Degree in Mathematics

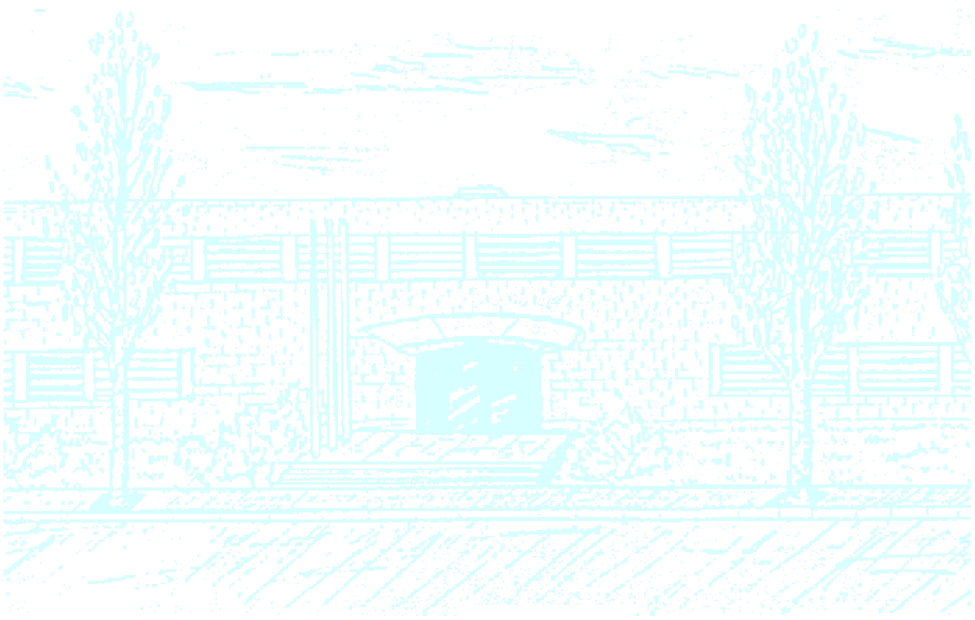
Title: Invariants of binary forms

Author: Enrique Jiménez Izquierdo

Advisor: Jordi Quer Bosor

Department: Mathematics

Academic year: 2016/17



Universitat Politècnica de Catalunya
Facultat de Matemàtiques i Estadística

Degree in Mathematics
Bachelor's Degree Thesis

Invariants of binary forms

Enrique Jimenez Izquierdo

Supervised by Jordi Quer Bosor

January, 2017

Abstract

In this work we want to give an introduction to the theory of invariants for binary forms, in order to later try to give a solution to some complex computational problems about invariants. The theory of invariants will be explained as viewed from its classical point of view, as it was studied by Hilbert (the main reference of this thesis), and we will explain all the concepts and results we consider necessaries for a basic understanding of the theory and for the approach to the problems we will implement. After the theory is exposed, we explain the computational problems we have faced in the thesis, and for which we have implemented a solution in Sage. We explain the details of the problems, as well as some details about our approach to the problems.

Keywords

Invariant theory, Binary forms, Invariant, Covariant, Transvectant

Contents

Introduction

The topic of this thesis is invariant theory for binary forms. Invariant theory is a field introduced in the XIX century, first studied roughly by Cayley (1845) and later developed by Clebsch, Gordan and Hilbert (just to mention a few). The invariant theory has some theoretical applications, as well as some practical applications such as studying the properties of some algebraic objects, the simplest case being hyperelliptic curves, which corresponds to the study of binary forms (what we will be studying in this thesis). In its origins, the theory of invariants was studied more computationally. The work made by Cayley, Clebsch or Gordan used a more constructive approach, more focused in the computational and practical aspects of the theory. The great contribution of Hilbert to the field was to give a non-constructive proof of the finiteness of the algebra of invariants of binary forms, which was generalizable to forms in any number of variables (Hilbert's basis theorem). This result had been proven by Gordan previously, but in a constructive way, and his proof was not generalizable to forms in more than two variables. The explicit computations used in the classical approach were quickly replaced by other methods, as the use of invariants required the manipulation of quite complicated polynomial expressions. In recent years, though, these explicit methods have regained interest, as the development of computers and mathematical software has given us a tool with which we can handle to some extent this complicated expressions.

The objective of this thesis is twofold. First, we want to give to the reader an introduction to invariant theory, focused on the study of binary forms. The theory explained here will be the invariant theory seen from its classical point of view, following up to certain point the exposition given by Hilbert in [2]. After this we want to give some notions to the reader about the practical applications of this theory. This we will do by exposing some of the complex problems of invariants that are only approachable computationally.

The modern approach to invariant theory studies actions of a group G on a finite vector space V, where the invariants are functions in the space of polynomials $I \in k[V]$ which are immutable by the action of G on V. The classical approach is particular case of this, which studies an action of GL_2 (or SL_2) on the space of homogeneous polynomials of given number of variables and total order. This is the approach used in the exposition by Hilbert in [2] and what we will use in our thesis.

We have divided the contents of our thesis in two chapters, which separate the different objectives of our exposition:

1. **Classical invariant theory:** In the first chapter of the thesis we explain all the concepts and results for a basic understanding of the theory of invariants applied to binary forms.

2. **Implementation:** In the second chapter, we explore the more practical aspects of the theory. Here, we list some of the computational problems that we can encounter studying invariant theory and its applications. For some of these problems we have implemented a solution using a mathematical software (the program "Sage"). We describe these problems in more detail, and about the details of our implementation (the pros and cons of our solution, as well as the other possible approaches that were considered).

1. Classical invariant theory

In this first section we want to give an introduction to classical invariant theory, following mostly the exposition from Hilbert in [2], focusing on the theory of invariants for binary forms. The objective is that the reader understands clearly the concepts of the theory of invariants (binary forms, invariants, covariants, a basis of invariants...) and the important results of the theory (characterization of invariants and covariants, the finiteness of the algebra of invariants...).

We will first introduce the basic concepts of invariant theory: forms, binary forms, invariants and covariants of forms of order n, invariants and covariants of a system of several simultaneous forms, etc... We will see that the system of invariants of binary forms of a certain order has the structure of a graded algebra and then see one of the most important results of invariant theory, that this algebra is finitely generated (Hilbert's theorem).

After the introduction of all these basic concepts, we will see the part of the theory more oriented to the practical applications. We will introduce two methods to construct invariants and covariants that will be useful when we face the computational part of the thesis: the use of the transvectants and seeing the invariants as functions of the roots of the forms. Later we see the information the invariants give us about the form, and how to use them to know if two forms are isomorphic to each other.

1.1 Forms and binary forms

Definition 1.1. We will call a **form** to an homogeneous polynomial (in n variables), or what is the same, a polynomial in which every term has the same degree.

We will call a form in two variables a **binary form**. The same way we will call a form in three variables a ternary form, in four variables a quaternary form, etc...

For all of our work we will be studying the particular case of binary forms, so in case we do not explicitly specify the number of variables of the form, we will always be referring to forms in two variables. It is easy to see that a great part of the concepts and of the results we will be dealing with in the following sections are easily extended to forms in an arbitrary number of variables.

An arbitrary binary form of order n can be written in the following way:

$$f(x_1, x_2) = a_0 x_1^n + \binom{n}{1} a_1 x_1^{n-1} x_2 + \dots + \binom{n}{i} a_i x_1^{n-i} x_2^i + \dots + a_n x_2^n \tag{1}$$

Which is the expression we will have in mind when talking about binary forms from now on. Also, when talking about the coefficients of a form, we will be talking about the $\{a_i\}$ in this expression (leaving out the binomial coefficients multiplying each term). Depending on the bibliography we are using we can see the binomial coefficients missing in the definition of a binary form. We will leave them since this is the way Hilbert presents it in [2] and most of the classical literature. The results for both ways of representing binary forms are the same, but this representation has the advantage that most of the expressions will end up being much simpler.

We will sometimes use special nomenclature to refer to binary forms of certain orders. For example, we will call binary forms of orders $2, 3, 4, 5, 6\dots$ binary quadratics, cubics, quartics, quintics, sextics, etc...

Before we continue we would want to make the following remark. For our work we will be working with the forms defined over an algebraically closed field with characteristic 0 (such as $\mathbb{C}$), but it is possible to work with binary forms and in general all of the theory defined on any field. This, though, could be a bit troublesome as some of the results we will explore along the thesis won't work over a field not so well behaved as $\mathbb{C}$. For example, we could not use the representation of binary forms just mentioned if we are working over a field of characteristic different from 0 (because some binomial coefficients might be zero). In the classical theory of invariants binary forms were always defined over $\mathbb{C}$, as Hilbert does in [2] (even if he does not says that explicitly), so that is what we will suppose for our work. So from now on, the field of definition k will be, unless said otherwise, an algebraically closed field of characteristic 0.

We can apply a linear transformation to the variables of a binary form in the following way:

$$\begin{aligned} x_1 &= \alpha_{11}x'_1 + \alpha_{12}x'_2 \\ x_2 &= \alpha_{21}x'_1 + \alpha_{22}x'_2 \end{aligned} \tag{2}$$

From which we obtain a new form in the new variables x'_1, x'_2 and with coefficients a'_i.

$$f'(x'_1, x'_2) = a'_0 x'^n_1 + ... + \binom{n}{i} a'_i x'^{n-i}_1 x'^i_2 + ... + a'_n x'^n_2$$

We will always want the transformations we apply to binary forms to be invertible, that is to say that $M = \left(\begin{smallmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \end{smallmatrix}\right) \in GL_2(k)$. We will also denote the transformed form as $f' = f|_M$.

This clearly defines an action of the group $GL_2(k)$ on the set of binary forms. In some bibliography, one can find the use of the group $SL_2(k)$ instead of $GL_2(k)$ for the study of invariants. The use of $SL_2(k)$ make some of the results for invariant theory a bit easier. This is because, as we will see later, several definitions and results in our thesis have the determinant of the transformation matrices as a factor, such as the definition of invariants 1.2 and covariants 1.3. In the case we were working with $SL_2(k)$ this factor would be just 1 and these expressions would be simpler. As we are working in $\mathbb{C}$ the results for both $SL_2(k)$ and $GL_2(k)$ are practically the same (except for this factor of the determinant), so we will work in $GL_2(k)$ for more generality and because this is what was used by Hilbert in [2].

Let f and g be binary forms of order n. We will call any matrix $M \in GL_2(k)$ such that $g = f|_M$ an isomorphism between f and g. Two forms f and g will be called isomorphic if there exists an isomorphism between them.

Let f be a binary form of order n. We will call any matrix $M \in GL_2(k)$ such that $f = f|_M$ an automorphism of f. The set of automorphisms of a form f is a subgroup of $GL_2(k)$.

1.2 Invariants and covariants of a form

On the center of invariant theory (as its name suggests) lie the concepts of invariant and covariant.

Definition 1.2. Let $I \in k[a_0, a_1, ..., a_n]$ be a polynomial function in the coefficients of forms of order n. We will say that I is an **invariant** if for any linear transformation M, I changes by a fixed power of the determinant of M, or what is the same:

$$I(a'_0, a'_1, ..., a'_n) = det(M)^p I(a_0, a_1, ..., a_n) \tag{3}$$

where a'_i are the new coefficients after the application of M.

Definition 1.3. Let $C \in k[a_0, a_1, ..., a_n; x_1, x_2]$ be a polynomial function in both the coefficients of forms of order n and the two variables x_1 and x_2. We will say that the expression C is a **covariant** if for any linear transformation M, C changes by a fixed power of the determinant of M, or what is the same:

$$C(a'_0, a'_1, ..., a'_n, x'_1, x'_2) = det(M)^p C(a_0, a_1, ..., a_n, x_1, x_2) \tag{4}$$

where a'_i and x'_1, x'_2 are the new coefficients and variables after the application of M.

To avoid misunderstandings about nomenclature in the work, we will explain briefly the few basic terms that we will be using when talking about forms, invariants and covariants that can be confused (especially between the terms **order** and **degree**):

1. Let $I = \sum Z a_0^{\nu_0} a_1^{\nu_1} ... a_n^{\nu_n}$ be an invariant for forms of order n, and let $t = Z a_0^{\nu_0} a_1^{\nu_1} ... a_n^{\nu_n}$ be a summand of the invariant. We will call to the expression:

   $$g = \nu_0 + \nu_1 + ... + \nu_n$$

   the "degree" of t. Also, as we will see later, an invariant must be an homogeneous polynomial in the $\{a_i\}$, what means that all its terms must have the same degree g. We will call the "degree" of I to the degree that share all of its terms.

2. Let $I = \sum Z a_0^{\nu_0} a_1^{\nu_1} ... a_n^{\nu_n}$ be an invariant for forms of order n, and let $t = Z a_0^{\nu_0} a_1^{\nu_1} ... a_n^{\nu_n}$ be a summand of the invariant. We will call to the expression:

   $$p = \nu_1 + 2\nu_2 + ... n\nu_n$$

   the "weight" of t. In the same way as with the degree, we will see later that an invariant must be an isobaric polynomial in the $\{a_i\}$. This means that all its terms must have the same weight p. We will call the "weight" of I to the weight that share all of its terms.

3. Let C be a covariant for forms of order n. As we will see later, a covariant must be an homogeneous polynomial in the x_i. We will call the order of C as a polynomial in the x the "order" of C and we will usually denote it with an m.

Note from the definitions of invariant and covariant that the first is a particular case of the second. So one could say that an invariant is just a covariant of order 0.

We can list some examples of invariants and covariants of simple binary forms:

Example 1.4. For the binary quadratic form $f(x_1, x_2) = a_0x_1^2 + 2a_1x_1x_2 + a_2x_2^2$ we can only make one invariant of degree 2 (except scalar multiples of it), which is known as the discriminant of the form:

$$I = a_0a_2 - a_1^2$$

Example 1.5. For the binary quartic $f(x_1, x_2) = a_0x_1^4 + 4a_1x_1^3x_2 + 6a_2x_1^2x_2^2 + 4a_3x_1x_2^3 + a_4x_2^4$ we can only make one invariant of degree 2 (except scalar multiples of it), which is:

$$I = a_0a_4 - 4a_1a_3 + 3a_2^2$$

Example 1.6. For any order n, the expression for a binary form is in itself a covariant $C = f = a_0x_1^n + \binom{n}{1} a_1x_1^{n-1}x_2 + ... + a_nx_2^n$. Easily we can see that C satisfies the covariant property: for any linear transformation $M \in GL_2(k)$, by applying it to C the covariant remains the same in the new coefficients $\{a'_i\}$ of the form.

We have not shown that these examples really satisfy the invariant property. We could check this directly by the definition of invariant, by applying a generalized linear transformation to the original form and checking that the transformed polynomial is the same (up to a power of the determinant of the transformation matrix). Instead of that, we will introduce the following concepts that will help us characterize when an expression is an invariant/covariant.

Definition 1.7. Let $I = \sum Z_{v_0 v_1 \dots v_n} a_0^{v_0} \dots a_n^{v_n}$ be a polynomial expression in the coefficients of forms of order n, we define the operators $\mathbf{D}$ and Δ as follows

$$\begin{aligned} \mathbf{D} &= a_0 \frac{\partial}{\partial a_1} + 2a_1 \frac{\partial}{\partial a_2} + \dots + n a_{n-1} \frac{\partial}{\partial a_n} \\ \Delta &= n a_1 \frac{\partial}{\partial a_0} + (n-1) a_2 \frac{\partial}{\partial a_1} + \dots + a_n \frac{\partial}{\partial a_{n-1}} \end{aligned} \tag{5}$$

These concepts can be used to find sufficient conditions for an expression to be an invariant by linear transformations. These are given in the following theorem:

Theorem 1.8. *Let $I(a_0, \dots, a_n)$ be a polynomial expression in the coefficients of forms of order n. The expression I is an invariant by linear transformations in $GL_2(k)$ with exponent p if and only if it satisfies the following properties:*

1. *It is an homogeneous polynomial of degree g, isobaric of weight p, and its degree and weight satisfy the relation $ng = 2p$.*

2. *It satisfies the equation $\mathbf{D}I = 0$.*

3. *It satisfies the equation $\Delta I = 0$.*

Proof. Every linear transformation $M = \left(\begin{smallmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \end{smallmatrix}\right) \in GL_2(K)$ can be expressed as a product of the three following types of linear transformations:

1. $\begin{pmatrix} \kappa & 0 \\ 0 & \lambda \end{pmatrix}$

2. $\begin{pmatrix} 1 & \nu \\ 0 & 1 \end{pmatrix}$

3. $\begin{pmatrix} 1 & 0 \\ \mu & 1 \end{pmatrix}$

the proof of this fact is a basic linear algebra exercise.

It follows from this statement that an expression in the coefficients of a binary form that satisfies the invariant property for any linear transformation of the types 1, 2 and 3 would be an invariant of the binary form. What we will do to prove the theorem is to see for each of these three types of linear transformations how they affect the coefficients $\{a_i\}$ of the form, and what properties needs to have an expression to be invariant by each kind of transformation. The conditions we give in the theorem for an expression to be an invariant will be a consequence from these facts.

1. The application of transformation 1 to the binary form $f = a_0 x_1^n + \binom{n}{1} a_1 x_1^{n-1} x_2 + ... + a_n x_2^n$ results in:

$$f' = f(\kappa x_1', \lambda x_2') = a_0 \kappa^n x_1'^n + \binom{n}{i} a_1 \kappa^{n-1} \lambda x_1'^{n-1} x_2' + ... + a_n \lambda^n x_2'^n$$
$$= a_0' x_1'^n + \binom{n}{i} a_1' x_1'^{n-1} x_2' + ... + a_n' x_2'^n$$

So if we compare the two expressions we end up with the $n+1$ relations:

$$a_i' = \kappa^{n-i} \lambda^i a_i$$

Now we study what happens to a polynomial expression on the $\{a_i\}$ when evaluated in the new coefficients. If we have one such expression, namely $I(\{a_i\}) = \sum Z a_0^{\nu_0} a_1^{\nu_1} ... a_n^{\nu_n}$, then we have:

$$I(\{a_i'\}) = \sum Z a_0'^{\nu_0} a_1'^{\nu_1} ... a_n'^{\nu_n}$$
$$= \sum Z a_0^{\nu_0} a_1^{\nu_1} ... a_n^{\nu_n} \kappa^{n\nu_0 + (n-1)\nu_1 + ... + \nu_{n-1}} \lambda^{\nu_1 + 2\nu_2 + ... + n\nu_n}$$

Now, for this expression to be an invariant respect to this kind of transformation, we would need that, according to the definition of invariant, for a certain number p:

$$I(\{a_i'\}) = det(M)^p I(\{a_i\}) = \kappa^p \lambda^p I(\{a_i\})$$

For this to work every term of the sum must fulfill the equality. Then, comparing the two expressions, we obtain the identities:

$$n\nu_0 + (n-1)\nu_1 + ... + (n-i)\nu_i + ... + \nu_{n-1} = p$$
$$\nu_1 + 2\nu_2 + ... + i\nu_i + ... + n\nu_n = p$$

Now we see that this is precisely the definition of weight we gave earlier. Also, if we add together the two equations, we obtain:

$$n(\nu_0 + \nu_1 + ... + \nu_n) = ng = 2p$$

So with this we have seen that, for a polynomial expression $I(\{a_i\})$ to be invariant by a transformation of type 1, being p the exponent of the transformation determinant by which the invariant is multiplied upon transformation, it has to be an isobaric function of weight equal to p, homogeneous of degree g satisfying both the equation $ng = 2p$.

This gives us the first point of the theorem.

2. Similarly to the previous one, we want to see what happens to the binary form $f = a_0 x_1^n + \binom{n}{1} a_1 x_1^{n-1} x_2 + ... + a_n x_2^n$ after applying a general linear transformation of the type 2:

$$f' = f(x_1' + \nu x_2', x_2') = a_0 (x_1' + \nu x_2')^n + \binom{n}{1} a_1 (x_1' + \nu x_2')^{n-1} x_2' + ... + a_n x_2'^n$$
$$= a_0' x_1'^n + \binom{n}{1} a_1' x_1'^{n-1} x_2 + ... + a_n' x_2'^n$$

We want to obtain, as in the previous case, a relation between the $\{a'_i\}$ and the $\{a_i\}$. If we compare the coefficients of the terms $x_1^i x_2^{n-i}$, we obtain:

$$\binom{n}{i} a'_i = \binom{n}{i} a_i + \binom{n}{i-1}(n-i+1)a_{i-1}\nu + ... + \binom{n}{i-k}\binom{n-i+k}{k} a_{i-k}\nu^k + ... + a_0\nu^i$$

and because of the identity:

$$\binom{n}{i-k}\binom{n-i+k}{k} = \binom{n}{i}\binom{i}{k}$$

we have the following relation between the transformed coefficients $\{a'_i\}$ and the old ones $\{a_i\}$:

$$a'_i = a_i + \binom{i}{1} a_{i-1}\nu + ... + \binom{i}{k} a_{i-k}\nu^k + ... + a_0\nu^i$$

Now, for a polynomial expression on the coefficients $I(\{a_i\})$ to be invariant by transformations of this type, one has to have:

$$I(\{a'_i\}) = det(M)^p I(\{a_i\}) = I(\{a_i\})$$

We want to find which properties has to have this expression to remain invariant for any transformation of type 2, so this equation should hold by any possible value of ν. With this being true, we can derive in both sides respect to ν, obtaining:

$$\frac{\partial I(\{a'_i\})}{\partial a'_0}\frac{\partial a'_0}{\partial \nu} + \frac{\partial I(\{a'_i\})}{\partial a'_1}\frac{\partial a'_1}{\partial \nu} + ... + \frac{\partial I(\{a'_i\})}{\partial a'_n}\frac{\partial a'_n}{\partial \nu} = 0$$

But if we look at the development of $\frac{\partial a'_i}{\partial \nu}$ we see that:

$$\begin{aligned}
\frac{\partial a'_0}{\partial \nu} &= 0 \\
\frac{\partial a'_1}{\partial \nu} &= a_0 = a'_0 \\
\frac{\partial a'_2}{\partial \nu} &= 2a_1 + 2a_0\nu = 2a'_1 \\
&... \\
\frac{\partial a'_i}{\partial \nu} &= ia_{i-1} + ... + \binom{i}{k} k a_{i-k}\nu^{k-1} + ... + ia_0\nu^{i-1} \\
&= ia_{i-1} + ... + \binom{i-1}{k-1} i a_{i-k}\nu^{k-1} + ... + ia_0\nu^{i-1} \\
&= ia'_{i-1}
\end{aligned}$$

which transforms the previous equation in:

$$\frac{\partial I(\{a'_i\})}{\partial a'_0} * 0 + \frac{\partial I(\{a'_i\})}{\partial a'_1} a'_0 + ... + \frac{\partial I(\{a'_i\})}{\partial a'_n} na'_{n-1} = 0$$

But this is the operator **D** we defined earlier. So, in the end, this gives us the second point in our theorem:

For a polynomial expression $I(\{a_i\})$ in the coefficients of a form to be invariant by linear transformations of the type 2, the expression I must satisfy the equation:

$$\mathbf{D}I = 0$$

3. The analysis for the third kind of transformations is very similar to the one done for the second type. We will proceed in the same fashion:

   For a binary form $f = a_0x_1^n + \binom{n}{1}a_1x_1^{n-1}x_2 + ... + a_nx_2^n$, if we apply to it a linear transformation of the type 3, we obtain:

$$\begin{aligned} f' = f(x_1', \mu x_1' + x_2') &= a_0x_1'^n + \binom{n}{1}a_1x_1'^{n-1}(\mu x_1' + x_2') + ... + a_n(\mu x_1' + x_2')^n \\ &= a_0'x_1'^n + \binom{n}{1}a_1'x_1'^{n-1}x_2' + ... + a_n'x_2'^n \end{aligned}$$

   With arguments similar to the ones applied to the transformations of type 2 one has the relations:

$$a_i' = a_i + \binom{n-i}{1}a_{i+1}\mu + ... + \binom{n-i}{k}a_{i+k}\mu^k + ... + a_n\mu^{n-i}$$

   And for later, if we derive these terms by μ, one has the equivalences:

$$\begin{aligned} \frac{\partial a_i'}{\partial \mu} &= (n-i)a_{i+1} + ... + \binom{n-i}{k}ka_{i+k}\mu^{k-1} + ... + (n-i)a_n\mu^{n-i-1} \\ &= (n-i)a_{i+1} + ... + \binom{n-i-1}{k-1}(n-i)a_{i+k}\mu^{k-1} + ... + (n-i)a_n\mu^{n-i-1} \\ &= (n-i)a_{i+1}' \end{aligned}$$

   For a polynomial expression $I(\{a_i\})$ to be invariant respect to transformations of the type 3 one must have:

$$I(\{a_i'\}) = det(M)^pI(\{a_i\}) = I(\{a_i\})$$

   and if we derive both sides by μ we obtain the equation:

$$\frac{\partial I(\{a_i'\})}{\partial a_0'}na_1' + ... + \frac{\partial I(\{a_i'\})}{\partial a_i'}(n-i)a_{i+1}' + ... + \frac{\partial I(\{a_i'\})}{\partial a_n'} * 0 = 0$$

   Which we see again that is equal to the operator Δ we defined before. So with this we have the third condition for our theorem:

   A polynomial expression $I(\{a_i\})$ in the coefficients of a binary form is invariant by transformations of the type 3 if it satisfies the equation:

$$\Delta I = 0$$

With this we have seen that in order for an expression $I(\{a_i\})$ to be invariant by linear transformations of the types 1, 2 and 3 it has to satisfy all the conditions listed in the theorem. Because any transformation $M \in GL_2(K)$ can be written as a product of transformations of these three types, any expression I that satisfies the conditions listed will be an invariant. □

The concepts introduced before are the same applied to polynomials in $k[\{a_i\}, x_0, x_1]$, so we can also apply them to determine the sufficient conditions for covariants of binary forms. But before we give these conditions for covariants we want to make a quick remark. We will suppose from now on, and without loss of generality, that covariants are polynomials homogeneous in the variables x_i. We do this because of the following: when applying a linear transformation, each x is replaced by a linear combination of the x_i', so each homogeneous function of the x_i transforms into an homogeneous function of the same order. This implies that if a polynomial function in $k[\{a_i\}, x_0, x_1]$ (not necessarily homogeneous) satisfies the covariant property, then each of its homogeneous parts are also covariants. So we can consider only the homogeneous covariants for our study.

With this in mind, let us see the set of conditions for an expression to be a covariant:

Theorem 1.9. *Let $C(a_0, ..., a_n, x_1, x_2)$ be a polynomial expression in the coefficients of forms of order n and the variables x_1, x_2, and that is homogeneous in the x_i. The expression C is a covariant by linear transformations in $GL_2(k)$ with exponent p if and only if it satisfies the following properties:*

1. *Written as $C = C_0x_1^m + ... + C_ix_1^{m-i}x_2^i + ... + C_mx_2^m$ each C_i is an homogeneous polynomial in the coefficients a_i of the same degree g and isobaric of weight $p+i$, and C satisfies the equation $m = ng - 2p$*
2. *It satisfies the equation $\mathbf{D}C = x_2\frac{\partial C}{\partial x_1}$*
3. *It satisfies the equation $\Delta C = x_1\frac{\partial C}{\partial x_2}$*

Proof. The proof for this theorem will follow the same steps as the previous one. We will see for each of the three types of linear transformations that we mentioned above, which properties make an expression to be a covariant respect to transformation of that type, and we will see that joining the three parts together we obtain our theorem.

1. We know from the last theorem's proof the way a binary form is modified by transformations of type 1 and the relation existing between the new coefficients $\{a_i'\}$ after the transformation and the original ones $\{a_i\}$. Now we want to know how does this affect to a polynomial expression like $C(a_0, a_1, ..., a_n, x_1, x_2)$, to find the conditions for it to be a covariant respect to this transformation. As we said earlier, we can consider the polynomial homogeneous in the variables x_i, so, let us call m the order of C in the variables x_i, and let us represent this polynomial as:

$$C = C_0x_1^m + C_1x_1^{m-1}x_2 + ... + C_mx_2^m$$

where each of the C_i is (as we denoted in the previous proof) $C_i = \sum Za_0^{\nu_0}a_1^{\nu_1}...a_n^{\nu_n}$. Now, for this expression to be a covariant, it must satisfy the covariant property for this kind of transformations, namely, for some number p:

$$C(\{a_i'\}, x_1', x_2') = det(M)^pC(\{a_i\}, x_1, x_2) = \kappa^p\lambda^pC(\{a_i\}, x_1, x_2)$$

If we work out the expression for $C(\{a_i'\}, x_1', x_2')$ with the relations between the $\{a_i'\}$ and the $\{a_i\}$ worked out in the previous proof, we obtain:

$$C(\{a_i'\}, x_1', x_2') = C(\{a_i\kappa^{n-i}\lambda^i\}, \frac{x_1}{\kappa}, \frac{x_2}{\lambda}) = C_0(\{a_i'\})\frac{x_1^m}{\kappa^m} + C_1(\{a_i'\})\frac{x_1^{m-1}}{\kappa^{m-1}}\frac{x_2}{\lambda} + ... + C_m(\{a_i'\})\frac{x_2^m}{\lambda^m}$$

And if we now compare the coefficients of $x_1^{m-i}x_2^i$ in both sides of the equation, we find out:

$$\begin{aligned} C_i(a_0', ..., a_n')\kappa^{i-m}\lambda^{-i} &= \sum Za_0^{\nu_0}a_1^{\nu_1}...a_n^{\nu_n}\kappa^{n\nu_0+(n-1)\nu_1+...+\nu_{n-1}}\lambda^{\nu_1+2\nu_2+...+n\nu_n}\kappa^{i-m}\lambda^{-i} \\ &= \kappa^p\lambda^p\sum Za_0^{\nu_0}a_1^{\nu_1}...a_n^{\nu_n} \end{aligned}$$

with the same argument as in the last proof, we obtain from this the following two equations:

$$\begin{aligned} n\nu_0 + (n-1)\nu_1 + ... + \nu_{n-1} + i - m = p \\ \nu_1 + 2\nu_2 + ... + n\nu_n - i = p \end{aligned}$$

and adding the two together we obtain the following relation:

$$n(\nu_0 + \nu_1 + ... + \nu_n) - m = ng - m = 2p$$

As every term on the sum must satisfy these equations, we can extract the following information from this:

For the expression C to be a covariant by the transformations of type 1, as these equations must hold for all terms, each of the terms C_i must be isobaric of weight equal to $p + i$ and homogeneous of degree g, which satisfies the equation $ng - m = 2p$

2. For the study of transformation of type 2 we proceed also in the same way as before. Because for transformations M of type 2 one has that $det(M) = 1$, then for a polynomial expression $C(\{a_i\}, x_1, x_2)$ to be a covariant respect to it, it must satisfy:

$$C(\{a_i'\}, x_1', x_2') = C(\{a_i\}, x_1, x_2) \tag{6}$$

All the relations computed in the previous proof are the same here, so we can use them in the same way:

$$\begin{aligned} a_i' &= a_i + \binom{i}{1} a_{i-1}\nu + ... + \binom{i}{k} a_{i-k}\nu^k + ... + a_0\nu^i \\ x_1' &= x_1 - \nu x_2 \\ x_2' &= x_2 \end{aligned}$$

We want to proceed in the same way as before, so we want to derivate both sides of the equation 6 with respect to ν, so we have:

$$\begin{aligned} &\frac{\partial C(\{a_i'\})}{\partial a_0'}\frac{\partial a_0'}{\nu} + ... + \frac{\partial C(\{a_i'\})}{\partial a_n'}\frac{\partial a_n'}{\nu} + \frac{\partial C(\{a_i'\})}{\partial x_1'}\frac{\partial x_1'}{\nu} + \frac{\partial C(\{a_i'\})}{\partial x_2'}\frac{\partial x_2'}{\nu} \\ &= \frac{\partial C(\{a_i'\})}{\partial a_0'} * 0 + \frac{\partial C(\{a_i'\})}{\partial a_1'} a_0' + ... + \frac{\partial C(\{a_i'\})}{\partial a_n'} n a_{n-1}' - \frac{\partial C(\{a_i'\})}{\partial x_1'} * x_2' + \frac{\partial C(\{a_i'\})}{\partial x_2'} * 0 \\ &= 0 \end{aligned}$$

As we see, this again contains the application of the operator **D**. For the same reasons as before, this equation must hold for all values of ν, so we can obtain from here the condition for an expression to be covariant by transformations of type 2:

A polynomial expression $C(\{a_i'\}, x_1', x_2')$ is covariant by transformations of type 2 if and only if it satisfies the equation:

$$\mathbf{D}C - x_2\frac{\partial C}{\partial x_1} = 0$$

which gives us the second condition in the theorem.

3. The same is done for transformations of type 3. Because transformations M of type 3 have $det(M) = 1$, for any polynomial expression $C(\{a_i\}, x_1, x_2)$ to be covariant respect to transformations of this type, it must satisfy again the equation 6.

   We can retrieve the relations between the primed variables and the original ones from the previous proof:

$$\begin{aligned} a_i' &= a_i + \binom{n-i}{1} a_{i+1}\mu + ... + \binom{n-i}{k} a_{i+k}\mu^k + ... + a_n\mu^{n-i} \\ x_1' &= x_1 \\ x_2' &= -\mu x_1 + x_2 \end{aligned}$$

   And from this we may proceed in the same fashion. We want to derive both sides of equation 6 by μ, as this equation must hold for every possible value of μ. Doing this we obtain:

$$\begin{aligned} &\frac{\partial C(\{a_i'\})}{\partial a_0'}\frac{\partial a_0'}{\mu} + ... + \frac{\partial C(\{a_i'\})}{\partial a_n'}\frac{\partial a_n'}{\mu} + \frac{\partial C(\{a_i'\})}{\partial x_1'}\frac{\partial x_1'}{\mu} + \frac{\partial C(\{a_i'\})}{\partial x_2'}\frac{\partial x_2'}{\mu} \\ &= \frac{\partial C(\{a_i'\})}{\partial a_0'} n a_1' + \frac{\partial C(\{a_i'\})}{\partial a_1'}(n-1)a_2' + ... + \frac{\partial C(\{a_i'\})}{\partial a_n'} * 0 + \frac{\partial C(\{a_i'\})}{\partial x_1'} * 0 - \frac{\partial C(\{a_i'\})}{\partial x_2'} * x_1' \\ &= 0 \end{aligned}$$

   Again we can see in the development of the equation the operator Δ. From this we obtain the third condition of the statement:

   For a polynomial expression $C(\{a_i\}, x_1, x_2)$ to be a covariant by transformations of type 3, it must satisfy the equation:

$$\Delta C - x_1 \frac{\partial C}{\partial x_2} = 0$$

□

This is the easy characterization of invariants and covariants, but this set of conditions can still be reduced. Next we present the minimum set of conditions for a polynomial expression to be an invariant/-covariant of a given form:

Theorem 1.10. *A polynomial expression I in the coefficients $\{a_i\}$ is an invariant for binary forms of order n if and only if:*

1. *It is an homogeneous and isobaric polynomial and its degree and weight satisfy the equation $ng = 2p$*
2. *It satisfies the equation $\mathbf{D}\ I = 0$*

Theorem 1.11. *A polynomial expression C in the coefficients $\{a_i\}$ and variables x_1, x_2 which is homogeneous in the x_i is a covariant for binary forms of order n if and only if:*

1. *Written as: $C = C_0 x_1^m + C_1 x_1^m - 1x_2 + ... + C_m x_2^m$, the term C_0 is a homogeneous polynomial of degree g and isobaric of weight p and satisfy the equation $m = ng - 2p$*
2. *The term C_0 satisfy the equation $\mathbf{D}C_0 = 0$*
3. *Each of the other terms C_i satisfies the condition:*

$$C_i = \frac{(m-i)!}{m!}\Delta^i C_0 \tag{7}$$

The term C_0 is called the source of the covariant. As we see from the last theorem, it is important because one can determine the whole covariant from it.

The proof of these two theorems can be found in [2, chap. I.6]. We omit the proof because in order to make it we would need to explain in detail some non-trivial properties of the two operators $\mathbf{D}$ and Δ. These properties can take a bit long to prove, and since they are only interesting to us as a mean to prove these theorems, we will leave them out.

1.3 Simultaneous invariants and covariants

We have defined an invariant as a polynomial in the coefficients of binary forms that satisfies the invariant property (that it is invariant by lineal transformations). We can generalize this concept to apply it to a system of more than one form, producing the concept of simultaneous invariants.

If we consider a system of more than one form at a time:

$$\begin{aligned} f &= a_0 x_1^{n_1} + \binom{n_1}{1} a_1 x_1^{n_1-1} x_2 + \ldots + a_{n_1} x_2^{n_1} \\ g &= b_0 x_1^{n_2} + \binom{n_2}{1} b_1 x_1^{n_2-1} x_2 + \ldots + b_{n_2} x_2^{n_2} \\ &\ldots \end{aligned}$$

we can apply a linear transformation to the whole system, obtaining:

$$\begin{aligned} f|_M &= a'_0 x_1'^{n_1} + \binom{n_1}{1} a'_1 x_1'^{n_1-1} x'_2 + \ldots + a'_{n_1} x_2'^{n_1} \\ g|_M &= b'_0 x_1'^{n_2} + \binom{n_2}{1} b'_1 x_1'^{n_2-1} x'_2 + \ldots + b'_{n_2} x_2'^{n_2} \\ &\ldots \end{aligned}$$

With this, we can define the same concepts of invariants and covariants for a system of several forms at once. We give the definition for a system of two simultaneous forms, since the generalization from here is obvious:

Definition 1.12. Let $I(\{a_i\}, \{b_i\})$ be a polynomial expression in the coefficients of a system of two forms of orders n_1 and n_2. The expression I is said to be an invariant if it satisfies the invariant property, namely:

For any linear transformation $M \in GL_2(k)$, the expression in the new coefficients $(\{a'_i\}, \{b'_i\})$ remains the same (up to a fixed power of the determinant of the transformation matrix):

$$I(\{a'_i\}, \{b'_i\}) = det(M)^p I(\{a_i\}, \{b_i\})$$

One can see that this concept is easily generalized from two binary forms to any number of forms, being then the invariant I an expression in the coefficients of all the binary forms involved.

The same way as with invariants, we can also generalize the concept of covariant to simultaneous covariants of any number of forms (in the same way we did with invariants).

Definition 1.13. Let $C(\{a_i\}, \{b_i\}, x_1, x_2)$ be a polynomial expression in the coefficients of a system of two binary forms of orders n_1 and n_2, as well as in the two variables x_1, x_2. Then, the expression C is said to be a covariant if:

For any linear transformation $M \in GL_2(k)$, the expression in the new coefficients $(\{a'_i\}, \{b'_i\})$ remains the same (up to a fixed power of the determinant of the transformation matrix):

$$C(\{a'_i\}, \{b'_i\}, x'_1, x'_2) = det(M)^p I(\{a_i\}, \{b_i\}, x_1, x_2)$$

This time again, is immediate to generalize this concept from two binary forms to any number of forms.

Now we may want from the invariants/covariants of two or more forms the same kind of characterization as we had for invariants/covariants of a single form (giving us a simple way to check if an expression was an invariant/covariant). We will explain the conditions for invariants/covariants of two forms and (again) it will be clear how to extend this characterization to any number of forms.

Definition 1.14. Given two binary forms f_1, f_2 with orders n, m and coefficients $\{a_i\}, \{b_i\}$ respectively, for a monomial $M = Za_0^{v_0}a_1^{v_1}...a_n^{v_n}b_0^{w_0}...b_m^{w_m}$ on the coefficients of the forms, we define its weight as:

$$p = v_1 + 2v_2 + ... + nv_n + w_1 + ... + mw_m \tag{8}$$

Definition 1.15. Given two binary forms f_1, f_2 with orders n, m and coefficients $\{a_i\}, \{b_i\}$ respectively, we define the following operators:

$$\mathbf{D}_a = a_0\frac{\partial}{\partial a_1} + 2a_1\frac{\partial}{\partial a_2} + ... + na_{n-1}\frac{\partial}{\partial a_n}$$

$$\mathbf{D}_b = b_0\frac{\partial}{\partial b_1} + 2b_1\frac{\partial}{\partial b_2} + ... + mb_{m-1}\frac{\partial}{\partial b_m}$$

$$\Delta_a = na_1\frac{\partial}{\partial a_0} + (n-1)a_2\frac{\partial}{\partial a_1} + ... + a_n\frac{\partial}{\partial a_n - 1}$$

$$\Delta_b = mb_1\frac{\partial}{\partial b_0} + (m-1)b_2\frac{\partial}{\partial b_1} + ... + b_m\frac{\partial}{\partial b_m - 1}$$

$$\mathbf{D} = \mathbf{D}_a + \mathbf{D}_b$$

$$\Delta = \Delta_a + \Delta_b$$

With these definitions we can now specify the conditions for an expression to be a simultaneous invariant/covariant of a set of binary forms (as in the rest of the section, we state the theorem for simultaneous invariants/covariants of two forms, as the extension to any number of forms is direct).

Theorem 1.16. *Let $I(\{a_i\}, \{b_i\})$ be a polynomial function in the coefficients of forms of orders n_1 and n_2 respectively. Then, the expression I is a simultaneous invariant if and only if:*

1. *it is an isobaric polynomial of weight p, as a polynomial of the $\{a_i\}$ is homogeneous of degree g_1, as a polynomial of the $\{b_i\}$ is homogeneous of degree g_2 and it satisfies the equation $n_1g_1 + n_2g_2 = 2p$*

2. *it satisfies the equation $\mathbf{D}I = 0$*

Theorem 1.17. *Let $C(\{a_i\}, \{b_i\}, x_1, x_2)$ be a polynomial function in the coefficients of forms of orders n_1 and n_2 and the variables x_1, x_2, which is homogeneous as a polynomial in the x_i. Then, C is a simultaneous covariant if and only if:*

1. *Written as $C = C_0 x_1^m + \ldots + C_m x_2^m$, the term C_0 is an isobaric polynomial of weight p, as a polynomial in the a_i is homogeneous of degree g_1 and as a polynomial in the b_i is homogeneous of degree g_2, and it satisfies the equation $m = n_1 g_1 + n_2 g_2 - 2p$*

2. *The term C_0 satisfies $\mathbf{D} C_0 = 0$*

3. *Each of the rest C_i satisfy the equation:*

$$C_i = \frac{(m-i)!}{m!} \Delta^i C_0$$

The complete proof of these two theorems can be found in [2, chap. I.9]. They follow the same steps as the proofs for the characterization of invariants/covariants of just one binary form, just generalizing each step for considering we are working with two binary forms, but the process is the same. We won't include them since they do not contribute anything new to the thesis.

1.4 Basis of invariants

The set of invariants of binary forms of order n will be an essential part of our work. So we want to know a bit more about its structure. Let us look at some of the properties that have the set of invariants of forms of order n:

1. Let I be an invariant of forms of order n that has degree g and weight p. An scalar multiple of the invariant λI is also an invariant for forms of order n with degree g and weight p.

2. Let I_1 and I_2 be two invariant of forms of order n that have degrees g_1, g_2 and weights p_1, p_2 respectively. The product $I_1 I_2$ of the two invariants is also an invariant for forms of order n which has degree $g_1 + g_2$ and weight $p_1 + p_2$.

   This is clear looking at it from the pure definition of invariant. Let f_1 and f_2 be binary forms of order n and $M \in GL_2(k)$ such that $f_2 = f_1|_M$. Then:

$$I_1(f_2) = det(M)^{p_1} I_1(f_1)$$
$$I_2(f_2) = det(M)^{p_2} I_2(f_1)$$

   and so:

$$(I_1 I_2)(f_2) = det(M)^{p_1+p_2} (I_1 I_2)(f_1)$$

3. Let I_1 and I_2 be two invariants of forms of order n with the same degree g and weight p. Then the sum of the two invariants, namely $I_1 + I_2$, is also an invariant of forms of order n with degree g and weight p.

   This is again clear just by looking at the definition:

$$(I_1 + I_2)(f_2) = det(M)^p I_1(f_1) + det(M)^p I_2(f_1) = det(M)^p (I_1 + I_2)(f_1)$$

With these properties we can state the following:

Theorem 1.18. *The set of invariants of binary forms of order n have the structure of a* **graded algebra***.*

We will make a brief remark about this statement. When we talk about invariants we want to move in this "algebra of invariants", but because the sum of two invariants of different degree is not really an invariant, the elements of the algebra might not be necessarily invariants. Instead, the elements of the algebra of invariants are sums of invariants of different degree, for which each homogeneous part is an invariant.

This result (that the invariants have structure of an algebra) is not only true for binary forms, but also for forms in any number of variables (ternary forms, quaternary forms etc...) and even for systems of simultaneous forms.

One of the most important questions that surged in invariant theory when it was first studied (Cayley) was if the algebra of invariants for a system of simultaneous forms is or not finitely generated. This is indeed true, and we can state it as follows:

Theorem 1.19. *Hilbert's theorem. The algebra of invariants for binary forms of order n is finitely generated, for any arbitrary value of n.*

This is one of the most important results of invariant theory. When Cayley began studying invariant theory, he made this conjecture but could not prove it more than for the particular cases of binary forms up to order 6. Later, Gordan proved this result for a system of an arbitrary number of binary forms, but could not prove it for forms in more than two variables. It was Hilbert who found a generalizable proof of this theorem for forms in any number of variables.

In [2, chap. II] one can find two proofs of this theorem from Hilbert. The first one is a proof of the theorem using the representation of invariants as function of the roots of the forms. The second, more complex, is a generalizable proof for forms of any number of variables. We do not include any of these proofs in our work because of their extension and complexity.

Because of this result we know that, for binary forms of order n, there exists a finite number of invariants that generate the algebra of invariants for these forms. We will call this finite set of invariants a **basis** for the algebra of invariants of order n. This is an important concept that we will use a lot in the thesis. We could list some examples:

Example 1.20. For binary forms of order 2, every invariant can be written as a polynomial function of one invariant of degree 2 (the discriminant $I = a_0a_2 - a_1^2$). This only invariant is the basis for the algebra of invariants of order 2.

Example 1.21. For binary forms of order 4, every invariant can be written as a polynomial function of two invariants of degrees 2 and 3:

$$\begin{aligned} I_2 &= a_0a_4 - 4a_1a_3 + 3a_2^2 \\ I_3 &= a_0a_2a_4 - a_0a_3^2 - a_1^2a_4 + 2a_1a_2a_3 - a_2^3 \end{aligned}$$

so these two invariants are a basis for the algebra of invariants of order 4. We will prove this particular result in the next section.

For the algebra of invariants of a given order, the basis of invariants is not unique. In fact, for some cases there are different interesting bases that have been used in the literature. As, for example, for the algebra of invariants of order 6, we can encounter in the literature the bases of Igusa ($I_2, I_4, I_6, I_{10}, I_{15}$), Clebsch ($C_2, C_4, C_6, C_{10}, R$), etc... which are named after the person who introduced them. We will be using these bases often in our work.

1.5 Number of invariants

Now that we have the characterization of invariants and covariants, the question arises of how many invariants can be found for a given form of a certain degree and weight.

First of all it must be noticed from the properties shown in the previous section that any linear combination of two invariants of the same weight is yet another invariant for the same base form. Thus, the invariants of degree g and weight p generate a vector space, and there are an infinite number of invariants for a given form. What we will be interested in, though, is the dimension of this space, or what is the same, the number of linearly independent invariants of degree g and weight p.

Let us call $M_n(g,p)$ the set of monomials of the ring $\mathbb{Z}[\{a_i\}]$ with coefficient 1 that have degree g and weight p, and let us call $m_n(g,p)$ the cardinal of this set.

Theorem 1.22. *The number of linearly independent invariants of degree g and weight p for forms of order n, which we will call $w_n(g,p)$ is given by the formula:*

$$w_n(g,p) = m_n(g,p) - m_n(g,p-1) \tag{9}$$

We can give an intuitive notion of why this holds. The invariants for forms of order n can be expressed as $I = \sum Za_0^{\nu_0}a_1^{\nu_1}...a_n^{\nu_n}$, and if we want them to be of degree g and weight p the exponents must satisfy:

$$\nu_0 + \nu_1 + ... + \nu_n = g$$
$$\nu_1 + 2\nu_2 + ... + n\nu_n = p$$

there are $m_n(g,p)$ possible values for the exponents, so the sum can have this number of summands. Now, for I to be an invariant it must satisfy the equation $\mathbf{D}I = \sum Z\mathbf{D}a_0^{\nu_0}a_1^{\nu_1}...a_n^{\nu_n} = 0$. As the operator $\mathbf{D}$ applied to a term of degree g and weight p results in an homogeneous polynomial of degree g and isobaric of weight $p-1$, we have $m_n(g,p-1)$ distinct summands in the expression of $\mathbf{D}I$. As every term of this sum must banish, we end up with $m_n(g,p-1)$ linear equations for the coefficients Z that must be equal to 0.

Now, we have $m_n(g,p-1)$ linear equations for the $m_n(g,p)$ coefficients Z of the sum. We could choose arbitrarily $m_n(g,p) - m_n(g,p-1)$ coefficients, assign them a value, and the rest would be uniquely determined. We can assign values for the $m_n(g,p) - m_n(g,p-1)$ coefficients Z chosen in as much as $m_n(g,p) - m_n(g,p-1)$ linearly independent ways. This gives the number of invariants one can generate of degree g and weight p.

This is not a proof of the theorem. Note that we have not argued that the system of $m_n(g,p-1)$ equations are linearly independent. We won't include the complete proof here, as it is a bit larger than this and needs use of some results that we have not introduced. But the full proof of the theorem can be found in [2, p. 50].

Now we want expressions for the quantities $m_n(g,p)$ and $w_n(g,p)$.

Because the weight of an invariant is determined given the order and degree by the equation $ng = 2p$, one can write the previous formula as:

$$w_n(g) = w_n(g, \frac{ng}{2}) = m_n(g, \frac{ng}{2}) - m_n(g, \frac{ng}{2} - 1) \tag{10}$$

Using combinatorial arguments one can find expressions for the numbers $m_n(g,p)$ and thus, also for the numbers $w_n(g)$.

Theorem 1.23. *The cardinal of the set $M_n(g,p)$ is given by the formula:*

$$m_n(g,p) = \left[\frac{(1-x^{n+1})(1-x^{n+2})...(1-x^{n+g})}{(1-x)(1-x^2)...(1-x^g)}\right]_{x^p} \tag{11}$$

where $[f]_{x^k}$ denotes the coefficient of degree k of the Taylor series of the function f.

Proof. First we will see that $m_n(g,p)$ admits the following representation:

$$m_n(g,p) = \left[\frac{1}{(1-y)(1-xy)(1-x^2y)...(1-x^ny)}\right]_{x^py^g} \tag{12}$$

This will follow from the combinatorial interpretation of $m_n(g,p)$. If we break the expression we get:

$$\frac{1}{(1-y)(1-xy)(1-x^2y)...(1-x^ny)} =$$
$$(1+y+y^2+...)(1+xy+x^2y^2+...)(1+x^2y+x^4y^2+...)...(1+x^ny+x^{2n}y^2+...)$$

and the general term of this expansion could be expressed as:

$$y^{\nu_0}(xy)^{\nu_1}...(x^ny)^{\nu_n} = x^{\nu_1+2\nu_2+...+n\nu_n}y^{\nu_0+\nu_1+...+\nu_n}$$

so for the general expansion, the coefficient for the term x^py^g will be the number of ways one can express p as $\nu_1+2\nu_2+...+n\nu_n$ and g as $\nu_0+\nu_1+...+\nu_n$ simultaneously. This is equal to the number $m_n(g,p)$.

We have proven that the number $m_n(g,p)$ satisfy the equation 12 as the coefficient of x^py^g of the Taylor expansion of a function in x and y. We will now show how we can get from this formula to the formula given in the statement.

We want to get rid of the variable y on our formula. We can expand the expression as a polynomial in this variable and write it as:

$$\frac{1}{(1-y)(1-xy)(1-x^2y)...(1-x^ny)} = 1 + C_1y + C_2y^2 + ...$$

on every C_i is a polynomial in the x only. Now, because we have:

$$(1-y)\frac{1}{(1-y)(1-xy)(1-x^2y)...(1-x^ny)} = (1-x^{n+1}y)\frac{1}{(1-xy)(1-x^2y)(1-x^3y)...(1-x^{n+1}y)}$$

expanding both sides as polynomials in y

$$(1-y)(1+C_1y+C_2y^2+...) = (1-x^{n+1}y)(1+C_1xy+C_2x^2y^2+...)$$

and comparing the coefficients in both sides of the term for y^g we obtain the equation:

$$C_g - C_{g-1} = x^gC_g - x^{n+g}C_{g-1}$$

and:

$$C_g = \frac{1-x^{n+g}}{1-x^g}C_{g-1}$$

If we apply iteratively this relation for $C_g, C_{g-1}...C_2, C_1$, then we obtain that:

$$C_g = \frac{(1-x^{n+1})(1-x^{n+2})...(1-x^{n+g})}{(1-x)(1-x^2)...(1-x^g)}$$

And we can see the statement in the theorem is a direct consequence of this:

$$\begin{aligned} m_n(g,p) &= \left[\frac{1}{(1-y)(1-xy)(1-x^2y)...(1-x^ny)}\right]_{x^py^g} \\ &= [C_g]_{x^p} = \left[\frac{(1-x^{n+1})(1-x^{n+2})...(1-x^{n+g})}{(1-x)(1-x^2)...(1-x^g)}\right]_{x^p} \end{aligned}$$

□

Theorem 1.24. *The number of linearly independent invariants of degree g for a form of order n is given by the formula:*

$$w_n(g) = \left[\frac{(1-x^{n+1})(1-x^{n+2})...(1-x^{n+g})}{(1-x^2)(1-x^3)...(1-x^g)}\right]_{x^{\frac{ng}{2}}} \tag{13}$$

Proof. With the result for $m_n(g,p)$ we can proof the result for $w_n(g)$ quite easily. If we expand the expression for $m_n(g,p)$ as a Taylor series:

$$m_n(g,p) = \left[\frac{(1-x^{n+1})(1-x^{n+2})...(1-x^{n+g})}{(1-x)(1-x^2)...(1-x^g)}\right]_{x^p} = \left[c_0 + c_1x + c_2x^2 + ... + c_px^p + ...\right]_{x^p} = c_p$$

So if we develop both sides of the equation in the statement separately we obtain, in one side:

$$w_n(g) = m_n(g,p) - m_n(g,p-1) = c_p - c_{p-1}$$

where here $p = \frac{ng}{2}$. In the other side, we obtain:

$$\begin{aligned} &\left[\frac{(1-x^{n+1})(1-x^{n+2})...(1-x^{n+g})}{(1-x^2)(1-x^3)...(1-x^g)}\right]_{x^p} \\ &= \left[(1-x)\frac{(1-x^{n+1})(1-x^{n+2})...(1-x^{n+g})}{(1-x)(1-x^2)(1-x^3)...(1-x^g)}\right]_{x^p} \\ &= \left[(1-x)(c_0 + c_1x + c_2x^2 + ... + c_px^p + ...)\right]_{x^p} \\ &= \left[c_0 + (c_1-c_0)x + (c_2-c_1)x^2 + ... + (c_p-c_{p-1})x^p + ...\right]_{x^p} = c_p - c_{p-1} \end{aligned}$$

So we have proven that:

$$w_n(g) = c_{\frac{ng}{2}} - c_{\frac{ng}{2}-1} = \left[\frac{(1-x^{n+1})(1-x^{n+2})...(1-x^{n+g})}{(1-x^2)(1-x^3)...(1-x^g)}\right]_{x^{\frac{ng}{2}}}$$

□

With this result we can find out easily how many invariants does a binary form has of a certain degree. But this result can also be used to work out (by working with the general expression in function of g) which invariants generate the algebra of invariants for forms of that order. Here we give an example of how to do this:

Example 1.25. For binary forms of order $n = 4$ we have that $p = 2g$, so one has that:

$$w_4(g) = \left[\frac{(1-x^5)(1-x^6)...(1-x^{4+g})}{(1-x^2)(1-x^3)...(1-x^g)}\right]_{x^{2g}}$$

or what is the same

$$w_4(g) = \left[\frac{(1-x^{g+1})(1-x^{g+2})...(1-x^{4+g})}{(1-x^2)(1-x^3)(1-x^4)}\right]_{x^{2g}}$$

If we operate this expression we obtain the following:

$$\begin{aligned}
w_4(g) &= \left[\frac{(1-x^{g+1})(1-x^{g+2})...(1-x^{4+g})}{(1-x^2)(1-x^3)(1-x^4)}\right]_{x^{2g}} \\
&= \left[\frac{1-x^{g+1}-x^{g+2}-x^{g+3}-x^{g+4}}{(1-x^2)(1-x^3)(1-x^4)}\right]_{x^{2g}} \\
&= \left[\frac{1}{(1-x^2)(1-x^3)(1-x^4)}\right]_{x^{2g}} - \left[\frac{x^{g+1}(1+x+x^2+x^3)}{(1-x^2)(1-x^3)(1-x^4)}\right]_{x^{2g}} \\
&= \left[\frac{1}{(1-x^2)(1-x^3)(1-x^4)}\right]_{x^{2g}} - \left[\frac{x^{g+1}}{(1-x)(1-x^2)(1-x^3)}\right]_{x^{2g}} \\
&= \left[\frac{1}{(1-x^2)(1-x^3)(1-x^4)}\right]_{x^{2g}} - \left[\frac{x}{(1-x)(1-x^2)(1-x^3)}\right]_{x^{g}} \\
&= \left[\frac{1}{(1-x^2)(1-x^3)(1-x^4)}\right]_{x^{2g}} - \left[\frac{x^2}{(1-x^2)(1-x^4)(1-x^6)}\right]_{x^{2g}} \\
&= \left[\frac{(1-x^6)-x^2(1-x^3)}{(1-x^2)(1-x^3)(1-x^4)(1-x^6)}\right]_{x^{2g}} \\
&= \left[\frac{1-x^2+x^3}{(1-x^2)(1-x^4)(1-x^6)}\right]_{x^{2g}} \\
&= \left[\frac{1-x^2}{(1-x^2)(1-x^4)(1-x^6)}\right]_{x^{2g}} + 0 \\
&= \left[\frac{1}{(1-x^4)(1-x^6)}\right]_{x^{2g}} \\
&= \left[\frac{1}{(1-x^2)(1-x^3)}\right]_{x^{g}}
\end{aligned}$$

What can be derived from this calculation is the following: The number of invariants for a binary quartic of degree g is

$$w_4(g) = \left[\frac{1}{(1-x^2)(1-x^3)}\right]_{x^g} = \left[(1+x^2+x^4+x^6+...)(1+x^3+x^6+x^9+...)\right]_{x^g}$$

and we will have that there are as many invariants of order g as there are non-negative integer solutions for k, l of the equation:

$$2k + 3l = g$$

For $g = 2$ and $g = 3$ we have just one invariant for the binary quartic, namely:

$$\begin{aligned}
I_2 &= a_0a_4 - 4a_1a_3 + 3a_2^2 \\
I_3 &= a_0a_2a_4 - a_0a_3^2 - a_1^2a_4 + 2a_1a_2a_3 - a_2^3
\end{aligned}$$

and we have that for every k, l the expression $I = I_2^k I_3^l$ is also an invariant of degree $g = 2k + 3l$. As we can do this for every k, l that add up to g, we can obtain all the invariants of degree g in this manner.

Therefor we have that the invariants I_2 and I_3 are generators of the algebra of invariants for the binary quartic, or what is the same, that all the invariants of the binary quartic can be expressed as a polynomial expression of the two invariants I_2 and I_3.

1.6 Transvections

Until now we have explored the properties of invariants and covariants, and found a simple way to determine if given expressions were or not invariants or covariants of binary forms. But what we do not have is a systematic way for finding the invariants and covariants of a form. Here we introduce the concept of transvectant (or transvection), which will allow us to do exactly that.

Definition 1.26. Given two binary forms f_1, f_2 of orders n_1, n_2 and coefficients $\{a_i\}, \{b_i\}$ we define the **transvectant** of order p (or p-th transvectant) as the simultaneous covariant of the two forms that has as its source the following expression:

$$C_0 = \sum_{i=0}^{p} (-1)^i \binom{p}{i} a_i b_{p-i}$$

We will denote the resulting covariant (the transvectant) as $(f_1, f_2)_p$, and it is a covariant of degree $g = 2$, weight p and order $m = n_1 + n_2 - 2p$.

One can check easily that this expression generates indeed a simultaneous covariant of the two forms. If we look at the characterization of simultaneous covariants in 1.17, we see that:

1. the first condition holds clearly. It is clear that the expression for C_0 is homogeneous both in the $\{a_i\}$ and in the $\{b_i\}$ and that it is an isobaric polynomial with weight equal to $i + p - i = p$. We can see that (with the same notation as in 1.17) $g_1 = g_2 = 1$, and then $m = n_1g_1 + n_2g_2 - 2p = n_1 + n_2 - 2p$.
2. If we operate the expression $\mathbf{D}C_0$, we get:
$$\begin{aligned}\mathbf{D}C_0 = \mathbf{D}\sum_{i=0}^{p} (-1)^i \binom{p}{i} a_i b_{p-i} &= \sum_{i=0}^{p} \mathbf{D}(-1)^i \binom{p}{i} a_i b_{p-i} \\ &= \sum_{i=0}^{p} (-1)^i \binom{p}{i} \left[ia_{i-1}b_{p-i} + (p-i)a_i b_{p-i-1}\right] \\ &= \sum_{i=1}^{p} (-1)^i \binom{p}{i} ia_{i-1}b_{p-i} + (-1)^{i-1}\binom{p}{i-1}(p-i+1)a_{i-1}b_{p-i} \\ &= \sum_{i=1}^{p} (-1)^i \left[\frac{p!}{i!(p-i)!}i - \frac{p!}{(i-1)!(p-i+1)!}(p-i+1)\right] a_{i-1}b_{p-i} = 0\end{aligned}$$
which satisfies the second condition.
3. The third condition holds directly because we have defined the transvectant by its source.

To realize the importance of the transvectants in computing covariants we have to note a few things. First, one can see a covariant as a form in itself. If we have the covariant $C = C_0x_1^m + \ldots + C_mx_2^m$ where each of the C_i are polynomials in the coefficients $\{a_i\}$ of forms of order n, one can take the C_i as new coefficients and consider the whole covariant C as a binary form of order m.

With this in mind we can state the following important result:

Theorem 1.27. *Let $X = \{C_1, C_2, ..., C_k\}$ be a set of k covariants for forms of order n, and let C be a simultaneous covariant of the set X considering those covariants as binary forms. Then C is also a covariant for forms of order n.*

This can also be expressed as the following:

Covariants of covariants or simultaneous systems of covariants are at the same time covariants of the same forms.

This proposition is also true for simultaneous covariants of more than one binary form. The proof of this can be found in [2, p. 92]. We do not include it because the proof is a bit messy and just consists on working with the expressions of the covariants and the covariants of covariants.

This highlights the importance of the transvection operation, because taking the covariants of a binary form as forms in themselves one can have the following:

Proposition 1.28. *Let C_1, C_2 be covariants for forms of order n, which have orders m_1, m_2 and degrees g_1, g_2 respectively. Then the p-th transvectant of C_1 and C_2, $(C_1, C_2)_p$ is also a covariant for forms of order n, and have degree $g_1 + g_2$ and order $m_1 + m_2 - 2p$.*

Now with this proposition we can obtain new covariants for binary forms from a set of covariants that we know. We can use this to develop a method to generate systematically covariants. We begin with the form f itself (which is a covariant), and we want to produce covariants of f. We start by computing the transvectants of f over itself. This produces some covariants of f. Then we compute the transvectants of these covariants with each other, and we obtain new covariants. We can repeat the process until we no longer obtain new covariants of f.

With this process we have come to what we wanted, a systematic way to construct new invariants/covariants of a binary form. But this method allows us not only to find new invariants/covariants of a form, it allows us to find all of them, what we can state as the following:

Theorem 1.29. *The set of covariants of a binary form is generated by the iterative application of the transvection operation over f, or what is the same:*

Given a binary form f. Let C be the closure of f by the operation of transvection, then every covariant of f is included in C.

The proof of this theorem can be found in [1].

1.7 Invariants as functions of the roots

We have presented in the previous section a way to construct the set of all invariants and covariants of a binary form. In this section we will explain another method to construct all the invariants of a given binary form, as functions of "the roots" of the form.

Given a binary form F:

$$F(x_1, x_2) = a_0x_1^n + a_1x_1^{n-1}x_2 + ... + a_{n-1}x_1x_2^{n-1} + a_nx_2^n$$

where this time we are considering the coefficients removing the binomial coefficients. We do this because for this section it is more convenient. It should be clear that the concepts introduced in previous sections (concepts of invariants, covariants etc...) remain the same this way, the only things that change a bit are the characterizations (and we won't need those for this section). Let

$$f(x) = F(x,1) = a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n$$

be the deshomogenized form. Let $\{\omega_i\}$ be the roots of f. When we mention the roots of a binary form we mean these $\{\omega_i\}$, the roots of its deshomogenized form. So we could write:

$$\begin{aligned} f(x) &= a_0(x-\omega_1)(x-\omega_2)...(x-\omega_n) \\ F(x_1,x_2) &= a_0(x_1-\omega_1x_2)(x_1-\omega_2x_2)...(x_1-\omega_nx_2) \end{aligned}$$

Then we have the following:

Theorem 1.30. *Let:*

$$I = a_0^m \sum \prod \omega_i - \omega_j \tag{14}$$

be a symmetric function on the $\{\omega_i\}$ such that:

1. *in each summand every ω_i appears exactly m times.*
2. *each summand is an homogeneous function on the differences.*

then I can be expressed as a polynomial function in the coefficients $\{a_i\}$ of the form and, as a function of the coefficients, I is an invariant.

Proof. First of all, the fact that the expression 14 can be expressed as a polynomial function in the coefficients $\{a_i\}$ of binary forms of order n follows directly from the fundamental theorem of symmetric polynomials.

Next, to proof that this function is indeed an invariant, let us take a look at how a linear transformation modify the roots of a binary form. Let $f = a_0(x_1-\omega_1x_2)(x_1-\omega_2x_2)...(x_1-\omega_nx_2)$ be a binary form of order n with roots $\{\omega_i\}$. Let $M = \left(\begin{smallmatrix}\alpha_{11} & \alpha_{12}\\ \alpha_{21} & \alpha_{22}\end{smallmatrix}\right)$ be a general linear transformation and let $g = f|_M = a'_0(x'_1-\omega'_1x'_2)(x'_1-\omega'_2x'_2)...(x'_1-\omega'_nx'_2)$ be the form f transformed by M. Then:

$$\begin{aligned} g = f|_M &= a_0((\alpha_{11}x'_1+\alpha_{12}x'_2)-\omega_1(\alpha_{21}x'_1+\alpha_{22}x'_2))...((\alpha_{11}x'_1+\alpha_{12}x'_2)-\omega_n(\alpha_{21}x'_1+\alpha_{22}x'_2)) \\ &= a_0((\alpha_{11}-\alpha_{21}\omega_1)x'_1+(\alpha_{12}-\alpha_{22}\omega_1)x'_2)...((\alpha_{11}-\alpha_{21}\omega_n)x'_1+(\alpha_{12}-\alpha_{22}\omega_n)x'_2) \\ &= a'_0\left(x'_1-\frac{(-\alpha_{12}+\alpha_{22}\omega_1)}{(\alpha_{11}-\alpha_{21}\omega_1)}x'_2\right)...\left(x'_1-\frac{(-\alpha_{12}+\alpha_{22}\omega_n)}{(\alpha_{11}-\alpha_{21}\omega_n)}x'_2\right) \\ &= a'_0(x'_1-\omega'_1x'_2)(x'_1-\omega'_2x'_2)...(x'_1-\omega'_nx'_2) \end{aligned} \tag{15}$$

With this development each linear factor $(x_1-\omega_ix_2)$ is transformed to another linear factor $(x'_1-\omega'_jx'_2)$. Without loss of generality we can suppose that each ω_i transforms into ω'_i. Then we end up with the following relation between the roots:

$$\omega'_i = \frac{(-\alpha_{12}+\alpha_{22}\omega_i)}{(\alpha_{11}-\alpha_{21}\omega_i)} \tag{16}$$

and the relation between the two coefficients a_0 and a'_0:

$$a'_0 = a_0(\alpha_{11}-\alpha_{21}\omega_1)(\alpha_{11}-\alpha_{21}\omega_2)...(\alpha_{11}-\alpha_{21}\omega_n) \tag{17}$$

If we now develop the expression for the differences:

$$\begin{aligned}
\omega'_i - \omega'_j &= \frac{(-\alpha_{12}+\alpha_{22}\omega_i)}{(\alpha_{11}-\alpha_{21}\omega_i)} - \frac{(-\alpha_{12}+\alpha_{22}\omega_j)}{(\alpha_{11}-\alpha_{21}\omega_j)} \\
&= \frac{(-\alpha_{12}+\alpha_{22}\omega_i)(\alpha_{11}-\alpha_{21}\omega_j) - (-\alpha_{12}+\alpha_{22}\omega_j)(\alpha_{11}-\alpha_{21}\omega_i)}{(\alpha_{11}-\alpha_{21}\omega_i)(\alpha_{11}-\alpha_{21}\omega_j)} \\
&= \frac{(-\alpha_{11}+\alpha_{12}\alpha_{21}\omega_j+\alpha_{11}\alpha_{22}\omega_i-\alpha_{21}\alpha_{22}\omega_i\omega_j) - (-\alpha_{11}+\alpha_{12}\alpha_{21}\omega_i+\alpha_{11}\alpha_{22}\omega_j-\alpha_{21}\alpha_{22}\omega_i\omega_j)}{(\alpha_{11}-\alpha_{21}\omega_i)(\alpha_{11}-\alpha_{21}\omega_j)} \\
&= \frac{(\alpha_{12}\alpha_{21}-\alpha_{11}\alpha_{22})\omega_j + (\alpha_{11}\alpha_{22}-\alpha_{12}\alpha_{21})\omega_i}{(\alpha_{11}-\alpha_{21}\omega_i)(\alpha_{11}-\alpha_{21}\omega_j)} \\
&= \frac{det(M)(\alpha_i-\alpha_j)}{(\alpha_{11}-\alpha_{21}\omega_i)(\alpha_{11}-\alpha_{21}\omega_j)}
\end{aligned}$$

Now, because we have that the expression 14 is homogeneous in the differences (let us say that it is homogeneous of degree p), then we have:

$$\begin{aligned}
I' = a_0'^m \sum\prod \omega'_i - \omega'_j &= a_0'^m \sum\prod \frac{det(M)(\alpha_i-\alpha_j)}{(\alpha_{11}-\alpha_{21}\omega_i)(\alpha_{11}-\alpha_{21}\omega_j)} \\
&= a_0'^m det(M)^p \sum\prod \frac{(\alpha_i-\alpha_j)}{(\alpha_{11}-\alpha_{21}\omega_i)(\alpha_{11}-\alpha_{21}\omega_j)}
\end{aligned}$$

now, as in every summand we have that each root ω_i appears exactly m times, we can put this as:

$$\begin{aligned}
I' &= a_0'^m det(M)^p \sum\prod \frac{(\alpha_i-\alpha_j)}{(\alpha_{11}-\alpha_{21}\omega_i)(\alpha_{11}-\alpha_{21}\omega_j)} \\
&= a_0'^m det(M)^p \sum \frac{1}{[(\alpha_{11}-\alpha_{21}\omega_1)(\alpha_{11}-\alpha_{21}\omega_2)...(\alpha_{11}-\alpha_{21}\omega_n)]^m} \prod(\omega_i-\omega_j) \\
&= a_0^m det(M)^p \sum\prod(\omega_i-\omega_j) = det(M)^p I
\end{aligned}$$

where we have used the relation between the two coefficients a_0 and a'_0.

This ends the proof of the theorem. □

This way of seeing the invariants of a form (as function of its roots) gives us another way to compute the invariants of a form. This is especially useful for computing the numerical value of the invariants of a form in a quick way. Also, one of the advantages of this representation is that the expressions for the invariants remain relatively simple even for forms of bigger n. In contrast, when we use the representation as polynomials in the coefficients of forms of order n, the expressions for the invariants get really complicated really quickly.

Example 1.31. For forms of order 4 we can construct the following invariants:

$$I_2 = a_0^2 \sum (\omega_1-\omega_2)^2(\omega_3-\omega_4)^2$$

$$I_3 = a_0^3 \sum (\omega_1-\omega_2)^2(\omega_3-\omega_4)^2(\omega_1-\omega_3)(\omega_2-\omega_4)$$

that make a basis of invariants for forms of order 4.

1.8 Classification of binary forms

Now we want to talk about classification of binary forms. With this we mean, to know when two binary forms are isomorphic (or linearly equivalent) to each other. Fortunately, the invariants of the forms gives us the information needed to know this.

We know that if two forms f_1 and f_2 are isomorphic to each other (there exists a linear transformation M between them), then the value of their invariants differ by a power of the determinant of M. This is due to the definition of invariant. Here we see that we also have the inverse: If the invariants differ by a fixed power of a number, then the two forms are equivalent. We state this as follows:

Theorem 1.32. *Let f_1 and f_2 be two binary forms of order n such that neither one has roots of multiplicity greater than $n/2$. Then the two forms f_1 and f_2 are linearly equivalent if and only if there exists an r such that for every invariant I of weight p we have that:*

$$I(f_1) = r^p I(f_2)$$

where r will be the determinant of the linear transformation existent between the two forms.

The proof of this theorem can be found in [5].

The fact that the two forms must have their roots with multiplicities not greater than $n/2$ should not concern us a lot. The reason for this is that for one of the most common applications of these theory, the study of hyperelliptic curves, this condition holds (as an hyperelliptic curve is defined by a binary form with all its roots different). We will make a few comments on hyperelliptic curves in 2.1.1.

The concept of absolute invariant that we will explain right now allows us to transform this result into a stronger one, in which the ugly r^p in the previous equation disappears.

Definition 1.33. Let f be a binary form of order n. We define an absolute invariant of f as a quotient of two invariants of f of the same degree.

Theorem 1.34. *Let f_1 and f_2 be two binary forms of order n such that neither one has roots of multiplicity greater than $n/2$. The forms f_1, f_2 are linearly equivalent if and only if every absolute invariant i satisfies the equation:*

$$i(f_1) = i(f_2)$$

Proof. Let I_1, I_2 be two invariants of the same weight p. Because we can form an absolute invariant $i = \frac{I_1}{I_2}$, we get that:

$$\frac{I_1(f_1)}{I_2(f_1)} = i(f_1) = i(f_2) = \frac{I_1(f_2)}{I_2(f_1)}$$
$$\frac{I_1(f_1)}{I_1(f_2)} = \frac{I_2(f_1)}{I_2(f_2)}$$

We can do this for any pair of invariants of weight p. So we obtain that for any invariant I of weight p:

$$\frac{I(f_1)}{I(f_2)} = \frac{I_1(f_1)}{I_1(f_2)} = \lambda_p$$

So, for invariants of weight p we have that $I(f_1) = \lambda_p I(f_2) = r_p^p I(f_2)$. If this number r_p were the same for all possible values of p, then the theorem would be proven, because we would have that for any invariant I of weight p, then $I(f_1) = r_p^p I(f_2) = r^p I(f_2)$.

Let I_p, I_q be two invariants of weights p and q respectively. Because of this we have that:

$$I_p(f_1) = r_p^p I_p(f_2)$$
$$I_q(f_1) = r_q^q I_q(f_2)$$

but the two invariants I_p^q and I_q^p are both invariants of weight pq, so we have:

$$I_p^q(f_1) = r_p^{pq} I_p^q(f_2) = r_{pq}^{pq} I_p^q(f_2)$$
$$I_q^p(f_1) = r_q^{pq} I_q^p(f_2) = r_{pq}^{pq} I_q^p(f_2)$$

and we obtain that $r_p = r_q = r_{pq}$. As we can do this for any weight, we obtain that $r_p = r$. And this ends the proof.

□

We know now that for two binary forms, the property of being isomorphic is equivalent to their invariants being equivalent (equal to a fixed power of the determinant). But since we know that the algebra of invariants for forms of order n is finitely generated, we can give this stronger result:

Proposition 1.35. *Let f_1 and f_2 be two binary forms of order n such that neither one has roots of multiplicity greater than $n/2$. Then the two forms f_1 and f_2 are linearly equivalent if and only if there exists an r such that for every invariant I in a fixed basis of invariants for forms of order n, satisfies:*

$$I(f_1) = r^p I(f_2)$$

where p is the weight of the invariant and r will be the determinant of the determinant of the linear transformation existent between the two forms.

Proof. Let $S = \{I_1, I_2...I_k\}$ be a basis of invariants for forms of order n. Because S is a basis, we have that every invariant I of forms of order n can be expressed as a polynomial function of the basis:

$$I = \phi(I_1, I_2, ..., I_k) = \sum Z I_1^{\nu_1} I_2^{\nu_2} ... I_k^{\nu_k}$$

where all the summands are invariants of the same weight, or what is the same: $\nu_1 p_1 + \nu_2 p_2 + ... + \nu_k p_k = p$ where p_i is the weight of the invariant I_i and p is the weight of I.

Because of this, and because for any invariant of the basis the relation $I_i(f_1) = r^{p_i} I_i(f_2)$, we obtain:

$$\begin{aligned} I(f_2) &= \phi(I_1(f_2), I_2(f_2), ..., I_k(f_k)) = \phi(r^{p_1} I_1(f_1), r^{p_2} I_2(f_1), ..., r^{p_k} I_k(f_1)) \\ &= \sum Z r^{\nu_1 p_1 + \nu_2 p_2 + ... + \nu_k p_k} I_1^{\nu_1}(f_1) I_2^{\nu_2}(f_1) ... I_k^{\nu_k}(f_1) \\ &= r^p \sum Z I_1^{\nu_1}(f_1) I_2^{\nu_2}(f_1) ... I_k^{\nu_k}(f_1) = r^p I(f_1) \end{aligned}$$

because this is true for any invariant, the theorem is proven. □

2. Implementation

In this chapter of the thesis we want to focus on the practical application of the theory explained in the first chapter. We will want to expose some problems of invariant theory that have to be approached computationally, and then we want to show the code we have developed to solve some of these problems, explaining what is the method that has been chosen to solve the problems, along with the difficulties we have encountered, and the pros and cons of each method considered.

For the computational part of this thesis we have decided to use the open source software "Sage". Sage is an open source math-oriented software with a python-based language, which makes it perfect for our purpose.

First we will list some of the interesting problems one can face when working with invariants, such as computing an explicit basis of invariants for forms of certain order. Then we will briefly mention the things we can encounter already programmed in Sage related to invariant theory and close to the topic of our thesis. After this we will go through the problems we have implemented for this work, discuss the difficulties that they present and give a quick impression of our solution (where the full code will be in the appendixes).

2.1 The computational problems of invariant theory

The objective in this section will be to give a quick review of the computational problems that we can encounter envolving directly or indirectly invariant theory.

First we will introduce briefly the concepts of hyperelliptic curves and their relation with binary forms and invariant theory. We will do this because some of the problems that we will enunciate later have a great interest in their application in hyperelliptic curves. Later we will enumerate some of the interesting computational problems that can be approached regarding invariant theory, such as finding a explicit basis of invariants for binary forms of a certain order or finding an explicit isomorphism between two binary forms. Later in this chapter we will see for some of these problems what was our approach, with a review on the details of the implementation and the difficulties we confronted when solving the problems.

2.1.1 Invariant theory in the study of hyperelliptic curves

In this section we explain the object of study of hyperelliptic curves and its relation with invariant theory. This is interesting in our thesis because, thanks to this relationship between hyperelliptic curves and binary forms, a lot of the problems we will show next are applicable to not only the study of binary forms but also the study of these hyperelliptic curves.

Definition 2.1. An hyperelliptic curve is an algebraic curve defined by an equation of the form:

$$y^2 = f(x) \tag{18}$$

Where $f(x)$ is a polynomial of degree $n > 4$, whose roots are all distinct. The degree of the polynomial f determines the genus of the curve over the field of complex numbers, and we have that a curve has genus g if it has degree $n = 2g + 1$ or $n = 2g + 2$.

Definition 2.2. Considering an hyperelliptic curve of genus g given by the model $y^2 = f(x)$, we define its associated binary form as:

$$F(x_1, x_2) = x_1^n f(x_1/x_2)$$

where $n = 2g + 1$ or $n = 2g + 2$ is the degree of f.

Similarly, given a binary form $F(x_1, x_2)$ of degree n we can define its associated hyperelliptic curve as the object defined by the equation:

$$y^2 = f(x) = F(x, 1)$$

With this last definition it is clear how hyperelliptic curves relate to invariant theory. One can study a lot of the properties of an specific hyperelliptic curve by studying its associated binary form and its invariants.

2.1.2 List of problems

Now let us take a look at some of the computational problems that we could address with the theory that we have studied in this thesis. Some of the problems that we will list right now we will approach later in more detail in 2.3, where we will see how we implemented a solution for these problems. The problems that were not implemented for this work are mentioned here for the purpose of showing the variety of applications that can have the theory that we have studied.

1. **Finding a basis of invariants**

   One of the first interesting things one can think of doing when working with invariants of a binary form is to find a basis of invariants for a form of order n.

   It is easy to see why it is interesting to find such a basis. Given a basis of invariants for a form, all the other invariants of that form are determined by the values of the basis. This for example allows us to determine if two binary forms are isomorphic, as explained in 1.35, just by looking to the values of the invariants of the basis.

   In the sections 1.6 and 1.7 we explored some ways of constructing systematically covariants and invariants of binary forms, by using transvections and by seeing the invariants as functions of the roots of the form. Both methods have their advantages and disadvantages that we will discuss thoroughly in the section about our implementation 2.3.

2. **Finding an isomorphism between two forms**

   Having a basis of invariants gives us an easy and quick way to check if two binary forms (respectively hyperelliptic curves) are isomorphic to each other. The immediate question that arises after this is, given two binary forms, known isomorphic to each other, if we can find an isomorphism between the two forms and how.

3. **Finding the group of automorphism**

   This is less of a computational problem. Given a binary form we would want to know what its group of automorphisms looks like (to what subgroup of $GL_2(k)$ it corresponds). Although it is not obvious (and we won't enter in much detail here), one can figure out the structure of the group of automorphism of a binary form (respectively an hyperelliptic curve) just by looking at the invariants of the form.

   We just give an example to see how things work. Let us consider we are studying binary forms of order 6, and we have the basis of invariants of Clebsch $(C_2, C_4, C_6, C_{10}, R)$. Then, a non-trivial result shows us that if the relations $C_4 = C_6 = C_{10} = 0$, then the group of automorphisms of the form is homeomorphic to the cyclic group C_5. With the same kind of relations, we could study all the possibilities for the group of automorphisms of forms of order 6 (which we are not interested in doing).

One can also approach the problem of finding the group of automorphisms itself (computationally). This combined with the previous problem would allow us to compute all the isomorphisms between two forms.

4. **Inverse invariant problem**

   Until now we have studied various ways to find invariant and covariants of binary forms, and studied what these were useful for and what did they told us about the forms in question. But it is also interesting at this point to consider the problem backwards: given a set of values, we would want to find a binary form whose invariants take these values.

   For example, considering what we said about the group of automorphisms in the previous problem, one could be interested in finding a binary form (or respectively an hyperelliptic curve) that has certain group of automorphisms. Knowing the conditions some invariants must fulfill to have each group of automorphisms, one only has to find a form (or a curve) with invariants satisfying the conditions required in each case.

2.2 What is already implemented

We want to take a look at the things that are already implemented in Sage, to have an idea of what has been done. Other mathematical software (such as Magma or Mathematica) have more done about the topic, but we will stick with what is implemented in Sage, because it is the software we will work with.

We have found features implemented related to invariant theory mainly in two modules: the "invariant_theory" module and a few modules about hyperelliptic curves. We will explain briefly what these contains.

1. **invariant_theory**: This module, as its name suggests, is exclusively focused in invariant theory. Although, almost all that is implemented is computing certain invariants and covariants of forms.

   We won't be listing all the classes and functions implemented in this module, because it is a very large list. We will make some comments about what is implemented overall.

   The module have one base class, "AlgebraicForm", to implement the basic functionalities of forms (such as transform by a linear transformation). For the rest of functionalities for forms, as the ones for computing the invariants, they are implemented only in the derived classes, which consists of specific kinds of forms. For example, if we wanted to compute the invariants of a binary form of order 4, we would have to use the derived class "BinaryQuartic".

   There are not many classes implemented. For instance, for binary forms, we can only work in binary quadratic and binary quartic forms. The good thing about this module, though, is that it does not restrict to binary forms, or even to only one form. It has implemented several functions to compute invariants of ternary forms, and allows to do some things with a set of more than one form.

2. **hyperelliptic curves**: There are a few classes on hyperelliptic curves implemented in Sage, but the one that touches invariants is the one that specializes in hyperelliptic curves of genus 2, "hyperelliptic_g2_generic". This is because for curves of bigger genus, these calculations tend to be far more complex.

Even for genus 2 curves there is not really much implemented. Regarding our topic there are four functions we can find, all related to computing the values of a basis of invariants for an hyperelliptic curve of genus 2.

The functions “clebsch_invariants()” and “igusa_clebsch_invariants()”, each compute respectively the Clebsch basis of invariants (A, B, C, D) and the Igusa-Clebsch basis (I_2, I_4, I_6, I_{10}). These are two different bases for the algebra of invariants for forms of order 6. The two bases can be used for the same purposes, the difference between them can be seen when working with binary forms defined in a field of characteristic not 0, where the Clebsch invariants do not work (because the way they are constructed).

The two other functions, “absolute_igusa_invariants_kohel()” and “absolute_igusa_invariants_wamelen()“ each return a list of three absolute invariants for an hyperelliptic curve of genus 2. We saw in the section about classification the importance of absolute invariants, because two isomorphic hyperelliptic curves (the same as binary forms) must have the same absolute invariants.

As we see, there is not much work done in Sage about invariants.

2.3 Our work

Now we want to explain in detail the code developed for this thesis. All the code has been implemented in Sage, following the standars and with all the documentation that Sage requires included in each function. All the relevant files for the work are included in the appendixes.

2.3.1 The main code

The main code of the work is in the file named *binary_form.sage*, here included in the appendix A. Here we will explain the important aspects of what there is implemented in this code.

For this work we have taken a classical view of invariant theory, and we have kept this also at implementation. As it is, the first thing we implemented was a main class "binary_form", which seemed adequate. A binary_form object has as its attributes the field in which it is defined, its order n and its coefficients $\{a_i\}$, and there are implemented the basic functions we would expect to work with and handle easily binary forms: We can add two forms of the same order, we can multiply two binary forms, or we can apply a linear transformation to a form to obtain an equivalent (an essential feature).

Example 2.3. Here we show some examples of work with the this class:

```
sage: f1 = binary_form(QQ,3,[1,1,1,1]); f1
x^3 + x^2*y + x*y^2 + y^3
sage: f2 = binary_form(QQ,3,[2,0,1,2]); f2
2*x^3 + x*y^2 + 2*y^3
sage: f3 = f1.transform([[1,1],[0,1]]); f3
x^3 + 4*x^2*y + 6*x*y^2 + 4*y^3
sage: f1 + f2
3*x^3 + x^2*y + 2*x*y^2 + 3*y^3
sage: f1 * f3
x^6 + 5*x^5*y + 11*x^4*y^2 + 15*x^3*y^3 + 14*x^2*y^4 + 10*x*y^5 + 4*y^6
```

In the main code we have implemented another three classes, that work in group in some sense and that serve together for the same purpose. These classes are the classes "Term", "Invariant" and "Covariant". These classes, as one could have guessed, serve the purpose to implement the concept of invariants and covariants of a form. Though, they are not the same as invariants and covariants (we will explain this now).

As we mentioned before, because we were approaching the problem in a classical way, we wanted to implement these concepts in its crude form, as polynomials in the coefficients of binary forms. And that is exactly what they are in our implementation.

Now we will explain briefly how these three classes are structured and what exactly represents each of them:

1. Term: The class "Term" here represents a monomial in the coefficients of a form, an expression of the type $ca_0^{v_0}a_1^{v_1}...a_n^{v_n}$, for a form of order n. Having this class is useful in order to have a cleaner and simpler code in the other two classes, but its function is purely auxiliary.

   It has as its attributes the field in which it is defined, the number n (the order of the forms for which it is defined), the coefficient c of the monomial and the list of exponents $\{v_i\}$ of the monomial. Both the field and the number n are common attributes of the three classes.

2. Invariant: The class "Invariant" represents a polynomial in the coefficients of a form. So this could be expressed as a sum of "Terms", which is exactly how it is implemented.

   Apart from the field and the order n of the base form, the Invariant class has as its only attribute a list of terms (each of the monomials of the polynomial).

3. Covariant: The class "Covariant" represents a polynomial in the coefficients of a form $\{a_i\}$ as well as in the variables x_1, x_2 which is homogeneous of order m in the x_i. As we are representing an homogeneous polynomial in the x_i, this could be expressed as: $C_0x_1^m + C_1x_1^{m-1}x_2 + ... + C_mx_2^m$, where each of the C_i is a polynomial in the $\{a_i\}$, or what is the same, an object of the class "Invariant".

   So the only attributes we have for the class "Covariant" are, apart from the field and order n of the base form, the order m of the polynomial and a list c of "Invariant" which represents the coefficients C_i.

The implementation is made this way to make it easier to work with invariants and covariants of binary forms. This way, invariants of a binary form are just an object of the class "Invariant" that satisfies the invariant property and covariants are the same for the class "Covariant". Moreover, the structure of the class "Covariant" is pretty useful for working with covariants, as many times we want to work with a covariant represented as the form $C_0x_1^m + ... + C_mx_2^m$, or just need the source of the covariant.

Now that we have explained about the structure of the classes, let us talk about what is implemented with them (the functions of the classes).

Aside from the basic functions of each class and some basic operational functions (such as multiplication and addition when it makes sense), we have programmed some functions to help us work later with the invariants and covariants based on the theory explained in the first section of the thesis. Now we list them and explain briefly its purpose:

1. operatorD: Implemented in both the “Term” and “Invariant” classes. As the name suggests, this applies the operator **D** to the object, returning the Invariant resultant of this operation.

2. operatorDelta: Same as the operatorD function, it applies the operator Δ to the object and returns the Invariant resultant.

3. isInvariant: Implemented in the “Invariant” class. It tells if the object satisfies or not the invariant property (if the object, which is a polynomial in the coefficients of a form, is really an invariant). This function checks the conditions listed in 1.10 to verify the invariant property.

4. covariantFromSource: Of the class “Invariant”. Applied to an “Invariant” x, this returns the covariant which has for source the Invariant x.

5. transvectant: Implemented for the class “Covariant”. This takes another object of the class “Covariant” x and a number p as parameters, and returns the p-th transvectant between the object and the x

2.3.2 Base of invariants for n = 6 and n = 8

We want to be able to compute the values of a certain basis of invariants for a given binary form. As we commented earlier, this is useful for various purposes, as the value of the invariants of a basis determine the values of all the invariants of the form, and having a basis allows us to know if two forms are isomorphic.

Here we explain how we dealt with the problem for invariants of binary forms of orders 6 and 8.

There are some reasons why we are only solving the problem for these cases. First of all, as we will see later, the invariants and covariants we can get at computing the basis can become really complex expressions, even to be handled by a computer, and if we get any farther away than order 8 the problem gets really hard (just to name some data, a basis of invariants for a binary form of order 10 is formed by 106 invariants), and there is not much work about it.

Although we will be only solving the problem for the cases of binary forms of orders 6 and 8, this is still interesting, as this corresponds to hyperelliptic curves of genus 2 and 3, and even the work for order 8 is not present in Sage (as in most softwares).

Now let us talk about the problem itself. We have two different approaches available to compute a basis of invariants of a form, explained briefly in previous sections: computing invariants as functions of the roots of the form and computing covariants as transvectants.

The great advantage of the first option (taking the invariants as a function of the roots) is the simplicity of the expressions for the invariants. Just to show an example, here are the expression of the Igusa invariants for a form of order 6:

$$\begin{aligned}
I_2 &= a_0^2 \sum (12)^2(34)^2(56)^2 \\
I_4 &= a_0^4 \sum (12)^2(23)^2(31)^2(45)^2(56)^2(64)^2 \\
I_6 &= a_0^6 \sum (12)^2(23)^2(31)^2(45)^2(56)^2(64)^2(14)^2(25)^2(36)^2 \\
I_{10} &= a_0^{10} \prod (ij)^2
\end{aligned} \tag{19}$$

Where here (ij) represents $(\omega_i - \omega_j)$ the difference between the two roots of the form, and the sum extends for all the permutations of the roots that give different expressions.

As we can see, these expressions are relatively simple, and much more simpler than the expressions of the same invariants represented in the classical way (as polynomials in the coefficients of the form). As these expressions are really simple it is feasible to check for ourselves that they satisfy the properties in 1.30 (and therefore are invariants), and even to find the expressions themselves for each invariant.

So to compute the basis of invariants of a certain binary form by this method we only have to compute the roots of the form and substitute in the computed expressions, which is straightforward. The problem here, and the real drawback of this method, is computing the roots of the form. Because we are dealing with roots of polynomials of order greater than four we have no other choice but to compute the roots numerically, which would produce some precision errors in our solutions. This could be no big problem if we just wanted to know the value of the invariants of a form or for certain other things if we gave a relative level of tolerance for the precision errors, but depending on our purpose, little errors at computing the roots could become real issues. This problem won't appear in the other method.

The classical representation of invariants give no precision issues, as we are working with polynomials with rational coefficients over the coefficients of the form themselves. Although the method of computing the invariants/covariants of the forms via transvectants has its own inconveniences (which we can deal with), this is the main reason we will use this method instead of the previous one just explained.

The main problem with this is that these expressions of the invariants/covariants get very complicated very quickly (just to give an example, if we were to write the expression of the invariant I_{10} mentioned above in this thesis it would occupy three pages). This does not allow us to introduce manually the expressions for the invariants, and forces us to compute them from scratch (by using transvectants).

As the operation of transvection is really expensive (computationally speaking in terms of time), what we will do is pre-compute all the relevant invariants using transvectants, and store their information in a file in a way we can recover them easily later. Thus, after doing this pre-calculation we are able later to compute the basis of invariants of a form quickly.

Previously on 1.29 we stated that the set of covariants (and therefore invariants) of a binary form was generated by the operation of transvection over the binary form itself. This tells us that if we continue to apply the operation of transvection to the form f and iteratively to the covariants of that form, we will end up finding all the covariants of the form (covariants of a basis), and identically all the invariants. But this does not tell us neither when to stop (when do we have all the invariants/covariants we need) nor which transvections gives us the invariants of the basis.

Thanks to the study on section 1.5 we can find the number of invariants that generate the algebra of invariants of order n, as well as the degree of those invariants (as well as the number of relations or syzygys that exists between them, but we won't be interested in those). Said in other words, we can know which invariants form a basis of invariants for certain order.

For example, if we work with the expression in 1.24 for order 6, we obtain that the number of invariants of degree g for a binary form of order 6 is given by the formula:

$$w_6(g) = \left[\frac{(1-x^{30})}{(1-x^2)(1-x^4)(1-x^6)(1-x^{10})(1-x^{15})} \right]_{x^g} \tag{20}$$

This result tells us that every invariant of a binary sextic can be expressed as a polynomial expression of five invariants, of degrees 2, 4, 6, 10 and 15 respectively, and that between these invariants exists one algebraic relation (or syzygy) of order 30.

So with this we know that the basis of invariants for a binary form of order 6 consists on five invariants of degrees 2, 4, 6, 10 and 15. A similar analysis shows us that for order 8, the basis of invariants is formed by nine invariants of degrees 2, 3, 4, 5, 6, 7, 8, 9, and 10.

Now we would want to know how to obtain this basis of invariants by doing transvectants. Thankfully there is already work done about this. In the appendix B we have included two tables that show how to get all the generating invariants and covariants for the algebras of orders 6 and 8 respectively. These tables tell us which transvectants we have to do in order to get each of the invariants and covariants necessaries for the basis. With this information, obtaining the required invariants for the basis is a straightforward process (if not a quick one).

In the appendixes C and D we have included the code of the two scripts we programmed, following the operations described in the tables that we have mentioned, to get all the invariants for forms of both orders 6 and 8. We have not computed all the covariants described in the tables, and that is because, even thought those are all the covariants that form the basis of covariants for each order, we only need to compute the covariants necessary for computing the invariants (what we are really after), and for that we already take a lot of time to do. The approximate time of execution of the scripts annexed was between 30 minutes and 90 minutes each.

The computed invariants are stored in the files “invariants_6.sage” and “invariants_8.sage”, in a format that will recover the invariants just by loading the file (*load*(“*invariants_6.sage*”)). We have not included the content of these two files in an appendix because of the their size, and because their exact content does not brings anything to the thesis. With this information we can easily obtain the invariant information for both forms of orders 6 and 8.

The most direct application of the computed invariants is to find out if two given binary forms are or not isomorphic to each other (if there exists a linear transformation that converts one to the other). This, as we have shown earlier, can be done just with the information of the invariants of a basis, so we have all we need to solve this problem.

In the main class of our code, “binary_form”, we have added a few methods to deal with this problem:

1. invariant_base(self): this method returns a list of objects of the class “Invariant”, which corresponds to a basis of invariants of the form.

2. eval_base(self): this method returns a list with the values that take the invariants of the basis in the coefficients of the binary form.

3. is_isomorphic(self,other): this method returns True if the two binary forms (“self” and “other”) are isomorphic.

These three functions are implemented to work for binary forms of orders 2, 3, 4, 6 and 8. For orders 6 and 8, we obtain the bases of invariants from the files we specified earlier. For orders 2 to 4, the respective basis of invariants of the forms are relatively simple, and we are able to include them directly in the code in just a few lines.

To find out if two forms are isomorphic we have to compare the invariants of the basis for the two forms. It is not so simple as checking if all the invariants are the same, because if the transformation between the forms is of determinant different from one, the value of the invariants is multiplied by the determinant to the power of the weight of the invariant.

In our code, we perform this check in the following way. First we check which invariants cancel for each form, and if one differs from the other, the forms are not isomorphic. After this, for each invariant that does not cancel, we compute the quotient between its value on each form powered to $(1/p)$ where p is the weight of the invariant. If this number is the same for all the invariants that does not cancel, then the two forms are equivalent, and moreover, this number is equal to the determinant of the linear transformation between them.

Example 2.4. Here we show some examples that use these functions:

```
sage: a = binary_form(QQ,2,[1,1,1]); a
x^2 + x*y + y^2
sage: b = binary_form(QQ,3,[1,1,1,1]); b
x^3 + x^2*y + x*y^2 + y^3
sage: c = binary_form(QQ,4,[1,1,1,1,1]); c
x^4 + x^3*y + x^2*y^2 + x*y^3 + y^4
sage: f = binary_form(QQ,6,[1,1,1,1,1,1,1]); f
x^6 + x^5*y + x^4*y^2 + x^3*y^3 + x^2*y^4 + x*y^5 + y^6
sage: a.invariant_base()
[a0*a2 - a1^2]
sage: b.invariant_base()
[a0^2*a3^2 - 6*a0*a1*a2*a3 + 4*a0*a2^3 + 4*a1^3*a3 - 3*a1^2*a2^2]
sage: c.invariant_base()
[a0*a4 - 4*a1*a3 + 3*a2^2, a0*a2*a4 - a0*a3^2 - a1^2*a4 + 2*a1*a2*a3 -
a2^3]
sage: f.invariant_base()[0]
a0*a6 - 6*a1*a5 + 15*a2*a4 - 10*a3^2
sage: a.eval_base()
[3/4]
sage: b.eval_base()
[16/27]
sage: c.eval_base()
[5/6, 25/432]
sage: f.eval_base()
[7/8, 1127/18000, 2401/4320000, 3411821/233280000000, 0]
sage: g = f.transform([[1,3],[2,7]]); g
127*x^6 + 2607*x^5*y + 22311*x^4*y^2 + 101891*x^3*y^3 + 261881*x^2*y^4
+ 359161*x*y^5 + 205339*y^6
sage: f.is_isomorphic(g)
True
```

```
sage: h = binary_form(QQ,6,[1,1,1,1,1,1,0]); h
x^6 + x^5*y + x^4*y^2 + x^3*y^3 + x^2*y^4 + x*y^5
sage: f.is_isomorphic(h)
False
```

2.3.3 Finding an isomorphism between two forms

After the work in the previous section we are able to tell if two given binary forms are or not isomorphic to each other (just for binary forms of orders 2, 3, 4, 6 or 8), or what is the same, we can tell if there exists a linear transformation between the two forms. The next logical step for us to make from here is, given two forms that we know are isomorphic, find an isomorphism between them. In this section we will explain how we have confronted this problem.

There are various algorithms that we could use to compute the isomorphism between the two forms. In [4] there are described two methods to do this, the first one works directly with the coefficients of the isomorphisms while the second one uses an interesting approach to compute the isomorphism by using covariants.

We will implement yet another method, which is based on the roots of the binary forms.

The idea behind the algorithm that we will use is that the linear transformation that we apply to a binary form maps the roots from the first form to the roots of the second (as seen in the section 1.7). We will take advantage of this fact to find an easy way to compute an isomorphism between the two forms. Now we will explain a bit how it works.

In the section 1.7, where we studied the invariants as functions of the roots, we saw how linear transformation affected the roots of a binary forms when applied.

Let $f = a_0(x_1 - \omega_1 x_2)(x_1 - \omega_2 x_2)...(x_1 - \omega_n x_2)$ be a binary form of order n with roots $\{\omega_i\}$. Let $M = \left(\begin{smallmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \end{smallmatrix}\right)$ be a linear transformation and let $g = f|_M = a'_0(x'_1 - \omega'_1 x'_2)(x'_1 - \omega'_2 x'_2)...(x'_1 - \omega'_n x'_2)$ be the transformed form. Then, the work we did showed that:

$$\begin{aligned} g = f|_M &= a'_0 \left(x'_1 - \frac{(-\alpha_{12} + \alpha_{22}\omega_1)}{(\alpha_{11} - \alpha_{21}\omega_1)} x'_2 \right) \dots \left(x'_1 - \frac{(-\alpha_{12} + \alpha_{22}\omega_n)}{(\alpha_{11} - \alpha_{21}\omega_n)} x'_2 \right) \\ &= a'_0(x'_1 - \omega'_1 x'_2)(x'_1 - \omega'_2 x'_2)...(x'_1 - \omega'_n x'_2) \end{aligned}$$

So, this shows that for a permutation σ of the roots, one has the relation:

$$\omega'_{\sigma_i} = \frac{(-\alpha_{12} + \alpha_{22}\omega_i)}{(\alpha_{11} - \alpha_{21}\omega_i)} \tag{21}$$

So a linear transformation between the two forms f and g would be a transformation $M = \left(\begin{smallmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \end{smallmatrix}\right)$ that satisfies:

$$\alpha_{12} - \alpha_{22}\omega_i + \alpha_{11}\omega'_{\sigma_i} - \alpha_{21}\omega_i\omega'_{\sigma_i} = 0$$

for all roots of f and for some permutation σ of the roots.

With this result, the algorithm will be straightforward. If we were given the permutation of the roots σ what we would end up with would be a system of linear equations in the α_{ij}, for which a solution would give us an isomorphism between the two binary forms.

Because we know that the two binary forms are isomorphic (which we can check with the code developed in the previous section), we know that there exists a permutation of the roots for which the system has a solution. We could iterate for every permutation of the roots until we find a solution. We are only treating binary forms of small order (the ones for which we can compute a basis of invariants), so this is feasible. But we will note the following things that will allow us to speed up the algorithm: First, that we can always force one of the coefficients to be 1. And second, that with this condition, any three roots determine uniquely the transformation. Effectively, with three roots we would have a system of three equations and three variables, uniquely determined.

So with these things in mind we will develop the following algorithm:

We will iterate for each set of three roots of the second form, which will be our pick for the mapping of the three first roots of the first form. With these three pairs of roots, we will solve the system of equations and obtain an isomorphism that maps the three roots of the first form to the three roots of the second. With this isomorphism computed, we check for the rest of the roots if there is a permutation for which this isomorphism maps the remaining roots of the first form to the remaining in the second. If this is true, then this is an isomorphism between the two binary forms and we are done.

We have implemented this idea in the file "isomorphism.sage", here included in the appendix E. We now give a few remarks about some technical aspects of the implementation.

1. For finding the roots of binary forms we have used the method ".roots()" of the class "Polynomial-Ring". In order to use this, we had to obtain the object of this class equivalent to the deshomogenized form, for which we implemented a short auxiliary function "poly_from_form". We will compute the roots over the complex field with 100 digits precision, to have all the roots of the form. As we computed the roots as numerical approximation of complex numbers, we will have some precision issues in our solution.

2. For iterating through the permutations of the roots we used the function "permutations" from the package "itertools", which ended up being really convenient to make the code cleaner and shorter.

Having said these things, let us talk about the code itself. The code for this problem works by two main functions, namely:

1. find_iso_root: takes as parameters two lists of complex numbers of the same length (a list of roots of two binary forms), and returns an isomorphism that maps the roots of the first list to the roots on the second. If this isomorphism does not exists it returns -1. In this function we have implemented the algorithm described above to find the isomorphism between roots.

2. find_iso_form: takes as parameters two binary forms and returns an isomorphism between the two binary forms. As in the previous one, if this isomorphism does not exists, it returns -1. It calls internally the function "find_iso_root" with the roots of the two binary forms, but one has to notice that the returned isomorphism might not be the same for these two functions. This might occur because a rescaled of the solution might be needed. This is because an scalar multiplication of a form does not change its roots, so an isomorphism that maps the roots of our first form to the roots of the second might not transform the first binary form into the second, but into an scalar multiple of it. Notice in 15 the coefficients a_0 and a'_0 multiplying the product of the roots, which point exactly to this issue. This is fixed by in our implementation by applying the transformation to the first form,

finding the ratio between the resultant form and the objective form, and multiplying the isomorphism by the n-th root of this ratio.

It is worth a mention that, even though we have included the case in which the isomorphism does not exists in the code, we would normally call this function only when we know that the isomorphism does in fact exists, as we have an easy way to find this out (by using the invariants computed in the previous section).

Also, the code is implemented to work for binary forms of arbitrary order *n*, but we would have to have some things into account. First, we have to notice that the algorithm above only works for forms of order greater or equal to 3, as the first step of the algorithm is to pick three of the roots of the form. The second thing we have to have in mind is that to check that the isomorphism really works we are iterating through the permutations of the roots, which leads to exponential complexity, which will make it difficult to process binary forms of big orders. This should not be a problem though, as we would be interested in computing isomorphism between forms that we know are isomorphic, and we can only perform this check to forms up to order 8, for which the code is able to find an isomorphism in a decent amount of time.

Example 2.5. Now we can show some examples on the use of these functions:

```
sage: load("binary_form.sage")
sage: load("isomorphism.sage")
sage: f1 = binary_form(QQ,3,[1,1,1,1]); f1
x^3 + x^2*y + x*y^2 + y^3
sage: g1 = f1.transform([[1,-2],[8,5]]); g1
585*x^3 + 879*x^2*y + 489*x*y^2 + 87*y^3
sage: f1.is_isomorphic(g1)
True
sage: m = find_iso_form(f1,g1); m
[[1, -2], [8, 5]]
sage: f2 = binary_form(QQ,6,[1,1,1,1,1,1,1]); f2
x^6 + x^5*y + x^4*y^2 + x^3*y^3 + x^2*y^4 + x*y^5 + y^6
sage: g2 = f2.transform([[1,1],[0,1]]); g2
x^6 + 7*x^5*y + 21*x^4*y^2 + 35*x^3*y^3 + 35*x^2*y^4 + 21*x*y^5 + 7*y^6
sage: f2.is_isomorphic(g2)
True
sage: m = find_iso_form(f2,g2); m
[[1, 1], [0, 1]]
sage: h2 = f2.transform([[1,-2],[8,5]]); h2
299593*x^6 + 1011129*x^5*y + 1446459*x^4*y^2 + 1113581*x^3*y^3 + 486339
*x^2*y^4 + 113757*x*y^5 + 11179*y^6
sage: f2.is_isomorphic(h2)
True
sage: m = find_iso_form(f2,h2); m
[[-1, 2], [-8, -5]]
sage: f2.transform(m)
299593*x^6 + 1011129*x^5*y + 1446459*x^4*y^2 + 1113581*x^3*y^3 + 486339
*x^2*y^4 + 113757*x*y^5 + 11179*y^6
```

References

[1] J. Grace, A. Young. 'The Algebra of Invariants', Chelsea Publishing Company, New York, 1903.

[2] D. Hilbert. 'Theory of algebraic invariants', Cambridge University Press, 1993.

[3] R. Lercier, C. Ritzenthaler. 'Hyperelliptic curves and their invariants: Geometric, arithmetic and algorithmic aspects', Journal of Algebra 372 (2012) 595636.

[4] R. Lercier, C. Ritzenthaler, J. Sijsling. 'Fast computation of isomorphisms of hyperelliptic curves and explicit galois descent', https://arxiv.org/pdf/1203.5440.pdf

[5] D. Mumford, J. Fogarty. 'Geometric Invariant Theory', second ed., Ergeb. Math. Grenzgeb., vol. 34, Springer-Verlag, Berlin, 1982.

[6] M. Olive. 'About Gordan's algorithm for binary forms', https://arxiv.org/pdf/1403.2283.pdf

[7] P. J. Olver. 'Classical invariant theory', Cambridge University Press, 1999.

A. binary_form.sage

```
from sage.rings.ring import Field
from sage.structure.unique_representation import UniqueRepresentation
from sage.structure.element import FieldElement

class binary_form(SageObject):
  def __init__(self, field, n, a):
    """
    We show some examples of initialization of binary forms.

    EXAMPLES::

      sage: f = binary_form(QQ,2,[1,1,1]); f
      x^2 + x*y + y^2
      sage: g = binary_form(RR,4,[1.0,1.1,1.2,1.3,1.4]); g
      x^4 + 1.10000000000000*x^3*y + 1.20000000000000*x^2*y^2 + 1.300000
      00000000*x*y^3 + 1.40000000000000*y^4
    """
    if (n not in ZZ) or n < 1:
      raise TypeError("n must be a positive integer")
    self._n = n
    if field not in Fields:
      raise TypeError("first argument must be a field")
    self._base = field
    if isinstance(a, list):
      if len(a) != n+1:
        raise TypeError("Must provide n+1 coefficients for a form of \
            order n")
      else:
        self._a = a
        for i in range(0,len(a)):
          if self._a[i] not in field:
            raise TypeError("Coefficients must be elements of \
                input field")
          else:
            self._a[i] = self._a[i]/binomial(self._n, i)
    else:
      raise TypeError("Coefficients of the binary form must be provided\
          in a list")
  def _repr_(self):
    """
    Return string representation of the binary form.

    EXAMPLES::
      sage: f = binary_form(QQ,3,[1,2,5,7])
      sage: f.__repr__()
      'x^3 + 2*x^2*y + 5*x*y^2 + 7*y^3'
    """
    return repr(self(self._base['x','y'].gens()))
  def __call__(self,args):
    """
    Return the result of replacing the variables of the binary form by
    the values of ''args''

    INPUT:

    - ''args'' -- A list of two elements of the base field of the binary
    form

    EXAMPLES::

      sage: R.<x,y> = QQ[]
      sage: f = binary_form(QQ,3,[1,2,5,7])
      sage: f.__call__([x,y])
      x^3 + 2*x^2*y + 5*x*y^2 + 7*y^3
      sage: f([x,y])
      x^3 + 2*x^2*y + 5*x*y^2 + 7*y^3
      sage: f([1,x])
      7*x^3 + 5*x^2 + 2*x + 1
      sage: f([1,2])
      81
    """
    if len(args) != 2:
      raise TypeError("You need two variables")
    x, y = args
    pol = 0;
    for i in range(0,self._n+1):
      pol += binomial(self._n,i)*self._a[i]*(y**i)*(x**(self._n-i))
    return pol
  def __cmp__(self,other):
    """
    Returns true if ''self'' and ''other'' are distinc.

    EXAMPLES::

      sage: a = binary_form(QQ,2,[1,1,1]);
      sage: b = binary_form(QQ,2,[1,1,1]);
      sage: c = binary_form(QQ,2,[3,1,1]);
      sage: d = binary_form(QQ,3,[1,1,1,1]);
      sage: e = binary_form(RR,2,[1,1,1]);
      sage: a.__cmp__(b)
      False
      sage: a.__cmp__(c)
      True
      sage: a.__cmp__(d)
      True
      sage: a.__cmp__(e)
      True
      sage: a == b
      True
      sage: a == c
      False
      sage: a == d
      False
      sage: a == e
      False
    """
    if not isinstance(other,binary_form):
      return True
    if self._n != other._n:
      return True
    if self._base != other._base:
      return True
    return self._a != other._a
  def __mul__(self,other):
    """
    Return string representation of the binary form.

    EXAMPLES::

      sage: f = binary_form(QQ,2,[1,1,1]); f
      x^2 + x*y + y^2
      sage: g = binary_form(QQ,2,[1,2,3]); g
      x^2 + 2*x*y + 3*y^2
      sage: f * g
      x^4 + 3*x^3*y + 6*x^2*y^2 + 5*x*y^3 + 3*y^4
    """
    if not isinstance(other,binary_form):
      raise TypeError("Must operate two binary forms")
    if self._base != other._base:
      raise TypeError("Forms must be of defined over the same field")
    B = self._base
    m = self._n+other._n
    b = [0]*(m+1)
    for i in range(0,self._n+1):
```

```
    for j in range(0,other._n+1):
      b[i+j] += (self._a[i]*other._a[j]*binomial(self._n,i)*
           binomial(other._n,j))
  return binary_form(B,m,b)
def __add__(self,other):
  """
  Return the sum of the two binary forms.

  EXAMPLES::

    sage: f = binary_form(QQ,2,[1,1,1]); f
    x^2 + x*y + y^2
    sage: g = binary_form(QQ,2,[1,2,3]); g
    x^2 + 2*x*y + 3*y^2
    sage: f + g
    2*x^2 + 3*x*y + 4*y^2
  """
  if not isinstance(other,binary_form):
    raise TypeError("Must operate two binary forms")
  if self._base != other._base:
    raise TypeError("Forms must be of defined over the same field")
  B = self._base
  if self._n != other._n:
    raise TypeError("You can only add forms of the same order")
  b = [0]*(self._n+1)
  for i in range(0,self._n+1):
    b[i] = binomial(self._n,i)*(self._a[i]+other._a[i])
  return binary_form(B,self._n,b)
def eval(self,other):
  """
  Returns the evaluation of the expression 'other' in the coefficients
  of the form

  INPUT:

  - ''other'' -- an instance of Term, Invariant or Covariant of the
  same type of binary form as 'self'

  EXAMPLES::

    sage: f = binary_form(QQ,2,[1,1,1]); f
    x^2 + x*y + y^2
    sage: I = Invariant(QQ,2,[[1,1,0,1],[-1,0,2,0]]); I
    a0*a2 - a1^2
    sage: f.eval(I)
    3/4
  """
  if (isinstance(other,Term) or isinstance(other,Invariant) or
    isinstance(other,Covariant)):
    if other._n != self._n:
      raise TypeError("expression must be defined on a form of the \
            same order")
    else:
      return other.eval(self)
  else:
    raise TypeError("No valid expression to evaluate")
def selfcovariant(self):
  """
  Returns the object ''Covariant'' equal to the form itself.

  EXAMPLES::

    sage: f = binary_form(QQ,2,[1,1,1]); f
    x^2 + x*y + y^2
    sage: C = f.selfcovariant(); C
    a0*x^2 + 2*a1*x*y + a2*y^2
  """
  return Invariant(self._base,self._n,[[1,1]+[0]*self._n]
                    ).covariantFromSource(self._n)
def transform(self,m):
  """
  Returns the binary form resultant of applying the linear
  transformation ''m'' to the form

  INPUT:

  - ''m'' -- a 2x2 matrix, a linear transformation of the variables of
  the form

  EXAMPLES::

    sage: f = binary_form(QQ,3,[1,1,1,1]); f
    x^3 + x^2*y + x*y^2 + y^3
    sage: g = f.transform([[1,1],[0,1]]); g
    x^3 + 4*x^2*y + 6*x*y^2 + 4*y^3
  """
  if (not isinstance(m,list)) or (len(m) != 2):
      raise TypeError("Not a matrix")
  b = [0]*(self._n+1)
  for i in range(0,self._n+1):
    for j in range(0,i+1):
      j2 = i-j
```

```
        for k in range(0,self._n-i+1):
          k2 = self._n-i-k
          b[j2+k2] += (self._a[i]*binomial(self._n,i)*binomial(i,j)*
                  binomial(self._n-i,k)*(m[0][0]**k)*(m[0][1]**k2)*
                  (m[1][0]**j)*(m[1][1]**j2))
    return binary_form(self._base,self._n,b)
  def invariant_base(self):
    """
    Returns a basis of invariants for the binary form, if the binary form
    is of order n = 2,3,4,6 or 8

    EXAMPLES::

      sage: a = binary_form(QQ,2,[1,1,1]); a
      x^2 + x*y + y^2
      sage: b = binary_form(QQ,3,[1,1,1,1]); b
      x^3 + x^2*y + x*y^2 + y^3
      sage: c = binary_form(QQ,4,[1,1,1,1,1]); c
      x^4 + x^3*y + x^2*y^2 + x*y^3 + y^4
      sage: f = binary_form(QQ,6,[1,1,1,1,1,1,1]); f
      x^6 + x^5*y + x^4*y^2 + x^3*y^3 + x^2*y^4 + x*y^5 + y^6
      sage: a.invariant_base()
      [a0*a2 - a1^2]
      sage: b.invariant_base()
      [a0^2*a3^2 - 6*a0*a1*a2*a3 + 4*a0*a2^3 + 4*a1^3*a3 - 3*a1^2*a2^2]
      sage: c.invariant_base()
      [a0*a4 - 4*a1*a3 + 3*a2^2, a0*a2*a4 - a0*a3^2 - a1^2*a4 + 2*a1*a2*
      a3 - a2^3]
      sage: bb = f.invariant_base(); bb[0]
      a0*a6 - 6*a1*a5 + 15*a2*a4 - 10*a3^2
    """
    if self._n == 2:
      return [Invariant(self._base,2,[[1,1,0,1],[-1,0,2,0]])]
    elif self._n == 3:
      return [Invariant(QQ,3,[[1,2,0,0,2],[-6,1,1,1,1],[4,1,0,3,0],[-3,0
              ,2,2,0],[4,0,3,0,1]])]
    elif self._n == 4:
      return [Invariant(QQ,4,[[1,1,0,0,0,1],[-4,0,1,0,1,0],[3,0,0,2,0,0]
              ]),Invariant(QQ,4,[[1,1,0,1,0,1],[-1,1,0,0,2,0],[-1,0,2,
              0,0,1],[-1,0,0,3,0,0],[2,0,1,1,1,0]])]
    elif self._n == 6:
      load("invariants_6.sage")
      return inv
    elif self._n == 8:
      load("invariants_8.sage")
      return inv
    else:
      raise TypeError("Not implemented for binary forms of this order")
  def eval_base(self):
    """
    Returns the evaluation of the basis of invariants of the form in the
    coefficients of it, if the order of the form is n = 2,3,4,6 or 8

    EXAMPLES::

      sage: a = binary_form(QQ,2,[1,1,1]); a
      x^2 + x*y + y^2
      sage: b = binary_form(QQ,3,[1,1,1,1]); b
      x^3 + x^2*y + x*y^2 + y^3
      sage: c = binary_form(QQ,4,[1,1,1,1,1]); c
      x^4 + x^3*y + x^2*y^2 + x*y^3 + y^4
      sage: f = binary_form(QQ,6,[1,1,1,1,1,1,1]); f
      x^6 + x^5*y + x^4*y^2 + x^3*y^3 + x^2*y^4 + x*y^5 + y^6
      sage: a.eval_base()
      [3/4]
      sage: b.eval_base()
      [16/27]
      sage: c.eval_base()
      [5/6, 25/432]
      sage: f.eval_base()
      [7/8, 1127/18000, 2401/4320000, 3411821/233280000000, 0]
    """
    if (self._n >= 2 and self._n < 5) or self._n == 6 or self._n == 8:
      inv_base = self.invariant_base()
      sol = []
      for i in inv_base:
        sol.append(self.eval(i))
      return sol
    else:
      raise TypeError("Not implemented for binary forms of this order")
  def is_isomorphic(self,other):
    """
    Returns true if the two binary forms are isomorphic, if the order of
    the binary form is n = 2,3,4,6 or 8

    INPUT:

    - ''other'' -- a binary form

    EXAMPLES::

      sage: f = binary_form(QQ,6,[1,1,1,1,1,1,1]); f
      x^6 + x^5*y + x^4*y^2 + x^3*y^3 + x^2*y^4 + x*y^5 + y^6
      sage: g = f.transform([[2,3],[1,7]]); g
      127*x^6 + 1803*x^5*y + 12231*x^4*y^2 + 52039*x^3*y^3 + 146801*x^2*
      y^4 + 254669*x*y^5 + 205339*y^6
      sage: h = g.transform([[1,1],[0,1]]); h
      127*x^6 + 2565*x^5*y + 23151*x^4*y^2 + 121533*x^3*y^3 + 396239*x^2
      *y^4 + 763089*x*y^5 + 673009*y^6
      sage: r = binary_form(QQ,6,[1,1,1,1,1,1,0]); r
      x^6 + x^5*y + x^4*y^2 + x^3*y^3 + x^2*y^4 + x*y^5
      sage: f.is_isomorphic(g)
      True
      sage: f.is_isomorphic(h)
      True
      sage: f.is_isomorphic(r)
      False
    """
    if not isinstance(other,binary_form):
      raise TypeError("Comparation must be performed between two binary\
        forms")
    if self._base != other._base or self._n != other._n:
      return False
    if self == other:
      return True
    if (self._n >= 2 and self._n < 5) or self._n == 6 or self._n == 8:
      bb = self.invariant_base()
      v1 = self.eval_base()
      v2 = other.eval_base()
      m = len(bb)
      for i in range(0,m):
        if (v1[i] == 0 and v2[i] != 0) or (v1[i] != 0 and v2[i] == 0):
          return False
      rat = []
      for i in range(0,m):
        if v1[i] != 0:
          rat.append((v2[i]/v1[i])**(1/bb[i]._t[0].weight()))
      for i in range(1,len(rat)):
        if rat[i] != rat[0]:
          return False
      return True
    else:
      raise TypeError("Not implemented for binary forms of this order")

class Term(SageObject):
  def __init__(self,field,n,c,v):
    """
    We show some examples of initialization of Term.
    EXAMPLES::

      sage: a = Term(QQ,3,4,[1,1,0,2]); a
      4*a0*a1*a3^2
      sage: b = Term(RR,2,3.4,[1,0,0]); b
      3.40000000000000*a0
    """
    if field not in Fields:
      raise TypeError("first argument must be a field")
    if c not in field:
      raise TypeError("Coefficient must be in the field of the binary \
          form")
    if (not isinstance(v,list)) or len(v) != n+1:
      raise TypeError("Bad input of coefficients")
    for i in v:
      if i not in ZZ:
        raise TypeError("Exponents must be integers")
    self._base = field
    self._n = n
    self._c = c
    self._v = []
    self._p = self._g = 0
    for i in range(0,len(v)):
      self._v.append(v[i])
      self._g += v[i]
      self._p += i*v[i]
  def _repr_(self):
    """
    Returns the string representation of the Term

    EXAMPLES::

      sage: a = Term(QQ,3,4,[1,1,0,2])
      sage: a.__repr__()
      '4*a0*a1*a3^2'
      sage: b = Term(QQ,3,0,[1,1,0,2])
      sage: b.__repr__()
      '0'
    """
    aux = []
    for i in range(0,self._n+1):
      aux.append('a'+str(i))
    return repr(self(self._base[aux].gens()))
  def __call__(self,args):
    """
```

```
        Return the result of replacing the variables of the Term (the
        'coefficients' a_i) by the values of ''args''

        INPUT:

        - ''args'' -- A list of n+1 elements of the field of the Term

        EXAMPLES::

            sage: a = Term(QQ,3,4,[1,1,0,2])
            sage: R.<a0,a1,a2,a3> = QQ[]
            sage: a.__call__([a0,a1,a2,a3])
            4*a0*a1*a3^2
            sage: a([a0,a1,a2,a3])
            4*a0*a1*a3^2
        """
        if len(args) != self._n+1:
            raise TypeError("Must have n+1 arguments")
        pol = self._c
        for i in range(0,self._n+1):
            pol *= args[i]**self._v[i]
        return pol
    def eval(self,bf):
        """
        Returns the evaluation of the Term in the coefficients of the binary
        form.

        INPUT:

        - ''bf'' -- An instance of binary form defined in the same base
        field and in the same order in which the Term 'self' was defined

        EXAMPLES::
            sage: a = Term(QQ,3,4,[1,1,0,2]); a
            4*a0*a1*a3^2
            sage: f = binary_form(QQ,3,[1,1,1,1]); f
            x^3 + x^2*y + x*y^2 + y^3
            sage: f._a
            [1, 1/3, 1/3, 1]
            sage: a.eval(f)
            4/3
        """
        if bf._base != self._base or bf._n != self._n:
            raise TypeError("Binary form must be or the same base field and \
                    order as the term")
        return self(bf._a)
```

```
def __mul__(self,other):
  """
  Returns the multiplication of the two Terms.

  EXAMPLES::

    sage: a = Term(QQ,3,4,[1,1,0,2]); a
    4*a0*a1*a3^2
    sage: b = Term(QQ,3,2,[1,0,0,1]); b
    2*a0*a3
    sage: a * b
    8*a0^2*a1*a3^3
  """
  if other in self._base:
    return Term(self._base,self._n,other*self._c,self._v)
  if not isinstance(other,Term):
    raise TypeError("Must operate two terms")
  if other._base != self._base or other._n != self._n:
    raise TypeError("No sense multiplying terms of different field \
          base or order")
  c = []
  for i in range(0,self._n+1):
    c.append(self._v[i]+other._v[i])
  return Term(self._base,self._n,self._c*other._c,c)
def degree(self):
  return self._g
def weight(self):
  return self._p
def operatorD(self):
  """
  Returns the result of applying the 'D' operator to the Term.

  EXAMPLES::

    sage: a = Term(QQ,3,4,[1,1,0,2]); a
    4*a0*a1*a3^2
    sage: a.operatorD()
    4*a0^2*a3^2 + 24*a0*a1*a2*a3
  """
  c = Invariant(self._base,self._n,[])
  for i in range(0,self._n):
    if self._v[i+1] == 0:
      continue
    aux = []
    for j in self._v:
      aux.append(j)
```

```
            aux[i+1] -= 1
            aux[i] += 1
            c += Term(self._base, self._n, self._c*(aux[i+1]+1)*(i+1),aux)
        return c
    def operatorDelta(self):
        """
        Returns the result of applying the 'Delta' operator to the Term.

        EXAMPLES::

            sage: a = Term(QQ,3,4,[1,1,0,2]); a
            4*a0*a1*a3^2
            sage: a.operatorDelta()
            8*a0*a2*a3^2 + 12*a1^2*a3^2
        """
        c = Invariant(self._base, self._n, [])
        for i in range(0,self._n):
            if self._v[i] == 0:
                continue
            aux = []
            for j in self._v:
                aux.append(j)
            aux[i+1] += 1
            aux[i] -= 1
            c += Term(self._base, self._n, self._c*(aux[i]+1)*(self._n-i),aux)
        return c

class Invariant(SageObject):
    def __init__(self, field, n, coef):
        """
        We show some examples of initialization of Invariants.

        EXAMPLES::

            sage: z1 = Term(QQ,2,1,[1,0,1]); z1
            a0*a2
            sage: z2 = Term(QQ,2,-1,[0,2,0]); z2
            -a1^2
            sage: I = Invariant(QQ,2,[z1,z2]); I
            a0*a2 - a1^2
            sage: I2 = Invariant(QQ,3,[[1,1,0,0,1],[2,0,1,1,0]]); I2
            a0*a3 + 2*a1*a2
        """
        if field not in Fields:
            raise TypeError("first argument must be a field")
        self._base = field
    if not isinstance(coef, list):
      raise TypeError("Bad input of coefficients")
    self._t = []
    self._n = n
    for i in coef:
      if isinstance(i,Term):
        self._t.append(Term(field,n,i._c,i._v))
      else:
        self._t.append(Term(field,n,i[0],i[1:]))
    for i in self._t:
      if i._c == 0:
        self._t.remove(i)
  def _repr_(self):
    """
    Returns the string representation of the Invariant.

    EXAMPLES::

      sage: I = Invariant(QQ,2,[[1,1,0,1],[-1,0,2,0]])
      sage: I.__repr__()
      'a0*a2 - a1^2'
    """
    aux = []
    for i in range(0,self._n+1):
      aux.append('a'+str(i))
    return repr(self(self._base[aux].gens()))
  def __call__(self,args):
    """
    Return the result of replacing the variables of the Invariant (the
    'coefficients' a_i) by the values of ''args''.

    INPUT:

    - ''args'' -- A list of n+1 elements of the field of the Invariant

    EXAMPLES::

      sage: R.<a0,a1,a2> = QQ[]
      sage: I = Invariant(QQ,2,[[1,1,0,1],[-1,0,2,0]])
      sage: I.__call__([a0,a1,a2])
      -a1^2 + a0*a2
      sage: I([a0,a1,a2])
      -a1^2 + a0*a2
    """
    if len(args) != self._n+1:
      raise TypeError("Must have n+1 arguments")
    pol = 0
    for i in self._t:
      pol += i(args)
    return pol
  def eval(self,bf):
    """
    Returns the evaluation of the invariant in the coefficients of the
    given binary form.

    INPUT:

    - ''bf'' -- An instance of binary form defined in the same base
    field and in the same order in which the Invariant 'self' was
    defined

    EXAMPLES::

      sage: I = Invariant(QQ,2,[[1,1,0,1],[-1,0,2,0]]); I
      a0*a2 - a1^2
      sage: f = binary_form(QQ,2,[1,1,1]); f
      x^2 + x*y + y^2
      sage: f._a
      [1, 1/2, 1]
      sage: I.eval(f)
      3/4
    """
    if bf._base != self._base or bf._n != self._n:
      raise TypeError("Binary form must be or the same base field and \
            order as the invariant")
    return self(bf._a)
  def __add__(self,other):
    """
    Returns the sum of the Invariant and ''other''.

    INPUT:

    - ''other'' -- An instance of Invariant or Term, which have the same
    base field and order as the Invariant 'self'

    EXAMPLES::

      sage: I1 = Invariant(QQ,2,[[1,1,0,1],[-1,0,2,0]]); I1
      a0*a2 - a1^2
      sage: I2 = Invariant(QQ,2,[[-1,1,0,1],[2,2,0,0]]); I2
      2*a0^2 - a0*a2
      sage: T1 = Term(QQ,2,-2,[2,0,0]); T1
      -2*a0^2
      sage: I3 = I1 + I2; I3
      2*a0^2 - a1^2
      sage: I3 + T1
      -a1^2
    """
    if isinstance(other,Term):
      if self._base != other._base or self._n != other._n:
        raise TypeError("Need to be of the same base field and order")
      aux = Invariant(self._base,self._n,self._t)
      for i in range(0,len(aux._t)):
        same = True
        for j in range(0,other._n+1):
          if other._v[j] != aux._t[i]._v[j]:
            same = False
            break
        if same == True:
          aux._t[i]._c += other._c
          if aux._t[i]._c == 0:
            aux._t.remove(aux._t[i])
          return aux
      aux._t.append(Term(other._base,other._n,other._c,other._v))
      return aux
    elif isinstance(other,Invariant):
      if self._base != other._base or self._n != other._n:
        raise TypeError("Need to be of the same base field and order")
      aux = Invariant(self._base,self._n,self._t)
      for i in other._t:
        aux += i
      return aux
    else:
      raise TypeError("Must operate with two Invariants")
  def __mul__(self,other):
    """
    Returns the product of the invariant by ''other''.

    INPUT:

    - ''other'' -- Can be either an instance of Invariant of the same
    field and order, or an element of the base field of the invariant.

    EXAMPLES::

      sage: I1 = Invariant(QQ,2,[[1,1,0,1],[-1,0,2,0]]); I1
      a0*a2 - a1^2
      sage: I2 = Invariant(QQ,2,[[1,1,0,0]]); I2
      a0
      sage: I3 = Invariant(QQ,2,[[3,0,1,0],[1,1,0,1]]); I3
      3*a1 + a0*a2
      sage: I1 * I2
      a0^2*a2 - a0*a1^2
      sage: I1 * I3
      3*a0*a1*a2 - 3*a1^3 + a0^2*a2^2 - a0*a1^2*a2
      sage: I2 * I2 * I3
      3*a0^2*a1 + a0^3*a2
      sage: I2 * (5/3)
      5/3*a0
    """
    if other in self._base:
      if other == 0:
        return Invariant(self._base,self._n,[])
      aux = Invariant(self._base,self._n,self._t)
      for i in range(0,len(self._t)):
        aux._t[i] *= other
      return aux
    if isinstance(other,Invariant):
      if self._base != other._base or self._n != other._n:
        raise TypeError("Need to be of the same base field and order")
      t = []
      for i in self._t:
        for j in other._t:
          t.append(i*j)
      return Invariant(self._base,self._n,t)
    else:
      raise TypeError("Must operate two Invariants")
  def operatorD(self):
    """
    Returns the result of applying the operator 'D' to the Invariant.

    EXAMPLES::

      sage: I1 = Invariant(QQ,2,[[1,1,1,1],[2,2,1,0]]); I1
      2*a0^2*a1 + a0*a1*a2
      sage: I2 = Invariant(QQ,2,[[1,1,0,1],[-1,0,2,0]]); I2
      a0*a2 - a1^2
      sage: I1.operatorD()
      2*a0^3 + a0^2*a2 + 2*a0*a1^2
      sage: I2.operatorD()
      0
    """
    aux = Invariant(self._base,self._n,[])
    for i in self._t:
      aux += i.operatorD()
    return aux
  def operatorDelta(self):
    """
    Returns the result of applying the operator 'Delta' to the Invariant

    EXAMPLES::

      sage: I1 = Invariant(QQ,2,[[1,1,1,1],[2,2,1,0]]); I1
      2*a0^2*a1 + a0*a1*a2
      sage: I2 = Invariant(QQ,2,[[1,1,0,1],[-1,0,2,0]]); I2
      a0*a2 - a1^2
      sage: I1.operatorDelta()
      2*a0^2*a2 + 8*a0*a1^2 + a0*a2^2 + 2*a1^2*a2
      sage: I2.operatorDelta()
      0
    """
    aux = Invariant(self._base,self._n,[])
    for i in self._t:
      aux += i.operatorDelta()
    return aux
  def isZero(self):
    """
    Returns True if the Invariant is equal to zero, False otherwise

    EXAMPLES::

      sage: I1 = Invariant(QQ,2,[[1,1,0,1],[-1,1,0,1]]); I1
      0
      sage: I1.isZero()
      True
      sage: I2 = Invariant(QQ,2,[[1,1,0,1],[-1,0,2,0]]); I2
      a0*a2 - a1^2
      sage: I2.isZero()
      False
    """
    return self.__repr__() == '0'
  def isInvariant(self):
    """
    Returns True if the Invariant satisfies the invariant property.

    EXAMPLES::

      sage: I1 = Invariant(QQ,2,[[1,1,0,1],[-1,0,2,0]]); I1
      a0*a2 - a1^2
      sage: I2 = Invariant(QQ,4,[[1,1,0,0,0,1],[-4,0,1,0,1,0],[3,0,0,2,0
        ....:  ,0]]); I2
        a0*a4 - 4*a1*a3 + 3*a2^2
        sage: I3 = Invariant(QQ,4,[[1,1,0,0,0,1],[4,0,1,0,1,0],[3,0,0,2,0,
        ....:  0]]); I3
        a0*a4 + 4*a1*a3 + 3*a2^2
        sage: I1.isInvariant()
        True
        sage: I2.isInvariant()
        True
        sage: I3.isInvariant()
        False
      """
      if len(self._t) == 0:
        return True
      g = self._t[0].degree()
      p = self._t[0].weight()
      for i in self._t[1:]:
        if i.degree() != g or i.weight() != p:
          return False
      if self._n*g != 2*p:
        return False
      return self.operatorD().isZero()
    def covariantFromSource(self,m):
      """
      Returns the covariant of order m which has for source the Invariant
      ''self''.

      EXAMPLES::

        sage: I1 = Invariant(QQ,4,[[1,1,0,1,0,0]]); I1
        a0*a2
        sage: I1.covariantFromSource(4)
        a0*a2*x^4 + 2*a0*a3*x^3*y + a0*a4*x^2*y^2 + 4*a1*a2*x^3*y + 8*a1*a
        3*x^2*y^2 + 4*a1*a4*x*y^3 + 6*a2^2*x^2*y^2 + 16*a2*a3*x*y^3
        + 7*a2*a4*y^4 + 8*a3^2*y^4
        sage: I2 = Invariant(QQ,4,[[1,1,0,0,0,0]]); I2
        a0
        sage: I2.covariantFromSource(4)
        a0*x^4 + 4*a1*x^3*y + 6*a2*x^2*y^2 + 4*a3*x*y^3 + a4*y^4
      """
      c = [Invariant(self._base,self._n,self._t)]
      for i in range(0,m):
        c.append(c[i].operatorDelta())
        c[i+1] *= ((m-i)^-1)
      return Covariant(self._base,self._n,m,c)
```

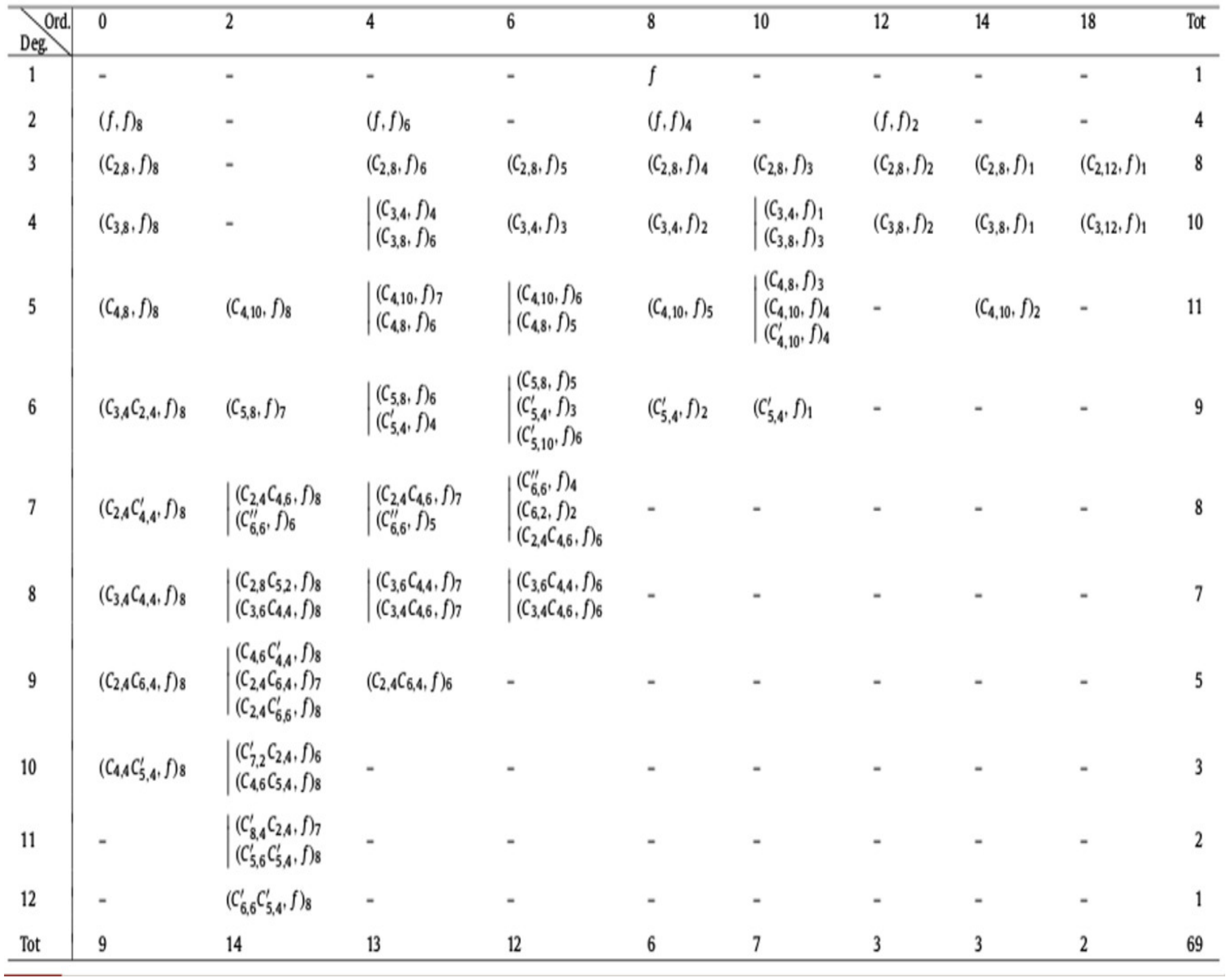
```
class Covariant(SageObject):
  def __init__(self, field ,n,m, coef):
    """
    We show some examples of initialization of Covariants.

    EXAMPLES::

      sage: A = Covariant(QQ,4 ,4 ,[[[1 ,1 ,0 ,1 ,0 ,0] ,[ -1 ,0 ,2 ,0 ,0 ,0]] ,[[2 ,1 ,0
      ....:  ,0 ,1 ,0] ,[2 ,0 ,1 ,1 ,0 ,0]] ,[[1 ,1 ,0 ,0 ,0 ,1] ,[2 ,0 ,1 ,0 ,1 ,0] ,[ -3 ,0 ,0 ,
      ....:  2 ,0 ,0]] ,[[2 ,0 ,1 ,0 ,0 ,1] ,[ -2 ,0 ,0 ,1 ,1 ,0]] ,[[1 ,0 ,0 ,1 ,0 ,1] ,[ -1 ,0 ,
      ....:  0 ,0 ,2 ,0]]]); A
      a0*a2*x^4 + 2*a0*a3*x^3*y + a0*a4*x^2*y^2 - a1^2*x^4 + 2*a1*a2*x^3
      *y + 2*a1*a3*x^2*y^2 + 2*a1*a4*x*y^3 - 3*a2^2*x^2*y^2 - 2*a2*a3*x*
      y^3 + a2*a4*y^4 - a3^2*y^4
      sage: I0 = Invariant(QQ,2 ,[[1 ,1 ,2 ,3] ,[ -1 ,2 ,1 ,3]]); I0
      -a0^2*a1*a2^3 + a0*a1^2*a2^3
      sage: I1 = Invariant(QQ,2 ,[[1 ,2 ,0 ,1] ,[1 ,1 ,2 ,0]]); I1
      a0^2*a2 + a0*a1^2
      sage: I2 = Invariant(QQ,2 ,[[1 ,1 ,1 ,1] ,[1 ,3 ,0 ,0]]); I2
      a0^3 + a0*a1*a2
      sage: B = Covariant(QQ,2 ,2 ,[I0 ,I1 ,I2 ])
      sage: B
      a0^3*y^2 + 2*a0^2*a2*x*y + 2*a0*a1^2*x*y + a0*a1*a2*y^2 - a0^2*a1*
      a2^3*x^2 + a0*a1^2*a2^3*x^2
    """
    if field not in Fields:
      raise TypeError("first argument must be a field")
    if (m not in ZZ) or m < 0:
      raise TypeError("second argument must be a non-negative integer")
    if not isinstance(coef, list):
      raise TypeError("Bad input of coefficients")
    if len(coef) != m+1:
      raise TypeError("Covariant must have m+1 terms")
    self._base = field
    self._n = n
    self._m = m
    self._c = []
    for i in range(0,len(coef)):
      if isinstance(coef[i], list):
        self._c.append(Invariant(self._base, self._n, coef[i]) *
                          (1/binomial(m, i)))
      elif isinstance(coef[i], Invariant):
        self._c.append(Invariant(self._base, self._n, coef[i]._t))
      else:
        raise TypeError("Bad input coefficients")
  def _repr_(self):
    """
    Returns the string representant of the Covariant

    EXAMPLES::

      sage: A = Covariant(QQ,3,2,[[[1,1,0,1,0],[-1,0,2,0,0]],[[1,1,0,0,1
      ....: ],[-1,0,1,1,0]],[[1,0,1,0,1],[-1,0,0,2,0]]])
      sage: A.__repr__()
      'a0*a2*x^2 + a0*a3*x*y - a1^2*x^2 - a1*a2*x*y + a1*a3*y^2 - a2^2*y
      ^2'
    """
    aux = []
    for i in range(0,self._n+1):
      aux.append('a'+str(i))
    aux.append('x')
    aux.append('y')
    aux = self._base[aux].gens()
    return repr(self(aux[:self._n+1],aux[self._n+1:]))
  def __call__(self,args,var):
    """
    Return the result of replacing the variables of the Covariant (the
    'coefficients' a_i and the x_i) by the values in ''args'' and
    ''var'' respectively.

    INPUT:

    - ''args'' -- A list of n+1 elements of the field of the covariant

    - ''var'' -- A list of two elements of the field of the Covariant

    EXAMPLES::

      sage: R.<a0,a1,a2,a3,x,y> = QQ[]
      sage: A = Covariant(QQ,3,2,[[[1,1,0,1,0],[-1,0,2,0,0]],[[1,1,0,0,1
      ....: ],[-1,0,1,1,0]],[[1,0,1,0,1],[-1,0,0,2,0]]])
      sage: A.__call__([a0,a1,a2,a3],[x,y])
      -a1^2*x^2 + a0*a2*x^2 - a1*a2*x*y + a0*a3*x*y - a2^2*y^2 + a1*a3*y
      ^2
      sage: A([a0,a1,a2,a3],[x,y])
      -a1^2*x^2 + a0*a2*x^2 - a1*a2*x*y + a0*a3*x*y - a2^2*y^2 + a1*a3*y
      ^2
      sage: A([1,a1,1,0],[x,y])
      -a1^2*x^2 - a1*x*y + x^2 - y^2
    """
    if len(args) != self._n+1:
      raise TypeError("Must have n+1 arguments")
    if len(var) != 2:
      raise TypeError("Must provide 2 variables")
    pol = 0
    for i in range(0,self._m+1):
      pol += (binomial(self._m,i)*self._c[i](args)*(var[0]**(self._m-i))
            *(var[1]**i))
    return pol
  def eval(self,bf):
    """
    Returns the evaluation of the covariant in the coefficients of the
    binary form.

    INPUT:

    - ''bf'' -- An instance of binary form defined in the same base
    field and in the same order in which the Covariant 'self' was
    defined

    EXAMPLES::

      sage: f = binary_form(QQ,3,[1,1,2,0]); f
      x^3 + x^2*y + 2*x*y^2
      sage: A = Covariant(QQ,3,2,[[[1,1,0,1,0],[-1,0,2,0,0]],[[1,1,0,0,1
      ....: ],[-1,0,1,1,0]],[[1,0,1,0,1],[-1,0,0,2,0]]]); A
      a0*a2*x^2 + a0*a3*x*y - a1^2*x^2 - a1*a2*x*y + a1*a3*y^2 - a2^2*y^
      2
      sage: A.eval(f)
      5/9*x^2 - 2/9*x*y - 4/9*y^2
    """
    if bf._base != self._base or bf._n != self._n:
      raise TypeError("Binary form must be or the same base field and \
            order as the covariant")
    return self(bf._a,bf._base['x','y'].gens())
  def __mul__(self,other):
    """
    Returns the product of the Covariant by other.

    INPUT:

    - ''other'' -- Can be either an instance of a Covariant of the same
    base field and order of ''self'', or an element of the base field of
    the Covariant.

    EXAMPLES::

      sage: A = Covariant(QQ,3,2,[[[1,1,0,1,0],[-1,0,2,0,0]],[[1,1,0,0,1
      ....: ],[-1,0,1,1,0]],[[1,0,1,0,1],[-1,0,0,2,0]]]); A
      a0*a2*x^2 + a0*a3*x*y - a1^2*x^2 - a1*a2*x*y + a1*a3*y^2 - a2^2*y^
      2
      sage: f = binary_form(QQ,3,[1,1,1,1])
      sage: g = f.selfcovariant(); g
      a0*x^3 + 3*a1*x^2*y + 3*a2*x*y^2 + a3*y^3
      sage: A * g
      a0^2*a2*x^5 + a0^2*a3*x^4*y - a0*a1^2*x^5 + 2*a0*a1*a2*x^4*y + 4*a
      0*a1*a3*x^3*y^2 + 2*a0*a2^2*x^3*y^2 + 4*a0*a2*a3*x^2*y^3 + a0*a3^2
      *x*y^4 - 3*a1^3*x^4*y - 6*a1^2*a2*x^3*y^2 +2*a1^2*a3*x^2*y^3 - 6*a
      1*a2^2*x^2*y^3 + 2*a1*a2*a3*x*y^4 + a1*a3^2*y^5 - 3*a2^3*x*y^4 - a
      2^2*a3*y^5
      sage: A * (3/4)
      3/4*a0*a2*x^2 + 3/4*a0*a3*x*y - 3/4*a1^2*x^2 - 3/4*a1*a2*x*y + 3/4
      *a1*a3*y^2 - 3/4*a2^2*y^2
    """
    if other in self._base:
      if other == 0:
        return Covariant(self._base,self._n,self._m,[])
      aux = Covariant(self._base,self._n,self._m,self._c)
      for i in range(0,len(self._c)):
        aux._c[i] *= other
      return aux
    if isinstance(other,Covariant):
      if self._base != other._base or self._n != other._n:
        raise TypeError("Need to be of the same base field and order")
      c = []
      for i in range(0,self._m+other._m+1):
        aux = Invariant(self._base,self._n,[])
      for j in range(0,i+1):
        k = i-j
        if j > self._m or k < 0 or k > other._m:
          continue
        aux += (self._c[j]*other._c[k]*(binomial(self._m,j)*
            binomial(other._m,k)/binomial(self._m+other._m,i)))
      c.append(aux)
      return Covariant(self._base,self._n,self._m+other._m,c)
    else:
      raise TypeError("Must operate two Invariants")
  def transvectant(self,other,p):
  """
    Returns the p-th transvectanct between the two covariants ''self''
    and ''other''

    INPUT:
```

```
- ''other'' -- An instance of Covariant of the same type of base
form (same base field and order of the form) as ''self''.

- ''p'' -- A positive integer satisfying that 2*p > self._m+other._m

EXAMPLES::

   sage: a = binary_form(QQ,6,[1,1,1,1,1,1,1]); a
   x^6 + x^5*y + x^4*y^2 + x^3*y^3 + x^2*y^4 + x*y^5 + y^6
   sage: f = a.selfcovariant(); f
   a0*x^6 + 6*a1*x^5*y + 15*a2*x^4*y^2 + 20*a3*x^3*y^3 + 15*a4*x^2*y^
   4 + 6*a5*x*y^5 + a6*y^6
   sage: I2 = f.transvectant(f,6); I2
   a0*a6 - 6*a1*a5 + 15*a2*a4 - 10*a3^2
"""
if not isinstance(other,Covariant):
   raise TypeError("must operate two covariants")
if self._base != other._base or self._n != other._n:
   raise TypeError("must operate two covariants of the same base \
         field and order")
if p <= 0 or 2*p > self._m+other._m:
   raise TypeError("transvectant p does not exists")
C0 = Invariant(self._base,self._n,[])
for i in range(0,p+1):
   aux = self._c[i]*other._c[p-i]
   aux *= (-1)^i
   aux *= binomial(p,i)
   aux *= 1/2
   C0 += aux
return C0.covariantFromSource(self._m+other._m-2*p)
```

B. Transvectant tables

deg\ord	0	2	4	6	8	10	12
1	-	-	-	f	-	-	-
2	$(f,f)_6$	-	$(f,f)_4$	-	$(f,f)_2$	-	-
3	-	$(C_{2,4},f)_4$	-	$(C_{2,4},f)_2$	$(C_{2,4},f)_1$	-	$(C_{2,8},f)_1$
4	$(C_{2,4},C_{2,4})_4$	-	$(C_{3,2},f)_2$	$(C_{3,2},f)_1$	-	$(C_{2,8},C_{2,4})_1$	-
5	-	$(C_{2,4},C_{3,2})_2$	$(C_{2,4},C_{3,2})_1$	-	$(C_{2,8},C_{3,2})_1$	-	-
6	$(C_{3,2},C_{3,2})_2$	-	-	$(C_{3,8},C_{3,2})_2$ $(C_{3,6},C_{3,2})_1$	-	-	-
7	-	$(f,C_{3,2}^2)_4$	$(f,C_{3,2}^2)_3$	-	-	-	-
8	-	$(C_{2,4},C_{3,2}^2)_3$	-	-	-	-	-
9	-	-	$(C_{3,8},C_{3,2}^2)_4$	-	-	-	-
10	$(C_{3,2}^3,f)_6$	$(C_{3,2}^3,f)_5$	-	-	-	-	-
12	-	$(C_{3,8},C_{3,2}^3)_6$	-	-	-	-	-
15	$(C_{3,8},C_{3,2}^4)_8$	-	-	-	-	-	-

Table 1: **Table for generating a basis of covariants for binary forms of order 6 using transvectants**



Deg. \ Ord.	0	2	4	6	8	10	12	14	18	Tot
1	-	-	-	-	f	-	-	-	-	1
2	$(f,f)_8$	-	$(f,f)_6$	-	$(f,f)_4$	-	$(f,f)_2$	-	-	4
3	$(C_{2,8},f)_8$	-	$(C_{2,8},f)_6$	$(C_{2,8},f)_5$	$(C_{2,8},f)_4$	$(C_{2,8},f)_3$	$(C_{2,8},f)_2$	$(C_{2,8},f)_1$	$(C_{2,12},f)_1$	8
4	$(C_{3,8},f)_8$	-	$(C_{3,4},f)_4$ $(C_{3,8},f)_6$	$(C_{3,4},f)_3$	$(C_{3,4},f)_2$	$(C_{3,4},f)_1$ $(C_{3,8},f)_3$	$(C_{3,8},f)_2$	$(C_{3,8},f)_1$	$(C_{3,12},f)_1$	10
5	$(C_{4,8},f)_8$	$(C_{4,10},f)_8$	$(C_{4,10},f)_7$ $(C_{4,8},f)_6$	$(C_{4,10},f)_6$ $(C_{4,8},f)_5$	$(C_{4,10},f)_5$	$(C_{4,8},f)_3$ $(C_{4,10},f)_4$ $(C'_{4,10},f)_4$	-	$(C_{4,10},f)_2$	-	11
6	$(C_{3,4}C_{2,4},f)_8$	$(C_{5,8},f)_7$	$(C_{5,8},f)_6$ $(C'_{5,4},f)_4$	$(C_{5,8},f)_5$ $(C'_{5,4},f)_3$ $(C'_{5,10},f)_6$	$(C'_{5,4},f)_2$	$(C'_{5,4},f)_1$	-	-	-	9
7	$(C_{2,4}C'_{4,4},f)_8$	$(C_{2,4}C_{4,6},f)_8$ $(C''_{6,6},f)_6$	$(C_{2,4}C_{4,6},f)_7$ $(C''_{6,6},f)_5$	$(C''_{6,6},f)_4$ $(C_{6,2},f)_2$ $(C_{2,4}C_{4,6},f)_6$	-	-	-	-	-	8
8	$(C_{3,4}C_{4,4},f)_8$	$(C_{2,8}C_{5,2},f)_8$ $(C_{3,6}C_{4,4},f)_8$	$(C_{3,6}C_{4,4},f)_7$ $(C_{3,4}C_{4,6},f)_7$	$(C_{3,6}C_{4,4},f)_6$ $(C_{3,4}C_{4,6},f)_6$	-	-	-	-	-	7
9	$(C_{2,4}C_{6,4},f)_8$	$(C_{4,6}C'_{4,4},f)_8$ $(C_{2,4}C_{6,4},f)_7$ $(C_{2,4}C'_{6,6},f)_8$	$(C_{2,4}C_{6,4},f)_6$	-	-	-	-	-	-	5
10	$(C_{4,4}C'_{5,4},f)_8$	$(C'_{7,2}C_{2,4},f)_6$ $(C_{4,6}C_{5,4},f)_8$	-	-	-	-	-	-	-	3
11	-	$(C'_{8,4}C_{2,4},f)_7$ $(C'_{5,6}C'_{5,4},f)_8$	-	-	-	-	-	-	-	2
12	-	$(C'_{6,6}C'_{5,4},f)_8$	-	-	-	-	-	-	-	1
Tot	9	14	13	12	6	7	3	3	2	69

Figure 1: **Table for generating a basis of covariants for binary forms of order 8 using transvectants**

C. base_generator_6.sage

```
load("binary_form.sage")
a = binary_form(QQ,6,[1,1,1,1,1,1,1])

f = a.selfcovariant()

c = {}
c[(2,0)] = f.transvectant(f,6)
c[(2,4)] = f.transvectant(f,4)
c[(3,2)] = c[(2,4)].transvectant(f,4)
c[(3,8)] = c[(2,4)].transvectant(f,1)
c[(4,0)] = c[(2,4)].transvectant(c[(2,4)],4)
c[(6,0)] = c[(3,2)].transvectant(c[(3,2)],2)
c322 = c[(3,2)]*c[(3,2)]
c323 = c322*c[(3,2)]
c[(10,0)] = c323.transvectant(f,6)
c324 = c323*c[(3,2)]
c[(15,0)] = c[(3,8)].transvectant(c324,8)

file1 = open("invariants_6.sage","w")
file1.write("#~ from __future__ import division\n")
bas = [2,4,6,10,15]
for i in bas:
  j = c[(i,0)]
  for k in j._c:
    aux = []
    for l in k._t:
      aux.append([l._c]+l._v)
    file1.write("I"+str(i)+" = Invariant(QQ,6,")
    file1.write(repr(aux))
    file1.write(")\n")
file1.write("inv = [I2,I4,I6,I10,I15]\n")
file1.close()
```

D. base_generator_8.sage

```
load("binary_form.sage")
a = binary_form(QQ,8,[1,1,1,1,1,1,1,1,1])

f = a.selfcovariant()

x = {}
x[(2,0)] = f.transvectant(f,8)
x[(2,4)] = f.transvectant(f,6)
x[(2,8)] = f.transvectant(f,4)
x[(3,0)] = x[(2,8)].transvectant(f,8)
x[(3,4)] = x[(2,8)].transvectant(f,6)
x[(3,8)] = x[(2,8)].transvectant(f,4)
x[(4,0)] = x[(3,8)].transvectant(f,8)
x[(4,41)] = x[(3,4)].transvectant(f,4)
x[(4,42)] = x[(3,8)].transvectant(f,6)
x[(4,8)] = x[(3,4)].transvectant(f,2)
x[(4,10)] = x[(3,4)].transvectant(f,1)
x[(5,0)] = x[(4,8)].transvectant(f,8)
x[(5,42)] = x[(4,8)].transvectant(f,6)
x[(5,8)] = x[(4,10)].transvectant(f,5)
x[(6,0)] = (x[(3,4)]*x[(2,4)]).transvectant(f,8)
x[(6,41)] = x[(5,8)].transvectant(f,6)
x[(7,0)] = (x[(2,4)]*x[(4,42)]).transvectant(f,8)
x[(8,0)] = (x[(3,4)]*x[(4,41)]).transvectant(f,8)
x[(9,0)] = (x[(2,4)]*x[(6,41)]).transvectant(f,8)
x[(10,0)] = (x[(4,41)]*x[(5,42)]).transvectant(f,8)

file1 = open("invariants_8.sage","w")
file1.write("#~ from __future__ import division\n")
for i in range(2,11):
  j = x[(i,0)]
  for k in j._c:
    aux = []
    for l in k._t:
      aux.append([l._c]+l._v)
    file1.write("I"+str(i)+" = Invariant(QQ,8,")
    file1.write(repr(aux))
    file1.write(")\n")
file1.write("inv = [I2,I3,I4,I5,I6,I7,I8,I9,I10]\n")
file1.close()
```

E. isomorphism.sage

```
from itertools import permutations
C100 = ComplexField(100)
R = PolynomialRing(C100,'x')
x = R.gens()[0]

def poly_from_form(f):
  p = 0
  for i in range(0,f._n+1):
    p += C100(f._a[i]*binomial(f._n,i))*x**(f._n-i)
  return p

def find_iso_form(f,g):
  eps = C100(1e-10)
  p1 = poly_from_form(f)
  p2 = poly_from_form(g)
  r1 = p1.roots()
  r2 = p2.roots()
  s1 = []
  s2 = []
  for i in range(0,len(r1)):
    for j in range(0,r1[i][1]):
      s1.append(r1[i][0])
  for i in range(0,len(r2)):
    for j in range(0,r2[i][1]):
      s2.append(r2[i][0])
  m = find_iso_root(s1,s2)
  if m == -1:
    return m
  else:
    for j in range(0,2):
      for k in range(0,2):
        if not (I in f._base):
          m[j][k] = f._base(m[j][k].real())
        else:
          m[j][k] = f._base(m[j][k])
  h = f.transform(m)
  for i in range(0,f._n+1):
    if h._a[i].abs() > eps:
      r = g._a[i]/h._a[i]
      r = r**(1/f._n)
      for j in range(0,2):
        for k in range(0,2):
          m[j][k] *= r
      return m
  return m

def find_iso_root(r1,r2):
  eps = C100(1e-10)
  idx = range(0,len(r2))
  take3 = permutations(idx,3)
  for aux in take3:
    eq = []
    for u in range(0,3):
      eq.append(1-a22*r1[u]+a11*r2[aux[u]]-a21*r1[u]*r2[aux[u]] == 0)
    s = solve(eq,a11,a21,a22)
    for sol in s:
      aux2 = [sol[0].rhs(),1,sol[1].rhs(),sol[2].rhs()]
      bad = False
      for u in range(0,3):
        if (aux2[1] - aux2[3]*r1[u] + aux2[0]*r2[aux[u]] -
            aux2[2]*r1[u]*r2[aux[u]]).abs() > eps:
          bad = True
          break
      if bad:
        break
      rest = []
      for i in range(0,len(r2)):
        if not (i in aux):
          rest.append(i)
      permrest = permutations(rest)
      for aux3 in permrest:
        for u in range(0,len(aux3)):
          if (aux2[1] - aux2[3]*r1[u+3] + aux2[0]*r2[aux3[u]] -
              aux2[2]*r1[u+3]*r2[aux3[u]]).abs() > eps:
            bad = True
            break
        if bad:
          bad = False
          continue
        for i in range(0,4):
          aux2[i] = C100(ComplexField(20)(aux2[i]))
          if aux2[i].abs() < eps:
            aux2[i] = 0
        return [[aux2[0],aux2[1]],[aux2[2],aux2[3]]]
  return -1
```