $p^1_{2,1}$ if $r(i) > 1$, are free points on the y-axis. The remaining ones are all satellite and proximity between them is as above.

Is clear that theorem 2.7.1 determines the multiplicity and the proximity relations of all the infinitely near points of γ, for a much better understanding of the above theorem see figure 2.7.

Theorem 2.7.1 splits the points in $\mathcal{S}_O(\gamma)$ in $k + 1$ pairwise disjoint sets of consecutive points. For $i = 1, \ldots, k$, we have the finite set $\{p^i_{0,1}, \ldots, p^i_{r(i),h^i_{r(i)}}\}$

whose points will be called *the points depending on the i-th characteristic exponent*. The $(k+1)$-th and last set is infinite and consist of all the points after $p^k_{r(k),h^k_{r(k)}}$, which are simple and free and will be called *the tail points*.

Each set of points depending on a characteristic exponent consist of a certain number of consecutive free points followed by satellite points grouped together in straight lines, usually called steps. There is at least one free point depending on each characteristic exponent as $r(i) > 0$ for all i. Analogously, there are satellite points depending on each characteristic exponent except for the case $i = 1$ and $m_1/n < 1$: in this case the points $p^i_{1,l}$ are all free. In particular, all points depending on the characteristic exponents are singular but for the case $m_1 = 1$ which corresponds to a smooth branch tangent to the y-axis.

Follows from the Enriques' theorem the invariance of the characteristic exponents.

Corollary 2.7.2 ([4, 5.5.3])**.** *Assume that the y-axis is not tangent to γ. The characteristic exponents of the Puiseux series of γ are determined by the multiplicities of the points of γ and their proximity relations. Hence, the characteristic exponents of a Puiseux series of γ do not depend on the coordinates.*

Corollary 2.7.3 ([4, 5.5.4])**.** *Irreducible germs γ and $\bar{\gamma}$ are equisingular if and only if they have the same characteristic exponents.*

Corollary 2.7.4 ([4, 5.5.5])**.** *Two reduced germs ξ and ζ are equisingular if and only if there is a one to one correspondence between branches of ξ and branches of ζ such that:*

1. *corresponding branches have the same characteristic exponents, and*
2. *any two branches of ξ have the same intersection multiplicity as their corresponding branches of ζ.*

This follows from the previous corollary and [4, 3.8.6].

2.8 Computing Enriques diagrams

In this section we will show the algorithms that compute the proximity matrix and the vector of multiplicities of the singular points of an irreducible germ of a curve from its Puiseux series and applying theorem 2.7.1.

In order to apply theorem 2.7.1 we need to compute the numerical values $h^i_j, n^i_j \in \mathbb{N}, i = 1, \ldots, k, j = 0, \ldots, r(i)$ defined in section 2.7. But first we need to compute the characteristic exponents of the input Puiseux series.

Algorithm 5 Characteristic exponents

Require: A Puiseux series $s = \sum_{j>0} a_j x^{j/n}$.

Ensure: The characteristic exponents $\{(0,n),(m_1,n^1),\ldots,(m_k,n^k)\}$.

function CHARACTERISTICEXPONENTS(s)
  $C \leftarrow \{(0,n)\}$
  $n^i \leftarrow n$
  while $n^i \neq 1$ **do**
    $m_i \leftarrow \min\{j \in \mathbb{N} \mid a_j \neq 0, j \notin (n^{i-1})\}$
    $n^i \leftarrow \gcd(n^i, m_i)$
    $C \leftarrow C \cup \{(m_i, n^i)\}$
  end while
  return C
end function

Algorithm 6 Enriques' theorem's values

Require: The characteristic exponents $\{(0,n),(m_1,n^1),\ldots,(m_k,n^k)\}$ of a Puiseux series.

Ensure: The integers $h^i_j, n^i_j \in \mathbb{N} \mid i = 1,\ldots,k, j = 0,\ldots,r(i)$ defined in section 2.7.

function ENRIQUESVALUES($\{(0,n),(m_1,n^1),\ldots,(m_k,n^k)\}$)
  $H \leftarrow \emptyset, N \leftarrow \emptyset$
  for $(m_i, n^i) \in \{(m_1,n^1),\ldots,(m_k,n^k)\}$ **do**
    $m \leftarrow m_i - m_{i-1}$ ▷ $(m_0, n^0) := (0, n)$
    $n \leftarrow n_{i-1}$
    while $n \neq 0$ **do**
      $H \leftarrow H \cup \{\lfloor m/n \rfloor\}$ ▷ $h^i_j := \lfloor m_j/n_j \rfloor$
      $N \leftarrow N \cup \{n\}$ ▷ $n^i_j := n_j$
      $r \leftarrow m \pmod n$
      $m \leftarrow n, n \leftarrow r$
    end while
  end for
  return (H, N)
end function

The values $m_i, n^i \in \mathbb{N}$ are the ones defined in section 2.6. Then, we can compute the values h_j^i and n_j^i from section 2.7 as explained in algorithm 6.

Algorithm 6 is a direct application of the definition of h_j^i in section 2.7 and the Euclidean algorithm. Next, we present algorithms 7 and 8, which, using the two previous algorithms, will compute the proximity matrix P of the singular points of an irreducible germ γ and its vector of multiplicities $e(\gamma)$.

Algorithm 7 Proximity matrix (irreducible)

Require: The values $h_j^i, i = 1, \ldots, k, j = 0, \ldots, r(i)$ from theorem 2.7.1 of an irreducible germ γ.

Ensure: The proximity matrix P of γ.

function ProximityMatrixIrreducible($\{h_j^i\}$)
  $n \leftarrow \sum_{i=1}^{k} \sum_{j=0}^{r(i)} h_j^i$
  $P \leftarrow I_n$
  $(P_{ij})_{i-j=1} \leftarrow -1$
  for $i \leftarrow 1$ **to** k **do**
    if $i = 0$ **and** $h_0^1 = 0$ **then**
      $j_0 \leftarrow 2$
    else
      $j_0 \leftarrow 1$
    end if
    $h_{r(i)}^i \leftarrow h_{r(i)}^i - 1$
    for $j \leftarrow j_0$ **to** $r(i)$ **do**
      $m \leftarrow \sum_{p=1}^{i-1} \sum_{q=0}^{r(p)} h_q^p + \sum_{q=0}^{j-1} h_q^i$
      for $l \leftarrow 1$ **to** h_j^i **do**
        $(P_{i'j'})_{i'=l+m, j'=m-1} \leftarrow -1$
      end for
    end for
    $h_{r(i)}^i \leftarrow h_{r(i)}^i + 1$
  end for
  return P
end function

For the special case of the curve $x = 0$, the y-axis, for which theorem 2.7.1 does not apply, the proximity matrix is trivial as it only contains free points. Algorithm 7 applies theorem 2.7.1 to fill the proximities in the proximity matrix. Clearly, the proximity matrix P will have ones in the diagonal and minus ones in the positions P_{ij} such that $i - j = 1$, hence we only have to use the values h_j^i to fill the remaining proximities corresponding to satellite points.

Algorithm 8 Vector of multiplicities (irreducible)

Require: The values $h_j^i, n_j^i \in \mathbb{N}, i = 1, \ldots, k, j = 0, \ldots, r(i)$ defined in section 2.7 of an irreducible germ γ.

Ensure: The vector of vector of multiplicities e of γ.

```
1: function VectorMultReduced(({h_j^i}, {n_j^i}))
2:     e ← ()
3:     for i = 0, ..., k do
4:         for j = 0, ..., r(i) do
5:             e ← e ‖ (n_j^i, ...., h_j^i, ..., n_j^i)
6:         end for
7:     end for
8:     return e
9: end function
```

Again, algorithm 8 is a direct application of theorem 2.7.1. We have denoted with $\|$ the concatenation of two vectors, and with () the empty vector. For the special case of the curve $x = 0$, which does not have a Puiseux series, the vector of multiplicities is trivial as only contains a single value $r \in \mathbb{C}$, the algebraic multiplicity of x in the equation of the curve.

Remark 2.8.1. For all the previous algorithms to work, all the Puiseux series returned by algorithm 4 have to contain enough terms to compute all the characteristic exponents. Indeed, checking the definition of characteristic exponents in section 2.6 and lemma 1.6.2, it is clear that the Newton-Puiseux algorithm will return, at least, series containing the term associated with the last characteristic exponent and all the preceding terms. This is what happened in example 1, where both Puiseux series have two characteristic exponents.

2.9 Comparing branches

We already know how to compute the Enriques diagram, or equivalently its proximity matrix, of the singular points of an irreducible germ. The next step is to compare the Enriques diagrams together to know how many infinitely near points each pair of branches share.

This comparison process can also be done using the information encoded in the Puiseux series of each branch. This process however turns out to be different for the free and the satellite points. The next three results completely determine how many points two branches share.

Let s be a Puiseux series of an irreducible germ γ. We will use the same notation as in theorem 2.7.1; however, it will be useful to slightly change the notation for

the points at the corners of the Enriques diagram: $p^i_{j,h^i_j+1} = p^i_{j+1,1}$ for $i = 1, \dots k$ and $j = 0, \dots, r(j) - 1$.

Proposition 2.9.1 (going through a free point, [4, 5.7.1]). *Fix a point $p^i_{0,l}$ on an irreducible germ γ, $1 \leq i \leq k+1, 1 \leq l \leq h^i_0 + 1$: $p^i_{0,l} = O$ if $i = l = 1$, otherwise it is a free point. Then, another irreducible germ ζ goes through $p^i_{0,l}$ if and only if it has a Puiseux series $\bar{s}$ such that*

$$[\bar{s}]_{(m_{i-1}+(l-1)n^{i-1})/n} = [s]_{(m_{i-1}+(l-1)n^{i-1})/n}$$

where n is the polydromy order of γ.

Also, $[\cdot]_\tau$ denotes the partial sum of degree τ of any power series s, and by $[\cdot]_{<\tau}$ we denote the sum of the monomials of of degree strictly less than τ. Notice that if the y-axis is tangent to γ, then the only point $p^1_{0,l}$ is $p^1_{0,1} = p^1_{1,1} = O$ and no claim is made about the free points $p^1_{1,l}$ on the y-axis.

Proposition 2.9.2 (still going through a free point, [4, 5.7.3]). *An irreducible germ ζ, with Puiseux series $\bar{s}$, goes through the last point depending on $m_i/n, p^i_{r(i),h^i_r(i)}$, and has a free point not on the y-axis in its first neighbourhood if and only if*

$$\bar{s} = [s]_{<m_i/n} + ax^{m_i/n} + \cdots$$

for some $a \in \mathbb{C}, a \neq 0$.

Let ζ be another irreducible germ of a curve that goes through $p^i_{1,1}$, i.e. the last free point depending on the i-th characteristic exponent in γ. According to proposition 2.9.2, ζ as a Puiseux series of the form:

$$\bar{s} = [s]_{<m_i/n} + bx^{\bar{m}/\bar{n}} + \cdots$$

where $\bar{n}$ is the polydromy order of $\bar{s}$ and $b \neq 0$. If $\bar{m}/\bar{n}$ is not a characteristic exponents, the point $p^i_{1,1}$ is not the last free points depending on the i-th characteristic exponent in ζ, so γ and ζ diverge in $p^i_{1,1}$ and they don't share any satellite point beyond that point. Hence, we can assume that $\bar{m} = \bar{m}_i$ is a characteristic exponent of ζ.

Proposition 2.9.3 (going through a satellite point, [4, 5.7.5]). *The irreducible germ ζ goes through $p^i_{t,j}$ if and only if $\bar{h}_0 = h^i_0, \dots, \bar{h}_{t-1} = h^i_{t-1}, \bar{h}_t \geq j-1$ if $\bar{r} > t$, or $\bar{h}_0 = h^i_0, \dots, \bar{h}_{t-1} = h^i_{t-1}, \bar{h}_t \geq j$ if $\bar{r} = t$*

This condition is simple saying that the part of the Enriques diagram of ζ depending on $\bar{m}_i/\bar{n}$ fits into the part of the Enriques diagram of γ on m_i/n, up to point $p^i_{t,j}$.

Assume that the y-axis is not tangent to γ, that is, $m_1/n > 1$. Let us call $\mathcal{J} = \mathcal{J}(m_1/n, \ldots, m_k/n)$ the set of indices allowed in a Puiseux series with characteristic exponents $m_1/n, \ldots, m_k/n$, that is,

$$\mathcal{J} = \{j \in \mathbb{Z} \mid j > 0 \text{ and } j \in (n^{i-1}) \text{ if } j < m_i, i = 1, \ldots, k\}$$

Then the corresponding Puiseux series s may be written as

$$s = \sum_{j \in \mathcal{J}} a_j x^{j/n},$$

where $a_j \in \mathbb{C}$ can be zero. On the other hand, let us call $\mathcal{F} = \mathcal{F}(\gamma)$ the set of all free points on γ, ordered by the natural order of infinitely near points. By mapping the r-th element of $\mathcal{J}$ to the r-th element of $\mathcal{F}$ we can get a one-to-one map from $\mathcal{J}$ to $\mathcal{F}$. Let us write $m_0 = n$, $m_{k+1} = \infty$, the we can associate to each coefficient a_j of s a point $p^i_{0,l}, l = (j - m_{i-1})/n^{i-1} + 1$. Conversely, given a free point $p^i_{0,l}$, $j = m_{i-1} + (l-1)n^{i-1}$.

We can now explicitly compute the proximity matrix of the singular points of an arbitrary germ of a curve ξ containing an arbitrary number of irreducible branches $\gamma_1, \ldots, \gamma_r$. The first steps towards that has been done in the last section by computing the Enriques diagrams of an irreducible branch. The second consists in computing the number of ordinary or infinitely near points that two branches will share. We will call this number the *contact number* a pair of branches and can be computed using the three propositions above.

Given a Puiseux series of an irreducible branch as in equation (2.5) we will introduce the following useful representations of the information encoded in a Puiseux series to compute contact numbers:

Definition 2.9.4. Using the above notation, we will encoded all the information in a Puiseux series in the following way

$$S = \left\{ \left(\{(a_j, j) \mid j \in (n), 0 \leq j < m_i\}, \{h^i_0, \ldots, h^i_{r(i)}\} \right) \mid i = 1, \ldots, k+1 \right\} \quad (2.6)$$

where $h^i_0, \ldots, h^i_{r(i)}$ are the numerical values from theorem 2.7.1 and $h^{k+1}_0 = \infty$ for $r(k+1) = 0$.

For branches tangent to the y-axis the first tuple of 2.6 is simply $(\{(0,0)\}$, $\{0, h^1_1, \ldots, h^1_{r(1)}\})$. For the special case of the y-axis curve, i.e. $x = 0$, which does not have a Puiseux series let us set $S = \left\{ (\{(0,0\}), \{0, \infty\}) \right\}$.

Algorithm 9 is pretty much a translation of propositions 2.9.1, 2.9.2 and 2.9.3. Once we know how to compute the contact number of two branches it will be useful for the next algorithm to store the contact numbers of all the branches in an

Algorithm 9 Contact number

Require: The Puiseux representation S and $\overline{S}$, as in 2.6, of two Puiseux series s and $\bar{s}$.

Ensure: The contact number c of the Puiseux series s and $\bar{s}$.

function ContactNumber($S, \overline{S}$)
  $c \leftarrow 1$
  for $i \leftarrow 1$ **to** $\min\{|S|, |\overline{S}|\}$ **do**
    $(\mathcal{F}_i, \mathcal{H}_i) \leftarrow \big(\{(a_j, j) \mid j \in (\nu(s)), 0 \leq j < m_i\}, \{h^i_0, \ldots, h^i_{r(i)}\}\big)$
    $(\overline{\mathcal{F}}_i, \overline{\mathcal{H}}_i) \leftarrow \big(\{(\bar{a}_\iota, \iota) \mid \iota \in (\nu(\bar{s})), 0 \leq \iota < \overline{m}_i\}, \{\bar{h}^i_0, \ldots, \bar{h}^i_{\bar{r}(i)}\}\big)$
    for $l \leftarrow 1$ **to** $\min\{|\mathcal{F}_i|, |\overline{\mathcal{F}}_i|\}$ **do** ▷ $|\mathcal{F}_i| = h^i_0, |\overline{\mathcal{F}}_i| = \bar{h}^i_0$
      if $(a_{j_l}, j_l) = (\bar{a}_{\iota_l}, \iota_l)$ **then** ▷ $(a_{j_l}, j_l) \in \mathcal{F}_i$
        $c \leftarrow c + 1$ ▷ $(\bar{a}_{\iota_l}, \iota_l) \in \overline{\mathcal{F}}_i$
      else
        return c
      end if
    end for
    if $|\mathcal{F}_i| \neq |\overline{\mathcal{F}}_i|$ **then**
      return c
    end if
    $h^i_{r(i)} \leftarrow h^i_{r(i)} - 1$
    $\bar{h}^i_{r(i)} \leftarrow \bar{h}^i_{r(i)} - 1$
    for $l \leftarrow 1$ **to** $\min\{|\mathcal{H}_i|, |\overline{\mathcal{H}}_i|\}$ **do** ▷ $|\mathcal{H}_i| = r(i), |\overline{\mathcal{H}}_i| = \bar{r}(i)$
      $c \leftarrow c + \min\{h^i_l, \bar{h}^i_l\}$ ▷ $\bar{h}^i_l \in \overline{\mathcal{H}}_i$
      if $h^i_l \neq \bar{h}^i_l$ **then** ▷ $h^i_l \in \mathcal{H}_i$
        return c
      end if
    end for
    if $|\mathcal{H}_i| \neq |\overline{\mathcal{H}}_i|$ **then**
      return c
    end if
  end for
  return c
end function

arbitrary germ of curve all together. Given an arbitrary germ ξ with irreducible branches $\gamma_1, \ldots, \gamma_r$ we define the *contact matrix* $C = (c_{ij})$ of ξ by setting c_{ij} to be equal to the contact number of γ_i and γ_j and let us set $c_{ii} = \infty$ for $i = 1, \ldots, r$.

2.10 Determining the equisingularity type

This section is devoted to present a new algorithm that computes the proximity matrix and the multiplicities of the singular points of a germ of a curve ξ given the proximity matrices, the vector of multiplicities and the contact matrix of its branches. Hence, the equisingularity type of ξ will be completely determined.

Algorithm 10 is a recursive algorithm that computes the proximity matrix of a cluster K given the proximity matrix of each branch in the cluster. If $p_i, i = 1, \ldots, l$ are the points in the first neighbourhood of the origin $O \in K$. Algorithm 10 computes the proximity matrix P of K from the proximity matrices P_i of K_i, where K_i is the sub-cluster $K_i \subset K$ with origin in p_i. Applying this idea recursively to each cluster $K_i, i = 1, \ldots, l$ one obtains algorithm 10.

Following the notation in algorithm 10: together with the proximity matrix P algorithm 10 returns a set of vectors

$$Q = \{(q_1^1, \ldots, q_{n_1}^1), \ldots, (q_1^r, \ldots, q_{n_r}^r) \mid q_j^i \in \mathbb{N}, i = 1, \ldots, r, j = 1, \ldots, n_i\}$$

where n_i is the number of points in the i-th branch. Each number q_j^i meaning that the j-th point in the i-th branch is now in position q_j^i inside P. This is useful for computing the vector of multiplicities of germs ξ from the vectors of multiplicities of $\gamma_1, \ldots, \gamma_r$.

- Lines 2-4 are the base case of the recursive algorithm, if there is only one branch, we already know its proximity matrix, and the position of the points is trivial. See remark 2.10.1 for more information on the base case.
- In lines 5-14 we compute which branches split in the current point, i.e. which branches belong to the same subcluster K_i. This information is stored in S as a partition of the branches going through the current points. Two branches will be in the same set inside S if they remain together after the current infinitely near point. The matrix J_r denotes the all-ones matrix. As we move one point down the Enriques diagram, we subtract one from all the contact numbers.
- Next, in lines 15-21, we first remove the current point from all the matrices $P_1, \ldots, P_r$, i.e. the first row and column. Then, we can call the function ProximityMatrix recursively for each partition of the branches in S, i.e. for each subcluster K_i.

Algorithm 10 Proximity matrix

Require: The proximity matrices $P_1, \dots, P_r$ and the contact matrix C of the irreducible branches $\gamma_1, \dots, \gamma_r$ of a curve ξ.

Ensure: The proximity matrix P of the curve ξ and the relative positions Q of the points in $\gamma_1, \dots, \gamma_r$ inside P.

function ProximityMatrix($\mathcal{P} = \{P_1, \dots, P_r\}, C$)
  if $r = 1$ **then**
    return $(P_1, \{1, \dots, r\})$
  end if
  $C \leftarrow C - J_r$
  $C' \leftarrow C$
  $B \leftarrow \{1, \dots, r\}, S \leftarrow \{\emptyset\}$
  while $B \neq \emptyset$ **do**
    $A \leftarrow \{i \in \mathbb{N} \mid c'_{1i} \neq 0\}$ ▷ $C' = (c'_{ij})$
    $\overline{A} \leftarrow \{i \in \mathbb{N} \mid c'_{1i} = 0\}$
    $S \leftarrow S \cup A$
    $B \leftarrow B - \overline{A}$
    $C' \leftarrow (c'_{ij})_{i \in \overline{A}, j \in \overline{A}}$
  end while
  $\mathcal{P} \leftarrow \left\{ \left(p^{(l)}_{ij}\right)_{i \neq 0, j \neq 0} \mid l = 1, \dots, r \right\}$ ▷ $P_l = \left(p^{(l)}_{ij}\right)$
  $R \leftarrow \emptyset$
  for $S_k \in S$ **do**
    $\mathcal{P}_k \leftarrow \{P_i \in \mathcal{P} \mid i \in S_k\}$
    $C_k \leftarrow (c_{ij})_{i \in S_k, j \in S_k}$
    $R \leftarrow R \cup$ ProximityMatrix($\mathcal{P}_k, C_k$)
  end for
  $n \leftarrow \sum_{(P'_k, Q_k) \in R} |P'_k| + 1$
  $(P, Q) \leftarrow (I_n, \emptyset)$
  $\alpha \leftarrow 1$
  for $\left(S_k, (P'_k, Q_k)\right) \in S \times R$ **do** ▷ $P = (p_{ij})$
    $(p_{ij})_{i = i' + \alpha, j = j' + \alpha} \leftarrow \left(p'^{(k)}_{i'j'}\right)$ ▷ $P'_k = \left(p'^{(k)}_{ij}\right)$
    $Q \leftarrow Q \cup \left\{(0, \mathbf{q}_{kl} + \alpha) \mid \mathbf{q}_{kl} \in Q_k\right\}$
    $\alpha \leftarrow \alpha + |P'_k|$
    for $l \in |S_k|$ **do**
      $(p_{i0})_{i \in \mathbf{q}_{kl}} \leftarrow \left(p^{(l)}_{i'0}\right)_{i' > 1}$
    end for
  end for
  $\sigma \leftarrow (s_{11}, s_{21}, \dots, s_{n_1 1}, \dots, s_{1k}, s_{2k}, \dots, s_{n_k k}, \dots)$ ▷ $s_{ik} \in S_k$
  return $(P, \sigma^{-1}(Q))$
end function

- Lines 22-32 merge the results from each call in line 20 to construct the final matrix P. We copy each sub-proximity matrix P'_k into P with the top-left corner in position (α, α) and updates the relative positions of the points in Q. Finally, line 30, updates the proximities of the current points. We use $|\cdot|$ to denote the size of a square matrix.
- Finally, lines 33-34, reorder Q so we return the positions of the points in the right order. Line 33, creates a permutation from the partitions in S, and we apply the inverse of that permutation to the set Q before returning.

Remark 2.10.1. We assume that all the proximity matrices are computed from the Puiseux series of ξ and that each pair of Puiseux series have been completely separated. This is true if one uses algorithm 4 with an equation of ξ as input. Because the Puiseux series are separated, after enough recursive steps of algorithm 10 we will hit line 3 and the algorithm will terminate. Notice, that this is no longer true if, for instance $\xi = \xi_1 + \xi_2$, and we compute the Puiseux series from the equations of ξ_1 and ξ_2 separately.

Remark 2.10.2. For algorithm 10 to work it is important that all the matrices $P_1, \ldots, P_r$ have size at least equal to the maximum contact number, i.e. the maximum entry of C. This is easy to achieve by expanding the matrices $P_1, \ldots, P_r$ with free points: $p_{ii}^{(l)} = 1, p_{ij}^{(l)} = -1, i > |P_l|, i - j = 1, l = 1, \ldots, r$.

Finally, algorithm 11 computes the vector of multiplicities of ξ. Algorithm 11 uses the set Q from algorithm 10 and the fact that,

$$e_p(\xi) = \alpha_i e_p(\gamma_1) + \cdots + \alpha_r e_p(\gamma_r)$$

for all the ordinary or infinitely points p, and where $\xi = \alpha_1\gamma_1 + \cdots + \alpha_r\gamma_r$.

2.11 Implementation details

We implemented algorithms 5, 6, 7, 8, 9, 10 and 11 in the Macaulay2 software using the implementation of Puiseux series and Newton-Puiseux algorithm from chapter 1.

In our implementation of the Newton-Puiseux algorithm we decided to use floating points arithmetic to make computations practical. However, this causes numerical approximations of the roots and hence, numerical approximations of the Puiseux series. We should justify that using floating points values does not affect the correctness of our program. From the way Puiseux series are constructed, the numerical approximations only affect the coefficients. Therefore, we can have a problem to determine when a coefficient is zero or not, and hence when an exponent will appear in the series. We can also have problems determining when two coefficients are equal or not, as it is required in line 7 of algorithm 9.

Algorithm 11 Vector of multiplicities

Require: The vector of multiplicities of each branch $\{e_1, \ldots, e_r\}$, the set Q from algorithm 10 containing the relative position of the points, the algebraic multiplicity of each branch $\{\alpha_1, \ldots, \alpha_r\}$, from algorithm 4, and the size n of the proximity matrix from algorithm 10.

Ensure: The vector of multiplicities e of ξ.

```
1: function VectorMult((\{e_1,...,e_r\}, Q, \{\alpha_1,...,\alpha_r\}, n))
2:     e ← (0, ....(n)..... , 0)
3:     for (q_1^i, ..., q_{n_i}^i) ∈ Q do
4:         for q_j^i ∈ (q_1^i, ..., q_{n_i}^i) do
5:             (e)_{q_j^i} ← (e)_{q_j^i} + α_i · (e_i)_j
6:         end for
7:     end for
8:     return e
9: end function
```

- In order to solve the first problem, we assume that the input bit precision is b and we perform all the operation with precision $2b$. Then we set as zero any $\alpha \in \mathbb{C}$ such that $|\alpha| < 2^{-b}$.

- The second problem is actually easier to solve. As we said in the previous chapter and in remark 2.10.1, we assume that the roots of each polynomial side F_Γ are separated numerically and all the Puiseux series come from the same output of algorithm 4. Therefore, even if we use an approximation of the root a_i, we can be sure that comparing it against other roots will yield the expected result.

The implementation of all these algorithms is straightforward once we can operate safely on Puiseux series, and no further comment is required.

By the time this work has been completed, we are only aware of another package that can compute the proximity matrix from the equation of a curve. The ALEXPOLY.LIB library from Singular [6] can compute the proximity matrix from a reduced equation, whether our implementation handles arbitrary equations. It is also important to notice that the ALEXPOLY.LIB does not use neither Puiseux series nor the Enriques theorem to compute the proximity matrices. It uses the Hamburger-Noether expansions of the equation, which is the analogue of Puiseux series over fields of finite characteristic [3].

Finally, we present an example of the usage and the output of these algorithms in our new Macaulay2 package. For the actual code, the reader is referred to Appendix C.

Example 5. Let's introduce the polynomial in example 4 in Macaulay2 and use our program to compute the proximity matrix and the vector of multiplicities:

```
i1 : ZZ[x, y];

i2 : f = x*y*(x - y)*(x^2 - y^3);

i3 : (P, e) = proximityMatrix(f, ExtraPoint => true)

o3 = (| 1  0  0 0  0 0 0 |, {| 1 |, | 1 |, | 2 |, | 1 |})
      | -1 1  0 0  0 0 0 |   | 1 |  | 0 |  | 1 |  | 0 |
      | 0  -1 1 0  0 0 0 |   | 1 |  | 0 |  | 0 |  | 0 |
      | -1 -1 0 1  0 0 0 |   | 0 |  | 0 |  | 1 |  | 0 |
      | 0  0  0 -1 1 0 0 |   | 0 |  | 0 |  | 1 |  | 0 |
      | -1 0  0 0  0 1 0 |   | 0 |  | 1 |  | 0 |  | 0 |
      | -1 0  0 0  0 0 1 |   | 0 |  | 0 |  | 0 |  | 1 |

i4 : sum e;

o4 = | 5 |
     | 2 |
     | 1 |
     | 1 |
     | 1 |
     | 1 |
     | 1 |
```

We request to the proximityMatrix function an extra point at the end of each branch so the dimensions of the output matrix match with example 4. We return the vector of multiplicities of each branch separately, but we can obtain the final vector of multiplicities by adding the vector of each branch.

Chapter 3

Base points of an ideal

In this section we will introduce the concept of linear system and the weighted cluster of base points associated to an ideal. The objective of the chapter is to describe an algorithm capable of computing the proximity matrix of that cluster and the multiplicities of generic elements of the ideal. In order to do that we first introduce the necessary concepts and we then state and prove a set of novel results to compute the weighted cluster of base points of an ideal. In order to make the computation feasible, we present a modified version of the Newton-Puiseux algorithms to obtain the weighted cluster of singular points of all the generators of an ideal.

3.1 Linear systems

We fix a point O on a smooth surface S and write $\mathcal{O} = \mathcal{O}_{S,O}, \mathcal{M} = \mathcal{M}_{S,O}$. A *linear system* of germs of curve is a set of germs of curve at a point O of the form $\mathcal{L} = \{\xi : f = 0 \mid f \in I - \{0\}\}$, defined by an ideal I of $\mathcal{O}$. Because $\mathbb{C}[[x, y]]$ is Noetherian, any ideal of $\mathcal{O}$ is finitely generated, and one may take a system of generators $f_1, \ldots, f_r$ of I and then $\mathcal{L}$ is the set of all germs of curve that have an equation $\sum_{i=1}^{r} a_i f_i = 0$ for some $a_i \in \mathcal{O}$.

We are in particular considering the linear system defined by the ideal $I = (1)$, clearly the only linear system containing the empty germ. It will be called the *irrelevant linear system*. If g is the greatest common divisor of all the elements in (or just the generators of) a linear system $\mathcal{L}$ of $\mathcal{O}$, one may write $\mathcal{L} = g\mathcal{L}'$ where $\mathcal{L}'$ is a linear system and the greatest common divisor of all elements in $\mathcal{L}'$ is 1. The germ $\xi : g = 0$ is called the *fixed part* of the linear system $\mathcal{L}$ defined by I. If $g = 1$ we say that $\mathcal{L}$ has no fixed part.

Let us set $e = e_O(\mathcal{L}) = \min\{e_O(\xi) \mid \xi \in \mathcal{L}\}$. In particular, $e_O(\mathcal{L}) = 0$ if and only if $\mathcal{L}$ is irrelevant. We will call $e = e_O(\mathcal{L})$ the multiplicity of $\mathcal{L}$ at O. It is clear that, if $\mathcal{G}$ is a set of germs generating $\mathcal{L}$, then also $e_O(\mathcal{L}) = \min\{e_O(\xi) \mid \xi \in \mathcal{G}\}$.

If p is a point in the first neighbourhood of O, denote by $\mathcal{O}_p$ its local ring, by $\phi_p : \mathcal{O} \longrightarrow \mathcal{O}_p$ the morphism induced by blowing up, and by z an equation at p of the exceptional divisor. The ideal generated by $z^{-e}\phi_p(I)$ obviously does not depend on the particular equation z we are using and defines a linear system $\mathcal{L}_p$ of germs at p that is generated by the virtual transforms, as defined in 2.5.1, of any set of germs generating $\mathcal{L}$, O being taken with virtual multiplicity e. The linear system $\mathcal{L}_p$ will be called the *transform of $\mathcal{L}$ with origin at p*. We extend this definition to all points on the successive neighbourhoods by using induction on the order of the neighbourhoods. We write $e_p(\mathcal{L}) = e_p(\mathcal{L}_p)$ and call this number the *multiplicity* of $\mathcal{L}$ at p.

The first important result is that all but finitely many transforms $\mathcal{L}_p$ of any given linear system $\mathcal{L}$ with no fixed part are irrelevant, or, equivalently, that $e_p(\mathcal{L}) = 0$ for all but finitely many points p.

Proposition 3.1.1 ([4, 7.2.3])**.** *If $\mathcal{L}$ is a linear system with no fixed part, for all but finitely many points p infinitely near to O its transform $\mathcal{L}_p$ is irrelevant.*

Now let $\mathcal{L}$ be a non-irrelevant linear system with no fixed part. We define the *weighted cluster of base points* of $\mathcal{L}$, $BP(\mathcal{L})$ by taking the points p equal or infinitely near to O for which $\mathcal{L}_p$ is not irrelevant, each p taken with virtual multiplicity $e_p(\mathcal{L})$. $BP(\mathcal{L})$ is actually a weighted cluster: if K denotes its set of points, it is non-empty because $\mathcal{L}$ is non-irrelevant and is finite by proposition 3.1.1; furthermore, if $p \in K$ then also $q \in K$ for all $q < p$ because all transforms of an irrelevant linear system are irrelevant too.

The points $BP(\mathcal{L})$ will be called the *base points* of $\mathcal{L}$. Note that p is a base point of $\mathcal{L}$ if and only if $e_p(\mathcal{L}) > 0$. In such a case $e_p(\mathcal{L})$, the virtual multiplicity of p in $BP(\mathcal{L})$, will be called the *multiplicity of the base point p*. Sometimes $BP(\mathcal{L})$ is also called the weighted cluster of base points of the ideal I (defining $\mathcal{L}$) and it is denoted by $BP(I)$. Both notations are common and their use depend on whether the context is more geometric or algebraic.

Remark 3.1.1. It follows from the definition that all germs in $\mathcal{L}$ go through $BP(\mathcal{L})$. It is also clear from the definition that if a point p effectively belongs to all germs in $\mathcal{L}$, then it is a base point. Even more, since the virtual transforms always contain the strict ones, if $e_p(\xi) \geq e > 0$ for all $\xi \in \mathcal{L}$, the virtual multiplicity of p in $BP(\mathcal{L})$ is non-less than e. The reader may notice that the converse is far from true as there may be germs in $\mathcal{L}$ failing to effectively go through some base points.

The following theorem completely characterizes the behaviour of germs of curve going through $BP(\mathcal{L})$.

Theorem 3.1.2 ([4, 7.2.13]). *Let $\mathcal{L}$ be a linear system of germs of curve at O with no fixed part and T a finite set of points infinitely near to O, no one a base point of $\mathcal{L}$. All germs on $\mathcal{L}$ go through $BP(\mathcal{L})$. The set of germs going sharply through $BP(\mathcal{L})$ and missing all points in T is a non-empty Zariski-open set, in particular, they are reduced and have the same equisingularity type.*

The following results will result useful in the next section.

Corollary 3.1.3 ([4, 7.2.16]). *A linear system $\mathcal{L}$ with no fixed part may be generated by finitely many germs going sharply through $BP(\mathcal{L})$, any two sharing no point other than the base ones.*

Lemma 3.1.4 ([4, 7.2.6]). *Let $\mathcal{G}$ be a set of germs that generate $\mathcal{L}$. If p is a base point, the virtual transform with origin at p of some $\xi \in \mathcal{G}$ has multiplicity $e_p(\mathcal{L})$ at p. If q is not a base point and lies in the first neighbourhood of a base point p, then the virtual transform with origin at q of some germ $\mathcal{G}$ is empty.*

3.2 Constructing the cluster of base points

In this section we will generalize the results in [1] for the case of an ideal with an arbitrary number of generators. First let us state some results that will be useful in the sequel and that follow directly from the definitions.

Proposition 3.2.1. *Let $\mathcal{K} = (K, \nu)$ a weighted cluster with origin at O and let $p \in K$. Let ξ any germ of curve with origin at O. Then:*

1. $\nu_p(\xi) = e_p(\xi) + \sum_{p \to q} \nu_q(\xi)$.
2. $e_p(\hat{\xi}_p) = v_p(\xi) - \sum_{p \to q} \nu_q$, *where $\hat{\xi}_p$ is the virtual transform of ξ at p relative to $\mathcal{K}_{\leq p}$.*
3. *ξ goes sharply through $\mathcal{K}$ if and only if $\nu_p(\xi) = \nu_p$ for all $p \in K$.*

where $\mathcal{K}_{\leq p}$ is the weighted cluster that consists of all points preceding p in $\mathcal{K}$ with the same weights ν.

Let O be a point on a smooth surface, and let $\xi_i : f_i = 0, i = 1, \ldots, r$ be arbitrary germs of curve at O. Consider the linear system $\mathcal{L}$ of the germs of curves $\xi_1, \cdots, \xi_r$ and the ideal $I = (f_1, \ldots, f_r)$. Consider $BP(I) = (B, \beta)$ weighted by the virtual values β. We will describe a weighted cluster $\mathcal{K} = (K, v)$, with v virtual

values, in terms of the infinitely near points and values of $\xi_1, \cdots, \xi_r$, and we will prove that $\mathcal{K}$ equals $BP(\mathcal{L})$.

First we assign to each point p equal or infinitely near to O two integers $v_p = \min_i\{v_p(\xi_i)\}$ and $h_p = \sum_{p \to q} v_q$, $h_O = 0$, the second one defined using induction on the order of neighborhood p is belonging to. Define K as the set of points p equal or infinitely near to O such that $h_p < v_p$, and for each $p \in K$ define $e_p = v_p - h_p$. We will see that if p belongs to K then all the points preceding p also belong to K. Hence, once it is proved that K is finite, we would have defined a weighted cluster $\mathcal{K} = (K, v) = (K, e)$ with virtual multiplicities e and virtual values v.

First, let us prove a technical lemma that will be useful in the sequel.

Lemma 3.2.2. *Let $f \in I = (f_1, \cdots, f_r)$, then*

$$v_p(f) \geq \min\{v_p(f_1), \cdots, v_p(f_r)\}$$

for any point ordinary or infinitely point p.

Proof. Since $v_p(f)$ is a valuation in $\mathcal{O}$ (see [4], 4.5), for any $g_1, g_2 \in I$ we have,

$$v_p(g_1 + g_2) \geq \min\{v_p(g_1), v_p(g_2)\},$$

$$v_p(g_1 g_2) = v_p(g_1) + v_p(g_2).$$

Then, since $f = g_1 f_1 + \cdots + g_r f_r$, with $g_1, \cdots, g_r \in \mathcal{O}$,

$$\begin{aligned} v_p(f) = v_p(g_1 f_1 + \cdots + g_r f_r) &\geq \min\{v_p(g_1 f_1), \cdots, v_p(g_r f_r)\} \\ = \min_i\{v_p(g_i) + v_p(f_i)\} &\geq \min\{v_p(f_1), \cdots, v_p(f_r)\} \end{aligned}$$

where in the last inequality we used that $v_p(g) \geq 0, \forall g \in \mathcal{O}$. □

Next, we should prove that our definition of the cluster $\mathcal{K}$ does not depend on the generators of the ideal.

Lemma 3.2.3. *The virtual values v_p do no depend on the generators $f_1, \ldots, f_r$ of the ideal I. Namely, $v_p = \min_{f \in I}\{v_p(f)\}$.*

Proof. It is clear that $v_p = \min_i\{v_p(f_i)\} \geq \min_{f \in I}\{v_p(f)\}$ since $\{f_1, \ldots, f_r\} \subset I$. On the other hand, by lemma 3.2.2, $v_p(f) \geq \min_i\{v_p(f_i)\}$, for all $f \in I$, hence $\min_{f \in I}\{v_p(f)\} \geq v_p$ and the result follows. □

Lemma 3.2.4. *For any p equal or infinitely near to O, it holds $h_p \leq v_p$.*

Proof. For $p = O$ it is clear. If p is free, take $q \leftarrow p$, then by proposition 3.2.1(1)

$$h_p = v_q = \min_i\{v_q(\xi_i)\} \leq \min_i\{v_p(\xi_i)\} = v_p.$$

If p is satellite, let $\{q, q'\}$ be the points p is proximate to. Then

$$\begin{aligned} h_p = v_q + v_{q'} &= \min_i\{v_q(\xi_i)\} + \min_i\{v_{q'}(\xi_i)\} \\ &\leq \min_i\{v_q(\xi_i) + v_{q'}(\xi_i)\} \end{aligned}$$

and again, by the same equality as before, it follows the desired inequality. □

Lemma 3.2.5. *If there exists a generator ξ_i such that $h_p = v_p = v_p(\xi_i)$, then $e_p(\xi_i) = 0$ and $v_q = v_q(\xi_i)$ for any $q \leftarrow p$.*

Proof. Suppose first p is free and take $q \leftarrow p$. Since $v_p(\xi_i) = v_p = h_p = v_q = \min_j\{v_q(\xi_j)\}$, we infer that $v_p(\xi_i) \leq v_q(\xi_i)$. Then by proposition 3.2.1(1) the equalities $v_p(\xi_i) = v_q(\xi_i)$, $e_p(\xi_i) = 0$ follow, and hence $v_q = v_q(\xi_i)$.

Assume now p is satellite and take $q \leftarrow p$, $q' \leftarrow p$. Since $v_p(\xi_i) = v_p = h_p = v_q + v_{q'}$, we infer that $v_p(\xi_i) \leq v_q(\xi_i) + v_{q'}(\xi_i)$. By proposition 3.2.1(1) the equalities $v_p(\xi_i) = v_q(\xi_i) + v_{q'}(\xi_i)$, $e_p(\xi) = 0$ follow. Now, if $v_q < v_q(\xi_i)$ or $v_{q'} < v_{q'}(\xi_i)$, then $h_p = v_q + v_{q'} < v_p(\xi_i)$, a contradiction. □

Proposition 3.2.6. *If $p \in K$, then any point q preceding p belongs to K.*

Proof. Let us show the converse: if $q \notin K$, i.e. $h_q = v_q$, then $h_p = v_p$ for any p in the first neighborhood of q, and inductively $h_p = v_p$, i.e. $p \notin K$, for any point p infinitely near to q.

Let p be a point in the first neighborhood of q, $q \notin K$. Assume the minimum v_q is reached by the germ ξ_i, thus $h_q = v_q = v_q(\xi_i)$.

If p is satellite, take $q' \leftarrow p$ (we already know $q \leftarrow p$). Then by lemma 3.2.5

$$h_p = v_q + v_{q'} = v_q(\xi_i) + v_{q'}(\xi_i) = v_p(\xi_i).$$

Hence and by lemma 3.2.4 $h_p = v_p(\xi_i) = v_p$.

If p is free, the same reasoning is valid, by taking $v_{q'} = v_{q'}(\xi_i) = 0$. □

Proposition 3.2.7. *If p is a point of B, then p belongs to K and the equality of virtual values $\beta_p = v_p$ is satisfied.*

Proof. Take $p \in B$. Let us first prove that $\beta_p = v_p$, and then we will infer that $h_p < v_p$, i.e. $p \in K$.

Taking $\xi : \sum_{i=0}^{r} g_i f_i = 0, g_1, \ldots, g_r \in \mathcal{O}$ as any germ going sharply through $BP(\mathcal{L})$, we obtain by proposition 3.2.1(3) and lemma 3.2.2

$$\beta_p = \ v_p(\xi) \geq \min_i\{v_p(\xi_i)\} = \min_i\{v_p(f_i)\} = v_p.$$

Now, by corollary 3.1.3, we admit a system of generators $(h_1, \cdots, h_s) = I$ such that $\zeta_i : h_i = 0$ goes sharply through $BP(I)$. Then,

$$v_p = \min_i\{v_p(f_i)\} \geq \min_i\{v_p(h_i)\} = \min_i\{v_p(\zeta_i)\} = \beta_p,$$

after applying again proposition 3.2.1(3) and lemma 3.2.2 to the elements f_i expressed in terms of $h_1, \cdots, h_s$.

Therefore the equality $v_p = \beta_p$ follows.

Since the same equality holds for all the points preceding p, we infer

$$v_p - h_p = \beta_p - \sum_{p \to q} \beta_q = b_p > 0,$$

i.e. $v_p > h_p$, since b_p it is the virtual multiplicity at p of the strictly consistent weighted cluster $BP(\mathcal{L})$. □

Theorem 3.2.8. *The weighted clusters $BP(\mathcal{L})$ and $\mathcal{K}$ are equal. In particular, K is finite.*

Proof. From 3.2.7, it only remains to show the inclusion $K \subset B$. Let $p \in K$, we will prove, by induction on the order of neighborhood p is belonging to, that $p \in B$. For $p = O$, it is clear that p belongs to both K and B. Now assume that the assertion is true for all the points preceding p (which are in K by proposition 3.2.6) and hence, by hypothesis of induction, they all belong to B.

Let $q \in B$ be the antecessor of p. By lemma 3.1.4, $p \in B$ if and only if $0 < \min_{\zeta \in \mathcal{P}}\{e_p(\widehat{\zeta_p})\}$, where $e_p(\widehat{\zeta_p})$ is the virtual multiplicity of the germ ζ at p relative to the weighted cluster $BP(\mathcal{L})_{\leq q}$. This is equivalent, by proposition 3.2.1(2), to

$$\min_{\zeta \in \mathcal{L}}\{v_p(\zeta)\} > \sum_{s \to p} \beta_s.$$

By lemma 3.2.3, $v_p = \min_{\xi \in \mathcal{L}}\{v_p(\xi)\}$. Thus, applying proposition 3.2.7 to the points preceding p, p belong to B if and only if

$$v_p > \sum_{p \to s} \beta_s = \sum_{p \to s} v_s = h_p.$$

□

Corollary 3.2.9. *Given an ideal $I = (f_1, \cdots, f_r)$, $BP(I) = (B, \beta)$. Any cluster of infinitely near points K' weighted by the values v, $v(p) = v_p = \min_i\{v_p(f_i)\}, \forall p \in K'$, or alternatively weighted by the multiplicities e, $e(p) = e_p = v_p - \sum_{p \to q} v_q, \forall p \in K'$, satisfying $e_p \neq 0$ for any $p \in K'$ is a subcluster of B.*

Proof. Since by definition $K = \{p \in \mathcal{N}_O \mid e_p > 0\}$, clearly $K' \subseteq K$. Now applying theorem 3.2.8, the result follows. □

3.3 An algorithm for computing base points

In this section we will describe an algorithm that, based on the results in the previous section, will compute the weighted cluster of base points of an ideal.

Let $I = (f_1, f_2, \ldots, f_r)$ be an ideal of $\mathcal{O}$ and we want to compute $BP(I) = (B, \beta)$. Let us assume for the moment that $\mathcal{K}' = (K', v)$ is a weighted cluster with system of values $v_p = \min_i\{v_p(f_i)\}$ for any $p \in K'$ and satisfying two additional properties: first, $K' \subset B$, and second, any point $p \in B$ singular for $\xi : f_1 \cdots f_r = 0$ is already in K'. Assuming this we can know that any missing base point, not already in K', will lie only in one or in zero generators. Indeed, if the point were in two or more generators, it would be singular in ξ.

Next, we state and prove two novel results that are useful for adding the remaining points to K' given only the virtual values of the generators in each infinitely near point.

The first result states that all the free points in $BP(I)$ lie on the generators.

Lemma 3.3.1. *If q is free and does not lie on any $\xi_1, \ldots, \xi_r$ then $q \notin BP(I)$.*

Proof. Let $q \to p$. If $q \notin \xi_j, \forall j$: then $v_q(\xi_j) = v_p(\xi_j)$ for $j = 1, \ldots r$ and $v_q = \min_j\{v_q(\xi_j)\} = \min_j\{v_p(\xi_j)\} = v_p$, hence $e_q = v_q - v_p = 0$ and $q \notin BP(I)$. □

The next result characterizes the free points in $BP(I)$ non-singular for ξ.

Proposition 3.3.2. *Keep the above hypothesis on K'. Let $q \notin K'$ be a free point proximate to $p \in K'$. Then q is in $BP(I)$ if and only if there exists a unique generator f_i such that $v_p(f_i) < v_p(f_j), \forall j \neq i$.*

Proof. We shall apply theorem 3.2.8 to characterize whether q belongs to $BP(I)$. By definition $v_q = \min_j\{v_q(f_j)\}$ and $v_q(f_j) = e_q(f_j) + v_p(f_j)$, for $j = 1, \ldots, r$. Let us prove first the reverse implication. Then:

- If $q \in f_j, j \neq i$: we know that $q \notin f_k$, for $k \neq j$. Then $v_q(f_j) = e_q(f_j) + v_p(f_j)$, with $e_q(f_j) > 0$ and $v_q(f_k) = v_p(f_k)$ for $k \neq j$. Hence $v_q = \min_l\{v_q(f_l)\} = \min_k\{v_p(f_k), e_q(f_j) + v_p(f_j)\} = v_p(f_i) = v_p$ and $q \notin BP(\mathcal{L})$.
- If $q \in f_i$: we know that $q \notin f_j$ for $j \neq i$. Then $v_q = \min_j\{v_p(f_j), e_q(f_i) + v_p(f_i)\} > v_p(f_i)$ which implies that $e_q > 0$ and hence $q \in BP(\mathcal{L})$ as we wanted to prove.

For the other implication we know that q can only belong to one generator f_i. This means that $e_q = v_q - v_p > 0$ and comparing $v_p = \min_j\{v_p(f_j), v_p(f_i)\}$ and $v_q = \min_j\{v_p(f_j), v_p(f_i) + e_q(f_i)\}$ we see that $v_p(f_i) = v_p$ and hence, it is unique. □

Corollary 3.3.3. *Let $p \in BP(I)$ such that $v_p = v_p(f_i)$ for a unique generator f_i and define $w_p = \min_{j \neq i}\{v_p(f_j)\}$. Then, there are $(w_p - v_p)$ free base points in the first neighbourhood of p with the same multiplicity.*

Proof. By definition $v_q = \min\{w_p, e_q(f_i) + v_p(f_i)\}$. If $v_q = w_p$ we are done, otherwise $v_q = e_q(f_i) + v_p(f_i)$ and we can use proposition 3.3.2 to add a free base point in the first neighbourhood of p as many times as $\lceil (w_p - v_p)/e_q(f_i) \rceil$. □

Keeping the above hypothesis on K', the next result deals with the missing satellite base points not already in K'. Notice that these missing satellite points will not lie on any generator, otherwise they would belong to the singular points of $\xi : f_1 \cdots f_r = 0$,

Proposition 3.3.4. *Keep the above hypothesis on K'. Let $q \notin K'$ be a satellite point proximate to $p, p' \in K'$. Then q is in $BP(I)$ if and only if for each generator f_i either $v_p(f_i) > v_p$ or $v_{p'}(f_i) > v_{p'}$.*

Proof. Let us start by proving the converse implication. We know that $q \notin f_j$ for $j = 1, \ldots, r$, otherwise q would be in K'. We want to see that $e_q = v_q - v_p - v_{p'} > 0$. Then $v_q = \min_j\{v_q(f_j)\} = \min_j\{v_p(f_j + f_j)\}$ and the last equality is true because $e_q(f_j) = 0$. By hypothesis, $v_p(f_j) + v_{p'}(f_j) > v_p + v_{p'}$, hence $v_q > v_p + v_{p'}$ as we wanted.

For the other implication, let us assume the contrary, that is, there exists a generator f_i such that $v_p(f_i) = v_p$ and $v_{p'}(f_i) = v_p$. We know that $q \notin f_i$, otherwise it would be in K. By definition, $v_q = \min_j\{v_q(f_j)\} = \min_j\{v_p(f_j) + v_{p'}(f_j)\} = v_p(f_i) + v_{p'}(f_i) = v_p + v_{p'}$, implying that $e_q = 0$, which is a contradiction with the fact that $q \in BP(I)$. □

Finally, we present the novel procedure to compute the weighted cluster of base points of and ideal I. The procedure works as follows:

1. Start with a set of generators $\{f_1, \ldots, f_r\}$ of the ideal I and compute $f = f_1 \cdots f_r$.

2. Find the cluster $\overline{K}$ of singular points of f and the system of virtual values $\{v_p(f_i)\}_{p \in \overline{K}}$ of any generator $f_i, i = 1, \cdots, r$.

3. Compute, recursively on the order of neighbourhood p is belonging to, $v_p = \min_i\{v_p(f_i)\}$ and $e_p = v_p - \sum_{p \to q} v_q$. Define $\mathcal{K}' = (K', v)$ with $K' \subset \overline{K}$ containing the points $p \in \mathcal{K}'$ such that $e_p \neq 0$ and the virtual values $v(p) := v_p$.

4. Define $\mathcal{K}'' = (K'', v)$ from $\mathcal{K}'$ by adding the missing free points using proposition 3.3.2 weighted by the values $v(p) = v_p = \min_i\{v_p(f_i)\}$ for each new $p \in K'' \setminus K'$.

5. Define $\mathcal{K} = (K, v)$ from $\mathcal{K}''$ by adding the missing satellite points using proposition 3.3.4, weighted by the values $v(p) = v_p = \min_i\{v_p(f_i)\}$ for each new $p \in K \setminus K''$.

6. Return $\mathcal{K}$, the weighted cluster of base points.

Theorem 3.3.5. *Keep the above notations. The above procedure actually computes $BP(I)$, the weighted cluster of base points of the ideal I.*

Proof. First of all, note that by computing the cluster of singular points of $f = f_1 \cdots f_r$ we are capturing all the singular infinitely near points in the germs of a curve $\xi = \xi_1 + \cdots + \xi_r$. By corollary 3.2.9, after removing the points $e_p = 0$ from $\overline{K}$, the resulting cluster K' is inside B. Since this set K' fulfils the hypothesis of 3.3.2 and proposition 3.3.4, we can use them to add the remaining base points.

At this point we have added all the base points: if we had to add a missing base points in the first neighbourhood of a point already in K', it would have to be free as we have added all the missing satellites in the last step. This free point would have to be in a generator, by lemma 3.3.1, and it would have to be after one of the new satellite points, otherwise we would have added it in the fourth step. But that is impossible because the new satellite points cannot lie on a generator and hence, neither can do any of its successors. □

3.4 Newton-Puiseux expansion for ideals

In this section we focus on an efficient way of computing the step 2 of the previous algorithm. From chapter 1 we know that we can use the Newton-Puiseux algorithm to compute the Puiseux factorization of a given polynomial and in chapter 2 we saw how to use the Puiseux factorization to compute the cluster of the desingularization.

In order to compute $v_p(f_i)$ for each $p \in \overline{K}$ and each generator f_i, we need to compute

$$v_p(f_i) = \alpha_{i1} v_p(g_{i1}) + \alpha_{i2} v_p(g_{i2}) + \cdots + \alpha_{ir_i} v_p(g_{ir_i}) \tag{3.1}$$

where $f_i = g_{i1}^{\alpha_{i1}} g_{i2}^{\alpha_{i2}} \cdots g_{ir_i}^{\alpha_{ir_i}}, g_{ij} \in \mathbb{C}[[x]][y]$ for $j = 1, \ldots, r_i$. We know from proposition 2.5.5 how to compute $v_p(g_{i1})$ from the multiplicities of g_{ij}. At the same time, we can compute those multiplicities from all the Puiseux series as we saw in algorithm 11. Thanks to the novel algorithm 4 we can also compute the algebraic multiplicities $\alpha_{i1}, \cdots, \alpha_{ir_i}$ of each branch $g_{i1}, \cdots, g_{r_i}$ of a fixed generator f_i.

The main problem here is that when we compute the $f = f_1 \cdots f_r$ we are losing the information about which generator each branch belongs to. This is due to the fact the several generators of the ideal can contain the same branch.

We present a novel algorithm, based on algorithm 4 that can keep track of repeated branches (i.e. Puiseux series) across different generators and their algebraic multiplicity in each generator. Then, once we know the multiplicity with which each branch belongs to a generator, we can compute equation (3.1) easily. For the detailed description of the algorithm see algorithm 12.

Algorithm 12 Newton-Puiseux algorithm for an ideal

Require: A list of bivariate polynomial $f_1, f_2, \ldots, f_r$.

Ensure: A set of pairs (s_j, A_j) where s_j is a Puiseux series of the product $f_1 f_2 \cdots f_r$ completely separated from the rest and the sets $A_j = \{(i_{j_1}, \alpha_{j_1}), \ldots, (i_{n_j}, \alpha_{n_j})\}$ contain pairs (i, α) meaning that $(y - s_j)^\alpha | f_i$.

1: **function** NewtonPuiseux($\{f_1, f_2, \ldots, f_r\}$)
2: $f \leftarrow f_1 f_2 \cdots f_r$
3: $S \leftarrow \emptyset$
4: $\mathbf{N}(f) \leftarrow$ NewtonPolygon(f) ▷ $\mathbf{N}(f) = \{(\alpha_0, \beta_0), \ldots, (\alpha_k, \beta_k)\}$
5: **if** $\alpha_0 > 0$ **then**
6: $\{\mathbf{N}(f_1), \ldots, \mathbf{N}(f_r)\}, \leftarrow \{$NewtonPolygon($f_i$) $\mid i = 1, \ldots r\}$
7: $S \leftarrow \{(x, \{(i, \alpha_{0i}) \mid (\alpha_{0i}, \beta_{0i}) \in \mathbf{N}(f_i), \alpha_{0i} \neq 0, i = 1, \ldots, r\})\}$
8: **end if**
9: $\widetilde{f} \leftarrow f / \gcd(f, f_y)$ ▷ $f_y(x, y) := \frac{d}{dy} f(x, y)$
10: $L \leftarrow \emptyset$
11: **for** $f_i \in \{f_1, f_2, \ldots, f_r\}$ **do**
12: **for** $(g_{ij}, \alpha_{ij}) \in$ SquareFreeFactorization(f_i) **do**
13: $L \leftarrow L \cup \{(g_{ij}, \alpha_{ij}, i)\}$
14: **end for**
15: **end for**
16: **return** $S \cup$ NewtonPuiseuxLoop($\widetilde{f}, L$)
17: **end function**

18: **function** NewtonPuiseuxLoop(f, L)
19: $N \leftarrow \{\text{NewtonPolygon}(g_{ij}) \mid (g_{ij}, \alpha_{ij}, i) \in L\}$
20: $L \leftarrow \{(g_{ij}, \alpha_{ij}, i) \in L \mid h(\mathbf{N}(g_{ij})) \neq 0, \mathbf{N}(g_{ij}) \in N\}$
21: $\mathbf{N}(f) \leftarrow \text{NewtonPolygon}(f)$ ▷ $\mathbf{N}(f) = \{(\alpha_0, \beta_0), \ldots, (\alpha_k, \beta_k)\}$
22: **if** $\beta_0 = 1$ **then** ▷ $h(\mathbf{N}(f)) = \beta_0$
23: **return** $\{(0, \{(i, \alpha_{ij}) \mid (g_{ij}, \alpha_{ij}, i) \in L\})\}$
24: **end if** ▷ $L = \{(g_{ij}, \alpha_{ij}, i) \mid g_{ij} \in \mathbb{C}[x][y], \alpha_{ij}, i \in \mathbb{N}\}$
25: $S \leftarrow \emptyset$
26: **if** $\beta_k > 0$ **then**
27: $S \leftarrow \{(0, \{(\alpha_{ij}, i) \in \mathbb{N}^2 \mid \beta_{k_i} > 0,\ (\alpha_{k_i}, \beta_{k_i}) \in \mathbf{N}(g_{ij}) \in N,$
$(g_{ij}, \alpha_{ij}, i) \in L\})\}$
28: **end if**
29: **for** $(\alpha_i, \beta_i), (\alpha_{i+1}, \beta_{i+1}) \in \mathbf{N}(f)$ **do**
30: $n \leftarrow \beta_i - \beta_{i+1}$
31: $m \leftarrow \alpha_{i+1} - \alpha_i$
32: $k \leftarrow \beta_i \alpha_{i+1} - \alpha_i \beta_{i+1}$
33: $\Gamma \leftarrow nx + my - k$ ▷ $\Gamma \in \mathbb{Z}[x, y]$
34: $F_\Gamma \leftarrow \sum_{(\alpha,\beta)\in\Gamma} A_{\alpha,\beta} Z^{\beta-\beta_0}$ ▷ $F_\Gamma \in \mathbb{C}[Z]$
35: **for** $a \in \{F_\Gamma(z^n) = 0 \mid z \in \mathbb{C}\}$ **do**
36: $\overline{f} \leftarrow x^{-k} f(x^n, x^m(a + y))$
37: $\overline{L} \leftarrow \{(x^{-k} g_{i'j}(x^n, x^m(a + y)), \alpha_{i'j}, i') \mid (g_{i'j}(x, y), \alpha_{i'j}, i') \in L\}$
38: $\overline{S} \leftarrow \text{NewtonPuiseuxReduced}(\overline{f}, \overline{L})$
39: $S \leftarrow S \cup \{(x^{m/n}(a + \bar{s}(x^{1/n})), A) \mid (\bar{s}, A) \in \overline{S}\}$
40: **end for**
41: **end for**
42: **return** S
43: **end function**

The basic idea behind algorithm 12 is the same as in algorithm 4. We apply the traditional Newton-Puiseux algorithm to both the product and to each square-free factor of each generator but also keeping track of the generator each square-free factor belongs to. When we reach the base case in line 22, we know that the current branch has been separated from the rest and we can return the zero Puiseux series, as usual, and both the generator index and the algebraic multiplicity of each reduced factor in the set L.

3.5 Implementation details

As all the other algorithms in this work, these two new algorithms have been implemented in Macaulay2. However, because these algorithms are novel we have not be able to compare them against any other package or library. The Macaulay2 code of algorithm 12 and the algorithm for computing the base points can be found in Appendices B and C, respectively.

To end this section, we will show a basic example of use of the algorithm that computes the base points of an ideal in Macaulay2.

Example 6. Consider the two-dimensional ideal $I = (x^2, y^3)$.

```
i1 : I = ideal(x^3, y^2)

              3   2
o2 = ideal (x , y )

i3 : basePoints I

o3 = (| 1  0  0 |, | 2 |, | 2 |)
      | -1 1  0 |  | 1 |  | 3 |
      | -1 -1 1 |  | 1 |  | 6 |
```

In our implementation we are returning both the virtual multiplicities and the virtual values defining $BP(I)$. We can see that although the generators of the ideal are smooth, the cluster of base points contains a satellite point in the second neighbourhood of the origin. The reader can check that $f = x^2 - y^3 \in I$ goes sharply through $BP(I)$.

Appendix A

Macaulay2 code: Puiseux series

```
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
---------------------------- CONSTRUCTORS -----------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------

PuiseuxSerie = new Type of HashTable;

puiseuxSerie = method(TypicalValue => PuiseuxSerie);
puiseuxSerie (RingElement, ZZ) := (f, m) -> (
  if not isPolynomialRing ring f then error "not a polynomial";
  if (degree f)#0 <= 0 then return new PuiseuxSerie from hashTable {p=>f, n=>1};
  return new PuiseuxSerie from hashTable {p=>f, n=>m};
)

-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
------------------------- OVERLOADED UNARY METHODS ---------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------

ring (PuiseuxSerie) := (f) -> ring f.p;

terms (PuiseuxSerie) := (f) ->
  reverse apply(terms f.p, term -> puiseuxSerie(term, f.n));

PuiseuxSerie _ PuiseuxSerie := (f, m) -> (
  if size m != 1 then error "expected a monomial";
```

```
  R := coefficientRing ring f;
  e := first exponents m;
  idx := position(exponents f, i -> i == e);
  if idx === null then return 0_R else return (listForm f)#idx#1;
)

listForm (PuiseuxSerie) := (f) ->
  return reverse apply(listForm f.p, (e, c) -> (e/f.n, c));

exponents (PuiseuxSerie) := (f) -> if f.p == 0 then return {{0}} else
  reverse apply(exponents f.p, e -> {(first e)/f.n});

coefficients (PuiseuxSerie) := (f) -> coefficients f.p;

size (PuiseuxSerie) := (f) -> size f.p;

lift (PuiseuxSerie, PolynomialRing) := (f, R) -> (
  if f.n != 1 then error "cannot lift to polynomial ring";
  return lift(f.p, R);
)

- PuiseuxSerie := (f) -> puiseuxSerie(-f.p, f.n);

rootUnity = method(TypicalValue => RingElement, Options => {Bits => 300})
rootUnity (QQ) := opts -> (kn) -> (
  bits := 2*opts.Bits;
  ppi := numeric_bits pi;
  root := exp(2*ppi*ii*kn);
  if (abs(realPart root) < 2.0^(-bits/2)) then root = (imaginaryPart root)*ii;
  if (abs(imaginaryPart root) < 2.0^(-bits/2)) then root = realPart root;
  return root;
)

conjugate (PuiseuxSerie) := (f) -> (
  x := first generators ring f;
  n := f.n;
  bits := precision coefficientRing ring f;
  return apply(toList(1..n), k -> sum apply(listForm f, (e, c) ->
    c*rootUnity((e#0)*k, Bits => bits//2)*x^(e#0)));
)
```

```
clean (PuiseuxSerie) := (f) -> (
  x := first generators ring f; y := last generators ring f;
  bits := precision coefficientRing ring f;
  s := puiseuxSerie(0*x, 1);
  s = s + sum apply(listForm f, (e, c) -> (
    if (abs(realPart c) < 2.0^(-bits)) then c = (imaginaryPart c)*ii;
    if (abs(imaginaryPart c) < 2.0^(-bits)) then c = realPart c;
    return c*x^(e#0)*y^(e#1);
  )); return s;
)

toPolynomial = method(TypicalValue => RingElement)
toPolynomial (PuiseuxSerie) := (s) -> (
  R := ring s;
  y := puiseuxSerie(last generators R, 1);
  factors := apply(conjugate s, si -> y - si);
  p := clean(product apply(conjugate s, si -> y - si));
  x := first generators R; y = last generators R;
  return sum apply(listForm p.p, (e, a) -> a*x^(e#0//p.n)*y^(e#1//p.n));
)

-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
------------------------- OVERLOADED BINARY METHODS ----------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------

PuiseuxSerie + PuiseuxSerie := (f, g) -> (
  n := lcm(f.n, g.n);
  fSubs := apply(generators ring f, gen -> gen => gen ^ (n // f.n));
  gSubs := apply(generators ring g, gen -> gen => gen ^ (n // g.n));
  ff := sub(f.p, fSubs);
  gg := sub(g.p, gSubs);
  return puiseuxSerie(ff + gg, n);
)

PuiseuxSerie - PuiseuxSerie := (f, g) -> f + (-g);

PuiseuxSerie * PuiseuxSerie := (f, g) -> (
```

```
  n := lcm(f.n, g.n);
  fSubs := apply(generators ring f, gen -> gen => gen ^ (n // f.n));
  gSubs := apply(generators ring g, gen -> gen => gen ^ (n // g.n));
  ff := sub(f.p, fSubs);
  gg := sub(g.p, gSubs);
  return puiseuxSerie(ff * gg, n);
)

Number + PuiseuxSerie := (n, f) -> puiseuxSerie(n + f.p, f.n);

PuiseuxSerie + Number := (f, n) -> n + f;

Number * PuiseuxSerie := (n, f) -> puiseuxSerie(n*f.p, f.n);

PuiseuxSerie * Number := (f, n) -> n * f;

RingElement + PuiseuxSerie := (p, f) -> puiseuxSerie(p, 1) + f;

PuiseuxSerie + RingElement := (f, p) -> f + puiseuxSerie(p, 1);

RingElement * PuiseuxSerie := (p, f) -> puiseuxSerie(p, 1) * f;

PuiseuxSerie * RingElement := (f, p) -> f * puiseuxSerie(p, 1);

PuiseuxSerie ^ ZZ := (f, k) -> puiseuxSerie(f.p ^ k, f.n);

RingElement ^ QQ := (f, q) -> (
  if size f > 1 then error "no method for binary operator ^ applied
    to polynomials with more than one monomial";
  return puiseuxSerie(f^(numerator q), denominator q);
)

PuiseuxSerie ^ QQ := (f, q) -> (
  if size f > 1 then error "no method for binary operator ^ applied
    to polynomials with more than one monomial";
  return puiseuxSerie(f.p^(numerator q), f.n*(denominator q));
)

substitute (PuiseuxSerie, Option) := (f, opt) -> (
  if not isPolynomialRing ring opt#1 or size opt#1 != 1 then
```

```
    error "substitution error";
  if class opt#1 === PuiseuxSerie then
    return puiseuxSerie(sub(f.p, opt#0 => (opt#1).p), f.n * (opt#1).n)
  else
    return puiseuxSerie(sub(f.p, opt), f.n);
)

-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
--------------------------- FORMATTING OUTPUT -------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------

expression PuiseuxSerie := f -> (
  gens := generators ring f;
  if length listForm f == 0 then return expression 0
  else return sum apply(listForm f,
      (exps, coef) -> coef * product apply(exps, gens, (e, g) ->
        (expression g)^e)) + O"("(expression first gens)^(1/f.n)")";

)

net PuiseuxSerie := f -> net expression f;

toString PuiseuxSerie := f -> toString expression f;

tex PuiseuxSerie := f -> tex expression f;

html PuiseuxSerie := f -> html expression f;

-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
```

Appendix B

Macaulay2 code: Newton-Puiseux algorithms

```
needs "puiseuxSeries.m2"

squareFreePart = method(TypicalValue => RingElement);
squareFreePart (RingElement) := (f) -> (
  y := last generators ring f;
  return f//gcd(f, diff(y, f));
)

squareFreeFactorization = method(TypicalValue => List);
squareFreeFactorization (RingElement) := (f) -> (
  -- Yun's algorithm
  y := last generators ring f;
  d := diff(y, f);
  squareFree := {};
  while degree f != {0} do (
    a := gcd(f, d);
    f = f//a;
    d = d//a - diff(y, f);
    squareFree = append(squareFree, a);
  );
  squareFree = drop(squareFree, 1);
  return apply(select(pack(2, mingle(squareFree, 1..#squareFree)),
    fact -> not isConstant fact#0), toSequence);
)

ZZ * InfiniteNumber := (n, inf) -> if n == 0 then 0 else
```

```
  if n > 0 then infinity else -infinity;

ccwTurn = method(TypicalValue => Boolean);
ccwTurn (BasicList, BasicList, BasicList) := (p1, p2, p3) -> (
  return (p2#0 - p1#0)*(p3#1 - p1#1) - (p2#1 - p1#1)*(p3#0 - p1#0) <= 0;
)

newtonPolygon = method(TypicalValue => List);
newtonPolygon (RingElement) := (f) -> (
  pol := {};
  apply(sort exponents f | {{infinity, 0}}, p -> (
    while (#pol >= 2 and ccwTurn(pol#(#pol - 2), pol#(#pol - 1), p)) do
      pol = drop(pol, -1);
    pol = append(pol, p);
  )); return drop(pol, -1);
)

newtonSide = method(TypicalValue => Sequence);
newtonSide (List, List, RingElement) := (p, q, f) -> (
  g := gcd(p#1 - q#1, q#0 - p#0);
  n := (p#1 - q#1)//g;
  m := (q#0 - p#0)//g;
  k := (p#1*q#0 - p#0*q#1)//g;

  -- Select which exponents are on the side generated by p & q.
  use ZZ[local X, local Y];
  side := n*X + m*Y - k;
  onSide := sort select(listForm f, (e, A) -> sub(side, {X=>e#0, Y=>e#1}) == 0);

  -- Construct the equation associated with the pq side.
  bits := precision ring f;
  use CC_bits[local Z];
  beta0 := (last onSide)#0#1;
  return (m, n, k, sum apply(onSide, (e, A) -> A*Z^((e#1 - beta0)//n)));
)

newtonSides = method(TypicalValue => List);
newtonSides (RingElement, List) := (f, points) -> (
  numSides := #points - 1;
  return apply(points_{0..numSides-1}, points_{1..numSides}, (p, q) ->
```

```
           newtonSide(p, q, f));
)

puiseuxExpansion = method(TypicalValue => List,
                          Options => { Terms => -1,
                                       Bits => 300 });
puiseuxExpansion (RingElement) := opts -> (f) -> (
  if not isPolynomialRing ring f then error "not a polynomial";
  if numgens ring f != 2 then error "not a bivariate polynomial";
  R := coefficientRing ring f;
  if R =!= ZZ and R =!= QQ then error "coefficient ring must be ZZ or QQ";
  z := first generators ring f; w := last generators ring f;
  use CC_(2*opts.Bits)[local x, local y];

  Nf := newtonPolygon f;
  yBranch := {};
  -- If Nf starts on the right of the y-axis, we have an x-factor.
  if (first Nf)#0 > 0 then yBranch = { (x, (first Nf)#0) };

  V := {z => x, w => y};
  return yBranch | apply(puiseuxExpansionLoop(sub(squareFreePart(f), V),
    apply(squareFreeFactorization(f), (g, m) -> (sub(g, V), m, 1)), opts.Terms),
      s -> (s#0, s#1#0#1));
)

puiseuxExpansion (List) := opts -> (L) -> (
  if not all(L, f -> numgens ring f == 2) then
    error "not bivariate polynomials";
  R := coefficientRing ring L#0;
  if R =!= ZZ and R =!= QQ then error "coefficient ring must be ZZ or QQ";
  z := first generators ring L#0; w := last generators ring L#0;
  use CC_(2*opts.Bits)[local x, local y];

  f := product L;
  Nf := newtonPolygon f;
  yBranch := {};
  -- If Nf starts on the right of the y-axis, we have an x-factor.
  if (first Nf)#0 > 0 then yBranch = { (x, select(apply(L, 1..#L, (g, i) ->
      (i, (first newtonPolygon g)#0)), (i, m) -> m != 0)) };
```

```
  V := {z => x, w => y};
  sqFreePart := sub(squareFreePart(f), V);
  sqFreeFact := flatten apply(apply(L, l -> apply(squareFreeFactorization l,
    (g, m) -> (sub(g, V), m))), 1..#L, (l, i) -> apply(l, (g, m) -> (g, m, i)));
  return yBranch | apply(puiseuxExpansionLoop(sqFreePart, sqFreeFact, -1));
)

puiseuxExpansionLoop = method(TypicalValue => List);
puiseuxExpansionLoop (RingElement, List, ZZ) := (f, L, num) -> (
  x := first generators ring f;
  y := last generators ring f;
  eps := 2.0^(-(precision ring f)/2);
  -- Select only those factors that contain the current branch.
  L = select(L, (g, m, i) -> (first newtonPolygon g)#1 != 0);

  Nf := newtonPolygon f;
  -- If the height is 1 we can stop here as this branch has been
  -- separated from the rest. (Except if more terms have been requested.)
  if (num < 0 and (first Nf)#1 == 1) or num == 0 then
    return { ((0*x)^(1_QQ), apply(L, (g, m, i) -> (i, m))) };

  -- Step (i.a): Select only those factors containing the 0 branch.
  if (last Nf)#1 > 0 then exactBranch := { ((0*x)^(1_QQ), apply(select(L,
    (g, m, i) -> (last newtonPolygon g)#1 > 0), (g, m, i) -> (i, m))) }
  else exactBranch = {};
  -- Step (i.b): For each side...
  return exactBranch | flatten apply(newtonSides(f, Nf), (m, n, k, F) ->
    -- For each root...
    flatten apply(roots(F, Unique => true), a -> (
      -- Get the solution, do & undo the change of variables.
      newVar := { x => x^n, y => x^m*(a^(1/n) + y) };
      apply(puiseuxExpansionLoop(
          clean(eps, sub(f, newVar)), apply(L, (g, m, i) ->
            (clean(eps, sub(g, newVar)), m, i)), num - 1),
        (s, I) -> (x^(m/n)*(a + sub(s, x => x^(1/n))), I))
    ))
  );
)
```

Appendix C

Macaulay2 code: Enriques diagrams

```
needs "puiseuxExpansion.m2"

basePoints = method(TypicalValue => Sequence)
basePoints (Ideal) := (I) -> (
  gen := first entries generators I;
  if not all(gen, f -> numgens ring f == 2) then
    error "not bivariate polynomials";
  -- Remove the fixed part.
  if #gen > 1 then gen = gen//gcd(gen)
  else (
    (P, e) := proximityMatrix(gen#0);
    return (P, sum e, P^-1*(sum e));
  );
  -- Compute the Puiseux expansion & the prox. matrix for the product.
  branches := puiseuxExpansion(gen);
  (P, e) = proximityMatrix(apply(branches, (s, l) -> (s, 1)),
              ExtraPoint => true);
 -- Multiplicities of each generator.
  ee := new MutableList from apply(1..#gen, i -> vector toList(numcols P:0));
  for i from 0 to #branches - 1 do (
    (s, l) := branches#i;
    scan(l, (j, n) -> ee#(j-1) = ee#(j-1) + n * e#i);
  ); ee = toList ee;
  -- Values for each generator.
  vv := P^-1*ee;
  -- Values for the points in the cluster.
  v := vector apply(entries matrix vv, min);
  -- Multiplicities for the cluster base points.
```

```
m := P*v;
-- Remove points not in the cluster of base points.
inCluster := positions(entries m, x -> x != 0);
P = P_inCluster^inCluster;
vv = apply(vv, vg -> vector apply(inCluster, i -> vg_i));
ee = P*vv;
v = vector apply(entries matrix vv, min);
-- Add NEW free points.
-- For each last free point on a branch...
apply(positions(sum entries P, i -> i == 1), p -> (
  -- Values for each generator in the point p.
  vvp := (entries matrix vv)#p;
  uniqueGen := #select(vvp, x -> x == v_p) == 1;
  -- Index of the generator achieving the minimum.
  g := minPosition(vvp);
  -- Multiplicity of that generator in the point p.
  mgp := (entries matrix ee)#p#g;
  -- If there exist a unique generator achieving the minimum
  -- value vp and the excess is positive add new points...
  if uniqueGen and mgp != 0 then (
    -- Minimum of the values for all the generators but g.
    wp := min(delete(v_p, vvp));
    -- Number of new free points.
    k := ceiling((wp - v_p)/mgp);
    -- Expand the prox. matrix.
    n := numcols P;
    P = expandMatrix(P, k);
    P_(n, n - 1) = 0; P_(n, p) = -1;
    P = matrix P;
    -- Expand the vector of mult. of each gen. with zeros.
    ee = new MutableList from apply(ee, e -> expandVector(e, k));
    -- For the generator g fill the new points with mult. mp.
    eeg := new MutableList from entries ee#g;
    for j from 0 to k - 1 do eeg#(n + j) = mgp;
    ee#g = vector toList eeg;
    ee = toList ee;
    -- Recompute all the other vectors for the next iteration.
    vv = P^-1*ee;
    v = vector apply(entries matrix vv, min);
  );
```

```
  ));
  -- Add NEW satellite points.
  points2test := numcols P - 1; p := 1;
  while points2test != 0 do (
    -- Values for the generators at point p.
    vvp := apply((entries matrix vv)#p, x -> x - v_p);
    -- Points p is prox to. && Points prox. to p
    Pprox := positions(flatten entries P^{p}, x -> x == -1);
    proxP := positions(flatten entries P_{p}, x -> x == -1);
    apply(select(Pprox, q -> sum flatten entries P^proxP_{q} == 0), q -> (
      vvq := apply((entries matrix vv)#q, x -> x - v_q);
      if product(vvq + vvp) != 0 then (
        -- Expand proximity matrix with a new point.
        n := numcols P;
        P = expandMatrix(P, 1);
        P_(n, n - 1) = 0; P_(n, p) = -1; P_(n, q) = -1;
        P = matrix P;
        -- Expand de vector of multiplicities of each generator.
        ee = apply(ee, e -> expandVector(e, 1));
        -- Recompute all the other vector for the next iteration.
        vv = P^-1*ee;
        v = vector apply(entries matrix vv, min);
        points2test = points2test + 1;
      );)
    ); points2test = points2test - 1; p = p + 1;
  ); return (P, P*v, v);
)

proximityMatrix = method(TypicalValue => Sequence,
                         Options => { ExtraPoint => false, Bits => 300 });
proximityMatrix (RingElement) := opts -> (f) -> (
  if not isPolynomialRing ring f then error "not a polynomial";
  if numgens ring f != 2 then error "not a bivariate polynomial";
  -- Get the Puiseux expansion of f.
  branches := puiseuxExpansion(f, Bits => opts.Bits);
  return proximityMatrix(branches,
    ExtraPoint => opts.ExtraPoint, Bits => opts.Bits);
)

proximityMatrix (List) := opts -> (branches) -> (
```

```
  -- Compute the proximity matrix and the contact matrix of each branch.
  contactMat := contactMatrix(branches);
  -- Proximity matrix of each branch.
  branchProx := apply(branches, 0..#branches - 1, (s, i) ->
    proximityMatrixBranch(s#0, max flatten entries contactMat^{i},
      ExtraPoint => opts.ExtraPoint));
  -- Compute the multiplicities of the infinitely near points of each branch.
  branchMult := apply(branches, 0..#branches - 1, (s, i) -> s#1 *
    multiplicityVectorBranch(s#0, max flatten entries contactMat^{i},
      ExtraPoint => opts.ExtraPoint));
  -- Get the proximity matrix of f and the position of each infinitely
  -- near point inside P.
  (P, p) := proximityMatrix(contactMat, branchProx);
  mult := {};
  -- Rearranges each point's multiplicity so its position is coherent with P.
  for i from 0 to #branches - 1 do (
    m := new MutableList from (numcols(P):0);
    for j from 0 to #p#i - 1 do m#(p#i#j) = branchMult#i_j;
    mult = mult | {vector toList m};
  ); return (P, mult);
)

proximityMatrix (Matrix, List) := opts -> (contactMat, branchProx) -> (
  ------------------------ Base case -------------------------------
  -- If there is only branch, return its prox. matrix.
  if #branchProx == 1 then
    return (branchProx#0, {toList(0..numcols(branchProx#0)-1)});
  ------------------ Compute the splits ----------------------------
  -- Substract one to all the contact numbers except the diagonal ones.
  contactMat = contactMat - matrix pack(#branchProx, (#branchProx)^2:1) +
    matrix mutableIdentity(ZZ, #branchProx);
  -- Idenitify each current branch with an ID from 0 to #brances.
  C := contactMat;
  remainingBranch := toList(0..numcols(C) - 1);
  -- Splits will contain lists of branches ID, where two branches will
  -- be in the same list iff they don't separate in the current node.
  splits := {};
  while #remainingBranch != 0 do (
    -- Get the contact number of the first remaining branch;
    branchContacts := first entries C;
```

```
  -- Get the positions of the branches with contact > 1 & contact = 1.
  sameBranchIndex := positions(branchContacts, c -> c != 0);
  otherBranchIndex := positions(branchContacts, c -> c == 0);
  -- Save the branches with contact > 1 together.
  splits = append(splits, remainingBranch_sameBranchIndex);
  -- Remove those branches since they've been splitted from the rest.
  remainingBranch = remainingBranch_otherBranchIndex;
  -- Compute the contact matrix of the remaining branches.
  C = submatrix(C, otherBranchIndex, otherBranchIndex);
);
---------- Compute the prox. matrix of each subdiagram -------------
-- Substract one to all the contact numbers and erase the
-- first point of the proximity matricies of the current
-- branches since we are moving down the Enriques diagram.
newBranchProx := apply(branchProx, P -> submatrix'(P, {0}, {0}));
-- Traverse each sub-diagram recursively.
splitResult := apply(splits, split -> (
  proximityMatrix(submatrix(contactMat, split, split), newBranchProx_split)));
-------------- Merge the prox. matrix of each split ----------------
-- Create the matrix that will hold the proximity branch of this subdiagram.
numPoints := sum apply(splitResult, (M, pos) -> numcols M) + 1;
P := mutableIdentity(ZZ, numPoints);
rowPoint := {}; k := 0;
-- For each set of branches that splits in this node...
for s from 0 to #splits - 1 do (
  -- Get the proximity matrix & the position of the points
  -- (relative to that prox. matrix) of the s-th subdiagram.
  (M, splitRowPoint) := splitResult#s;
  -- Copy the submatrix M inside P with the top left entry in (k+1, k+1)
  copySubmatrix(P, M, k + 1);
  -- Sum k+1 and add the new point ({0}) to the position of the
  -- points relative to the prox. matrix of the subdiagram.
  splitRowPoint = apply(splitRowPoint, pp -> {0} | apply(pp, p -> p + k + 1));
  rowPoint = rowPoint | splitRowPoint;
  -- Use the information in splitRowPoint to set the proximities of
  -- the current point into the new prox. matrix (P):
  -- For each branch in this subdiagram...
  for i from 0 to #(splits#s) - 1 do (
    Q := branchProx#(splits#s#i);
    -- For each element int the first column...
```

```
      for j from 1 to numcols(Q) - 1 do P_((splitRowPoint#i)#j, 0) = Q_(j, 0);
    ); k = k + numcols(M);
  );
  -- Make sure rowPoint is returned in the original order.
  splits = flatten splits;
  splits = toList apply(0..#splits-1, i -> position(splits, j -> j == i));
  return (matrix P, rowPoint_splits);
)

contactMatrix = method(TypicalValue => Matrix);
contactMatrix (List) := (branches) -> (
  -- Add a dummy term so compare exact branches is easier.
  x := first generators ring (first branches)#0;
  maxExp := max apply(branches, (s, m) -> ceiling((last exponents s)#0 + 1));
  branches = apply(branches, (s, m) -> (s + x^maxExp, m));
  branchesInfo := apply(branches, (s, m) -> puiseuxInfo s);
  contact := mutableIdentity(ZZ, #branches);
  -- For each pair of branches compute their contact number.
  for i from 0 to #branches - 1 do (
    for j from i + 1 to #branches - 1 do (
      contactNum := contactNumber(branchesInfo#i, branchesInfo#j);
      contact_(i,j) = contact_(j,i) = contactNum;
    );
  ); return matrix contact;
)

contactNumber = method(TypicalValue => ZZ);
contactNumber (List, List) := (branchInfoA, branchInfoB) -> (
  contactNumber := 0;
  -- For each characteristic exponent...
  for r from 0 to min(#branchInfoA, #branchInfoB) - 1 do (
    -- Get the contact number of this char. exponent and whether
    -- or not we should compare more points.
    (numExp, compNext) := contactNumberExp(branchInfoA#r, branchInfoB#r);
    contactNumber = contactNumber + numExp;
    if not compNext then break;
  ); return contactNumber;
)

contactNumberExp = method(TypicalValue => Sequence);
```

```
contactNumberExp (Sequence, Sequence) := (expInfoA, expInfoB) -> (
  contactNum := 0;
  -- Free points associated with the char. exponent.
  freeA := expInfoA#0;
  freeB := expInfoB#0;
  -- Satellite points associated with the char. exponent.
  satelliteA := new MutableList from expInfoA#1;
  satelliteB := new MutableList from expInfoB#1;
  -- Compare free points.
  for i from 0 to min(#freeA, #freeB) - 1 do (
    if freeA#i == freeB#i then contactNum = contactNum + 1
    else return (contactNum, false);
  );
  -- If the number of free points is not the same, no more points can be shared.
  if #freeA != #freeB then return (contactNum, false);
  -- Compare satellite points.
  satelliteA#-1 = satelliteA#-1 - 1;
  satelliteB#-1 = satelliteB#-1 - 1;
  for i from 1 to min(#satelliteA, #satelliteB) - 1 do (
    contactNum = contactNum + min(satelliteA#i, satelliteB#i);
    if satelliteA#i != satelliteB#i then return (contactNum, false);
  );
  -- If the number of stairs is not the same, no more points can be shared.
  if #satelliteA != #satelliteB then return (contactNum, false);
  -- Otherwise, all the points are shared.
  return (contactNum, true);
)

puiseuxInfo = method(TypicalValue => List);
puiseuxInfo (PuiseuxSerie) := (s) -> (
  pInfo := {}; allExps := charExponents(s);
  if tailExponents(s) != {} then allExps = allExps | { last tailExponents(s) };
  for i from 1 to #allExps - 1 do (
    (mj, nj) := allExps#(i-1);
    mi := allExps#i#0;
    h0 := (mi - mj)//nj;
    free := apply(apply(0..h0, l -> (mj + l*nj)/s.n), e -> (e, s_(x^e)));
    satellite := first euclides(mi - mj, nj);
    pInfo = append(pInfo, (toList free, satellite));
  ); return pInfo;
```

```
)

puiseuxInfo (RingElement) := (x) -> {({(0,0)}, {0, infinity})}

multiplicityVectorBranch = method(TypicalValue => List,
                                  Options => { ExtraPoint => false });
multiplicityVectorBranch (PuiseuxSerie, ZZ) := opts -> (s, maxContact) -> (
  mult := (); charExps := charExponents(s);
  for i from 1 to #charExps - 1 do (
    (mj, nj) := charExps#(i-1);
    mi := charExps#i#0;
    (hs, ns) := euclides(mi - mj, nj);
    scan(hs, ns, (h,n) -> mult = mult | (h:n));
  );
  mult = mult | ((maxContact - #mult):1);
  if opts.ExtraPoint then mult = mult | (1:1);
  return vector toList mult;
)

multiplicityVectorBranch (RingElement, ZZ) := opts -> (x, maxContact) -> (
  if opts.ExtraPoint then maxContact = maxContact + 1;
  return vector toList (maxContact:1);
)

charExponents = method(TypicalValue => List);
charExponents (PuiseuxSerie) := (s) -> (
  exps := reverse exponents s.p;
  charExps := {(0, s.n)};
  ni := s.n;
  while ni != 1 do (
    -- m_i = min{ j | a_j != 0 and j \not\in (n_{i-1}) }
    idx := position(exps, e -> e#0 % ni != 0);
    mi := exps#idx#0;
    -- n_i = gcd(n, m_1, ..., m_k)
    ni = gcd(ni, mi);
    charExps = append(charExps, (exps#idx#0, ni));
  ); return charExps;
)

tailExponents = method(TypicalValue => List);
```

```
tailExponents (PuiseuxSerie) := (s) -> (
  exps := reverse exponents s.p;
  (mk, nk) := last charExponents(s);
  -- If smooth branch take all, otherwise look for the last char. exp.
  if mk == 0 then idx := 0 else idx = position(exps, e -> e#0 == mk);
  return take(apply(exps, e -> (e#0, 1)), {idx, #exps});
)

euclides = method(TypicalValue => List);
euclides (ZZ, ZZ) := (m, n) -> (
  hs := {}; ns := {};
  while n != 0 do (
    hs = hs | {m // n};
    ns = ns | {n};
    r := m % n; m = n; n = r;
  ); return (hs, ns);
)

proximityMatrixBranch = method(TypicalValue => Matrix,
                               Options => {ExtraPoint => false});
proximityMatrixBranch (PuiseuxSerie, ZZ) := opts -> (branch, maxContact) -> (
  h := drop(apply(puiseuxInfo branch, charExps -> charExps#1), -1);
  numPoints := max(sum flatten h, maxContact);
  if opts.ExtraPoint then numPoints = numPoints + 1;
  -- Construct a prioximity matrix with free points only.
  prox = expandMatrix(matrix {{}}, numPoints);
  -- Fill in satellite points proximities.
  for i from 0 to #h - 1 do (
    -- Inverted branch case.
    if i == 0 and h#0#0 == 0 then start := 2 else start = 1;
    hi := new MutableList from h#i; hi#-1 = hi#-1 - 1;
    for j from start to #hi - 1 do (
      l := sum flatten h_{0..i-1} + sum (toList hi)_{0..j-1};
      for k from 1 to hi#j do prox_(l+k, l-1) = -1;
    );
  ); return matrix prox;
)

proximityMatrixBranch (RingElement, ZZ) := opts -> (branch, maxContact) -> (
  if opts.ExtraPoint then maxContact = maxContact + 1;
```

```
  return matrix expandMatrix(matrix {{}}, maxContact);
)

-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------

copySubmatrix = method(TypicalValue => Nothing);
copySubmatrix (MutableMatrix, Matrix, ZZ) := (A, B, k) -> (
  for i from 0 to numcols(B) - 1 do
    for j from 0 to numcols(B) - 1 do
      A_(k + i, k + j) = B_(i, j);
)

expandMatrix = method(TypicalValue => MutableMatrix)
expandMatrix (Matrix, ZZ) := (A, k) -> (
  newA := mutableIdentity(ZZ, numcols A + k);
  for i from 1 to numcols(newA) - 1 do newA_(i, i-1) = -1;
  copySubmatrix(newA, matrix A, 0);
  return newA;
)

expandVector = method(TypicalValue => Vector)
expandVector (Vector, ZZ) := (v, k) -> return vector(entries v | toList(k:0));
```

Bibliography

[1] M. Alberich-Carramiñana. An algorithm for computing the singularity of the generic germ of a pencil of plane curves. *Communications in Algebra*, 32(4):1637–1646, 2004.

[2] A. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, December 1979.

[3] A. Campillo. *Algebroid Curves in Positive Characteristic*. Number 813 in Lecture Notes in Mathematics. Springer-Verlag, Berlin Heidelberg, first edition, 1980.

[4] E. Casas-Alvero. *Singularities of Plane Curves*. Number 276 in London Mathematical Society Lecture Note Series. Cambridge University Press, Cambridge, UK, first edition, 2000.

[5] C. Chevalley. Intersections of algebraic and algebroid varieties. *Transactions of the American Mathematical Society*, 57(1):1–85, January 1945.

[6] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann. SINGULAR 4.0.2 — A computer algebra system for polynomial computations. Available at http://www.singular.uni-kl.de, 2015.

[7] D. R. Grayson and M. E. Stillman. Macaulay2, a software system for research in algebraic geometry. Available at http://www.math.uiuc.edu/Macaulay2/, 2015.

[8] Maplesoft, a division of Waterloo Maple Inc. Maple 2015. Available at http://www.maplesoft.com/products/Maple/, 2015.

[9] The PARI Group. PARI/GP version `2.7.4`, 2015. Available at http://pari.math.u-bordeaux.fr/.

[10] D. Y. Yun. On square-free decomposition algorithms. *Precedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, pages 26–35, August 1976.