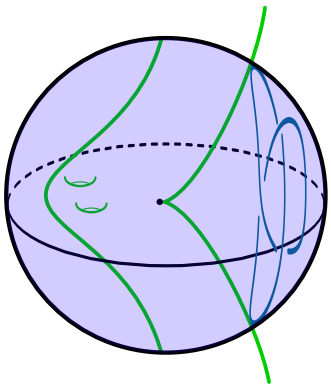
Trabajo Fin de Máster

Máster de Iniciación a la Investigación en Matemáticas

2011-2012



Funciones zeta y poliedros de Newton: Aspectos teóricos y computacionales

Autor:
Juan Viu Sos

Director:
Prof. Enrique Artal Bartolo

Universidad Zaragoza
1542

Índice

Introducción

Sea $f \in \mathbb{Z}[x_1, \ldots, x_n]$ y p un número primo.
De manera clásica, para el estudio del número de soluciones:

$$N_k := \{a \in \left(\mathbb{Z}/p\mathbb{Z}\right)^n \mid f(a) \equiv 0 \mod p^k\}$$

se asocia al polinomio f la *serie de Poincaré* generadora:

$$P(f,T) = \sum_{k=0}^{\infty} N_k T^k$$

Shafarevich conjeturó que $P(f,T)$ es una función racional en T. Igusa [**Igu74**] demostró esta conjetura mediante una relación con la integral p-ádica:

$$Z_f(s) := \int_{\mathbb{Z}_p^n} |f(x)|_p^s |\mathrm{d}x|, \qquad s \in \mathbb{C} \text{ con } \mathrm{Re}(s) > 0$$

donde $|\mathrm{d}x|$ denota la medida de Haar en $\mathbb{Q}_p^n$ normalizada tal que $\mathbb{Z}_p^n$ (el *anillo de enteros* de $\mathbb{Q}_p^n$) tiene medida 1.
Pese a ser un problema puramente combinatorio, Igusa utilizó resoluciones encajadas de singularidades sobre el lugar de ceros $f^{-1}(0) \subseteq \mathbb{C}^n$ para probar la igualdad

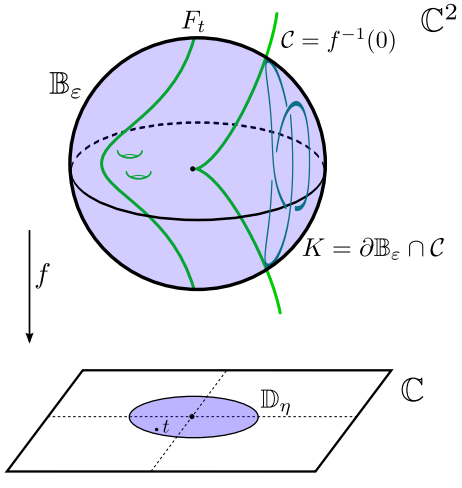
$$P(p^{-(n+s)}) = \frac{1 - p^{-s} Z_f(s)}{1 - p^{-s}}$$

viendo que $Z_f(s)$, la *función zeta de Igusa*, es una función racional en p^{-s}. Este procedimiento abre una nueva vía de análisis en la *teoría de singularidades*.
En este estudio, se observa que cada divisor excepcional en la resolución encajada de $f^{-1}(0)$ da un candidato a polo de $Z_f(s)$, sin embargo se comprueba que muchos de ellos no lo son realmente.

Siguiendo a [**Mil**], supongamos que f tiene una singularidad aislada en el origen. Para un $\varepsilon > 0$ podemos definir la *fibración de Milnor* de la función holomorfa f en el origen como la $\mathcal{C}^\infty$-fibración localmente trivial

$$f_{|\mathbb{B}_\varepsilon \cap f^{-1}(\mathbb{D}_\eta^*)} : \mathbb{B}_\varepsilon \cap f^{-1}(\mathbb{D}_\eta^*) \to \mathbb{D}_\eta^*$$

donde $\mathbb{B}_\varepsilon$ es la bola abierta centrada en el origen de radio ε, $\mathbb{D}_\eta = \{z \in \mathbb{C} \mid |z| < \eta\}$ y $\mathbb{D}_\eta^* = \mathbb{D}_\eta \setminus \{0\}$ con $(0 < \eta \ll \varepsilon)$. Para $0 < |t| < \eta$, la fibra $F_t = f^{-1}(t)$ se llama la *fibra de Milnor* de f en el origen, y tiene el tipo de homotopía de un ramillete de μ esferas de dimensión $n-1$ donde a μ se le llama el *número de Milnor* de la singularidad.
Si tomamos $\mathbb{S}_\varepsilon = \partial\mathbb{B}_\varepsilon$ la esfera de dimensión $2n-1$, obtenemos un enlace como resultado de la intersección $K = \mathbb{S}_\varepsilon \cap f^{-1}(0)$. El par $(\mathbb{S}_\varepsilon, K)$ determina la topología local de la hipersuperficie $f^{-1}(0)$ en la singularidad, no dependiendo del $\varepsilon > 0$ si éste es suficientemente pequeño.
La *transformación de monodromía* $h : F_t \to F_t$ es un difeomorfismo bien definido (salvo isotopía) de F_t, inducida por un giro pequeño alrededor del



Fibración de Milnor en el origen para $f(x,y) = x^2 - y^3$.

origen en $\mathbb{D}_\eta$. La *monodromía algebraica compleja* de f en el origen es la acción que induce la monodromía en los grupos de homología de la fibra, es decir, la aplicación lineal correspondiente $h_* : H_\bullet(F_t, \mathbb{C}) \to H_\bullet(F_t, \mathbb{C})$.

En el caso de singularidades aisladas, el polinomio característico de h_* se calcula también en función de la resolución. De nuevo, cada divisor excepcional da un candidato a valor propio que puede no serlo realmente.

Igusa dio una conjetura relacionando los polos de la integral $Z_f(s)$ con los valores propios de la monodromía compleja.

Conjetura 1 (Monodromía, Igusa)**.** *Sea $f \in \mathbb{K}[x_1, \ldots, x_n]$ no constante, para $\mathbb{K}$ cuerpo de números contenido en $\mathbb{C}$.*
Se tiene que, para casi toda compleción p-ádica $\mathbb{K}_p$ de $\mathbb{K}$, si s_0 es un polo de $Z_{f,\mathbb{K}_p}(s)$, entonces $\exp(2i\pi \operatorname{Re}(s_0))$ *es valor propio de la acción de monodromía local de f en algún punto de $f^{-1}(0)$.*

Esto es cierto para el caso de curvas ($n = 2$) y fue probado por Loeser [**Loe**] mediante una conjetura más fuerte, escrita en términos de polinomios de Bernstein.

Conjetura 2 (Monodromía fuerte, Igusa)**.** *Sea $f \in \mathbb{K}[x_1, \ldots, x_n]$ no constante, para $\mathbb{K}$ cuerpo de números contenido en $\mathbb{C}$.*
Se tiene que, para casi toda compleción p-ádica $\mathbb{K}_p$ de $\mathbb{K}$, si s_0 es un polo de $Z_{f,\mathbb{K}_p}(s)$, entonces $\operatorname{Re}(s_0)$ *es raíz del polinomio de Bernstein $b_f(s)$ de f.*

Para f polinomio complejo, la *función zeta topológica* $Z_{\mathrm{top},f}(s)$ fue introducida por Denef y Loeser (ver [**DenLoe92**]) como una cierta clase de límite de funciones zeta de Igusa, pudiendo ser expresada en términos de las multiplicidades N_i y $\nu_i - 1$ de la componentes irreducibles E_i en los divisores de $\pi^* f$ y $\pi^* \omega$ de una resolución encajada π de $f^{-1}(0)$, siendo ω la n-forma diferencial canónica en $\mathbb{C}^n$.
La función zeta topológica es un invariante analítico de la singularidad, pero no topológico (se puede ver un contraejemplo en [**ArCaLuMe01**]). Estando relacionados los polos de la función zeta de Igusa con los de la función zeta topológica, dándonos además información sobre una resolución encajada, Denef y Loeser conjeturaron:

Conjetura 3 (Monodromía, Topológica)**.** *Sea $f \in \mathbb{C}[x_1, \ldots, x_n]$ no constante. Si s_0 es polo de $Z_{top,f}(s)$, entonces* $\exp(2i\pi s_0)$ *es valor propio de la acción de monodromía local de f en algún punto de $f^{-1}(0)$.*

De manera análoga, se pueden definir las respectivas funciones zeta locales en el origen con $f : (\mathbb{C}^n, 0) \to (\mathbb{C}, 0)$ germen de función analítica.

Cálculo de funciones zeta en `Sage`

Existen tres problemas a considerar a la hora de intentar probar o refutar la conjetura usando resoluciones de singularidades:

- Cálculo explícito de una resolución encajada de la hipersuperficie $f^{-1}(0)$.
- Eliminación de los candidatos a polos de $Z_{\mathrm{top},f}(s)$ que no lo sean realmente.
- Cálculo explícito de los autovalores de la monodromía algebraica compleja (o del polinomio característico de la acción asociada a la monodromía) en función de los datos de la resolución.

Estos cálculos son costosos en general, sobre todo los relativos a encontrar resoluciones encajadas para dimensiones superiores $n > 2$ como se puede ver en demostraciones parciales de la conjetura dadas por [**Veys**] (caso $n = 2$) [**ArCaLuMe02**] (singularidades superaisladas) y [**ArCaLuMe05**] (polinomios cuasiordinarios). Sin embargo, existe una clase importante de polinomios, los *no degenerados con respecto las caras de su poliedro de Newton* (ver **Definición 1.10**), donde es posible calcular una resolución encajada del respectivo lugar de ceros, ver [**Var**]. A partir de esto, Denef-Loeser [**DenLoe92**], Hoornaert [**DenHoo**] y Varchenko [**Var**] dan fórmulas para las respectivas funciones zeta y la monodromía en el origen.

Hoornaert [**HooLoo**] da un un programa escrito en `Maple` donde implementa estas fórmulas. Sin embargo, nos encontramos con el problema de que `Maple` es un software privativo en el que se cambia constantemente de sintaxis de programación, y en donde Hoornaert ha tenido que acudir a una libreria externa de análisis convexo para poder desarrollar su programa.

Basándonos en este programa y en los trabajos citados anteriormente, se ha preparado un programa escrito en `Sage` para el cálculo de la función zeta de Igusa dado un primo p (abstracto o explícito), la función zeta topológica, y la monodromía en el origen de polinomios (o gérmenes de funciones analíticas en el origen) que cumplan tal condición de no degeneración. Se da acceso además a toda la información relativa al poliedro de Newton, sus caras, los conos asociados, particiones simpliciales,...
El programa de Hoornaert fue modificado por Artal-Melle para introducir el caso en el que se calcula la función zeta topológica respecto a la n-forma diferencial algebraica $\omega = x_1^{\omega_1} \cdots x_n^{\omega_n} \mathrm{d}x_1 \wedge \cdots \wedge \mathrm{d}x_n$ (ver [**NemVe**]), donde $(\omega_1, \ldots, \omega_n)$ son los *pesos*, y poder aplicarlo en el caso de los polinomios cuasiordinarios a funciones degeneradas mediante un proceso inductivo.

`Sage` (http://www.sagemath.org) es un sistema algebraico computacional de código abierto escrito en `Python` que incorpora software libre ya conocido para cálculo algebraico como `GAP`, `Maxima` ó `SINGULAR`, aprovechando su potencial y permitiendo desarrollar nuevo código sobre ellos.

En el desarrollo del programa `ZetaFunctions.sage` se ha aprovechado el potencial dado por las clases y métodos ya incorporados en su última versión (v5.2), como las clases geométricas `Polyhedron` ó `Cone`, que resultaron muy útiles en el desarrollo del programa al estar ya implementadas sobre la librería externa utilizada por Hoornaert antes citada. Sin embargo, en el caso de la clase `Polyhedron`, se detectó un error en el código fuente del método que devolvía la dimensión para las distintas caras del poliedro. Se creó una función propia corregida (`dim_face`), y se ha procedido al envío del reporte de error con la nueva propuesta al centro de desarrollo de `Sage`. También se han implementado métodos nuevos como descomposiciones simpliciales de conos.

El objetivo principal de este trabajo es el poder implementar todos estos cálculos en `Sage`, por lo que este código y sus métodos serán enviados al centro de desarrollo para que puedan aparecer en las versiones posteriores del programa.

A lo largo del desarrollo teórico del presente trabajo, después de cada definición y resultados obtenidos, se añade la parte de código en `Sage` correspondiente a la parte teórica integrado en `ZetaFunctions`. No se incluirán fuera del apéndice las funciones del código relativas a salidas a pantalla de la información de cada objeto o métodos.

La parte correspondiente al cálculo de la función zeta para la monodromía y el polinomio característico (para singularidades aisladas) se pueden encontrar en [**Var**].

Para la presentación del código en LaTeX, se utilizó el módulo `Pygments` de `Python`, diseñado para mostrar código en distintos formatos (http://pygments.org).

Todos los gráficos y dibujos son originales y fueron hechos con el programa open-source de gráficos vectoriales `Inkscape` (http://inkscape.org).

1. Preliminares

En esta primera sección introduciremos los conceptos y herramientas básicos para definir las diferentes funciones zeta y los resultados implementados en el código.

1.1. Los números p-ádicos

Sea p un número primo. Antes de introducir la función zeta de Igusa, debemos concretar el marco aritmético en el que surge: los números p-ádicos.

Definición de $\mathbb{Q}_p$ y propiedades.

Definición 1.1. Definimos el *orden p-ádico (o valuación p-ádica)* como la función:

$$\begin{aligned} \mathrm{ord}_p : \quad & \mathbb{Q} \longrightarrow \mathbb{Z} \cup \{\infty\} \\ & x \longmapsto \mathrm{ord}_p(x) = \mathrm{ord}_p\left(p^n \frac{a}{b}\right) = n \quad \text{con} \quad p \nmid ab \\ & 0 \longmapsto \infty \end{aligned}$$

Nota 1.1. Notar que n es el mayor entero que cumple $x \equiv 0 \mod p^n$ para $x \in \mathbb{Z}$.

Propiedad 1.1. *Sean $x, y \in \mathbb{Q}$, es sencillo comprobar que el orden p-ádico cumple las siguientes propiedades:*

1. $\mathrm{ord}_p(xy) = \mathrm{ord}_p(x) + \mathrm{ord}_p(y)$.
2. $\mathrm{ord}_p(x+y) \geq \mathrm{mín}\{\mathrm{ord}_p(x), \mathrm{ord}_p(y)\}$.

Definición 1.2. Para $x \in \mathbb{Q}$, tomamos la *norma p-ádica de x* como:

$$|x|_p = \begin{cases} p^{-\mathrm{ord}_p x} & \text{si } x \neq 0 \\ 0 & \text{si } x = 0 \end{cases}$$

Proposición 1.1. *$|\cdot|_p$ induce una ultramétrica sobre $\mathbb{Q}$.*

Demostración Sean $x, y \in \mathbb{Q}$, se cumple a partir de las propiedades del orden p-ádico:

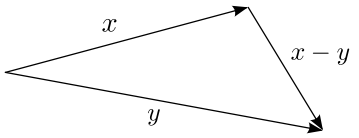
1. $|0|_p = p^{-\infty} = 0$.
2. $|xy|_p = p^{-\mathrm{ord}_p(xy)} = p^{-\mathrm{ord}_p(x)} p^{-\mathrm{ord}_p(y)} = |x|_p |y|_p$.
3. $|x-y|_p = p^{-\mathrm{ord}_p((-1)(y-x))} = p^{-\mathrm{ord}_p(-1)-\mathrm{ord}_p(y-x)} = p^{-\mathrm{ord}_p(y-x)} = |y-x|_p$.
4. $|x+y|_p = p^{-\mathrm{ord}_p(x+y)} \leq p^{-\mathrm{mín}\{\mathrm{ord}_p(x), \mathrm{ord}_p(y)\}} = \mathrm{máx}\{p^{-\mathrm{ord}_p(x)}, p^{-\mathrm{ord}_p(y)}\} = \mathrm{máx}\{|x|_p, |y|_p\}$.

□

Nota 1.2.

1. A la propiedad 4 sobre la suma en la demostración anterior se le llama la $\triangle$-*desigualdad fuerte*.

2. La $\triangle$-desigualdad fuerte en un espacio métrico $(X, \|\cdot\|)$ es equivalente a decir que "todos los triángulos son isósceles".



   - $\|x\| < \|y\|$.
   - $\|x - y\| \leq \text{máx}\{\|x\|, \|y\|\} = \|y\|$.
   - $\|y\| = \|y - x + x\| \leq \text{máx}\{\|x - y\|, \|x\|\} = \|x - y\|$ (ya que, en otro caso $\|y\| \leq \|x\|$).

   Luego $\|y\| = \|x - y\|$.

3. Consideremos la sucesión $(p^n)_{n=1}^{\infty}$. Vemos que con respecto a la norma p-ádica: $|p^n|_p = p^{-n} \longrightarrow 0$ cuando $n \to \infty$, converge a 0.

Definición 1.3. Se define el *cuerpo de los números p-ádicos* $\mathbb{Q}_p$ a la compleción de $\mathbb{Q}$ con la norma $|\cdot|_p$.
Llamaremos *anillo de los enteros p-ádicos* al subconjunto:

$$\mathbb{Z}_p := \{a \in \mathbb{Q}_p \mid |a|_p \leq 1\}$$

Propiedad 1.2. *Todo elemento $0 \neq a \in \mathbb{Q}_p$ tiene una representación única de la forma:*

$$a = p^{\text{ord}_p(a)} \sum_{i=0}^{\infty} a_i p^i$$

con $a_i \in \{0, 1, \ldots, p-1\}$ y $a_0 \neq 0$.

Nota 1.3. Se tiene $\mathbb{Z}_p^{\times} = \{a \in \mathbb{Z}_p \mid |a|_p = 1\} = \mathbb{Z}_p / p\mathbb{Z}_p$ son las unidades de $\mathbb{Z}_p$.

Uno de los resultados más importante sobre polinomios p-ádicos viene dado por el siguiente lema.

Lema 1.1 (Hensel)**.** *Sea $f \in \mathbb{Z}_p[x]$.*
Si para cierto $a_0 \in \mathbb{Z}_p$ se tiene $f(a_0) \equiv 0 \ (mod\ p)$ y $f'(a_0) \not\equiv 0 \ (mod\ p)$, entonces existe un único $a \in \mathbb{Z}_p$ tal que $f(a) = 0$ tal que $a \equiv a_0 \ (mod\ p)$.

Topología, medida e integrales en $\mathbb{Q}_p$.

Propiedad 1.3. *A partir de la $\triangle$-desigualdad fuerte, se tienen las siguientes propiedades topológicas:*

1. *Todo punto contenido en una bola en $\mathbb{Q}_p$ es centro de la propia bola.*
2. *Toda bola en $\mathbb{Q}_p$ es abierta y cerrada.*
3. *Para cualquier par de bolas en $\mathbb{Q}_p$, o bien son disjuntas, o bien una está contenida en la otra.*

El espacio métrico $\mathbb{Q}_p$ tiene una base de abiertos formada por elementos de la forma:

$$a + p^n\mathbb{Z}_p = \left\{ x \in \mathbb{Q}_p \mid |x - a|_p \leq \frac{1}{p^n} \right\}$$

con $a \in \mathbb{Q}_p$ y $n \in \mathbb{Z}$. Vamos a introducir una medida Borel en $\mathbb{Q}_p$ utilizando esta base de abiertos sobre la que definir integración: la medida de Haar.

Definición 1.4. Sea G un grupo topológico localmente compacto. Un a *medida de Haar* definida sobre G es una medida Borel μ que satisface:

1. $\mu(xE) = \mu(E)$, para todo $x \in G$ y para todo medible-Borel $E \subset G$.
2. $\mu(U) > 0$ para todo abierto $U \subseteq G$.
3. $\mu(K) < +\infty$ para todo compacto $K \subseteq G$.

Definición 1.5. Llamaremos *medida de Haar en $\mathbb{Q}_p$ normalizada sobre $\mathbb{Z}_p$* a la medida definida sobre la base de abiertos que cumple:

$$\mu_{\text{Haar}}(a + p^n\mathbb{Z}_p) = \frac{1}{p^n}$$

Propiedad 1.4. *Sea E medible-Borel de $\mathbb{Q}_p$.*
Se tiene:

1. *Invarianza por traslación: $\mu_{Haar}(x + E) = \mu_{Haar}(E),\ \forall x \in \mathbb{Q}_p$.*
2. *$\mu_{Haar}(\mathbb{Z}_p) = 1$.*

Nota 1.4.

1. De la misma manera, podemos generalizar la medida de Haar en $\mathbb{Q}_p^n$ normalizada sobre $\mathbb{Z}_p^n$ mediante la medida producto.
2. Bajo signo de integración, representaremos la medida de Haar normalizada con $|\mathrm{d}x|$.

Ejemplo 1.1. Sabemos que podemos poner las unidades de $\mathbb{Z}_p$ como unión disjunta:

$$\mathbb{Z}_p^\times = \bigcup_{a=1}^{p-1} a + p\mathbb{Z}_p$$

Por lo que podemos obtener fácilmente:

$$\mu_{\text{Haar}}(\mathbb{Z}_p^\times) = \sum_{a=1}^{p-1} \int_{a+p\mathbb{Z}_p} |\mathrm{d}x| = (p-1) \int_{p\mathbb{Z}_p} |\mathrm{d}x| = \frac{p-1}{p} = 1 - p^{-1}$$

1.2. El poliedro de Newton

Definición 1.6. Sea $f(x) = f(x_1, \ldots, x_n) = \sum_{\omega \in \mathbb{N}^n} a_\omega x_1^{\omega_1} \ldots x_n^{\omega_n}$ polinomio definido sobre un anillo conmutativo $R[x_1, \ldots, x_n]$ tal que $f(0) = 0$.
Llamaremos *soporte de* f al conjunto

$$\operatorname{supp}(f) = \{\omega \in \mathbb{N}^n | a_\omega \neq 0\}$$

Definimos el *poliedro de Newton global de* f, $\Gamma_{\text{gl}}(f)$, como la envolvente convexa de $\operatorname{supp}(f)$.
Sea ahora $f \in R\left[[x_1, \ldots, x_n]\right]$ y tomemos $\mathbb{R}^+ = \{x \in \mathbb{R} | x \geqslant 0\}$, definimos el *poliedro de Newton de* f, $\Gamma(f)$, como la envolvente convexa en $(\mathbb{R}^+)^n$ del conjunto

$$\bigcup_{\omega \in \operatorname{supp}(f)} \omega + (\mathbb{R}^+)^n$$

Nota 1.5. Es fácil ver que $\Gamma(f) = \Gamma_{\text{gl}}(f) + (\mathbb{R}^+)^n$.

```
def support_points(f):
    """
    Support of f: points of ZZ^n corresponding to the exponents of the monomial into ``f``.
    """
    points = f.exponents()
    return points
```

```
def newton_polyhedron(f):
    """
    Construction of Newton's Polyhedron Gamma(f) for the polynomial ``f``.
    """
    P = Polyhedron(vertices = support_points(f), rays=VectorSpace(QQ,f.parent().ngens()).basis())
    return P
```

Definición 1.7. Llamaremos *cara* de $\Gamma(f)$ (resp. $\Gamma_{\text{gl}}(f)$) a todo subconjunto convexo τ que se pueda obtener mediante la intersección de $\Gamma(f)$ (resp. $\Gamma_{\text{gl}}(f)$) y un hiperplano H de $\mathbb{R}^n$ tal que alguno de los semiespacios definidos por H contiene a $\Gamma(f)$ (resp. $\Gamma_{\text{gl}}(f)$).

Nota 1.6. Estamos considerando al poliedro total ($\Gamma(f)$ ó $\Gamma_{\text{gl}}(f)$) como cara. Para toda cara distinta al vacío o al total hablaremos de *caras propias*.

```
def faces(P):
    """
    Returns a LatticePoset of the faces in the polyhedron ``P`` with a relation of order
    (content between the faces).
    """
    P_lattice = LatticePoset(P.face_lattice())
    return P_lattice
```

```
def proper_faces(P):
    """
    Returns a list with the proper faces of the polyhedron ``P`` sorted in increasing order
    dimension.
    """
    L = faces(P).list()[1:-1]
    return L
```

Definición 1.8. Definimos *dimensión de la cara* τ como la dimensión del subespacio afín engendrado por τ. Llamaremos *vértices* a las caras de dimensión 0.

Nota 1.7. Podemos dar una expresión de las caras del poliedro de Newton en forma de suma de Minkowski:

$$\tau = \text{convex}\{v_1, \ldots, v_k\} + \sum_{i=1}^{l} \mathbb{R}^+ r_i$$

donde:

- $\{v_1, \ldots, v_k\} \subseteq \text{supp}(f)$ son los vértices de la cara.
- $\{r_1, \ldots, r_l\} \subseteq \{e_1, \ldots, e_n\}$ son los *rayos* contenidos en la cara.

Es claro que una cara en $\mathbb{R}^n$ es compacta si y sólo si ésta no contiene rayos.

```
def face_Vinfo(tau):
    """
    Returns a list containing the descriptions of the face in terms of vertices and rays.
    """
    return tau.element.ambient_Vrepresentation()
```

```
def face_Hinfo(tau):
    """
    Returns a list containing the descriptions of the face in terms of the inequalities of the
    facets who intersects into the face.
    """
    return tau.element.ambient_Hrepresentation()
```

```
def vertices(tau):
    """
    Returns a list with the vertices of the face.
    """
    L = map(lambda i:i.vector(),filter(lambda j:j.is_vertex(),face_Vinfo(tau)))
    return L
```

```
def rays(tau):
    """
    Returns a list with the rays of the face.
    """
    L = map(lambda i:i.vector(),filter(lambda j:j.is_ray(),face_Vinfo(tau)))
    return L
```

```
def contains_a_ray(tau):
    """
    Checks if the face contains some ray.
    """
    bool = False
    Vrep = face_Vinfo(tau)
    for e in Vrep:
        if e.is_ray() == True:
            bool = True
            break
    return bool
```

```
def compact_faces(P):
    """
    Returns a list with the compact faces of the polyhedron ``P`` sorted in increasing order
    dimension.
    """
    pfaces = proper_faces(P)
    return filter(lambda i: not contains_a_ray(i), pfaces)
```

```
def translate_points(points_list):
    """
    Returns a list of points taking the first point in the original list how the origin and
    rewriting the other points in terms of new origin.
    """
    origin = points_list[0]
    L = map(lambda v: v - origin, points_list)
    return L
```

```
def dim_face(tau):
    """
    Gives the dimension of the face.
    """
    vertices_tau = vertices(tau)
    rays_tau = rays(tau)
    if len(vertices_tau) == 0 and len(vertices_tau) == 0: dim_tau = -1
    else:
        v_list = translate_points(vertices_tau)
        dim_tau = matrix(v_list + rays_tau).rank()
    return dim_tau
```

Definición 1.9. Sea $f(x) = f(x_1, \ldots, x_n) = \sum_{\omega \in \mathbb{N}^n} a_\omega x_1^{\omega_1} \ldots x_n^{\omega_n}$ polinomio definido sobre un anillo R tal que $f(0) = 0$.
Para toda cara τ del poliedro de Newton $\Gamma(f)$, definimos el *polinomio asociado a* τ como:

$$f_\tau(x) = \sum_{\omega \in \tau} a_\omega x^\omega.$$

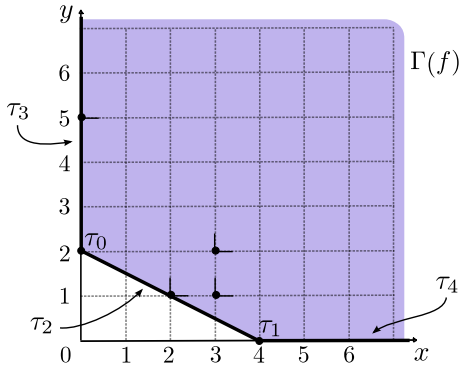
Nota 1.8. Observar que f_τ es un polinomio cuasihomogéneo para toda cara propia de $\Gamma(f)$.

```
def point_in_face(point,tau):
    """
    Checks if point belongs to the face.
    """
    bool = Polyhedron(vertices = vertices(tau), rays = rays(tau)).contains(point)
    return bool
```

```
def support_points_in_face(f, tau):
    """
    Returns a list of support points of ``f`` contained in the face.
    """
    L = filter(lambda i: point_in_face(i,tau),support_points(f))
    return L
```

```
def ftau(f,tau):
    """
    Returns the polynomial f_tau associated to the face ``tau`` of the Newton's Polyhedron.
    """
    from sage.rings.polynomial.polydict import ETuple
    # We take a dicctionary between the exponents and the coefficients of monomials.
    D=f.dict()
    vars = f.parent().gens()
    nvars = len(vars)
    g = 0
    for v in support_points_in_face(f,tau):
        mon = 1
        for i in range(nvars): mon = mon*vars[i]^v[i]
        g = g + D[ETuple(v)]*mon
    return g
```

Ejemplo 1.2. Poliedro de Newton de $f(x,y) = 2x^3y^2 + y^5 + x^4 - 2xy^3 - 2x^2y + y^2$:



Como caras propias, tenemos:

- $\dim \tau = 0$:

$$\tau_0 = \{(0,2)\}, \quad \tau_1 = \{(4,0)\}$$

- $\dim \tau = 1$:

$$\tau_2 = \{(1-\lambda)(0,2)+\lambda(1,0) \mid 0 < \lambda < 1\}, \quad \tau_3 = \{(0,2)+\mathbb{R}^+(0,1)\}, \quad \tau_4 = \{(4,0)+\mathbb{R}^+(1,0)\}$$

Los polinomios asociados a cada cara:

$$\begin{array}{rl} f_{\tau_0} = & y^2 \\ f_{\tau_1} = & x^4 \end{array}, \qquad \begin{array}{rl} f_{\tau_2} = & x^4 - 2x^2y + y^2 \\ f_{\tau_3} = & y^2 + y^5 \\ f_{\tau_4} = & x^4 \end{array}$$

El poliedro de Newton codifica importantes propiedades sobre las singularidades del lugar de ceros del polinomio y las resoluciones: relación con anillos locales, desarrollos de Puiseux, número de ramas del gérmen de una función analítica,... (ver [**BriKno**]).
Nos vamos a centrar en aquellos polinomios que están relacionados de una cierta forma con las caras de su poliedro de Newton y que son el objeto central de estudio en este trabajo.

Definición 1.10. Sea $f \in \mathbb{K}[x_1, \ldots, x_n]$ para $\mathbb{K}$ cuerpo y S un subconjunto de caras de $\Gamma(f)$. Diremos que f *es no degenerado sobre* $\mathbb{K}$ *con respecto a* S si para toda cara $\tau \in S$ el lugar de ceros de f_τ no tiene singularidades en $(\mathbb{K}^\times)^n$.

Ahora, introduciremos un nuevo concepto que nos permitirá obtener propiedades y relaciones sobre el poliedro de Newton y sus caras mediante ciertas particiones de $(\mathbb{R}^+)^n$: el **cono dual**.

Definición 1.11. Sea $\Gamma(f)$ poliedro de Newton de f definidos como antes.
Para $a \in (\mathbb{R}^+)^n$, definimos la función:

$$m(a) := \inf_{x \in \Gamma(f)} \{a \cdot x\}$$

Lema 1.2. *Sea $a \in (\mathbb{R}^+)^n$, se tiene que la función $\phi_a : \Gamma(f) \to \mathbb{R}$ con $\phi_a(x) = a \cdot x$ toma un mínimo sobre una cara de $\Gamma(f)$.*
En particular, éste se alcanza sobre el conjunto de vértices de $\Gamma(f)$.

Demostración Si $a = 0$, trivial. Supongamos $a \neq 0$, es claro que ϕ_a continua y que alcanza un mínimo restringida sobre $\Gamma_{\text{gl}}(f)$ por ser cerrado y acotado (compacto) en $\mathbb{R}^n$. Este mínimo también lo es sobre el poliedro de Newton ya que $\Gamma(f) = \Gamma_{\text{gl}}(f) + (\mathbb{R}^+)^n$, por lo que todo elemento se puede poner de la forma $p + \lambda$ con $p \in \Gamma_{\text{gl}}(f)$ y $\lambda \in (\mathbb{R}^+)^n$, y se tiene:

$$a \cdot (p + \lambda) = a \cdot p + a \cdot \lambda \geq a \cdot p$$

por estar $a \in (\mathbb{R}^+)^n$.
Ahora, sea $m(a)$ este valor mínimo. La ecuación $a \cdot x = m(a)$ nos define un hiperplano de soluciones en $\mathbb{R}^n$, de tal forma que que $\Gamma(f)$ está contenido en el semiespacio positivo $\{a \cdot x \geq m(a)\}$, ya que $m(a)$ es mínimo sobre el poliedro de Newton. Luego $\{a \cdot x = m(a)\} \cap \Gamma(f)$ es una cara del poliedro. □

Nota 1.9. A partir del lema anterior, podemos deducir:

$$m(a) = \min_{\omega \in \text{supp}(f)} \{a \cdot \omega\} = \min_{\substack{x \text{ vértice} \\ \text{de } \Gamma(f)}} \{a \cdot x\}$$

```
def m(v,P):
    """
    Returns min{v.x | x in polyhedron P}.
    """
    L = [vector(v).dot_product(vector(x)) for x in P.vertices()]
    return min(L)
```

Definición 1.12. Sea $f \in R[x_1, \ldots, x_n]$ anillo conmutativo tal que $f(0) = 0$ y $a \in (\mathbb{R}^+)^n$.
Definimos el *first meet locus de* a como el conjunto:

$$F(a) := \{x \in \Gamma(f) \mid a \cdot x = m(a)\}$$

Propiedad 1.5. *$F(a)$ es una cara de $\Gamma(f)$. En particular, $F(0) = \Gamma(f)$ y $F(a)$ es una cara propia de $\Gamma(f)$ si $a \neq 0$.*
Además, $F(a)$ es una cara compacta si y sólo si $a \in \mathbb{R}^n_{>0}$.

Demostración Cierto a partir del lema anterior. □

Definición 1.13. Un vector de $\mathbb{R}^n$ se dice *primitivo* si sus componentes son enteros relativamente primos entre sí.

Definición 1.14. A partir de la relación de equivalencia en $(\mathbb{R}^+)^n$ dada por

$$a \sim a' \iff F(a) = F(a')$$

podemos definir el *cono asociado a* τ, para τ cara de $\Gamma(f)$, como:

$$\Delta_\tau := \{a \in (\mathbb{R}^+)^n \mid F(a) = \tau\}$$

Observación 1.1. Notar que $\Delta_{\Gamma(f)} = \{0\}$.

Vamos a estudiar la geometría de los conos con respecto a las caras del poliedro de Newton. Debido a que el poliedro de Newton es un poliedro, sabemos que para toda τ cara propia:

$$\tau = \bigcap_{\substack{\tau \subseteq \gamma \\ \dim \gamma = n-1}} \gamma$$

siendo esta intersección finita. Además, para toda cara $(n-1)$-dimensional de $\Gamma(f)$, existe un único vector primitivo en $\mathbb{N}^n \setminus \{0\}$ perpendicular a ésta.

Lema 1.3. *Sea τ cara propia de $\Gamma(f)$ y $\gamma_1, \ldots, \gamma_e$ las caras $(n-1)$-dimensionales de $\Gamma(f)$ que la contienen. Sean $a_1, \ldots, a_e$ los únicos vectores primitivos perpendiculares a $\gamma_1, \ldots, \gamma_e$ respectivamente.*
Se tiene que:

$$\Delta_\tau = \{\lambda_1 a_1 + \ldots + \lambda_e a_e \mid \lambda_i \in \mathbb{R}_{>0}\}$$

con $\dim \Delta_\tau = n - \dim \tau$.

```
def prim(v):
    """
    Returns the primitivitation of an integral vector.
    """
    return v/gcd(v)
```

```
def primitive_vectors(tau):
    """
    Returns a list of primitive vectors of a face (normal vectors of the hyperplanes who defines
    the face, components are relatively primes).
    """
    L = map(lambda i:prim(i.A()),face_Hinfo(tau))
    return L
```

```
def cone_from_face(tau):
    """
    Construction of the dual cone of the face. In particular, for the total face it gives a cone
    generated by the zero vector.
    """
    gens = primitive_vectors(tau)
    if len(gens) == 0: cone = Cone([vertices(tau)[0].parent()(0)])
    else: cone = Cone(gens)
    return cone
```

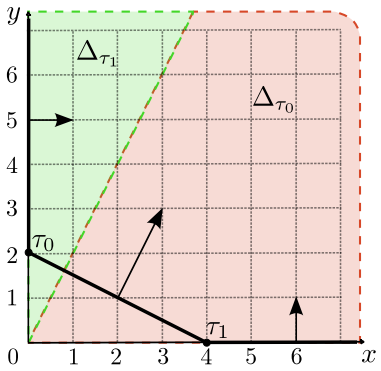
Ejemplo 1.3. Siguiendo con el ejemplo anterior para $f(x,y) = 2x^3y^2+y^5+x^4-2xy^3-2x^2y+y^2$, calculamos el vector primitivo perpendicular de cada cara propia de dimensión maximal:

- $\tau_2 = \{(1-\lambda)(0,2) + \lambda(1,0) \mid 0 < \lambda < 1\} \longrightarrow a_2 = (1,2)$.
- $\tau_3 = \{(0,2) + \mathbb{R}^+(0,1)\} \longrightarrow a_3 = (1,0)$.
- $\tau_4 = \{(4,0) + \mathbb{R}^+(1,0)\} \longrightarrow a_4 = (0,1)$.

Luego, utilizando el lema anterior, podemos describir los conos asociados a cada cara:

$$\Delta_{\tau_0} = \mathbb{R}_{>0}(1,2) + \mathbb{R}_{>0}(1,0), \quad \Delta_{\tau_1} = \mathbb{R}_{>0}(1,2) + \mathbb{R}_{>0}(0,1)$$

$$\Delta_{\tau_2} = \mathbb{R}_{>0}(1,2), \quad \Delta_{\tau_3} = \mathbb{R}_{>0}(1,0), \quad \Delta_{\tau_4} = \mathbb{R}_{>0}(0,1)$$



Definición 1.15. Sean $a_1, \ldots, a_e \in \mathbb{R}. \setminus \{0\}$, definimos el conjunto

$$\Delta = \{\lambda_1 a_1 + \ldots + \lambda_e a_e \mid \lambda_i \in \mathbb{R}_{>0}\}$$

como el *cono estrictamente positivamente generado por los vectores* $a_1, \ldots, a_e$.
Si $a_1, \ldots, a_e$ son linealmente independientes sobre $\mathbb{R}$, diremos que Δ es un *cono simplicial*. Si además $a_1, \ldots, a_e \in \mathbb{Z}^n$, decimos que Δ es un *cono simplicial racional*.
Si $\{a_1, \ldots, a_e\}$ es un subconjunto de una base del $\mathbb{Z}$-módulo $\mathbb{Z}^n$, decimos que Δ es un *cono simple*.

Lema 1.4. *Sea Δ un cono estrictamente positivamente generado por los vectores $a_1, \ldots, a_e \in \mathbb{R}^n \setminus \{0\}$. Existe una partición finita de Δ en conos Δ_i tal que cada cono es estrictamente positivamente generado por un subconjunto de vectores linealmente independientes de $\{a_1, \ldots, a_e\}$. Además, si Δ es un cono simplicial racional, entonces existe una partición en conos simples.*

Nota 1.10. Dada τ una cara propia de $\Gamma(f)$ y $a_1, \ldots, a_e$ los vectores propios primitivos asociados, a partir del lema anterior, podemos dar una partición del cono Δ_τ compuesta por un número finito de conos simpliciales racionales tal que cada cono Δ_i está generado por vectores del conjunto $\{a_1, \ldots, a_e\}$. Además, introduciendo nuevos rayos (generadores), podemos obtener igualmente una partición en conos simples.

```
def same_facet(lcone_gens, p, bcone_gens):
    """
    Checks if ``lcone_gens`` (a cone respresented by their generators) and the fix point
    ``p`` belongs to the same facet of ``bcone_gens``.
    """
    bool = False
    for face_cone in Cone(bcone_gens).facets():
        rays_lcone = set(map(tuple,lcone_gens))
        rays_face = set(map(tuple,primitive_vectors_cone(face_cone)))
        if ({tuple(p)}.union(rays_lcone)).issubset(rays_face):
            bool = True
            break
    return bool
```

```
def simplicial_partition(cone):
    """
    Returns a list with the subcones who forms the simplicial partition of ``cone``.
    """
    L = [cone]
    if not cone.is_simplicial():
        dict = {}
        F = Fan(L)
        list_subcones = []
        #Ordered list of subcones by ascending dimension
        for ls in F.cone_lattice().level_sets()[1:-1]: list_subcones = list_subcones + ls
        for subcone in list_subcones:
            if subcone.element.is_simplicial(): dict[subcone] = [set(subcone.element.rays())]
            else:
                partition = []
                fixpoint = subcone.element.rays()[0]
                for subsubcone in filter(lambda x: x<subcone, list_subcones):
                    if not same_facet(subsubcone.element.rays(), fixpoint, \
                                      subcone.element.rays()):
                        for part in dict[subsubcone]: partition = partition + \
                                                                  [part.union({fixpoint})]
                dict[subcone] = partition
        total_cone = list_subcones[-1]
        L = map(Cone,dict[total_cone])
    return L
```

Definición 1.16. Sean $a_1, \ldots, a_r$ vectores en $\mathbb{Z}^n$ linealmente independientes sobre $\mathbb{R}$. Definimos la *multiplicidad* de $a_1, \ldots, a_r$ como el índice del retículo $\mathbb{Z}a_1 + \ldots + \mathbb{Z}a_1$ en el grupo de puntos enteros del espacio vectorial real generado por $a_1, \ldots, a_r$.

Propiedad 1.6. *Sean $a_1, \ldots, a_r \in \mathbb{Z}^n$ linealmente independientes sobre $\mathbb{R}$. Se tiene:*

1. *La multiplicidad de $a_1, \ldots, a_r$ es igual al número de puntos enteros contenidos en el conjunto*

$$\left\{ \sum_{i=1}^{r} \lambda_i a_i \mid 0 \leq \lambda_i < 1 \right\}.$$

2. *Sea $A = \left(a_1 | \ldots | a_r\right)$, la multiplicidad de $a_1, \ldots, a_r$ es igual al producto de los elementos en la diagonal de la forma de Smith de A.*

```
def multiplicity(scone):
    """
    Returns the multiplicity of a simple cone.
    """
    L = primitive_vectors_cone(scone)
    A = matrix(ZZ, L)
    S = A.smith_form()[0]
    result = 1
    for i in range(len(L)):
        result = result*S[i,i]
    return result
```

```
def integral_vectors(scone):
    """
    Returns a list of integral vectors contained in {Sum lambda_j*a_j | 0<= lambda_j <1, a_j
    basis of the simple cone}.
    """
    origin = VectorSpace(QQ,scone.lattice_dim()).zero_vector()
    if scone.dim() == 0: integrals = [origin]
    else:
        cone_gens = primitive_vectors_cone(scone)
        ngens = len(cone_gens)
        A = transpose(matrix(ZZ, cone_gens))
        D, U, V = A.smith_form()
        diag = D.diagonal()
        coords = mrange(diag, vector)
        #Aux function for escale the vectors in the list
        def escale(v):
            v = vector(QQ, v)
            for i in range(ngens):
                if diag[i] != 0: v[i] = v[i]/diag[i]
                else: v[i] = 0
            return v
        #Aux function 'floor' for vectors component by component
        def floor(v):
            for i in range(ngens):
                v[i] = v[i] - v[i].floor()
            return v
        #Now, we escale and we return to the canonical basis
        L = map(lambda v: V*v, map(escale, coords))
        #Finally, we find the integral vectors of own region
        integrals = map(lambda v: matrix(QQ,A)*v, map(floor,L))
    return integrals
```

1.3. Resolución de singularidades

Introduciremos nociones y notaciones básicas relativas a la resolución de singularidades y resoluciones encajadas.
Sea X variedad algebraica sobre el cuerpo $\mathbb{K}$ algebraicamente cerrado y con $\operatorname{char}\mathbb{K} = 0$. Denotemos por $\operatorname{Sing}(X)$ el conjunto de puntos singulares de X.

Definición 1.17. Una *resolución* de X es un morfismo propio $\pi : Y \to X$ donde:

(i) Y es una variedad lisa (es decir, $\operatorname{Sing}(Y) = \varnothing$).

(ii) La restricción $\pi_{|Y\setminus\pi^{-1}(\mathrm{Sing}(X))} : Y \setminus \pi^{-1}(\mathrm{Sing}(X)) \to X \setminus \mathrm{Sing}(X)$ es un isomorfismo birracional.

Diremos que la resolución es *buena* si además cumple:

(iii) Para todo $p \in \pi^{-1}(\mathrm{Sing}(X))$, existe una carta

$$\begin{aligned} \mathbf{x}: \quad U &\xrightarrow{\cong} V \\ p &\longmapsto 0 \end{aligned}$$

con $U \subseteq Y$ y $V \subseteq \mathbb{K}^n$, tal que $U \cap \pi^{-1}(\mathrm{Sing}(X)) = \{x_{i_1}, \ldots, x_{i_r} = 0\}$ para ciertos $0 < i_1 < \ldots < i_r \leq n$.

Definición 1.18. Sea X variedad algebraica lisa, $f : X \to \mathbb{K}$ polinómica.
Una *resolución encajada* de f es un morfismo propio $\pi : Y \to X$ donde:

(i) Y es una variedad lisa (es decir, $\mathrm{Sing}(Y) = \varnothing$).

(ii) La restricción $\pi_{|Y\setminus\pi^{-1}(\mathrm{Sing}(f^{-1}(0)))} : Y \setminus \pi^{-1}(\mathrm{Sing}(f^{-1}(0))) \to X \setminus \mathrm{Sing}(f^{-1}(0))$ es un isomorfismo birracional.

(iii) Para todo $p \in \pi^{-1}(\mathrm{Sing}(X))$, existe una carta

$$\begin{aligned} \mathbf{x}: \quad U &\xrightarrow{\cong} V \\ p &\longmapsto 0 \end{aligned}$$

con $U \subseteq Y$ y $V \subseteq \mathbb{K}^n$, sobre la cual $\pi^* f = x_{i_1}^{N_1} \cdots x_{i_r}^{N_r} u$ con $u(0) \neq 0$ y $N_i \geq 0$.

Sea $f \in \mathbb{C}[x_1, \ldots, x_n]$ un polinomio no nulo. Sea $\pi : X \to \mathbb{C}^n$ una resolución de f, denotamos:

- $(E_i)_{i\in J}$ las componentes irreducibles de $\pi^{-1}(f^{-1}(0))$:

  $$\pi^*(f^{-1}(0)) = \sum_{i=1}^{r} N_j E_j$$

  con N_j la multiplicidad con la que $\pi^* f$ se anula en el punto genérico de E_j.
- $\mathring{E}_i = E_i \setminus \bigcup_{j\neq i} E_j$, para $i \in J$.
- $E_I = \bigcap_{i\in I} E_i$ y $\mathring{E}_I = E_I \setminus \bigcup_{j\notin I} E_j$, para $I \subset J$.

Para $i \in I$ fija, elegiremos una forma diferencial algebraica ω de grado n, definida sobre un entorno genérico del punto genérico $\pi(E_i)$ sobre $\mathbb{C}^n$, de tal manera que no se anula sobre este punto. Denotemos $\alpha(\omega)$ la multiplicidad con la que $\pi^*\omega$ se anula en el punto genérico de E_i, $\alpha(\omega)$ no depende de i. Denotamos:

$$\nu_i = \alpha(\omega) + 1.$$

Es decir, si $\omega = \mathrm{d}x_1 \wedge \cdots \wedge \mathrm{d}x_n$, entonces respecto a las carta coordenada centrada en el origen de (iii)

$$\pi^*\omega = x_{i_1}^{\nu_1 - 1} \cdots x_{i_r}^{\nu_r - 1} \mathrm{d}x_1 \wedge \cdots \wedge \mathrm{d}x_n$$

con ν_i asociada a la componente irreducible E_i.

Respectivamente, si tenemos $f : (\mathbb{C}^n, 0) \to (\mathbb{C}, 0)$ un germen de función analítica no nula, tomaremos un representante $f : B \to \mathbb{C}$ de la clase del germen definida sobre B bola abierta centrada en el origen. De esta forma, consideraremos análogamente una resolución π del germen f como la resolución $\pi : X \to B$ del representante $f : B \to \mathbb{C}$, con todas las notaciones anteriores.
La existencia de la resolución para cuerpos de característica 0 está garantizada por [**Hiro**].

2. La función zeta de Igusa

Sea p un cierto primo y denotemos como antes $\mathbb{Q}_p$ al cuerpo de los números p-ádicos, $\mathbb{Z}_p$ el anillo de los enteros p-ádicos y $\mathbb{F}_p$ al cuerpo finito de p elementos.

2.1. Definición y relaciones con el poliedro de Newton.

Definición 2.1. Sea $f(x) = f(x_1, \ldots, x_n) \in \mathbb{Z}_p[x_1, \ldots, x_n]$.
Definimos la *función zeta de Igusa asociada a* f de la siguiente forma:

$$Z_f(s) = \int_{\mathbb{Z}_p^n} |f(x)|_p^s |\mathrm{d}x| \in \mathbb{Q}(p^{-s}), \qquad s \in \mathbb{C} \text{ con } \mathrm{Re}(s) > 0$$

donde $|\mathrm{d}x|$ denota la medida de Haar en $\mathbb{Q}_p^n$ normalizada tal que $\mathbb{Z}_p^n$ tiene medida 1.

Nota 2.1. Denotaremos igualmente $Z_f(s)$ a la extensión meromorfa de la función zeta de Igusa.

Nota 2.2. Para $f \in \mathbb{Z}_p[x_1, \ldots, x_n]$ y τ cara del poliedro de Newton $\Gamma(f)$, denotaremos $\bar{f}_\tau(x)$ al polinomio asociado a la cara τ con coeficientes en $\mathbb{F}_p$, es decir, reduciendo cada coeficiente a_ω de f_τ módulo $p\mathbb{Z}_p$.

Nota 2.3.

1. La fórmula para la función zeta de Igusa que estamos estudiando se da sobre familias de polinomios que cumplen condiciones de no degeneración sobre $\mathbb{K} = \mathbb{F}_p$ ó $\mathbb{Q}_p$. Los conjuntos de caras sobre los que se cumple esta condición depende del enfoque global (conjunto total de caras) o local (caras compactas del poliedro de Newton) de la función zeta en el origen.

2. La condición de no degeneración sobre $\mathbb{F}_p$ en una cara τ es equivalente a que el sistema de ecuaciones en congruencias:

$$\begin{cases} f_\tau \equiv 0 & \text{mód } p \\ \frac{\partial f_\tau}{\partial x_i} \equiv 0 & \text{mód } p, \qquad i = 1, 2, 3, \ldots \end{cases}$$

   no tiene solución en $(\mathbb{Z}_p^\times)^n$.

```
def solve_in_Fp_x(f,p):
    """
    For f a integral polynomial, retruns a list [{a in (F_p^x)^d | f*(a)=0}, {vars of f*}] with
    f* being f with coefficients in F_p(f), for a given rime number ``p``.
    """
    g = f.change_ring(GF(p))
    vars = g.variables()
    nvars = g.nvariables()
    h = (GF(p)[vars])(g)
    if len(h.exponents()) == 1: sols = []    #If f_tau is a monomial
    else:
        Fp_x_nvars = list(Tuples(range(1,p), nvars))    #(Fp-0)^nvars
        if h == 0: return [Fp_x_nvars, vars]
        sols = filter(lambda a:h(tuple(a))==0,Fp_x_nvars)
    return [sols,vars]
```

```
def is_degenerated(f_tau, p = None, method = 'default'):
    """
    Checks if the polynomial ``f_tau`` is degenerated over F_p, for p a given prime number
    ``p``.\n
    If ``p = None``, checks degeneration  over CC and (equivalent to be degenerated over F_p with
    p>>0).\n
    For finite fields (``p`` is a given prime):\n
        - ``method = 'default'`` check the condition using evaluation over the (F_p^x)^n in the
    system of equations.\n
        - ``method = 'ideals'`` check the condition using ideals over the finite field.
    """
    bool = False
    if type(p) != Integer:
        vars = f_tau.parent().gens()
        S = QQ[vars]
        I = S(f_tau).jacobian_ideal() + S*(S(f_tau))
        bool = prod(S.gens()) not in I.radical()
    else:
        if method == 'ideals':
            S = GF(p)[vars]
            I = S(f_tau).jacobian_ideal() + S*(f_tau)
            for xi in vars:
                I = I + S*(xi^(p-1)-1)    #xi unity in Fp iff xi^{(p-1)-1}=0
            bool = 1 not in I    #True if I in NOT the ring (ie, sist. has solution)
        else:
            [candidates, vars] = solve_in_Fp_x(f_tau,p)
            if vars == []: bool = True
            else:
                S = GF(p)[vars]
                g = f_tau.change_ring(GF(p))
                for xi in S.gens():
                    df_tau = S(g).derivative(xi)
                    candidates = filter(lambda a: df_tau(tuple(a)) == 0, candidates)
                    if len(candidates) != 0:
                        bool = True
                        break
    return bool
```

Propiedad 2.1. *Sea $0 \neq f \in \mathbb{Z}[x_1, \ldots, x_n]$ con $f(0) = 0$.*

1. *Si f es no degenerado sobre $\mathbb{Q}_p$ con respecto a todas las caras de su poliedro de Newton entonces , si p suficiente grande ($p \gg 0$), f es no degenerado sobre $\mathbb{F}_p$ con respecto a todas las caras de su poliedro de Newton.*

2. *Si f es no degenerado sobre $\mathbb{C}$ con respecto a todas las caras de su poliedro de Newton entonces f es no degenerado sobre $\mathbb{F}_p$ con respecto a todas las caras de su poliedro de Newton, para casi todo primo p (es decir, para todo primo excepto en un conjunto finito).*

Ahora, introduciremos una serie de resultados clave en la fórmula que se da para la función zeta de Igusa bajo las condiciones de no degeneración que hemos visto para los polinomios.

Propiedad 2.2. *Sea $f \in \mathbb{Z}_p[x_1, \ldots, x_n]$ y $a \in \mathbb{Z}_p^n$.*
Supongamos que el sistema de congruencias

$$\begin{cases} f \equiv 0 & \text{mód } p \\ \frac{\partial f}{\partial x_i} \equiv 0 & \text{mód } p, \qquad i = 1, 2, 3, \ldots \end{cases}$$

no tiene solución en el coset $a + (p\mathbb{Z}_p)^n$.
Entonces, para $s \in \mathbb{C}$ *tal que* $\mathrm{Re}(s) > 0$, *se tiene que:*

$$\int_{a+(p\mathbb{Z}_p)^n} |f(x)|_p^s |\mathrm{d}x| = \begin{cases} p^{-n} & \text{si } f(a) \not\equiv 0 \mod p \\ p^{-n}(p-1)\frac{p^{-(s+1)}}{1-p^{-(s+1)}} & \text{si } f(a) \equiv 0 \mod p \end{cases}$$

donde $|\mathrm{d}x|$ *denota la medida de Haar en* $\mathbb{Q}_p^n$ *normalizada tal que* $\mathbb{Z}_p^n$ *tiene medida 1.*

Demostración Se puede encontrar en [**DenHoo**] una demostración elemental de esta propiedad usando el Lema de Hensel. □

Corolario 2.1. *Sea* $f \in \mathbb{Z}_p[x_1, \ldots, x_n]$ *y* $a \in \mathbb{Z}_p^n$. *Denotemos por* $\bar{f}$ *el polinomio sobre* $\mathbb{F}_p$ *obtenido reduciendo los coeficientes de* f *módulo* $p\mathbb{Z}_p$ *y sea* $N = \#\{a \in (\mathbb{F}_p^\times)^n \mid \bar{f}(a) = 0\}$.
Supongamos que el sistema de congruencias

$$\begin{cases} f \equiv 0 & \mod p \\ \frac{\partial f}{\partial x_i} \equiv 0 & \mod p, \qquad i = 1, 2, 3, \ldots \end{cases}$$

no tiene solución en $(\mathbb{Z}_p^\times)^n$.
Entonces, para $s \in \mathbb{C}$ *tal que* $\mathrm{Re}(s) > 0$, *se tiene que:*

$$\int_{(\mathbb{Z}_p^\times)^n} |f(x)|_p^s |\mathrm{d}x| = p^{-n}\left((p-1)^n - pN\frac{p^s - 1}{p^{s+1} - 1}\right)$$

donde $|\mathrm{d}x|$ *denota la medida de Haar en* $\mathbb{Q}_p^n$ *normalizada tal que* $\mathbb{Z}_p^n$ *tiene medida 1.*

Demostración Podemos dar una partición en cosets del dominio de integración:

$$(\mathbb{Z}_p^\times)^n = \bigcup_{a \in \{1,\ldots,p-1\}^n} a + (p\mathbb{Z}_p)^n$$

Por lo que:

$$\begin{aligned} \int_{(\mathbb{Z}_p^\times)^n} |f(x)|_p^s |\mathrm{d}x| &= \sum_{\substack{a \in \{1,\ldots,p-1\}^n \\ f(a) \not\equiv 0 \mod p}} \int_{a+(p\mathbb{Z}_p)^n} |f(x)|_p^s |\mathrm{d}x| \\ &+ \sum_{\substack{a \in \{1,\ldots,p-1\}^n \\ f(a) \equiv 0 \mod p}} \int_{a+(p\mathbb{Z}_p)^n} |f(x)|_p^s |\mathrm{d}x| \end{aligned}$$

De esta forma, se cumplen las hipótesis de la **Propiedad 2.2** para cada $a \in \{1, \ldots, p-1\}^n$ y contando entonces aquellos en los que $f(a) \equiv 0 \mod p$:

$$\begin{aligned} \int_{(\mathbb{Z}_p^\times)^n} |f(x)|_p^s |\mathrm{d}x| &= ((p-1)^n - N)p^{-n} + Np^{-n}(p-1)\frac{p^{-(s+1)}}{1-p^{-(s+1)}} \\ &= p^{-n}\left((p-1)^n - N\frac{1-p^{-s}}{1-p^{-(s+1)}}\right) \end{aligned}$$

□

2.2. Fórmula sobre las caras de $\Gamma(f)$.

Vamos a dar finalmente la fórmula para la función zeta de Igusa para polinomios que sean no degenerados sobre $\mathbb{F}_p$ con respecto a todas las caras de su poliedro de Newton (caso global), o con respecto a sus caras compactas (caso local en el origen).

```
def is_all_degenerated(f,P, p = None, local = False, method = 'default'):
    """
    Checks if own polynomial ``f`` is degenerated over F_p (``p`` prime) with respect the
    faces of the polyhedron ``P``.\n
    If ``p = None``, checks degeneration  over CC and (equivalent to be degenerated over F_p
    with p>>0).\n
    ``local = True`` checks degeneration for local case (only with respect the compact faces).\n
    For finite fields (``p`` is a given prime):\n
        - ``method = 'default'`` check the condition using evaluation over the (F_p^x)^n in the
    system of equations.\n
        - ``method = 'ideals'`` check the condition using ideals over the finite field.
    """
    bool = False
    if local == True: faces_set = compact_faces(P)
    else: faces_set = faces(P)[1:]
    for tau in faces_set:
        f_tau = ftau(f,tau)
        if is_degenerated(f_tau, p, method) == True:
            bool = True
            print "The formula for Igusa Zeta function is not valid:"
            if type(p) != Integer: print "The polynomial is degenerated at least with respect "\
                                         "to the face tau = {" + face_info_output(tau) + "} "\
                                         "over the complex numbers!"
            else: print "The polynomial is degenerated at least with respect to the face tau = "\
                        "{" + face_info_output(tau) + "} over GF(" + str(p) + ")!"
            break
    return bool
```

Primero, vamos a introducir las funciones con las que se expresará la fórmula para la función zeta de Igusa, en relación con las caras del poliedro de Newton, los conos duales asociados como partición de $(\mathbb{R}^+)^n$ y los puntos enteros contenidos en éstos.

Definición 2.2. Para $k = (k_1, \ldots, k_n) \in \mathbb{R}^n$, consideramos la suma de componentes:

$$\sigma(k) = \sum_{i=1}^{n} k_i$$

```
def sigma(v, weights = None):
    """
    Returns the pondered sum of the components in vector.
    """
    if weights == None: result = sum(v)
    else: result = vector(v).dot_product(vector(weights))
    return result
```

Definición 2.3. Sea τ cara de $\Gamma(f)$, p primo y $s \in \mathbb{C}$ tal que $\mathrm{Re}(s) > 0$. Consideraremos las siguientes expresiones:

- $N_\tau := \#\{a \in (\mathbb{F}_p^\times)^n \mid \bar{f}_\tau(a) = 0\}$
- $L_\tau(s) := p^{-n}\left((p-1)^n - pN_\tau \dfrac{p^s - 1}{p^{s+1} - 1}\right)$
- $S_{\Delta_\tau}(s) := \displaystyle\sum_{k \in \mathbb{N}^n \cap \Delta_\tau} p^{-\sigma(k) - m(k)s}$

Notar que estamos tomando también $\tau = \Gamma(f)$.

```
def Ntau(f,tau,p):
    """
    Returns the number Ntau = #{a in (F_p^x)^d | f*_tau(a)=0} with f*_tau being f_tau with
    coefficients in F_p(f_tau) for tau face.
    """
    n = f.parent().ngens()
    f_tau = ftau(f,tau)
    if type(p) != Integer:
        print "You must to give a 'Dictionary' with the number of solutions in GF(" + str(p) + \
              ")^" + str(n) + " associated to each face."
    else:
        [sols,vars] = solve_in_Fp_x(f_tau,p)
        nsols = len(sols)*(p-1)^(n - len(vars))
    return nsols
```

```
def Ltau(f,tau,p,abs_Ntau,s):
    """
    Returns a list [L_tau, N_tau] in terms of variable ``s``.\n
    ``abs_Ntau`` is the corresponding Ntau's values for abstract prime ``p``.
    """
    n = f.parent().ngens()
    if type(p) != Integer: N_tau = abs_Ntau
    else: N_tau = Ntau(f,tau,p)
    result = p^(-n)*((p-1)^n - p*N_tau*((p^s-1)/(p^(s+1)-1)))
    result = factor(result)
    return [result, N_tau]
```

```
def Lgamma(f,p,abs_Ngamma,s):
    """
    Returns the value Ntau for the total polyhedron in terms of variable ``s``.\n
    ``abs_Ngamma`` is the corresponding Ngamma value for abstract prime ``p``.
    """
    n = f.parent().ngens()
    if type(p) != Integer: N_gamma = abs_Ngamma
    else:
        [sols,vars] = solve_in_Fp_x(f,p)
        N_gamma = len(sols)*(p-1)^(n - len(vars))
    result = p^(-n)*((p-1)^n - p*N_gamma*((p^s-1)/(p^(s+1)-1)))
    return result
```

Nota 2.4. Para calcular $S_{\Delta_\tau} \equiv S_{\Delta_\tau}(s)$, consideraremos una partición de Δ_τ en conos simpliciales racionales (notar que estamos tomando también los subconos simpliciales maximales que aparecen

como intersección de los conos de mayor dimensión, rellenando de esta manera todo el interior de Δ_τ). De esta forma, sobre cada cono de la partición:

$$S_{\Delta_\tau} = \sum S_{\Delta_i} \quad \text{con} \quad S_{\Delta_i} = \sum_{k \in \mathbb{N}^n \cap \Delta_i} p^{-\sigma(k)-m(k)s}$$

Propiedad 2.3. *En las condiciones anteriores, sea Δ_i un cono simplicial estrictamente generado por $a_1, \ldots, a_e \in \mathbb{N}^n$ linealmente independientes.*
Entonces:

$$S_{\Delta_i} = \frac{\sum_h p^{\sigma(h)+m(h)s}}{(p^{\sigma(a_1)+m(a_1)s} - 1) \cdots (p^{\sigma(a_e)+m(a_e)s} - 1)}$$

donde h recorre los puntos enteros de $\{\sum_{i=1}^r \lambda_i a_i \mid 0 \leq \lambda_i < 1\}$.

```
def Stau(f,P,tau,p, weights,s):
    """
    Returns a list [S_tau, cone_info] with ``cone_info`` containing a string of information
    about the cones, simplicial partition, multiplicity and integral points.\n
    Value S_tau is in terms of variable ``s``.
    """
    c = cone_from_face(tau)
    dim_cone = c.dim()
    F = simplicial_partition(c)
    result = 0
    for scone in F:
        num = 0
        den = 1
        for h in integral_vectors(scone): num = num + p^(sigma(h, weights) + m(h,P)*s)
        for a in primitive_vectors_cone(scone): den = den*(p^(sigma(a, weights) + m(a,P)*s) - 1)
        result = factor(simplify(expand(result + num/den)))
    info = cone_info_output(c,F)+ "\n" + "multiplicities = " + str(map(multiplicity,F)) + \
           ", integral points = " + str(map(integral_vectors,F))
    return [result, info]
```

Demostración Fijemos x_τ en la cara τ, se tiene que $m(k) = k \cdot x_\tau$ para todo $k \in \Delta_\tau$. Luego:

$$S_{\Delta_i} = \sum_{k \in \mathbb{N}^n \cap \Delta_i} p^{-\sigma(k)-k\cdot x_\tau s}$$

Consideremos dos casos:

- Δ_i **cono simple.** Esto es equivalente a que

  $$\mathbb{N}^n \cap \Delta_i = \mathbb{N}_{>0} a_1 + \ldots + \mathbb{N}_{>0} a_e$$

  Puesto que $a_1, \ldots, a_e$ linealmente independientes:

  $$\begin{aligned} S_{\Delta_i} &= \sum_{\lambda_1, \ldots, \lambda_e \in \mathbb{N}_{>0}} p^{-\sigma(\lambda_1 a_1 + \ldots + \lambda_e a_e) - (\lambda_1 a_1 + \ldots + \lambda_e a_e)\cdot x_\tau s} \\ &= \sum_{\lambda_1=1}^{\infty} (p^{-\sigma(a_1)-a_1\cdot x_\tau s})^{\lambda_1} \cdots \sum_{\lambda_e=1}^{\infty} (p^{-\sigma(a_e)-a_e\cdot x_\tau s})^{\lambda_e} \end{aligned}$$

Ya que estamos considerando $\mathrm{Re}(s) > 0$ y $p > 1$, tenemos que $|p^{-\sigma(a_j)-a_j\cdot x_\tau s}| < 1$ para $j = 1, \ldots, e$, por lo que las series geométricas convergen y:

$$\begin{aligned} S_{\Delta_i} &= \frac{p^{-\sigma(a_1)-a_1\cdot x_\tau s}}{1-p^{-\sigma(a_1)-a_1\cdot x_\tau s}} \cdots \frac{p^{-\sigma(a_e)-a_e\cdot x_\tau s}}{1-p^{-\sigma(a_e)-a_e\cdot x_\tau s}} \\ &= \frac{1}{(p^{\sigma(a_1)+a_1\cdot x_\tau s}-1)\cdots(p^{\sigma(a_e)+a_e\cdot x_\tau s}-1)} \end{aligned}$$

Y $a_j \cdot x_\tau = m(a_j)$ ya que $a_j \in \bar{\Delta}_\tau = \{a \in (\mathbb{R}^+). \mid F(a) \supseteq \tau\}$, para todo $j = 1, \ldots, e$.

- **Caso general.** Consideremos el conjunto:

$$\mathbb{Z}^n \cap \{\sum_{j=1}^{e} \mu_j a_j \mid 0 < \mu_j \leq 1\} \tag{1}$$

En el caso general, podemos poner:

$$\mathbb{N}^n \cap \Delta_i = \bigcup^{\varnothing} (g + \mathbb{N}a_1 + \ldots + \mathbb{N}a_e)$$

con g recorriendo el conjunto (1). Luego:

$$S_{\Delta_i} = \left(\sum_g p^{-\sigma(g)-g\cdot x_\tau s}\right) \sum_{\lambda_1,\ldots,\lambda_e \in \mathbb{N}_{>0}} p^{-\sigma(\lambda_1 a_1+\ldots+\lambda_e a_e)-(\lambda_1 a_1+\ldots+\lambda_e a_e)\cdot x_\tau s}$$

Y como $\mathrm{Re}(s) > 0$, tenemos:

$$\begin{aligned} S_{\Delta_i} &= \left(\sum_g p^{-\sigma(g)-g\cdot x_\tau s}\right) \frac{p^{\sigma(a_1+\ldots+a_e)+(a_1+\ldots+a_e)\cdot x_\tau s}}{(p^{\sigma(a_1)+a_1\cdot x_\tau s}-1)\cdots(p^{\sigma(a_e)+a_e\cdot x_\tau s}-1)} \\ &= \frac{\sum_g p^{\sigma(a_1+\ldots+a_e-g)+(a_1+\ldots+a_e-g)\cdot x_\tau s}}{(p^{\sigma(a_1)+a_1\cdot x_\tau s}-1)\cdots(p^{\sigma(a_e)+a_e\cdot x_\tau s}-1)} \end{aligned}$$

con g recorriendo el conjunto (1). Los elementos a_j y $(a_1 + \ldots + a_e - g)$ pertenecen a $\bar{\Delta}_\tau$, luego $a_j \cdot x_\tau = m(a_j)$ y $(a_1 + \ldots + a_e - g) \cdot x_\tau = m(a_1 + \ldots + a_e - g)$.
De esta forma, reescribiendo $h = a_1 + \ldots + a_e - g$:

$$S_{\Delta_i} = \frac{\sum_h p^{\sigma(h)+m(h)s}}{(p^{\sigma(a_1)+m(a_1)s}-1)\cdots(p^{\sigma(a_e)+m(a_e)s}-1)}$$

donde h recorre $\mathbb{Z}^n \cap \{\sum_{i=1}^{r} \lambda_i a_i \mid 0 \leq \lambda_i < 1\}$.

□

Teorema 2.1. *Sea p primo y $f \in \mathbb{Z}_p[x_1, \ldots, x_n]$ polinomio no degenerado sobre $\mathbb{F}_p$ con respecto a todas las caras de su poliedro de Newton $\Gamma(f)$.*
Se tiene que:

$$Z_f(s) = L_{\Gamma(f)}(s) + \sum_{\substack{\tau \text{ cara propia} \\ \text{de } \Gamma(f)}} L_\tau(s) S_{\Delta_\tau}(s)$$

para $s \in \mathbb{C}$ con $\mathrm{Re}(s) > 0$.

Nota 2.5. Usando argumentos similares, podemos probar que si f es no degenerado sobre $\mathbb{F}_p$ con respecto a las caras compactas de su poliedro de Newton $\Gamma(f)$ entonces, para la forma local de la función zeta de Igusa:

$$Z_{f,0}(s) = \int_{(p\mathbb{Z}_p)^n} |f(x)|_p^s |\mathrm{d}x| = \sum_{\substack{\tau \text{ cara compacta} \\ \text{de } \Gamma(f)}} L_\tau(s) S_{\Delta_\tau}(s)$$

para $s \in \mathbb{C}$ con $\mathrm{Re}(s) > 0$.

```
def igusa_zeta(f, p = None, dict_Ntau = {}, local = False, weights = None, info = False):
    """
    The Igusa's Zeta Function por ``p`` prime (given or abstract), in terms of variable ``s``.\n
    For the abstract case (``p = None``), you must to give a dictionary ``dist_Ntau`` where the
    polynomes ftau for the faces of the Newton Polyhedron are the keys and the abstract value
    N_tau (depending of var p) as associated item. If ftau for face ``tauk`` is not in the
    dictionary, program introduces a new variable ``N_tauk``.\n
    ``local = True`` calcules the local (in the origin) Topological Zeta Function.\n
    ``weights`` is a list of weights if you want to considerate some ponderation.\n
    ``info = True`` gives information of face tau, cone of tau (all), L_tau, S_tau in the
    process.
    """
    s = var('s')
    if type(p) != Integer: p = var('p')
    P = newton_polyhedron(f)
    abs_Ngamma = None
    abs_Ntau = None
    if is_all_degenerated(f,P,p,local) == True: return NaN
    if local == True:
        faces_set = compact_faces(P)
        result = 0
    else:
        faces_set = proper_faces(P)
        if type(p) != Integer:
            abs_Ngamma = dict_Ntau.get(f)
            if abs_Ngamma == None: abs_Ngamma = var('N_Gamma')
        result = Lgamma(f,p,abs_Ngamma,s)
        if info == True: print "Gamma: total polyhedron\n" + "L_gamma = " + str(result) + "\n\n"
    for tau in faces_set:
        i = proper_faces(P).index(tau)
        if type(p) != Integer:
            f_tau = ftau(f, tau)
            if len(f_tau.exponents()) == 1: abs_Ntau = 0    #If f_tau is a monomial
            else:
                abs_Ntau = dict_Ntau.get(f_tau)
                if abs_Ntau == None: abs_Ntau = var('N_tau' + str(i))
        [L_tau, N_tau] = Ltau(f,tau,p,abs_Ntau,s)
        [S_tau, cone_info] = Stau(f,P,tau,p,weights,s)
        if info == True:
            print "tau" + str(i) + ": " + face_info_output(tau) + "\n" + cone_info + "\n" +\
                  "N_tau = " + str(N_tau) + ", L_tau = " + str(L_tau) + " , S_tau = " +\
                  str(S_tau) + "\n\n"
        result = factor(simplify(expand(result + L_tau*S_tau)))
    return result
```

Demostración Sabemos que los conos asociados a las caras de $\Gamma(f)$ forman una partición de

$(\mathbb{R}^+)^n$:

$$(\mathbb{R}^+)^n = \{0\} \cup \bigcup_{\substack{\tau \text{ cara propia} \\ \text{de } \Gamma(f)}} \Delta_\tau$$

De esta forma, tenemos:

$$\begin{aligned} Z_f(s) &= \int_{\mathbb{Z}_p^n} |f(x)|_p^s |\mathrm{d}x| = \sum_{k\in\mathbb{N}^n} \int_{\substack{x\in\mathbb{Z}_p^n \\ \mathrm{ord}_p\, x=k}} |f(x)|_p^s |\mathrm{d}x| \\ &= \int_{(\mathbb{Z}_p^\times)^n} |f(x)|_p^s |\mathrm{d}x| + \sum_{\substack{\tau \text{ cara propia} \\ \text{de } \Gamma(f)}} \sum_{k\in\mathbb{N}^n\cap\Delta_\tau} \int_{\substack{x\in\mathbb{Z}_p^n \\ \mathrm{ord}_p\, x=k}} |f(x)|_p^s |\mathrm{d}x| \end{aligned}$$

Donde ord_p se toma componente a componente.
Vamos a hacer un cambio de variable. Supongamos que τ es cara propia de $\Gamma(f)$, $k \in \mathbb{N}^n \cap \Delta_\tau$ y $x = (x_1, \ldots, x_n) \in \mathbb{Z}_p^n$ tal que $\mathrm{ord}_p\, x = k$. Pongamos $x_j = p^{k_j} u_j$ con $u_j \in \mathbb{Z}_p^\times$.
Tenemos que:

$$|\mathrm{d}x| = p^{-\sigma(k)} |\mathrm{d}u| \quad \text{y} \quad x^\omega = x_1^{\omega_1} \ldots x_n^{\omega_n} = p^{k\cdot\omega} u^\omega$$

Sabemos que $m(a) = \min_{\omega\in\mathrm{supp}(f)}\{k\cdot\omega\}$ y que $F(k) = \tau$, luego $k\cdot\omega = m(k)$ para todo $\omega \in \mathrm{supp}(f)\cap\tau$ y $k\cdot\omega > m(k)$ para todo $\omega \in \mathrm{supp}(f)\setminus\tau$. De esta forma, podemos expresar nuestro polinomio f de la siguiente manera con el cambio de variable:

$$f(x) = p^{m(k)}(f_\tau(u) + p\tilde{f}_{\tau,k}(u))$$

donde $\tilde{f}_{\tau,k} \in \mathbb{Z}_p[u_1, \ldots, u_n]$ dependiente de f, τ y k. Luego:

$$Z_f(s) = \int_{(\mathbb{Z}_p^\times)^n} |f(x)|_p^s |\mathrm{d}x| + \sum_{\substack{\tau \text{ cara propia} \\ \text{de } \Gamma(f)}} \sum_{k\in\mathbb{N}^n\cap\Delta_\tau} p^{-\sigma(k)-m(k)s} \int_{(\mathbb{Z}_p^\times)^n} |f_\tau(u) + p\tilde{f}_{\tau,k}(u)|_p^s |\mathrm{d}u|$$

Pongamos:

- $L_\Gamma(f) = \displaystyle\int_{(\mathbb{Z}_p^\times)^n} |f(x)|_p^s |\mathrm{d}x|$
- $L_\tau = \displaystyle\int_{(\mathbb{Z}_p^\times)^n} |f_\tau(u) + p\tilde{f}_{\tau,k}(u)|_p^s |\mathrm{d}u|$

Por hipótesis, f es no degenerada sobre $\mathbb{F}_p$ con respecto a todas las caras del poliedro de Newton, por lo que los sistemas de congruencias:

$$\begin{cases} f(x) \equiv 0 & \text{mód } p \\ \frac{\partial f}{\partial x_i}(x) \equiv 0 & \text{mód } p, \quad i = 1, \ldots \end{cases} \qquad \begin{cases} f_\tau(u) + p\tilde{f}_{\tau,k}(u) \equiv 0 & \text{mód } p \\ \frac{\partial f_\tau + p\tilde{f}_{\tau,k}}{\partial u_i}(u) \equiv 0 & \text{mód } p, \quad i = 1, \ldots \end{cases}$$

no tienes soluciones en $(\mathbb{Z}_p^\times)^n$. Luego, por el **Corolario 2.1** se tiene:

- $L_\Gamma(f) = p^{-n}\left((p-1)^n - pN_{\Gamma(f)}\dfrac{p^s-1}{p^{s+1}-1}\right)$
- $L_\tau = p^{-n}\left((p-1)^n - pN_\tau\dfrac{p^s-1}{p^{s+1}-1}\right)$

para toda τ cara propia de $\Gamma(f)$. Notar que L_τ no depende de k, solamente de la cara.
Por lo que, finalmente:

$$Z_f(s) = L_{\Gamma(f)} + \sum_{\substack{\tau \text{ cara propia} \\ \text{de } \Gamma(f)}} L_\tau S_{\Delta_\tau} \quad \text{con} \quad S_{\Delta_\tau} = \sum_{k \in \mathbb{N}^n \cap \Delta_\tau} p^{-\sigma(k) - m(k)s}$$

□

3. La función zeta Topológica

Vamos a introducir un invariante para gérmenes de polinomios complejos f que llamaremos *funciones zeta topológicas* $Z_{\text{top},f}$ locales y globales.

Las funciones $Z_{\text{top},f}$ son funciones racionales que se definen mediante resoluciones encajadas π del lugar de ceros $V(f) = \{f = 0\}$, en función de la característica de Euler de los estratos de $\pi^{-1}(f^{-1}(0))$ y las multiplicidades de las componentes irreducibles. Además, se prueba que éstas son independientes de la resolución π escogida.
Estas funciones zeta topológicas se obtienen como caso límite de las funciones zeta de Igusa para cuerpos p-ádicos asociadas a f, siendo los polos de $Z_{f,\text{top}}$ también polos de una infinidad de funciones zeta locales de Igusa p-ádicas. Como en el caso anterior, podremos dar una fórmula para las funciones zeta topológicas utilizando el poliedro de Newton, bajo ciertas condiciones de no degeneración sobre sus caras.

Sea un entero $d \geq 1$, denotamos

$$S^{(d)} = \{I \subset J \mid \forall i \in I,\ d | N_i\}$$

En particular, notar que $S^{(1)} = \mathcal{P}(J)$.

Definición 3.1. Sea un entero $d \geq 1$.

1. Sea $f \in \mathbb{C}[x_1, \ldots, x_n]$ polinomio no nulo y π resolución de f, definimos:

$$Z^{(d)}_{\text{top},f,\pi}(s) = \sum_{I \in S^{(d)}} \chi(\mathring{E}_I) \prod_{i \in I} \frac{1}{N_i s + \nu_i}$$

2. Sea $f : (\mathbb{C}^n, 0) \to (\mathbb{C}, 0)$ germen de función analítica no nula y $\pi : X \to B$ resolución de f, definimos:

$$Z^{(d)}_{\text{top},f,\pi,0}(s) = \sum_{I \in S^{(d)}} \chi(\mathring{E}_I \cap \pi^{-1}(0)) \prod_{i \in I} \frac{1}{N_i s + \nu_i}$$

Teorema 3.1. *Las funciones $Z^{(d)}_{top,f,\pi}$ y $Z^{(d)}_{top,f,\pi}$ son independientes de la resolución π elegida.*

Nota 3.1. Posteriormente, Denef-Loeser redemostraron estos resultados introduciendo la función zeta motívica (ver [**DenLoe98**]).
Denotaremos a estas funciones $Z^{(d)}_{\text{top},f}$ y $Z^{(d)}_{\text{top},f,0}$: *funciones zeta topológicas global y local*, respectivamente.

Análogamente a la función zeta de Igusa, vamos a dar una fórmula de las funciones zeta topológicas en función del poliedro de Newton.

Nota 3.2. Sea γ una envolvente convexa de puntos de $\mathbb{Z}^n$, denotamos por ω_γ la forma de volumen sobre el espacio afín generado por γ, $\text{Aff}(\gamma)$, tal que el paralelepípedo engendrado por una base del $\mathbb{Z}$-modulo asociado a $\text{Aff}(\gamma) \cap \mathbb{Z}^n$ tenga volumen 1.

Definición 3.2. Sea τ cara del poliedro de Newton, definimos $\text{Vol}(\tau)$ como el volumen de $\tau \cap \Gamma_{\text{gl}}(f)$ sobre la forma de volumen ω_τ.
Si $\dim \tau = 0$, ponemos $\text{Vol}(\tau) = 1$.

Nota 3.3. Si τ es cara compacta, $\tau \cap \Gamma_{\text{gl}}(f) = \tau$.

```
def face_volume(f,tau):
    """
    Returns the value Vol(tau)*(dim tau)!, for a face tau.\n
    The points of the face tau are contained in RR^dim and Vol(tau) is defined as follows:
    Let omega[tau] be the volume form on Aff(tau) such that the parallelopiped spanned by a
    lattice  basis of ZZ^n intersect (aff tau)_0 has volume 1. Then Vol(tau) is the volume of
    tau intersection the Global Newton Polyhedron with respect to omega[tau].
    """
    n = f.parent().ngens()
    dim_tau = dim_face(tau)
    result = 0
    if dim_tau != 0:
        tau_in_global = Polyhedron(vertices = support_points_in_face(f,tau))
        vertices_in_global = map(vector,tau_in_global.vertices())
        trans_vertices = translate_points(vertices_in_global)
        if matrix(ZZ,trans_vertices).rank() == dim_tau:
            V = QQ^n
            basis_aff = V.submodule(trans_vertices).intersection(ZZ^n).basis()
            W = V.submodule_with_basis(basis_aff)
            coords_list = map(W.coordinate_vector, trans_vertices)
            p = PointConfiguration(coords_list)
            result = p.volume()       # Returns dimtau!*n-volumen de tau
    else:
        result = 1
    return result
```

Definición 3.3. Para cada cara τ de $\Gamma(f)$, vamos a asociarle la función racional $J(\tau, s)$ como sigue: Tomemos una descomposición del cono asociado a la cara $\Delta_\tau = \bigcup_{i=1}^{r} \Delta_i$ en conos simples de dimensión $\dim \Delta_\tau = l$ tales que $\dim(\Delta_i \cap \Delta_j) < l, \forall i \neq j$. Ahora:

$$J(\tau, s) = \sum_{i=1}^{r} J_{\Delta_i}(s) \quad \text{con} \quad J_{\Delta_i}(s) = \frac{\text{mult}(\Delta_i)}{(\sigma(a_{i_1}) + m(a_{i_1})s) \cdots (\sigma(a_{i_l}) + m(a_{i_l})s)}$$

siendo $a_{i_1}, \ldots, a_{i_l} \in \mathbb{N}^n$ los vectores primitivos linealmente independientes que generan Δ_i.
Si $\tau = \Gamma(f)$, tomamos $J(\tau, s) = 1$.

```
def Jtau(tau,P,weights,s):
    """
    Returns a list [J_tau, cone_info] with ``cone_info`` containing a string of information
    about the cones, simplicial partition, multiplicity and integral points.\n
    Value J_tau is in terms of variable ``t``.
    """
    c = cone_from_face(tau)
    dim_cone = c.dim()
    F = simplicial_partition(c)
    if dim_cone == 0: result = 1
    else:
        result = 0
        for scone in filter(lambda i: i.dim()==dim_cone, F):
            num = multiplicity(scone)
            den = 1
            for a in primitive_vectors_cone(scone): den = den*(m(a,P)*s + sigma(a,weights))
            result = factor(simplify(expand(result + num/den)))
    cone_info = cone_info_output(c,F)+ "\n" + "multiplicities = " + str(map(multiplicity,F)) + \
                ", integral points = " + str(map(integral_vectors,F))
    return [result, cone_info]
```

Lema 3.1. *La función $J(\tau, s)$ es independiente de la descomposición de Δ_τ en conos simpliciales. Además, los polos de $J(\tau, s)$ son de la forma $-\frac{\sigma(a)}{m(a)}$ con a un vector primitivo generador de Δ_τ.*

Demostración Se puede comprobar que

$$J(\tau, s) = \int_{k\in\Delta_\tau} e^{-(\sigma(k)+m(k)s)}\omega_{\Delta_\tau}$$

la cual es independiente de la descomposición.
Fijemos x_τ en la cara τ, se tiene que $m(k) = k\cdot x_\tau$ para todo $k \in \Delta_\tau$. Tomemos una descomposición cualquiera de Δ_τ en conos simpliciales Δ_i (los conos de dimensión inferior tendrán volumen 0 respecto a ω_{Δ_τ}). Ahora, supongamos Δ_i cono simplicial estrictamente generado por $a_1, \ldots, a_l \in \mathbb{N}^n$ linealmente independientes, se tiene que $\omega_{\Delta_\tau} = \omega_{\Delta_i}$, ya que los rayos generadores de Δ_i forman una base del $\mathbb{Z}$-módulo:

$$\int_{k\in\Delta_i} e^{-(\sigma(k)+k\cdot x_\tau s)}\omega_{\Delta_\tau} = \int_{k\in\Delta_i} e^{-(\sigma(k)+k\cdot x_\tau s)}\omega_{\Delta_i}$$

Haciendo un cambio de variable a la base del cono simple, por la **Proposición 1.6** sabemos que el determinante del cambio es la multiplicidad del cono:

$$\begin{aligned}
&\int_{\{\lambda_j\in\mathbb{R}_{>0},\forall j\}} e^{-(\sigma(\lambda_1 a_l+\ldots+\lambda_l a_l)+(\lambda_1 a_1+\ldots+\lambda_l a_l)\cdot x_\tau s)}\,\mathrm{mult}(\Delta_i)\mathrm{d}\lambda_1\cdots\mathrm{d}\lambda_l \\
&\quad= \mathrm{mult}(\Delta_i)\int_{\{\lambda_j\in\mathbb{R}_{>0},\forall j\}} e^{-(\sum_{j=1}^l \sigma(\lambda_j a_j)+\sum_{j=1}^l \lambda_j a_j\cdot x_\tau s)}\mathrm{d}\lambda_l\cdots\mathrm{d}\lambda_r \\
&\quad= \mathrm{mult}(\Delta_i)\int_{\{\lambda_j\in\mathbb{R}_{>0},\forall j\}} \prod_{j=1}^l e^{-(\lambda_j\sigma(a_j)+\lambda_j a_j\cdot x_\tau s)}\mathrm{d}\lambda_1\cdots\mathrm{d}\lambda_l \\
&\quad= \mathrm{mult}(\Delta_i)\prod_{j=1}^l\int_0^\infty e^{-\lambda_j(\sigma(a_j)+a_j\cdot x_\tau s)}\mathrm{d}\lambda_j \\
&\quad= \mathrm{mult}(\Delta_i)\prod_{j=1}^l\left[-\frac{e^{-\lambda_j(\sigma(a_j)+a_j\cdot x_\tau s)}}{\sigma(a_j)+a_j\cdot x_\tau s}\right]_0^\infty \\
&\quad= \mathrm{mult}(\Delta_i)\prod_{j=1}^l -\frac{1}{\sigma(a_j)+a_j\cdot x_\tau s} \\
&\quad= \frac{\mathrm{mult}(\Delta_i)}{(\sigma(a_1)+a_1\cdot x_\tau s)\cdots(\sigma(a_l)+a_l\cdot x_\tau s)}
\end{aligned}$$

Y $a_j \cdot x_\tau = m(a_j)$ ya que $a_j \in \bar{\Delta}_\tau$, para todo $j = 1, \ldots, l$. La segunda afirmación se justifica a que podemos tomar una descomposición racional simplicial por conos simples generados por subconjuntos de los vectores generadores de Δ_τ. □

Nota 3.4. Para $A \subset (\mathbb{R}^+)^n$, consideraremos $m(A) = \gcd\{m(a) \mid a \in A \cap \mathbb{Z}^n\}$.

Teorema 3.2. *Sea $f \in \mathbb{C}[x_1, \ldots, x_n]$ polinomio no degenerado sobre $\mathbb{C}$ con respecto a todas las*

caras de su poliedro de Newton global $\Gamma_{gl}(f)$*, entonces:*

- $$Z^{(1)}_{top,f}(s) = \sum_{\substack{\tau \text{ vértice} \\ \text{de } \Gamma(f)}} J(\tau,s) + \left(\frac{s}{s+1}\right) \sum_{\substack{\tau \text{ cara de } \Gamma(f) \\ \dim\tau \geq 1}} (-1)^{\dim\tau}(\dim\tau)!\,\mathrm{Vol}(\tau)J(\tau,s)$$
- $$Z^{(d)}_{top,f}(s) = \sum_{\substack{\tau \text{ cara de } \Gamma(f) \\ d \mid m(\Delta_\tau)}} (-1)^{\dim\tau}(\dim\tau)!\,\mathrm{Vol}(\tau)J(\tau,s), \quad \textit{si } d > 1.$$

Nota 3.5. De forma similar, para $f : (\mathbb{C}^n, 0) \to (\mathbb{C}, 0)$ un germen de función analítica no degenerada sobre $\mathbb{C}$ con respecto a todas las caras del poliedro de Newton, entonces tenemos fórmulas análogas para $Z^{(1)}_{\mathrm{top},f}$ y $Z^{(d)}_{\mathrm{top},f}$ con $d > 1$ pero considerando únicamente las caras compactas de $\Gamma(f)$ en los respectivos sumatorios.

```
def is_global_degenerated(f, p = None, method = 'default'):
    """
    Checks if own polynome ``f`` over F_p with respect all the faces of the Global Newton's
    Polyhedron.
    If p = None, checks degeneration  over CC and (equivalent to be degenerated over F_p with
    p>>0).
    For finite fields (``p`` is a given prime):
        - ``method = 'default'`` check the condition using evaluation over the (F_p^x)^n in the
    system of equations
        - ``method = 'ideals'`` check the condition using ideals over the finite field.
    """
    Q = f.newton_polytope()    #Global Newton Polyhedron of f
    bool = False
    for tau in faces(Q)[1:]:
        f_tau = ftau(f,tau)
        if is_degenerated(f_tau, p, method) == True:
            bool = True
            print "The formula for Topological Zeta function is not valid:"
            if type(p) != Integer: print "The polynomial is degenerated at least with respect "\
                                         "to the face tau = {" + face_info_output(tau) + \
                                         "} over the complex numbers!"
            else: print "The polynomial is degenerated at least with respect to the face tau "\
                        "= {" + face_info_output(tau) + "} over GF(" + str(p) + ")!"
            break
    return bool
```

```
def face_divisors(d,faces_set,P):
    """
    Returns a list of faces in ``faces_set`` such that d divides m(Delta_tau) = gcd{m(a)| a in
    Delta_tau and ZZ^n}.
    """
    if d ==1: return faces_set
    L_faces = list()
    dim_total = P.dim()
    for tau in faces_set:
        c = cone_from_face(tau)
        F = simplicial_partition(c)
        L_vectors = list()
        #We need to evaluate m over the basis of the cone and the integral points views above.
        for scone in F:
            L_vectors = L_vectors + integral_vectors(scone) + primitive_vectors_cone(scone)
        l = gcd(map(lambda i: m(i,P), L_vectors))
        if d.divides(l): L_faces.append(tau)
    return L_faces
```

```
def topological_zeta(f, d = 1, local = False, weights = None, info = False):
    """
    The Topological Zeta Function Z_{top, f}^(d) for ``d``>=1, in terms of variable ``s``.\n
    ``local = True`` calcules the local (in the origin) Topological Zeta Function.\n
    ``weights`` is a list of weights if you want to considerate some ponderation.\n
    ``info = True`` gives information of face tau, cone of tau (all), L_tau, S_tau in the
    process.
    """
    s = var('s')
    P = newton_polyhedron(f)
    result = NaN
    if is_global_degenerated(f) == False:
        result = 0
        if local == True: faces_set = compact_faces(P)
        else:
            faces_set = proper_faces(P)
            if d == 1:
                total_face = faces(P)[-1]
                dim_gamma = dim_face(total_face)
                vol_gamma = face_volume(f,total_face)
                result = (s/(s+1))*((-1)^dim_gamma)*vol_gamma
                if info == True: print "Gamma: total polyhedron\n" + "J_gamma = 1 , "\
                                       "dim_Gamma!*Vol(Gamma) = " + str(vol_gamma) +"\n\n"
        faces_set = face_divisors(d,faces_set,P)
        for tau in faces_set:
            [J_tau, cone_info] = Jtau(tau,P,weights,s)
            dim_tau = dim_face(tau)
            vol_tau = face_volume(f,tau)
            if info == True:
                i = proper_faces(P).index(tau)
                print "tau" + str(i) + ": " + face_info_output(tau) + "\n" + cone_info + "\n" +\
                      "J_tau = " + str(J_tau) + " , dim_tau!*Vol(tau) = " + str(vol_tau) + "\n\n"
            if d == 1:
                if dim_tau == 0: term = J_tau
                else: term = (s/(s+1))*((-1)^dim_tau)*vol_tau*J_tau
            else:
                term = ((-1)^dim_tau)*vol_tau*J_tau
            result = simplify(expand(result + term))
            if result != 0: result = factor(result)
    return result
```

Demostración Se puede encontrar en [**DenLoe92**] usando las resoluciones encajadas introducidas por [**Var**]. □

4. Anexo: Ejemplos de ZetaFunctions.sage

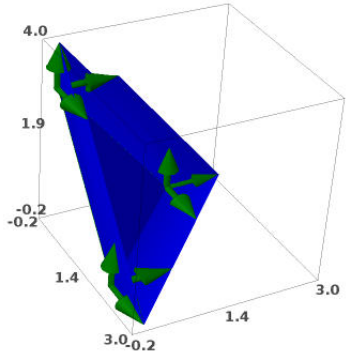
A modo de comparativa con el programa original de Hoornaert en Maple, se incluyen varios ejemplos sacados de la literatura y que ya fueron analizados en [**HooLoo**].

Ejemplos para la **Funcion Zeta de Igusa**

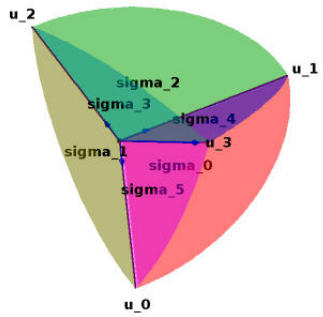
Ejemplo 3: $x^2 - y^2 + z^3$

```
R.<x,y,z> = QQ[]
zex3 = ZetaFunctions(x^2 - y^2 + z^3)

zex3.give_info_newton(faces = True)

Newton's polyhedron of z^3 + x^2 - y^2:
        support points = [(0, 0, 3), (2, 0, 0), (0, 2, 0)]
        vertices = [(0, 0, 3), (0, 2, 0), (2, 0, 0)]
        number of proper faces = 13
        Facet 1: y >= 0
        Facet 2: z >= 0
        Facet 3: x >= 0
        Facet 4: 3*x + 3*y + 2*z - 6 >= 0
Information about faces:
tau0: dim 0,  vertices = [(0, 0, 3)],  rays = []

tau1: dim 0,  vertices = [(0, 2, 0)],  rays = []

tau2: dim 0,  vertices = [(2, 0, 0)],  rays = []

tau3: dim 1,  vertices = [(0, 0, 3)],  rays = [(0, 0, 1)]

tau4: dim 1,  vertices = [(0, 0, 3), (0, 2, 0)],  rays = []

tau5: dim 1,  vertices = [(0, 2, 0)],  rays = [(0, 1, 0)]

tau6: dim 1,  vertices = [(0, 0, 3), (2, 0, 0)],  rays = []

tau7: dim 1,  vertices = [(0, 2, 0), (2, 0, 0)],  rays = []

tau8: dim 1,  vertices = [(2, 0, 0)],  rays = [(1, 0, 0)]

tau9: dim 2,  vertices = [(0, 0, 3), (2, 0, 0)],  rays = [(0, 0, 1), (1, 0, 0)]

tau10: dim 2,  vertices = [(0, 2, 0), (2, 0, 0)],  rays = [(0, 1, 0), (1, 0, 0)]

tau11: dim 2,  vertices = [(0, 0, 3), (0, 2, 0)],  rays = [(0, 0, 1), (0, 1, 0)]

tau12: dim 2,  vertices = [(0, 0, 3), (0, 2, 0), (2, 0, 0)],  rays = []
```

```
zex3.newton_plot()
```




```
zex3.cones_plot()
```



$p = 3$:

```
zex3.igusa_zeta(3)
```

```
2*(3^(2*s + 4) - 3^(s + 1) + 2)*3^(2*s)/((3^(s + 1) - 1)*(3^(3*s + 4) - 1))
```

p arbitrario, sin informacin sobre las caras:

```
zex3.igusa_zeta()
```

```
(N_Gamma - N_tau10 - N_tau11 - N_tau12 + N_tau4 + N_tau6 + N_tau7 - N_tau9 + p^(7*s +
12) - p^(7*s + 11) - p^(7*s + 9) + p^(7*s + 8) - p^(6*s + 11) + p^(6*s + 10) + p^(6*s
+ 8) - p^(6*s + 7) + p^(5*s + 9) - p^(5*s + 8) - p^(5*s + 7) + p^(5*s + 6) - p^(4*s +
8) + 2*p^(4*s + 7) - p^(4*s + 6) + p^(3*s + 5) - 2*p^(3*s + 4) + p^(3*s + 3) - p^(2*s
+ 5) + p^(2*s + 4) + p^(2*s + 3) - p^(2*s + 2) - p^s*N_Gamma + p^s*N_tau10 +
p^s*N_tau11 + p^s*N_tau12 - p^s*N_tau4 - p^s*N_tau6 - p^s*N_tau7 + p^s*N_tau9 -
N_Gamma*p - N_Gamma*p^(7*s + 9) + N_Gamma*p^(7*s + 8) + N_Gamma*p^(6*s + 9) -
N_Gamma*p^(6*s + 8) + N_Gamma*p^(s + 1) - N_tau10*p^(7*s + 8) + N_tau10*p^(6*s + 8) -
N_tau11*p^(7*s + 8) + N_tau11*p^(6*s + 8) + N_tau12*p - N_tau12*p^(s + 1) -
N_tau7*p^(5*s + 6) + N_tau7*p^(4*s + 6) - N_tau7*p^(3*s + 3) + N_tau7*p^(2*s + 3) -
N_tau9*p^(7*s + 8) + N_tau9*p^(6*s + 8))/((p^(s + 1) - 1)*(p^(3*s + 4) - 1)*(p^(3*s +
4) + 1)*(p - 1)*p^2)
```

p arbitrario, con numero de soluciones sobre las caras

```
dNtau3 = { x^2-y^2+z^3 : (p-1)*(p-3), -y^2+z^3 : (p-1)^2, \
           x^2+z^3 : (p-1)^2, x^2-y^2 : 2*(p-1)^2 }

zex3.igusa_zeta(dict_Ntau = dNtau3)
```

```
(p - 1)*(p + p^(2*s + 4) - p^(s + 1) - 1)*p^(2*s)/((p^(s + 1) - 1)*(p^(3*s + 4) - 1))
```

Ejemplo 4: $(x-y)^2+z$

```
zex4 = ZetaFunctions((x - y)^2 + z)
```

$p = 7$:

```
zex4.igusa_zeta(7)
```

```
The formula for Igusa Zeta function is not valid:
The polynomial is degenerated at least with respect to the face tau = {dim 1,
vertices = [(0, 2, 0), (2, 0, 0)],  rays = []} over GF(7)!
NaN
```

p arbitrario:

```
zex4.igusa_zeta()
```

```
The formula for Igusa Zeta function is not valid:
The polynomial is degenerated at least with respect to the face tau = {dim 1,
vertices = [(0, 2, 0), (2, 0, 0)],  rays = []} over the complex numbers!
NaN
```

Ejemplo 5: x^2+yz+z^2

```
zex5 = ZetaFunctions(x^2 + y*z + z^2)
```

$p = 3 \bmod 4$, podemos dar soluciones sobre las caras:

```
dNtau5 = { x^2+y*z+z^2 : (p-1)^2, y*z+z^2 : (p-1)^2,\
           x^2+y*z : (p-1)^2, x^2+z^2 : 0 }
zex5.igusa_zeta(dict_Ntau = dNtau5, info = True)
```

```
Gamma: total polyhedron
L_gamma = -((p - 1)^2*(p^s - 1)*p/(p^(s + 1) - 1) - (p - 1)^3)/p^3


tau0: dim 0,  vertices = [(0, 0, 2)],  rays = []
generators of cone = [(0, 1, 0), (1, 0, 0), (1, 1, 1)], partition into simplicial
cones = [[(0, 1, 0), (1, 0, 0), (1, 1, 1)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
```

```
N_tau = 0, L_tau = (p - 1)^3/p^3 , S_tau = 1/((p^(2*s + 3) - 1)*(p - 1)^2)


tau1: dim 0,  vertices = [(0, 1, 1)],  rays = []
generators of cone = [(1, 0, 0), (1, 0, 2), (1, 1, 1)], partition into simplicial
cones = [[(1, 0, 0), (1, 0, 2), (1, 1, 1)]]
multiplicities = [2], integral points = [[(0, 0, 0), (1, 0, 1)]]
N_tau = 0, L_tau = (p - 1)^3/p^3 , S_tau = (p^(s + 2) + 1)/((p^(2*s + 3) - 1)^2*(p -
1))


tau2: dim 0,  vertices = [(2, 0, 0)],  rays = []
generators of cone = [(0, 1, 0), (0, 0, 1), (1, 0, 2), (1, 1, 1)], partition into
simplicial cones = [[(1, 0, 2), (0, 1, 0)], [(1, 0, 2), (0, 0, 1), (0, 1, 0)], [(1,
0, 2), (1, 1, 1), (0, 1, 0)]]
multiplicities = [1, 1, 1], integral points = [[(0, 0, 0)], [(0, 0, 0)], [(0, 0, 0)]]
N_tau = 0, L_tau = (p - 1)^3/p^3 , S_tau = (p^(s + 2) - 1)*(p^(s + 2) + 1)/((p^(2*s +
3) - 1)^2*(p - 1)^2)


tau3: dim 1,  vertices = [(0, 0, 2)],  rays = [(0, 0, 1)]
generators of cone = [(0, 1, 0), (1, 0, 0)], partition into simplicial cones = [[(0,
1, 0), (1, 0, 0)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
N_tau = 0, L_tau = (p - 1)^3/p^3 , S_tau = (p - 1)^(-2)


tau4: dim 1,  vertices = [(0, 0, 2), (0, 1, 1)],  rays = []
generators of cone = [(1, 0, 0), (1, 1, 1)], partition into simplicial cones = [[(1,
0, 0), (1, 1, 1)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
N_tau = (p - 1)^2, L_tau = (p - 1)^2*(p^(s + 2) - 2*p^(s + 1) + 1)/((p^(s + 1) -
1)*p^3) , S_tau = 1/((p^(2*s + 3) - 1)*(p - 1))


tau5: dim 1,  vertices = [(0, 1, 1)],  rays = [(0, 1, 0)]
generators of cone = [(1, 0, 0), (1, 0, 2)], partition into simplicial cones = [[(1,
0, 0), (1, 0, 2)]]
multiplicities = [2], integral points = [[(0, 0, 0), (1, 0, 1)]]
N_tau = 0, L_tau = (p - 1)^3/p^3 , S_tau = (p^(s + 2) + 1)/((p^(2*s + 3) - 1)*(p -
1))


tau6: dim 1,  vertices = [(0, 0, 2), (2, 0, 0)],  rays = []
generators of cone = [(0, 1, 0), (1, 1, 1)], partition into simplicial cones = [[(0,
1, 0), (1, 1, 1)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
N_tau = 0, L_tau = (p - 1)^3/p^3 , S_tau = 1/((p^(2*s + 3) - 1)*(p - 1))
```

```
tau7: dim 1,  vertices = [(2, 0, 0)],  rays = [(0, 1, 0)]
generators of cone = [(0, 0, 1), (1, 0, 2)], partition into simplicial cones = [[(0,
0, 1), (1, 0, 2)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
N_tau = 0, L_tau = (p - 1)^3/p^3 , S_tau = 1/((p^(2*s + 3) - 1)*(p - 1))


tau8: dim 1,  vertices = [(0, 1, 1), (2, 0, 0)],  rays = []
generators of cone = [(1, 0, 2), (1, 1, 1)], partition into simplicial cones = [[(1,
0, 2), (1, 1, 1)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
N_tau = (p - 1)^2, L_tau = (p - 1)^2*(p^(s + 2) - 2*p^(s + 1) + 1)/((p^(s + 1) -
1)*p^3) , S_tau = (p^(2*s + 3) - 1)^(-2)


tau9: dim 1,  vertices = [(2, 0, 0)],  rays = [(1, 0, 0)]
generators of cone = [(0, 1, 0), (0, 0, 1)], partition into simplicial cones = [[(0,
1, 0), (0, 0, 1)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
N_tau = 0, L_tau = (p - 1)^3/p^3 , S_tau = (p - 1)^(-2)


tau10: dim 2,  vertices = [(0, 0, 2), (2, 0, 0)],  rays = [(0, 0, 1), (1, 0, 0)]
generators of cone = [(0, 1, 0)], partition into simplicial cones = [[(0, 1, 0)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
N_tau = 0, L_tau = (p - 1)^3/p^3 , S_tau = 1/(p - 1)


tau11: dim 2,  vertices = [(0, 0, 2), (0, 1, 1)],  rays = [(0, 0, 1), (0, 1, 0)]
generators of cone = [(1, 0, 0)], partition into simplicial cones = [[(1, 0, 0)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
N_tau = (p - 1)^2, L_tau = (p - 1)^2*(p^(s + 2) - 2*p^(s + 1) + 1)/((p^(s + 1) -
1)*p^3) , S_tau = 1/(p - 1)


tau12: dim 2,  vertices = [(2, 0, 0)],  rays = [(0, 1, 0), (1, 0, 0)]
generators of cone = [(0, 0, 1)], partition into simplicial cones = [[(0, 0, 1)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
N_tau = 0, L_tau = (p - 1)^3/p^3 , S_tau = 1/(p - 1)


tau13: dim 2,  vertices = [(0, 1, 1), (2, 0, 0)],  rays = [(0, 1, 0)]
generators of cone = [(1, 0, 2)], partition into simplicial cones = [[(1, 0, 2)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
N_tau = (p - 1)^2, L_tau = (p - 1)^2*(p^(s + 2) - 2*p^(s + 1) + 1)/((p^(s + 1) -
1)*p^3) , S_tau = 1/(p^(2*s + 3) - 1)
```

```
tau14: dim 2,  vertices = [(0, 0, 2), (0, 1, 1), (2, 0, 0)],  rays = []
generators of cone = [(1, 1, 1)], partition into simplicial cones = [[(1, 1, 1)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
N_tau = (p - 1)^2, L_tau = (p - 1)^2*(p^(s + 2) - 2*p^(s + 1) + 1)/((p^(s + 1) -
1)*p^3) , S_tau = 1/(p^(2*s + 3) - 1)
```

```
(p^(s + 3) - 1)*(p - 1)*p^(2*s)/((p^(s + 1) - 1)*(p^(2*s + 3) - 1))
```

$p = 1 \bmod 4$:

```
dNtau5bis = { x^2+y*z+z^2 : (p-1)*(p-3), y*z+z^2 : (p-1)^2, \
              x^2+y*z : (p-1)^2, x^2+z^2 : 2*(p-1)^2 }
zex5.igusa_zeta(dict_Ntau = dNtau5bis)
```

```
(p^(s + 3) - 1)*(p - 1)*p^(2*s)/((p^(s + 1) - 1)*(p^(2*s + 3) - 1))
```

Ejemplo 6: $x^2 * z + y^2 * z + u^3$

```
S.<x,y,z,u> = QQ[]
zex6 = ZetaFunctions(x^2*z + y^2*z + u^3)
```

$p = 1 \bmod 4$ con soluciones sobre las caras:

```
dNtau6 = { x^2*z+y^2*z+u^3 : (p-1)^2*(p-3), x^2*z+u^3 : (p-1)^3, \
           y^2*z + u^3: (p-1)^3,  x^2*z+y^2*z : 2*(p-1)^3}
zex6.igusa_zeta(dict_Ntau = dNtau6)
```

```
(p - 1)*(p^(4*s + 8) - 3*p^(3*s + 5) + 2*p^(3*s + 4) + 3*p^(2*s + 5) - 6*p^(2*s + 4)
+ 3*p^(2*s + 3) + 2*p^(s + 4) - 3*p^(s + 3) + 1)*p^(3*s)/((p^(s + 1) - 1)*(p^(3*s +
4) - 1)^2)
```

Local con $p = 1 \bmod 4$ con soluciones sobre las caras:

```
zex6.igusa_zeta(local = True, dict_Ntau = dNtau6)
```

```
(p - 1)*(p^(4*s + 8) - 3*p^(3*s + 5) + 2*p^(3*s + 4) + 3*p^(2*s + 5) - 6*p^(2*s + 4)
+ 3*p^(2*s + 3) + 2*p^(s + 4) - 3*p^(s + 3) + 1)/((p^(s + 1) - 1)*(p^(3*s + 4) -
1)^2*p^4)
```

Local con $p = 3 \bmod 4$ con soluciones sobre las caras:

```
dNtau6bis = { x^2*z+y^2*z+u^3 : (p-1)^3, x^2*z+u^3 : (p-1)^3, \
              y^2*z + u^3: (p-1)^3,  x^2*z+y^2*z : 0}
zex6.igusa_zeta(local = True, dict_Ntau = dNtau6bis, info = True)
```

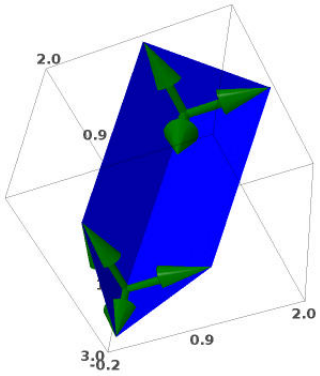
```
tau0: dim 0,  vertices = [(0, 0, 0, 3)],  rays = []
generators of cone = [(0, 1, 0, 0), (0, 0, 3, 1), (1, 0, 0, 0), (0, 0, 1, 0), (3, 3,
0, 2)], partition into simplicial cones = [[(1, 0, 0, 0), (0, 0, 3, 1), (0, 1, 0,
0)], [(0, 1, 0, 0), (1, 0, 0, 0), (0, 0, 3, 1), (0, 0, 1, 0)], [(3, 3, 0, 2), (1, 0,
0, 0), (0, 0, 3, 1), (0, 1, 0, 0)]]
multiplicities = [1, 1, 6], integral points = [[(0, 0, 0, 0)], [(0, 0, 0, 0)], [(0,
0, 0, 0), (3, 3, 1, 2), (2, 2, 2, 2), (2, 2, 0, 1), (1, 1, 1, 1), (1, 1, 2, 1)]]
N_tau = 0, L_tau = (p - 1)^4/p^4 , S_tau = (p^(6*s + 10) + p^(6*s + 9) - p^(6*s + 8)
+ 2*p^(3*s + 6) - p^(3*s + 5) - p^(3*s + 4) - 1)/((p^(3*s + 4) - 1)^2*(p^(3*s + 4) +
1)*(p - 1)^3)


tau1: dim 0,  vertices = [(0, 2, 1, 0)],  rays = []
generators of cone = [(0, 0, 0, 1), (0, 0, 3, 1), (1, 0, 0, 0), (3, 3, 0, 2)],
partition into simplicial cones = [[(0, 0, 0, 1), (0, 0, 3, 1), (1, 0, 0, 0), (3, 3,
0, 2)]]
multiplicities = [9], integral points = [[(0, 0, 0, 0), (1, 1, 0, 1), (2, 2, 0, 2),
(0, 0, 1, 1), (1, 1, 1, 1), (2, 2, 1, 2), (0, 0, 2, 1), (1, 1, 2, 2), (2, 2, 2, 2)]]
N_tau = 0, L_tau = (p - 1)^4/p^4 , S_tau = (p^(6*s + 8) + p^(5*s + 7) + 2*p^(4*s + 6)
+ p^(3*s + 4) + 2*p^(2*s + 3) + p^(s + 2) + 1)/((p^(3*s + 4) - 1)^2*(p^(3*s + 4) +
1)*(p - 1)^2)


tau2: dim 0,  vertices = [(2, 0, 1, 0)],  rays = []
generators of cone = [(0, 1, 0, 0), (0, 0, 0, 1), (0, 0, 3, 1), (3, 3, 0, 2)],
partition into simplicial cones = [[(0, 1, 0, 0), (0, 0, 0, 1), (0, 0, 3, 1), (3, 3,
0, 2)]]
multiplicities = [9], integral points = [[(0, 0, 0, 0), (1, 1, 0, 1), (2, 2, 0, 2),
(0, 0, 1, 1), (1, 1, 1, 1), (2, 2, 1, 2), (0, 0, 2, 1), (1, 1, 2, 2), (2, 2, 2, 2)]]
N_tau = 0, L_tau = (p - 1)^4/p^4 , S_tau = (p^(6*s + 8) + p^(5*s + 7) + 2*p^(4*s + 6)
+ p^(3*s + 4) + 2*p^(2*s + 3) + p^(s + 2) + 1)/((p^(3*s + 4) - 1)^2*(p^(3*s + 4) +
1)*(p - 1)^2)


tau11: dim 1,  vertices = [(0, 0, 0, 3), (0, 2, 1, 0)],  rays = []
generators of cone = [(0, 0, 3, 1), (1, 0, 0, 0), (3, 3, 0, 2)], partition into
simplicial cones = [[(0, 0, 3, 1), (1, 0, 0, 0), (3, 3, 0, 2)]]
multiplicities = [3], integral points = [[(0, 0, 0, 0), (1, 1, 1, 1), (2, 2, 2, 2)]]
N_tau = (p - 1)^3, L_tau = (p - 1)^3*(p^(s + 2) - 2*p^(s + 1) + 1)/((p^(s + 1) -
1)*p^4) , S_tau = (p^(6*s + 8) + p^(3*s + 4) + 1)/((p^(3*s + 4) - 1)^2*(p^(3*s + 4) +
1)*(p - 1))


tau17: dim 1,  vertices = [(0, 0, 0, 3), (2, 0, 1, 0)],  rays = []
generators of cone = [(0, 1, 0, 0), (0, 0, 3, 1), (3, 3, 0, 2)], partition into
simplicial cones = [[(0, 1, 0, 0), (0, 0, 3, 1), (3, 3, 0, 2)]]
multiplicities = [3], integral points = [[(0, 0, 0, 0), (1, 1, 1, 1), (2, 2, 2, 2)]]
N_tau = (p - 1)^3, L_tau = (p - 1)^3*(p^(s + 2) - 2*p^(s + 1) + 1)/((p^(s + 1) -
```

```
1)*p^4) , S_tau = (p^(6*s + 8) + p^(3*s + 4) + 1)/((p^(3*s + 4) - 1)^2*(p^(3*s + 4) +
1)*(p - 1))


tau20: dim 1,  vertices = [(0, 2, 1, 0), (2, 0, 1, 0)],  rays = []
generators of cone = [(0, 0, 0, 1), (0, 0, 3, 1), (3, 3, 0, 2)], partition into
simplicial cones = [[(0, 0, 0, 1), (0, 0, 3, 1), (3, 3, 0, 2)]]
multiplicities = [9], integral points = [[(0, 0, 0, 0), (0, 0, 1, 1), (0, 0, 2, 1),
(1, 1, 0, 1), (1, 1, 1, 1), (1, 1, 2, 2), (2, 2, 0, 2), (2, 2, 1, 2), (2, 2, 2, 2)]]
N_tau = 0, L_tau = (p - 1)^4/p^4 , S_tau = (p^(6*s + 8) + p^(5*s + 7) + 2*p^(4*s + 6)
+ p^(3*s + 4) + 2*p^(2*s + 3) + p^(s + 2) + 1)/((p^(3*s + 4) - 1)^2*(p^(3*s + 4) +
1)*(p - 1))


tau22: dim 2,  vertices = [(0, 0, 0, 3), (0, 2, 1, 0), (2, 0, 1, 0)],  rays = []
generators of cone = [(0, 0, 3, 1), (3, 3, 0, 2)], partition into simplicial cones =
[[(0, 0, 3, 1), (3, 3, 0, 2)]]
multiplicities = [3], integral points = [[(0, 0, 0, 0), (1, 1, 1, 1), (2, 2, 2, 2)]]
N_tau = (p - 1)^3, L_tau = (p - 1)^3*(p^(s + 2) - 2*p^(s + 1) + 1)/((p^(s + 1) -
1)*p^4) , S_tau = (p^(6*s + 8) + p^(3*s + 4) + 1)/((p^(3*s + 4) - 1)^2*(p^(3*s + 4) +
1))


(p^(s + 2) + 1)*(p - 1)*(p^(3*s + 6) - p^(2*s + 4) - p^(2*s + 3) + p^(s + 3) + p^(s +
2) - 1)/((p^(s + 1) - 1)*(p^(3*s + 4) - 1)*(p^(3*s + 4) + 1)*p^4)
```

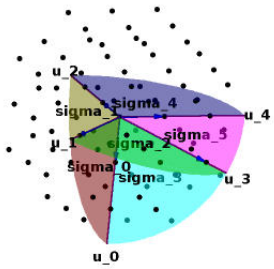
Ejemplos para la **Funcion Zeta Topolgica**

Ejemplo 10: $x^2 + yz$

```
zex10 = ZetaFunctions(R(x^2 + y*z))

zex10.give_info_newton()

Newton's polyhedron of x^2 + y*z:
        support points = [(2, 0, 0), (0, 1, 1)]
        vertices = [(0, 1, 1), (2, 0, 0)]
        number of proper faces = 13
        Facet 1: x >= 0
        Facet 2: y >= 0
        Facet 3: z >= 0
        Facet 4: x + 2*z - 2 >= 0
        Facet 5: x + 2*y - 2 >= 0


zex10.newton_plot()
```




```
zex10.cones_plot()
```




```
zex10.give_expected_pole_info()
```

```
The candidate poles of the (local) topological zeta function (with d = 1) of x^2 +
y*z in function of s are:

-3/2 with expected order: 2
The responsible face of maximal dimension is ‘‘tau_0‘‘ = minimal face who intersecs
with the diagonal of ambient space:
         tau6: dim 1,  vertices = [(0, 1, 1), (2, 0, 0)],  rays = []
        generators of cone = [(1, 0, 2), (1, 2, 0)], partition into simplicial cones
= [[(1, 0, 2), (1, 2, 0)]]

-1 with expected order: 1
(If all Vol(tau) are 0, where tau runs through the selected faces that are no
vertices, then the expected order of -1 is 0).
```

```
zex10.topological_zeta(info = True)
```

```
Gamma: total polyhedron
J_gamma = 1 , dim_Gamma!*Vol(Gamma) = 0
```

```
tau0: dim 0,  vertices = [(0, 1, 1)],  rays = []
generators of cone = [(1, 0, 0), (1, 0, 2), (1, 2, 0)], partition into simplicial
cones = [[(1, 0, 0), (1, 0, 2), (1, 2, 0)]]
multiplicities = [4], integral points = [[(0, 0, 0), (1, 1, 0), (1, 0, 1), (1, 1,
1)]]
J_tau = 4/(2*s + 3)^2 , dim_tau!*Vol(tau) = 1


tau1: dim 0,  vertices = [(2, 0, 0)],  rays = []
generators of cone = [(0, 1, 0), (0, 0, 1), (1, 0, 2), (1, 2, 0)], partition into
simplicial cones = [[(1, 0, 2), (0, 1, 0)], [(1, 0, 2), (0, 0, 1), (0, 1, 0)], [(1,
0, 2), (1, 2, 0), (0, 1, 0)]]
multiplicities = [1, 1, 2], integral points = [[(0, 0, 0)], [(0, 0, 0)], [(0, 0, 0),
(1, 1, 1)]]
J_tau = (2*s + 5)/(2*s + 3)^2 , dim_tau!*Vol(tau) = 1


tau2: dim 1,  vertices = [(0, 1, 1)],  rays = [(0, 0, 1)]
generators of cone = [(1, 0, 0), (1, 2, 0)], partition into simplicial cones = [[(1,
0, 0), (1, 2, 0)]]
multiplicities = [2], integral points = [[(0, 0, 0), (1, 1, 0)]]
J_tau = 2/(2*s + 3) , dim_tau!*Vol(tau) = 0


tau3: dim 1,  vertices = [(0, 1, 1)],  rays = [(0, 1, 0)]
generators of cone = [(1, 0, 0), (1, 0, 2)], partition into simplicial cones = [[(1,
0, 0), (1, 0, 2)]]
multiplicities = [2], integral points = [[(0, 0, 0), (1, 0, 1)]]
J_tau = 2/(2*s + 3) , dim_tau!*Vol(tau) = 0


tau4: dim 1,  vertices = [(2, 0, 0)],  rays = [(0, 0, 1)]
generators of cone = [(0, 1, 0), (1, 2, 0)], partition into simplicial cones = [[(0,
1, 0), (1, 2, 0)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
J_tau = 1/(2*s + 3) , dim_tau!*Vol(tau) = 0


tau5: dim 1,  vertices = [(2, 0, 0)],  rays = [(0, 1, 0)]
generators of cone = [(0, 0, 1), (1, 0, 2)], partition into simplicial cones = [[(0,
0, 1), (1, 0, 2)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
J_tau = 1/(2*s + 3) , dim_tau!*Vol(tau) = 0


tau6: dim 1,  vertices = [(0, 1, 1), (2, 0, 0)],  rays = []
```

```
generators of cone = [(1, 0, 2), (1, 2, 0)], partition into simplicial cones = [[(1,
0, 2), (1, 2, 0)]]
multiplicities = [2], integral points = [[(0, 0, 0), (1, 1, 1)]]
J_tau = 2/(2*s + 3)^2 , dim_tau!*Vol(tau) = 1


tau7: dim 1,  vertices = [(2, 0, 0)],  rays = [(1, 0, 0)]
generators of cone = [(0, 1, 0), (0, 0, 1)], partition into simplicial cones = [[(0,
1, 0), (0, 0, 1)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
J_tau = 1 , dim_tau!*Vol(tau) = 0


tau8: dim 2,  vertices = [(0, 1, 1)],  rays = [(0, 0, 1), (0, 1, 0)]
generators of cone = [(1, 0, 0)], partition into simplicial cones = [[(1, 0, 0)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
J_tau = 1 , dim_tau!*Vol(tau) = 0


tau9: dim 2,  vertices = [(2, 0, 0)],  rays = [(0, 0, 1), (1, 0, 0)]
generators of cone = [(0, 1, 0)], partition into simplicial cones = [[(0, 1, 0)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
J_tau = 1 , dim_tau!*Vol(tau) = 0


tau10: dim 2,  vertices = [(2, 0, 0)],  rays = [(0, 1, 0), (1, 0, 0)]
generators of cone = [(0, 0, 1)], partition into simplicial cones = [[(0, 0, 1)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
J_tau = 1 , dim_tau!*Vol(tau) = 0


tau11: dim 2,  vertices = [(0, 1, 1), (2, 0, 0)],  rays = [(0, 1, 0)]
generators of cone = [(1, 0, 2)], partition into simplicial cones = [[(1, 0, 2)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
J_tau = 1/(2*s + 3) , dim_tau!*Vol(tau) = 0


tau12: dim 2,  vertices = [(0, 1, 1), (2, 0, 0)],  rays = [(0, 0, 1)]
generators of cone = [(1, 2, 0)], partition into simplicial cones = [[(1, 2, 0)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
J_tau = 1/(2*s + 3) , dim_tau!*Vol(tau) = 0


(s + 3)/((s + 1)*(2*s + 3))
```

Ejemplo 11: $x^2 + yz$
$d = 2$:

```
zex11 = zex10
```

```
zex11.give_expected_pole_info(d = 2)
```

```
-3/2 with expected order: 2
The responsible face(s) of maximal dimension is/are:
         tau6: dim 1,  vertices = [(0, 1, 1), (2, 0, 0)],  rays = []
        generators of cone = [(1, 0, 2), (1, 2, 0)], partition into simplicial cones
= [[(1, 0, 2), (1, 2, 0)]]
```

```
zex11.topological_zeta(d = 2, info = True)
```

```
tau1: dim 0,  vertices = [(2, 0, 0)],  rays = []
generators of cone = [(0, 1, 0), (0, 0, 1), (1, 0, 2), (1, 2, 0)], partition into
simplicial cones = [[(1, 0, 2), (0, 1, 0)], [(1, 0, 2), (0, 0, 1), (0, 1, 0)], [(1,
0, 2), (1, 2, 0), (0, 1, 0)]]
multiplicities = [1, 1, 2], integral points = [[(0, 0, 0)], [(0, 0, 0)], [(0, 0, 0),
(1, 1, 1)]]
J_tau = (2*s + 5)/(2*s + 3)^2 , dim_tau!*Vol(tau) = 1


tau4: dim 1,  vertices = [(2, 0, 0)],  rays = [(0, 0, 1)]
generators of cone = [(0, 1, 0), (1, 2, 0)], partition into simplicial cones = [[(0,
1, 0), (1, 2, 0)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
J_tau = 1/(2*s + 3) , dim_tau!*Vol(tau) = 0


tau5: dim 1,  vertices = [(2, 0, 0)],  rays = [(0, 1, 0)]
generators of cone = [(0, 0, 1), (1, 0, 2)], partition into simplicial cones = [[(0,
0, 1), (1, 0, 2)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
J_tau = 1/(2*s + 3) , dim_tau!*Vol(tau) = 0


tau6: dim 1,  vertices = [(0, 1, 1), (2, 0, 0)],  rays = []
generators of cone = [(1, 0, 2), (1, 2, 0)], partition into simplicial cones = [[(1,
0, 2), (1, 2, 0)]]
multiplicities = [2], integral points = [[(0, 0, 0), (1, 1, 1)]]
J_tau = 2/(2*s + 3)^2 , dim_tau!*Vol(tau) = 1


tau7: dim 1,  vertices = [(2, 0, 0)],  rays = [(1, 0, 0)]
generators of cone = [(0, 1, 0), (0, 0, 1)], partition into simplicial cones = [[(0,
1, 0), (0, 0, 1)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
J_tau = 1 , dim_tau!*Vol(tau) = 0


tau8: dim 2,  vertices = [(0, 1, 1)],  rays = [(0, 0, 1), (0, 1, 0)]
```

```
generators of cone = [(1, 0, 0)], partition into simplicial cones = [[(1, 0, 0)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
J_tau = 1 , dim_tau!*Vol(tau) = 0


tau9: dim 2,  vertices = [(2, 0, 0)],  rays = [(0, 0, 1), (1, 0, 0)]
generators of cone = [(0, 1, 0)], partition into simplicial cones = [[(0, 1, 0)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
J_tau = 1 , dim_tau!*Vol(tau) = 0


tau10: dim 2,  vertices = [(2, 0, 0)],  rays = [(0, 1, 0), (1, 0, 0)]
generators of cone = [(0, 0, 1)], partition into simplicial cones = [[(0, 0, 1)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
J_tau = 1 , dim_tau!*Vol(tau) = 0


tau11: dim 2,  vertices = [(0, 1, 1), (2, 0, 0)],  rays = [(0, 1, 0)]
generators of cone = [(1, 0, 2)], partition into simplicial cones = [[(1, 0, 2)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
J_tau = 1/(2*s + 3) , dim_tau!*Vol(tau) = 0


tau12: dim 2,  vertices = [(0, 1, 1), (2, 0, 0)],  rays = [(0, 0, 1)]
generators of cone = [(1, 2, 0)], partition into simplicial cones = [[(1, 2, 0)]]
multiplicities = [1], integral points = [[(0, 0, 0)]]
J_tau = 1/(2*s + 3) , dim_tau!*Vol(tau) = 0


1/(2*s + 3)
```

Ejemplo 12: $x^2y^2z + xyz^2$
$d = 2$:

```
zex12 = ZetaFunctions(R(x^2*y^2*z + x*y*z^2))

zex12.give_expected_pole_info(d = 2)

There will be no poles for the (local) topological zeta function (with d = 2) of
x^2*y^2*z + x*y*z^2.


zex12.topological_zeta(d = 2)

0
```

Ejemplo 14: $xyz + uvw + xyw + zuv$

```
S2.<x,y,z,u,v,w> = QQ[]
zex14 = ZetaFunctions(x*y*z + u*v*w + x*y*w + z*u*v)

zex14.give_info_newton()

Newton's polyhedron of x*y*z + z*u*v + x*y*w + u*v*w:
        support points = [(1, 1, 1, 0, 0, 0), (0, 0, 1, 1, 1, 0), (1, 1, 0, 0, 0, 1),
(0, 0, 0, 1, 1, 1)]
        vertices = [(0, 0, 0, 1, 1, 1), (0, 0, 1, 1, 1, 0), (1, 1, 0, 0, 0, 1), (1,
1, 1, 0, 0, 0)]
        number of proper faces = 203
        Facet 1: z + w - 1 >= 0
        Facet 2: w >= 0
        Facet 3: u >= 0
        Facet 4: v >= 0
        Facet 5: x + v - 1 >= 0
        Facet 6: x + u - 1 >= 0
        Facet 7: y + u - 1 >= 0
        Facet 8: y + v - 1 >= 0
        Facet 9: y >= 0
        Facet 10: x >= 0
        Facet 11: z >= 0


zex14.give_expected_pole_info()

The candidate poles of the (local) topological zeta function (with d = 1) of x*y*z +
z*u*v + x*y*w + u*v*w in function of s are:

-2 with expected order: 4
The responsible face of maximal dimension is ``tau_0`` = minimal face who intersecs
with the diagonal of ambient space:
         tau178: dim 2,  vertices = [(0, 0, 0, 1, 1, 1), (0, 0, 1, 1, 1, 0), (1, 1,
0, 0, 0, 1), (1, 1, 1, 0, 0, 0)],  rays = []
        generators of cone = [(0, 0, 1, 0, 0, 1), (1, 0, 0, 0, 1, 0), (1, 0, 0, 1, 0,
0), (0, 1, 0, 1, 0, 0), (0, 1, 0, 0, 1, 0)], partition into simplicial cones = [[(0,
0, 1, 0, 0, 1), (1, 0, 0, 0, 1, 0), (0, 1, 0, 1, 0, 0)], [(0, 0, 1, 0, 0, 1), (1, 0,
0, 0, 1, 0), (1, 0, 0, 1, 0, 0), (0, 1, 0, 1, 0, 0)], [(0, 0, 1, 0, 0, 1), (0, 1, 0,
0, 1, 0), (0, 1, 0, 1, 0, 0), (1, 0, 0, 0, 1, 0)]]

-1 with expected order: 1
(If all Vol(tau) are 0, where tau runs through the selected faces that are no
vertices, then the expected order of -1 is 0).


zex14.topological_zeta()

The formula for Topological Zeta function is not valid:
The polynomial is degenerated at least with respect to the face tau = {dim 2,
```

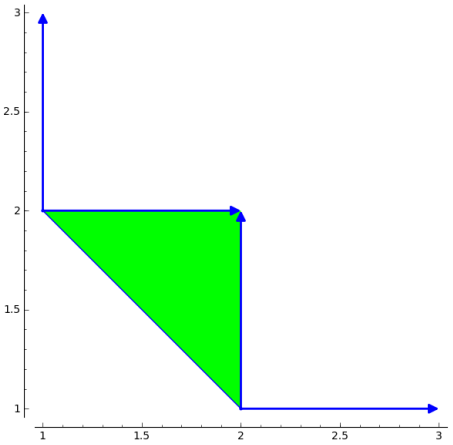
```
vertices = [(0, 0, 0, 1, 1, 1), (0, 0, 1, 1, 1, 0), (1, 1, 0, 0, 0, 1), (1, 1, 1, 0,
0, 0)],  rays = []} over the complex numbers!
NaN
```

Ejemplo 15: $xy^3 + xy^2 + x^2y$

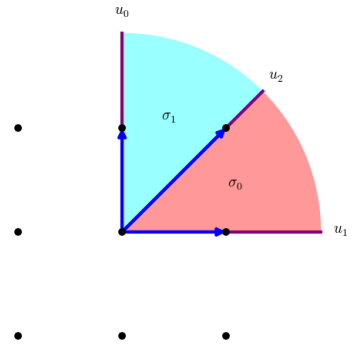
```
R2.<x,y> = QQ[]
zex15 = ZetaFunctions(x*y^3 + x*y^2 + x^2*y)
```

(Existe un problema en Sage a la hora de representar poliedros con rayos)

```
zex15.newton_plot()
```




```
zex15.cones_plot()
```



Local:

```
zex15.give_expected_pole_info(local = True)
```

```
The candidate poles of the (local) topological zeta function (with d = 1) of x*y^3 +
x^2*y + x*y^2 in function of s are:
```

```
-2/3 with expected order: 1
The responsible face of maximal dimension is ``tau_0`` = minimal face who intersecs
with the diagonal of ambient space:
         tau4: dim 1,  vertices = [(1, 2), (2, 1)],  rays = []
        generators of cone = [(1, 1)], partition into simplicial cones = [[(1, 1)]]

-1 with expected order: 1
The responsible face(s) of maximal dimension is/are:
         tau1: dim 0,  vertices = [(2, 1)],  rays = []
        generators of cone = [(0, 1), (1, 1)], partition into simplicial cones =
[[(0, 1), (1, 1)]]

         tau0: dim 0,  vertices = [(1, 2)],  rays = []
        generators of cone = [(1, 0), (1, 1)], partition into simplicial cones =
[[(1, 0), (1, 1)]]
```

```
zex15.topological_zeta(local = True, info = True)
```

```
tau0: dim 0,  vertices = [(1, 2)],  rays = []
generators of cone = [(1, 0), (1, 1)], partition into simplicial cones = [[(1, 0),
(1, 1)]]
multiplicities = [1], integral points = [[(0, 0)]]
J_tau = 1/((s + 1)*(3*s + 2)) , dim_tau!*Vol(tau) = 1


tau1: dim 0,  vertices = [(2, 1)],  rays = []
generators of cone = [(0, 1), (1, 1)], partition into simplicial cones = [[(0, 1),
(1, 1)]]
multiplicities = [1], integral points = [[(0, 0)]]
J_tau = 1/((s + 1)*(3*s + 2)) , dim_tau!*Vol(tau) = 1


tau4: dim 1,  vertices = [(1, 2), (2, 1)],  rays = []
generators of cone = [(1, 1)], partition into simplicial cones = [[(1, 1)]]
multiplicities = [1], integral points = [[(0, 0)]]
J_tau = 1/(3*s + 2) , dim_tau!*Vol(tau) = 1


-(s - 2)/((s + 1)*(3*s + 2))
```

Ejemplo 19: $x_1x_2x_3^2x_4 + x_1x_2^2x_3x_4 + x_1^2x_2x_3x_4^2$

```
T.<x_1,x_2,x_3,x_4> = QQ[]
zex19 = ZetaFunctions(x_1*x_2*x_3^2*x_4 + x_1*x_2^2*x_3*x_4 + x_1^2*x_2*x_3*x_4^2)

zex19.give_info_newton()
```

```
Newton's polyhedron of x_1^2*x_2*x_3*x_4^2 + x_1*x_2^2*x_3*x_4 + x_1*x_2*x_3^2*x_4:
        support points = [(2, 1, 1, 2), (1, 2, 1, 1), (1, 1, 2, 1)]
        vertices = [(1, 1, 2, 1), (1, 2, 1, 1), (2, 1, 1, 2)]
        number of proper faces = 33
        Facet 1: x_2 - 1 >= 0
        Facet 2: x_3 - 1 >= 0
        Facet 3: x_1 - 1 >= 0
        Facet 4: x_4 - 1 >= 0
        Facet 5: x_2 + x_3 + x_4 - 4 >= 0
        Facet 6: x_1 + x_2 + x_3 - 4 >= 0
```

```
zex19.give_expected_pole_info()
```

```
The candidate poles of the (local) topological zeta function (with d = 1) of
x_1^2*x_2*x_3*x_4^2 + x_1*x_2^2*x_3*x_4 + x_1*x_2*x_3^2*x_4 in function of s are:

-3/4 with expected order: 2
The responsible face of maximal dimension is ``tau_0`` = minimal face who intersecs
with the diagonal of ambient space:
         tau26: dim 2,  vertices = [(1, 1, 2, 1), (1, 2, 1, 1), (2, 1, 1, 2)],  rays
= []
        generators of cone = [(0, 1, 1, 1), (1, 1, 1, 0)], partition into simplicial
cones = [[(0, 1, 1, 1), (1, 1, 1, 0)]]

-1 with expected order: 3
The responsible face(s) of maximal dimension is/are:
         tau5: dim 1,  vertices = [(1, 1, 2, 1)],  rays = [(0, 0, 1, 0)]
        generators of cone = [(0, 1, 0, 0), (1, 0, 0, 0), (0, 0, 0, 1)], partition
into simplicial cones = [[(0, 1, 0, 0), (1, 0, 0, 0), (0, 0, 0, 1)]]

         tau9: dim 1,  vertices = [(1, 2, 1, 1)],  rays = [(0, 1, 0, 0)]
        generators of cone = [(0, 0, 1, 0), (1, 0, 0, 0), (0, 0, 0, 1)], partition
into simplicial cones = [[(0, 0, 1, 0), (1, 0, 0, 0), (0, 0, 0, 1)]]
```

```
zex19.topological_zeta()
```

```
(s^3 - 5*s^2 + 6*s + 9)/((s + 1)^3*(4*s + 3)^2)
```

Ejemplo 21: $x_1^2 + x_2^3x_4^3 + x_3^3x_5^3$

```
T2.<x_1,x_2,x_3,x_4,x_5> = QQ[]
zex21 = ZetaFunctions(x_1^2 + x_2^3*x_4^3 + x_3^3*x_5^3)

zex21.give_info_newton()
```

```
Newton's polyhedron of x_2^3*x_4^3 + x_3^3*x_5^3 + x_1^2:
        support points = [(0, 3, 0, 3, 0), (0, 0, 3, 0, 3), (2, 0, 0, 0, 0)]
        vertices = [(0, 0, 3, 0, 3), (0, 3, 0, 3, 0), (2, 0, 0, 0, 0)]
        number of proper faces = 85
        Facet 1: x_2 >= 0
        Facet 2: x_4 >= 0
        Facet 3: x_3 >= 0
        Facet 4: x_5 >= 0
        Facet 5: x_1 >= 0
        Facet 6: 3*x_1 + 2*x_3 + 2*x_4 - 6 >= 0
        Facet 7: 3*x_1 + 2*x_4 + 2*x_5 - 6 >= 0
        Facet 8: 3*x_1 + 2*x_2 + 2*x_5 - 6 >= 0
        Facet 9: 3*x_1 + 2*x_2 + 2*x_3 - 6 >= 0
```

```
zex21.give_expected_pole_info()
```

```
The candidate poles of the (local) topological zeta function (with d = 1) of
x_2^3*x_4^3 + x_3^3*x_5^3 + x_1^2 in function of s are:

-7/6 with expected order: 3
The responsible face of maximal dimension is ``tau_0`` = minimal face who intersecs
with the diagonal of ambient space:
         tau57: dim 2,  vertices = [(0, 0, 3, 0, 3), (0, 3, 0, 3, 0), (2, 0, 0, 0,
0)],  rays = []
        generators of cone = [(3, 0, 2, 2, 0), (3, 0, 0, 2, 2), (3, 2, 0, 0, 2), (3,
2, 2, 0, 0)], partition into simplicial cones = [[(3, 0, 2, 2, 0), (3, 2, 0, 0, 2)],
[(3, 0, 0, 2, 2), (3, 2, 0, 0, 2), (3, 0, 2, 2, 0)], [(3, 2, 0, 0, 2), (3, 2, 2, 0,
0), (3, 0, 2, 2, 0)]]

-1 with expected order: 1
(If all Vol(tau) are 0, where tau runs through the selected faces that are no
vertices, then the expected order of -1 is 0).
```

```
zex21.topological_zeta()
```

```
(108*s^3 + 456*s^2 + 647*s + 343)/((s + 1)*(6*s + 7)^3)
```

Ejemplos para la **Funcion Zeta de la Monodromia en el origen**

```
zexmon1 = ZetaFunctions(R2(y^7+x^2*y^5+x^5*y^3))
zexmon1.monodromy_zeta(char = True)
```

```
The characteristic polynomial of the monodromy is (T - 1)^3*(T^6 + T^5 + T^4 + T^3 +
T^2 + T + 1)*(T^18 + T^17 + T^16 + T^15 + T^14 + T^13 + T^12 + T^11 + T^10 + T^9 +
T^8 + T^7 + T^6 + T^5 + T^4 + T^3 + T^2 + T + 1)

1/((t^7 - 1)*(t^19 - 1))
```

```
zexmon2 = ZetaFunctions(R(x*y + z^3))
zexmon2.monodromy_zeta(char = True)

The characteristic polynomial of the monodromy is T^2 + T + 1

-t^3 + 1


zexmon3 = ZetaFunctions(R((3*x+5*z)*(x+2*z)+y^3))
zexmon3.monodromy_zeta(char = True)

The characteristic polynomial of the monodromy is T^2 + T + 1

-t^3 + 1


zexmon4 = ZetaFunctions(R(x*(y+x)+x^2*z+z^3))
zexmon4.monodromy_zeta(char = True)

The characteristic polynomial of the monodromy is T^2 + T + 1

-t^3 + 1


zexmon4 = ZetaFunctions(R(x*y*(x+y)+z^4))
zexmon4.monodromy_zeta(char = True)

The characteristic polynomial of the monodromy is (T + 1)^2*(T^2 + 1)^2*(T^2 - T +
1)*(T^4 - T^2 + 1)

-(t^4 - 1)*(t^12 - 1)/(t^3 - 1)
```

5. Apéndice: Código completo de ZetaFunctions.sage para Sage

```
class ZetaFunctions(object):
    def __init__(self, poly):
        #Polynome
        self._f = poly
        #Newton's polyhedron
        self._Gammaf = newton_polyhedron(poly)

    def give_info_facets(self):
        """
        Prints a relation of facets in Newton's polyhedron and their inequalities.
        """
        give_all_facets_info(self._f,self._Gammaf)

    def give_info_newton(self, faces = False, cones = False):
        """
        Prints information about the the Newton's polyhedron of ``f``:\n
            - Support points of f.
            - Vertices of Newton's polyhedron
            - Numer of proper faces
            - Inequations defining facets
        ``faces = True`` prints information about each face of polyhedron.
        ``cones = True`` prints information about each cone associated to faces of polyhedron.\n
        """
        print "Newton's polyhedron of " + str(self._f) + ":"
        print "\t" + "support points = " + str(support_points(self._f))
        print "\t" + "vertices = " + str(map(tuple,self._Gammaf.vertices()))
        print "\t" + "number of proper faces = " + str(len(proper_faces(self._Gammaf)))
        give_all_facets_info(self._f,self._Gammaf)
        if faces or cones:
            print "Information about faces:"
            faces_set = proper_faces(self._Gammaf)
            for tau in faces_set:
                face_info, cone_info = str(), str()
                i = faces_set.index(tau)
                if faces: face_info = face_info_output(tau) + "\n"
                if cones: cone_info = cone_info_output(cone_from_face(tau)) + "\n"
                print "tau" + str(i) + ": " + face_info + cone_info

    def newton_plot(self):
        """
        Plot Newton's polyhedron (for n = 2 , 3).
        """
        show(self._Gammaf.plot())

    def cones_plot(self):
        """
        Plot the Fan of cones associated to Newton's polyhedron (for n = 2 , 3).
        (Cones can be no simplicial).
        """
        F = fan_all_cones(self._Gammaf)
        show(F.plot())

    def give_expected_pole_info(self,d = 1, local = False, weights = None):
        """
        Prints information about the candidate real poles for the topological zeta function of
        ``f``: the orders and responsible faces of highest dimension.
        ``local = True`` calcules the local (in the origin) Topological Zeta Function.\n
```

```
        ``weights`` is a list of weights if you want to considerate some ponderation.\n
        """
        give_expected_pole_info(self._f,d, local, weights)

    def igusa_zeta(self, p = None, dict_Ntau = {}, local = False, weights = None, info = False):
        """
        The Igusa's Zeta Function por ``p`` prime (given or abstract), in terms of variable
        ``s``.\n
        For the abstract case (``p = None``), you must to give a dictionary ``dist_Ntau`` where
        the polynomes ftau for the faces of the Newton Polyhedron are the keys and the abstract
        value N_tau (depending of var p) as associated item. If ftau for face ``tauk`` is not in
        the dictionary, program introduces a new variable ``N_tauk``.\n
        ``local = True`` calcules the local (in the origin) Topological Zeta Function.\n
        ``weights`` is a list of weights if you want to considerate some ponderation.\n
        ``info = True`` gives information of face tau, cone of tau (all), L_tau, S_tau
        in the process.
        """
        return igusa_zeta(self._f, p, dict_Ntau, local, weights, info)

    def topological_zeta(self, d = 1, local = False, weights = None, info = False):
        """
        The Topological Zeta Function Z_{top, f}^(d) for ``d``>=1, in terms of variable ``s``.\n
        ``local = True`` calcules the local (in the origin) Topological Zeta Function.\n
        ``weights`` is a list of weights if you want to considerate some ponderation.\n
        ``info = True`` gives information of face tau, cone of tau (all), L_tau, S_tau
        in the process.
        """
        return topological_zeta(self._f, d, local, weights, info)

    def monodromy_zeta(self, weights = None, char = False, info = False):
        """
        The Monodromy Zeta Function in the origin, in terms of variable ``s``.\n
        ``weights`` is a list of weights if you want to considerate some ponderation.\n
        ``char = True`` prints the characteriristic polynomial of the monodromy (only if ``f``
        has an isolated singularity in the origin).\n
        ``info = True`` gives information of face tau, cone of tau (all), L_tau, S_tau in the
        process.
        """
        return monodromy_zeta(self._f, weights, char, info)

###-----------------------FUNCIONES AUXILIARES-----------------------###
##---NEWTON'S POLYHEDRON
def support_points(f):
    """
    Support of f: points of ZZ^n corresponding to the exponents of the monomial into ``f``.
    """
    points = f.exponents()
    return points

def newton_polyhedron(f):
    """
    Construction of Newton's Polyhedron Gamma(f) for the polynomial ``f``.
    """
    P = Polyhedron(vertices = support_points(f), rays=VectorSpace(QQ,f.parent().ngens()).basis())
    return P


##--- FACES
def faces(P):
```

```
    """
    Returns a LatticePoset of the faces in the polyhedron ``P`` with a relation of order
    (content between the faces).
    """
    P_lattice = LatticePoset(P.face_lattice())
    return P_lattice

def proper_faces(P):
    """
    Returns a list with the proper faces of the polyhedron ``P`` sorted in increasing order
    dimension.
    """
    L = faces(P).list()[1:-1]
    return L

#-- Informations about faces
def face_Vinfo(tau):
    """
    Returns a list containing the descriptions of the face in terms of vertices and rays.
    """
    return tau.element.ambient_Vrepresentation()

def face_Hinfo(tau):
    """
    Returns a list containing the descriptions of the face in terms of the inequalities of the
    facets who intersects into the face.
    """
    return tau.element.ambient_Hrepresentation()

def contains_a_ray(tau):
    """
    Checks if the face contains some ray.
    """
    bool = False
    Vrep = face_Vinfo(tau)
    for e in Vrep:
        if e.is_ray() == True:
            bool = True
            break
    return bool

def compact_faces(P):
    """
    Returns a list with the compact faces of the polyhedron ``P`` sorted in increasing order
    dimension.
    """
    pfaces = proper_faces(P)
    return filter(lambda i: not contains_a_ray(i), pfaces)

def vertices(tau):
    """
    Returns a list with the vertices of the face.
    """
    L = map(lambda i:i.vector(),filter(lambda j:j.is_vertex(),face_Vinfo(tau)))
    return L

def rays(tau):
    """
    Returns a list with the rays of the face.
```

```
    """
    L = map(lambda i:i.vector(),filter(lambda j:j.is_ray(),face_Vinfo(tau)))
    return L

def translate_points(points_list):
    """
    Returns a list of points taking the first point in the original list how the origin and
    rewriting the other points in terms of new origin.
    """
    origin = points_list[0]
    L = map(lambda v: v - origin, points_list)
    return L

def dim_face(tau):
    """
    Gives the dimension of the face.
    """
    vertices_tau = vertices(tau)
    rays_tau = rays(tau)
    if len(vertices_tau) == 0 and len(vertices_tau) == 0: dim_tau = -1
    else:
        v_list = translate_points(vertices_tau)
        dim_tau = matrix(v_list + rays_tau).rank()
    return dim_tau

def facets(P):
    """
    Returns a list of facets ((n-1)-dimensional faces) of the polyhedron ``P``.
    """
    dim = P.dim()
    L = filter(lambda i: dim_face(i) == dim-1, proper_faces(P))
    return L

def facet_info(f, facet):
    """
    Returns a string with the inequation of the facet write in form
    a_1*x_1 + a_2*x_2 + ... + a_n*x_n + b >= 0.
    """
    rep = face_Hinfo(facet)[0]
    message = str(vector(rep.A()).dot_product(vector(f.parent().gens())) + rep.b())
    message = message  + " >= 0"
    return message

def give_all_facets_info(f,P):
    """
    Prints a relation of facets in ``P`` and their inequalities.
    """
    i = 1
    for facet in facets(P):
        print "\tFacet " + str(i) + ": " + facet_info(f, facet)
        i = i + 1

def face_info_output(tau):
    """
    Returns a string containing a descrition of vertices and rays in face.
    """
    info =  "dim " + str(dim_face(tau)) + ",  vertices = " + str(vertices(tau)) + \
            ",  rays = " + str(rays(tau))
    return info
```

```
#-- Relations Polyhedron-points
def point_in_face(point,tau):
    """
    Checks if point belongs to the face.
    """
    bool = Polyhedron(vertices = vertices(tau), rays = rays(tau)).contains(point)
    return bool

def support_points_in_face(f, tau):
    """
    Returns a list of support points of ``f`` contained in the face.
    """
    L = filter(lambda i: point_in_face(i,tau),support_points(f))
    return L


##---CONES, FANES AND SIMPLE CONES
def prim(v):
    """
    Returns the primitivitation of an integral vector.
    """
    return v/gcd(v)

def primitive_vectors(tau):
    """
    Returns a list of primitive vectors of a face (normal vectors of the hyperplanes who defines
    the face, components are relatively primes).
    """
    L = map(lambda i:prim(i.A()),face_Hinfo(tau))
    return L

def cone_from_face(tau):
    """
    Construction of the dual cone of the face. In particular, for the total face it gives a cone
    generated by the zero vector.
    """
    gens = primitive_vectors(tau)
    if len(gens) == 0: cone = Cone([vertices(tau)[0].parent()(0)])
    else: cone = Cone(gens)
    return cone

def primitive_vectors_cone(cone):
    """
    Returns a list of primitive vectors (rays generators) of cone.
    """
    L = map(lambda i:prim(i.sparse_vector()),cone.rays())
    return L

def all_cones(P):
    """
    Returns a list with all the cones generated by the faces of ``P``.
    """
    L = map(cone_from_face, faces(P)[1:])
    return L

def fan_all_cones(P):
    """
    Fan of all cones of a Polyhedron.
```

```
    """
    F = Fan(all_cones(P),discard_faces=True)
    return F

def same_facet(lcone_gens, p, bcone_gens):
    """
    Checks if ``lcone_gens`` (a cone respresented by their generators) and the fix point
    ``p`` belongs to the same facet of ``bcone_gens``.
    """
    bool = False
    for face_cone in Cone(bcone_gens).facets():
        rays_lcone = set(map(tuple,lcone_gens))
        rays_face = set(map(tuple,primitive_vectors_cone(face_cone)))
        if ({tuple(p)}.union(rays_lcone)).issubset(rays_face):
            bool = True
            break
    return bool

def simplicial_partition(cone):
    """
    Returns a list with the subcones who forms the simplicial partition of ``cone``.
    """
    L = [cone]
    if not cone.is_simplicial():
        dict = {}
        F = Fan(L)
        list_subcones = []
        #Ordered list of subcones by ascending dimension
        for ls in F.cone_lattice().level_sets()[1:-1]: list_subcones = list_subcones + ls
        for subcone in list_subcones:
            if subcone.element.is_simplicial(): dict[subcone] = [set(subcone.element.rays())]
            else:
                partition = []
                fixpoint = subcone.element.rays()[0]
                for subsubcone in filter(lambda x: x<subcone, list_subcones):
                    if not same_facet(subsubcone.element.rays(), fixpoint, \
                                      subcone.element.rays()):
                        for part in dict[subsubcone]: partition = partition + \
                                                                  [part.union({fixpoint})]
                dict[subcone] = partition
        total_cone = list_subcones[-1]
        L = map(Cone,dict[total_cone])
    return L

def cone_info_output(cone, F = None):
    """
    Returns a string containing information about the generators of the cone and his simplicial
    partition.
    """
    if F == None: F = simplicial_partition(cone)
    info = "generators of cone = " + str(primitive_vectors_cone(cone)) + ", partition into "\
           "simplicial cones = " + str(map(primitive_vectors_cone,F))
    return info

def integral_vectors(scone):
    """
    Returns a list of integral vectors contained in {Sum lambda_j*a_j | 0<= lambda_j <1, a_j
    basis of the simple cone}.
    """
```

```
    origin = VectorSpace(QQ,scone.lattice_dim()).zero_vector()
    if scone.dim() == 0: integrals = [origin]
    else:
        cone_gens = primitive_vectors_cone(scone)
        ngens = len(cone_gens)
        A = transpose(matrix(ZZ, cone_gens))
        D, U, V = A.smith_form()
        diag = D.diagonal()
        coords = mrange(diag, vector)
        #Aux function for escale the vectors in the list
        def escale(v):
            v = vector(QQ, v)
            for i in range(ngens):
                if diag[i] != 0: v[i] = v[i]/diag[i]
                else: v[i] = 0
            return v
        #Aux function 'floor' for vectors component by component
        def floor(v):
            for i in range(ngens):
                v[i] = v[i] - v[i].floor()
            return v
        #Now, we escale and we return to the canonical basis
        L = map(lambda v: V*v, map(escale, coords))
        #Finally, we find the integral vectors of own region
        integrals = map(lambda v: matrix(QQ,A)*v, map(floor,L))
    return integrals

def multiplicity(scone):
    """
    Returns the multiplicity of a simple cone.
    """
    L = primitive_vectors_cone(scone)
    A = matrix(ZZ, L)
    S = A.smith_form()[0]
    result = 1
    for i in range(len(L)):
        result = result*S[i,i]
    return result

#-- Sigma and m functions defined in the paper
def sigma(v, weights = None):
    """
    Returns the pondered sum of the components in vector.
    """
    if weights == None: result = sum(v)
    else: result = vector(v).dot_product(vector(weights))
    return result

def m(v,P):
    """
    Returns min{v.x | x in polyhedron P}.
    """
    L = [vector(v).dot_product(vector(x)) for x in P.vertices()]
    return min(L)


##--- MONOMIALS ASSOCIATED TO A FACE
def ftau(f,tau):
    """
```

```
    Returns the polynomial f_tau associated to the face ``tau`` of the Newton's Polyhedron.
    """
    from sage.rings.polynomial.polydict import ETuple
    # We take a dicctionary between the exponents and the coefficients of monomials.
    D=f.dict()
    vars = f.parent().gens()
    nvars = len(vars)
    g = 0
    for v in support_points_in_face(f,tau):
        mon = 1
        for i in range(nvars): mon = mon*vars[i]^v[i]
        g = g + D[ETuple(v)]*mon
    return g

def solve_in_Fp_x(f,p):
    """
    For f a integral polynomial, retruns a list [{a in (F_p^x)^d | f*(a)=0}, {vars of f*}] with
    f* being f with coefficients in F_p(f), for a given rime number ``p``.
    """
    g = f.change_ring(GF(p))
    vars = g.variables()
    nvars = g.nvariables()
    h = (GF(p)[vars])(g)
    if len(h.exponents()) == 1: sols = []    #If f_tau is a monomial
    else:
        Fp_x_nvars = list(Tuples(range(1,p), nvars))    #(Fp-0)^nvars
        if h == 0: return [Fp_x_nvars, vars]
        sols = filter(lambda a:h(tuple(a))==0,Fp_x_nvars)
    return [sols,vars]

def is_degenerated(f_tau, p = None, method = 'default'):
    """
    Checks if the polynomial ``f_tau`` is degenerated over F_p, for p a given prime number
    ``p``.\n
    If ``p = None``, checks degeneration  over CC and (equivalent to be degenerated over F_p with
    p>>0).\n
    For finite fields (``p`` is a given prime):\n
        - ```method = 'default'``` check the condition using evaluation over the (F_p^x)^n in the
    system of equations.\n
        - ```method = 'ideals'``` check the condition using ideals over the finite field.
    """
    bool = False
    if type(p) != Integer:
        vars = f_tau.parent().gens()
        S = QQ[vars]
        I = S(f_tau).jacobian_ideal() + S*(S(f_tau))
        bool = prod(S.gens()) not in I.radical()
    else:
        if method == 'ideals':
            S = GF(p)[vars]
            I = S(f_tau).jacobian_ideal() + S*(f_tau)
            for xi in vars:
                I = I + S*(xi^(p-1)-1)    #xi unity in Fp iff xi^{(p-1)-1}=0
            bool = 1 not in I    #True if I in NOT the ring (ie, sist. has solution)
        else:
            [candidates, vars] = solve_in_Fp_x(f_tau,p)
            if vars == []: bool = True
            else:
                S = GF(p)[vars]
```

```
                g = f_tau.change_ring(GF(p))
                for xi in S.gens():
                    df_tau = S(g).derivative(xi)
                    candidates = filter(lambda a: df_tau(tuple(a)) == 0, candidates)
                    if len(candidates) != 0:
                        bool = True
                        break
    return bool

def is_all_degenerated(f,P, p = None, local = False, method = 'default'):
    """
    Checks if oum polynomial ``f`` is degenerated over F_p (``p`` prime) with respect the
    faces of the polyhedron ``P``.\n
    If ``p = None``, checks degeneration  over CC and (equivalent to be degenerated over F_p
    with p>>0).\n
    ``local = True`` checks degeneration for local case (only with respect the compact faces).\n
    For finite fields (``p`` is a given prime):\n
        - ``method = 'default'`` check the condition using evaluation over the (F_p^x)^n in the
    system of equations.\n
        - ``method = 'ideals'`` check the condition using ideals over the finite field.
    """
    bool = False
    if local == True: faces_set = compact_faces(P)
    else: faces_set = faces(P)[1:]
    for tau in faces_set:
        f_tau = ftau(f,tau)
        if is_degenerated(f_tau, p, method) == True:
            bool = True
            print "The formula for Igusa Zeta function is not valid:"
            if type(p) != Integer: print "The polynomial is degenerated at least with respect "\
                                         "to the face tau = {" + face_info_output(tau) + "} "\
                                         "over the complex numbers!"
            else: print "The polynomial is degenerated at least with respect to the face tau = "\
                        "{" + face_info_output(tau) + "} over GF(" + str(p) + ")!"
            break
    return bool


##--- IGUSA'S ZETA FUNCTION
#-- Values Ntau, Ltau and Stau defined in the paper
def Ntau(f,tau,p):
    """
    Returns the number Ntau = #{a in (F_p^x)^d | f*_tau(a)=0} with f*_tau being f_tau with
    coefficients in F_p(f_tau) for tau face.
    """
    n = f.parent().ngens()
    f_tau = ftau(f,tau)
    if type(p) != Integer:
        print "You must to give a 'Dictionary' with the number of solutions in GF(" + str(p) + \
              ")^" + str(n) + " associated to each face."
    else:
        [sols,vars] = solve_in_Fp_x(f_tau,p)
        nsols = len(sols)*(p-1)^(n - len(vars))
    return nsols

def Ltau(f,tau,p,abs_Ntau,s):
    """
    Returns a list [L_tau, N_tau] in terms of variable ``s``.\n
    ``abs_Ntau`` is the corresponding Ntau's values for abstract prime ``p``.
```

```
    """
    n = f.parent().ngens()
    if type(p) != Integer: N_tau = abs_Ntau
    else: N_tau = Ntau(f,tau,p)
    result = p^(-n)*((p-1)^n - p*N_tau*((p^s-1)/(p^(s+1)-1)))
    result = factor(result)
    return [result, N_tau]

def Lgamma(f,p,abs_Ngamma,s):
    """
    Returns the value Ntau for the total polyhedron in terms of variable ``s`.\n
    ``abs_Ngamma`` is the corresponding Ngamma value for abstract prime ``p``.
    """
    n = f.parent().ngens()
    if type(p) != Integer: N_gamma = abs_Ngamma
    else:
        [sols,vars] = solve_in_Fp_x(f,p)
        N_gamma = len(sols)*(p-1)^(n - len(vars))
    result = p^(-n)*((p-1)^n - p*N_gamma*((p^s-1)/(p^(s+1)-1)))
    return result

def Stau(f,P,tau,p, weights,s):
    """
    Returns a list [S_tau, cone_info] with ``cone_info`` containing a string of information
    about the cones, simplicial partition, multiplicity and integral points.\n
    Value S_tau is in terms of variable ``s``.
    """
    c = cone_from_face(tau)
    dim_cone = c.dim()
    F = simplicial_partition(c)
    result = 0
    for scone in F:
        num = 0
        den = 1
        for h in integral_vectors(scone): num = num + p^(sigma(h, weights) + m(h,P)*s)
        for a in primitive_vectors_cone(scone): den = den*(p^(sigma(a, weights) + m(a,P)*s) - 1)
        result = factor(simplify(expand(result + num/den)))
    info = cone_info_output(c,F)+ "\n" + "multiplicities = " + str(map(multiplicity,F)) + \
           ", integral points = " + str(map(integral_vectors,F))
    return [result, info]


def igusa_zeta(f, p = None, dict_Ntau = {}, local = False, weights = None, info = False):
    """
    The Igusa's Zeta Function por ``p`` prime (given or abstract), in terms of variable ``s``.\n
    For the abstract case (``p = None``), you must to give a dictionary ``dist_Ntau`` where the
    polynomes ftau for the faces of the Newton Polyhedron are the keys and the abstract value
    N_tau (depending of var p) as associated item. If ftau for face ``tauk`` is not in the
    dictionary, program introduces a new variable ``N_tauk``.\n
    ``local = True`` calcules the local (in the origin) Topological Zeta Function.\n
    ``weights`` is a list of weights if you want to considerate some ponderation.\n
    ``info = True`` gives information of face tau, cone of tau (all), L_tau, S_tau in the
    process.
    """
    s = var('s')
    if type(p) != Integer: p = var('p')
    P = newton_polyhedron(f)
    abs_Ngamma = None
    abs_Ntau = None
```

```
    if is_all_degenerated(f,P,p,local) == True: return NaN
    if local == True:
        faces_set = compact_faces(P)
        result = 0
    else:
        faces_set = proper_faces(P)
        if type(p) != Integer:
            abs_Ngamma = dict_Ntau.get(f)
            if abs_Ngamma == None: abs_Ngamma = var('N_Gamma')
        result = Lgamma(f,p,abs_Ngamma,s)
        if info == True: print "Gamma: total polyhedron\n" + "L_gamma = " + str(result) + "\n\n"
    for tau in faces_set:
        i = proper_faces(P).index(tau)
        if type(p) != Integer:
            f_tau = ftau(f, tau)
            if len(f_tau.exponents()) == 1: abs_Ntau = 0    #If f_tau is a monomial
            else:
                abs_Ntau = dict_Ntau.get(f_tau)
                if abs_Ntau == None: abs_Ntau = var('N_tau' + str(i))
        [L_tau, N_tau] = Ltau(f,tau,p,abs_Ntau,s)
        [S_tau, cone_info] = Stau(f,P,tau,p,weights,s)
        if info == True:
            print "tau" + str(i) + ": " + face_info_output(tau) + "\n" + cone_info + "\n" +\
                  "N_tau = " + str(N_tau) + ", L_tau = " + str(L_tau) + " , S_tau = " +\
                  str(S_tau) + "\n\n"
        result = factor(simplify(expand(result + L_tau*S_tau)))
    return result


##--- TOPOLOGICAL ZETA FUNCTION
#-- Calculation of the numbers Jtau and Mtau
def Jtau(tau,P,weights,s):
    """
    Returns a list [J_tau, cone_info] with ``cone_info`` containing a string of information
    about the cones, simplicial partition, multiplicity and integral points.\n
    Value J_tau is in terms of variable ``t``.
    """
    c = cone_from_face(tau)
    dim_cone = c.dim()
    F = simplicial_partition(c)
    if dim_cone == 0: result = 1
    else:
        result = 0
        for scone in filter(lambda i: i.dim()==dim_cone, F):
            num = multiplicity(scone)
            den = 1
            for a in primitive_vectors_cone(scone): den = den*(m(a,P)*s + sigma(a,weights))
            result = factor(simplify(expand(result + num/den)))
    cone_info = cone_info_output(c,F)+ "\n" + "multiplicities = " + str(map(multiplicity,F)) + \
                ", integral points = " + str(map(integral_vectors,F))
    return [result, cone_info]

def Mtau(tau,P,t):
    """
    Returns the value Mtau for ``tau`` face in ``P`` in terms of variable ``t``.
    """
    s = var('s')
    c = cone_from_face(tau)
    dim_cone = c.dim()
```

```
    F = simplicial_partition(c)
    result = 1
    for scone in filter(lambda i: i.dim()==dim_cone, F):
        mult = multiplicity(scone)
        den = SR(1)
        for a in primitive_vectors_cone(scone): den = den*(m(a,P)*s + sigma(a,weights = None))
        if den.degree(s) == 1:
            M = factor(simplify(s*den.subs(s=1/s)))
            result = result*(1-t^(M.subs(s=0)/mult))
    return result

def face_volume(f,tau):
    """
    Returns the value Vol(tau)*(dim tau)!, for a face tau.\n
    The points of the face tau are contained in RR^dim and Vol(tau) is defined as follows:
    Let omega[tau] be the volume form on Aff(tau) such that the parallelopiped spanned by a
    lattice  basis of ZZ^n intersect (aff tau)_0 has volume 1. Then Vol(tau) is the volume of
    tau intersection the Global Newton Polyhedron with respect to omega[tau].
    """
    n = f.parent().ngens()
    dim_tau = dim_face(tau)
    result = 0
    if dim_tau != 0:
        tau_in_global = Polyhedron(vertices = support_points_in_face(f,tau))
        vertices_in_global = map(vector,tau_in_global.vertices())
        trans_vertices = translate_points(vertices_in_global)
        if matrix(ZZ,trans_vertices).rank() == dim_tau:
            V = QQ^n
            basis_aff = V.submodule(trans_vertices).intersection(ZZ^n).basis()
            W = V.submodule_with_basis(basis_aff)
            coords_list = map(W.coordinate_vector, trans_vertices)
            p = PointConfiguration(coords_list)
            result = p.volume()       # Returns dimtau!*n-volumen de tau
    else:
        result = 1
    return result

def face_divisors(d,faces_set,P):
    """
    Returns a list of faces in ``faces_set`` such that d divides m(Delta_tau) = gcd{m(a)| a in
    Delta_tau and ZZ^n}.
    """
    if d ==1: return faces_set
    L_faces = list()
    dim_total = P.dim()
    for tau in faces_set:
        c = cone_from_face(tau)
        F = simplicial_partition(c)
        L_vectors = list()
        #We need to evaluate m over the basis of the cone and the integral points views above.
        for scone in F:
            L_vectors = L_vectors + integral_vectors(scone) + primitive_vectors_cone(scone)
        l = gcd(map(lambda i: m(i,P), L_vectors))
        if d.divides(l): L_faces.append(tau)
    return L_faces

def is_global_degenerated(f, p = None, method = 'default'):
    """
    Checks if own polynome ``f`` over F_p with respect all the faces of the Global Newton's
```

```
    Polyhedron.
    If p = None, checks degeneration  over CC and (equivalent to be degenerated over F_p with
    p>>0).
    For finite fields (``p`` is a given prime):
        - ``method = 'default'`` check the condition using evaluation over the (F_p^x)^n in the
    system of equations
        - ``method = 'ideals'`` check the condition using ideals over the finite field.
    """
    Q = f.newton_polytope()    #Global Newton Polyhedron of f
    bool = False
    for tau in faces(Q)[1:]:
        f_tau = ftau(f,tau)
        if is_degenerated(f_tau, p, method) == True:
            bool = True
            print "The formula for Topological Zeta function is not valid:"
            if type(p) != Integer: print "The polynomial is degenerated at least with respect "\
                                         "to the face tau = {" + face_info_output(tau) + \
                                         "} over the complex numbers!"
            else: print "The polynomial is degenerated at least with respect to the face tau "\
                        "= {" + face_info_output(tau) + "} over GF(" + str(p) + ")!"
            break
    return bool

#-- Topological Zeta Function of f,  Z_{top, f}^(d) for d>=1 and Monodromy Zeta:
def topological_zeta(f, d = 1, local = False, weights = None, info = False):
    """
    The Topological Zeta Function Z_{top, f}^(d) for ``d``>=1, in terms of variable ``s``.\n
    ``local = True`` calcules the local (in the origin) Topological Zeta Function.\n
    ``weights`` is a list of weights if you want to considerate some ponderation.\n
    ``info = True`` gives information of face tau, cone of tau (all), L_tau, S_tau in the
    process.
    """
    s = var('s')
    P = newton_polyhedron(f)
    result = NaN
    if is_global_degenerated(f) == False:
        result = 0
        if local == True: faces_set = compact_faces(P)
        else:
            faces_set = proper_faces(P)
            if d == 1:
                total_face = faces(P)[-1]
                dim_gamma = dim_face(total_face)
                vol_gamma = face_volume(f,total_face)
                result = (s/(s+1))*((-1)^dim_gamma)*vol_gamma
                if info == True: print "Gamma: total polyhedron\n" + "J_gamma = 1 , "\
                                       "dim_Gamma!*Vol(Gamma) = " + str(vol_gamma) +"\n\n"
        faces_set = face_divisors(d,faces_set,P)
        for tau in faces_set:
            [J_tau, cone_info] = Jtau(tau,P,weights,s)
            dim_tau = dim_face(tau)
            vol_tau = face_volume(f,tau)
            if info == True:
                i = proper_faces(P).index(tau)
                print "tau" + str(i) + ": " + face_info_output(tau) + "\n" + cone_info + "\n" +\
                      "J_tau = " + str(J_tau) + " , dim_tau!*Vol(tau) = " + str(vol_tau) + "\n\n"
            if d == 1:
                if dim_tau == 0: term = J_tau
                else: term = (s/(s+1))*((-1)^dim_tau)*vol_tau*J_tau
```

```
            else:
                term = ((-1)^dim_tau)*vol_tau*J_tau
            result = simplify(expand(result + term))
            if result != 0: result = factor(result)
    return result

def monodromy_zeta(f, weights = None, char = False, info = False):
    """
    The Monodromy Zeta Function in the origin, in terms of variable ``s``.\n
    ``weights`` is a list of weights if you want to considerate some ponderation.\n
    ``char = True`` prints the characteriristic polynomial of the monodromy (only if ``f`` has
    an isolated singularity in the origin).\n
    ``info = True`` gives information of face tau, cone of tau (all), L_tau, S_tau in the
    process.
    """
    n = f.parent().ngens()
    t = var('t')
    P = newton_polyhedron(f)
    result = 1
    i = 0
    for tau in compact_faces(P):
        zeta_tau = Mtau(tau,P,t)
        dim_tau = dim_face(tau)
        vol_tau = face_volume(f,tau)
        if info == True:
            print "tau" + str(i) + ": " + str(face_Vinfo(tau)) + "\n" + "M_tau = " + \
                  str(zeta_tau) + " , dim_tau!*Vol(tau) = " + str(vol_tau) + "\n\n"
            i = i + 1
        result = result*zeta_tau^((-1)^(dim_tau)*vol_tau)
    if char == True:
        result_aux = copy(result)
        T = var('T')
        mu = (-1)^(n - 1)*(result_aux.numerator().degree(t) - \
            result_aux.denominator().degree(t) - 1)
        factor(result_aux)
        aux = result_aux.subs(t = 1/T)
        char = factor(simplify(expand(T^mu*(T/(T - 1)*aux)^((-1)^(n - 1)))))
        print "The characteristic polynomial of the monodromy is " + str(char) +\
              "\n"
    return result


# -- INFORMATION ABOUT POLES IN TOPOLOGICAL ZETA
def dict_info_poles(f,d = 1, weights = None, local = False):
    """
    Returns a dictionary where the keys are the candidate real poles of the chosen zeta function.
    Items are lists containing, by order:\n
    1.- The list of perpendicular vectors to the facets that are responsible for the candidate
    real pole.\n
    2.- A list of the faces of maximal dimension that are responsible for the expected order.\n
    3.- The expected order.\n
    4.- Boolean: for the candidate pole -1 the factor L_tau or s/(s+1) can contribute to the
    order. If this is is the case, we increase the expected order due to the S_Delta_tau by 1
    and this record gets the value ``True``. In all other cases we don't increase this expected
    order and this record gets the value ``False``.
    """
    P = newton_polyhedron(f)
    if local == True: faces_set = compact_faces(P)
```

```
else: faces_set = proper_faces(P)
faces_set = face_divisors(d,faces_set,P)
dict_poles = {}
all_prim_vect = set([])
for tau in faces_set: all_prim_vect = all_prim_vect.union(set(map(tuple, \
                                                    primitive_vectors(tau))))
valid_prim_vect = filter(lambda v: m(v,P)!=0, all_prim_vect)
for v in valid_prim_vect:
    realpole = -sigma(v,weights)/m(v,P)
    #We initialize a list of attributes if the pole is not detected yet
    if dict_poles.get(realpole) == None: dict_poles[realpole] = [set([v]), [], 0, False]
    else: dict_poles[realpole][0].add(v)
#We calculate the maximal expected of each pole and the faces of higher dimension
#responsibles of this order
poles_set = dict_poles.keys()
#If d=1, we have the face tau_0
#(tau_0 is the smallest face who contains the intersecion between diagonal and polyhedron)
if d == 1:
    max_pole = max(poles_set)
    for tau in faces_set:
        gens_cone = map(tuple,primitive_vectors(tau))
        if set(gens_cone) ==  dict_poles[max_pole][0]:
            dict_poles[max_pole][1] = [tau]
            dict_poles[max_pole][2] = rank(matrix(gens_cone))
            dict_poles[max_pole][3] = False
            break
    poles_set.remove(max_pole)
for pole in poles_set:
    maxorder = 0
    responsible_faces = set([])
    for tau in faces_set:
        prim_vect = primitive_vectors(tau)
        interscone = set(map(tuple, prim_vect)).intersection(dict_poles[pole][0])
        if len(interscone) != 0:
            diminters = matrix(QQ, list(interscone)).rank()
            if diminters > maxorder:
                maxorder = diminters
                responsible_faces.add(tau)
            elif diminters == maxorder:
                responsible_faces.add(tau)
    #We find the maximal elements in set of responsible faces
    max_faces = set(responsible_faces)
    for face in responsible_faces:
        if face in max_faces:
            if len(filter(lambda i: face<i, max_faces)): max_faces.remove(face)
    dict_poles[pole][1] = list(max_faces)
    #Max order of pole is max dim of the asociated cones
    dict_poles[pole][2] = maxorder
    #We convert the set of vectors into a list
    dict_poles[pole][0] = map(vector, dict_poles[pole][0])
#Special pole -1 has sometimes a larger order
if -1 in dict_poles.keys():
    faces_minus_one = dict_poles[-1][1]
    if max(map(lambda tau: len(support_points_in_face(f,tau)), faces_minus_one)) > 1:
        dict_poles[-1][2] = dict_poles[-1][2] + 1
        dict_poles[-1][3] = True
return dict_poles
```

```
def give_expected_pole_info(f,d = 1, local = False, weights = None):
    """
    Prints information about the candidate real poles for the topological zeta function of ``f``:
    the orders and responsible faces of highest dimension.
    ``local = True`` calcules the local (in the origin) Topological Zeta Function.\n
    ``weights`` is a list of weights if you want to considerate some ponderation.\n
    """
    dict_poles = dict_info_poles(f,d, weights, local)
    P = newton_polyhedron(f)
    if local == True: faces_set = compact_faces(P)
    else: faces_set = proper_faces(P)
    faces_set = face_divisors(d,faces_set,P)
    n_supp_by_face = map(lambda tau: len(support_points_in_face(f,tau)), faces_set)
    if dict_poles == {}:
        if ( d == 1 and max(n_supp_by_face) == 1 ) or d!=1:
            print "There will be no poles for the (local) topological zeta function " + \
                    "(with d = " + str(d) + ") of " + str(f) + ".\n"
        else:
            print "The candidate poles of the (local) topological zeta function (with d = " + \
                    str(d) + ") of " + str(f) + " in function of s are:\n"
            print "-1 with expected order: 1"
            print "(If all Vol(tau) are 0, where tau runs through the selected faces " + \
                    "that are no vertices, then the expected order of -1 is 0)\n"
    else:
        poles_set = dict_poles.keys()
        #We reconstruct the list of all faces accessing to element
        some_face = dict_poles[poles_set[0]][1][0]
        list_all_faces = some_face.parent().list()[1:-1]
        if d == 1:
            print "The candidate poles of the (local) topological zeta function (with d = " + \
                    str(d) + ") of " + str(f) + " in function of s are:\n"
            max_pole = max(poles_set)
            print str(max_pole) + " with expected order: " + str(dict_poles[max_pole][2])
            if max_pole == -1:
                if dict_poles[-1][3] == True:
                    print "(If all the Vol(tau) of the faces tau that are no vertices and " + \
                            "contained in Gamma are 0, then the expected order of -1 is " + \
                            str(dict_poles[-1][3]-1) + ")."
            tau_0 = dict_poles[max_pole][1][0]
            print "The responsible face of maximal dimension is ``tau_0`` = minimal face " + \
                    "who intersecs with the diagonal of ambient space:"
            i = list_all_faces.index(tau_0)
            print "\t tau" + str(i) + ": " +  face_info_output(tau_0) + "\n\t" + \
                    cone_info_output(cone_from_face(tau_0)) + "\n"
            poles_set.remove(max_pole)
            if -1 not in poles_set:
                print "-1 with expected order: 1"
                print "(If all Vol(tau) are 0, where tau runs through the selected faces that " +\
                        "are no vertices, then the expected order of -1 is 0).\n"
        elif local == True:
            print "The candidate poles of the local topological zeta function (with d = " +\
                    str(d) + ") of " + str(f) + " in function of s are:\n"
            if max(n_supp_by_face) > 1 and -1 not in poles_set:
                print "-1 with expected order: 1"
                print "(If all Vol(tau) are 0, where tau runs through the selected faces that " +\
                        "are no vertices, then the expected order of -1 is 0).\n"
        for pole in poles_set:
            print str(pole) + " with expected order: " + str(dict_poles[pole][2])
            if dict_poles[pole][3] == True:
```

```
        print "(If all the Vol(tau) of the faces that are no vertices and contained"
        print "one or more of the faces below are 0, then the expected order of -1 is " +\
             str(dict_poles[pole][3]-1) + ")."
    print "The responsible face(s) of maximal dimension is/are:"
    for tau in dict_poles[pole][1]:
        i = list_all_faces.index(tau)
        print "\t tau" + str(i) + ": " + face_info_output(tau) + "\n\t" + \
                cone_info_output(cone_from_face(tau)) + "\n"
```

Referencias

[**ArCaLuMe01**] E. Artal Bartolo, P. Cassou-Noguès, I. Luengo, A. Melle Hernández: *Denef–Loeser zeta function is not a topological invariant*, J. London Math. Soc. (2001)

[**ArCaLuMe02**] E. Artal Bartolo, P. Cassou-Noguès, I. Luengo, A. Melle Hernández: *Monodromy conjecture for some surface singularities*, Ann. Scient. Ec. Norm. Sup. 35 (2002), 605–640.

[**ArCaLuMe05**] E. Artal Bartolo, P. Cassou-Noguès, I. Luengo, A. Melle Hernández: *Quasi-ordinary power series and their zeta functions*, Mem. Amer. Math. Soc. 178 no 841, 2005.

[**BriKno**] E. Brieskorn, H. Knorrer: *Plane Algebraic Curves*, Birkhauser,(1986).

[**Den**] J. Denef: *Report on Igusa's zeta function*, Asterisque 201-203 (1991), 359 386.

[**DenHoo**] J. Denef and K. Hoornaert: *Newton polyhedra and Igusa's local zeta function*, J. Number Theory 89 (2001), no. 1, 31–64.

[**DenLoe92**] J. Denef and F. Loeser: *Caractéristiques d'Euler-Poincaré, Fonctions Zêta Locales et Modifications Analytiques*, J. Amer. Math. Soc. 5(4), pp. 705-720, 1992.

[**DenLoe98**] J. Denef and F. Loeser: *Motivic Igusa zeta functions*, J. Alg. Geom. 7, 1998, 505–537.

[**Hiro**] H. Hironaka , *Resolution of singularities of an algebraic variety over a field of characteristic zero. I*, Ann. Of Math. (2) 79 (1): 109–203, and p*art II*, pp. 205–326 (1964).

[**HooLoo**] K. Hoornaert and D. Loots: *Computer program written in Maple for the calculation of Igusa's local zeta function*. http://www.wis.kuleuven.ac.be/algebra/kathleen.htm, 2000.

[**Igu74**] J. I. Igusa: *Complex powers and asymptotic expansions*, I, J. Reine Angew. Math. 268/269 (1974), 110-130; II, 278/279 (1975), 307-321.

[**Igu78**] J. I. Igusa: *Lectures on Forms of Higher Degree*, Tata Inst. Fund. Research,Springer-Verlag, Heidelberg/New York/Berlin, 1978.

[**Loe**] F. Loeser: *Fonctions d'Igusa p-adiques et polynômes de Bernstein*, Amer. J. Math. 110 (1) (1988) 1–21.

[**Mil**] J. Milnor: *Singular Points of Complex Hypersurfaces*. Ann. of Math. Study 61, Princeton University Press (1968).

[**NemVe**] A. Némethi, W. Veys: *Generalized monodromy conjecture in dimension two*. Geom. Topol. 16 (2012), no. 1, 155–217.

[**Rock**] R. Tyrrell Rockafellar: *Convex Analysis*, Princeton Univ. Press, Princeton, NJ, 1970.

[**Var**] A. N. Varchenko: *Zeta-function monodromy Newton's of and diagram*, Invent. Math.37 (1976), 253-262.

[**Veys**] W. Veys: *Determination of the poles of the topological zeta function for curves*, Manuscripta Math. 87 (1995), 435–448.