UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

INSTITUTO DE MATEMÁTICA

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

FILIPE RAMOS FERREIRA

BIBLIOTECA DO AXIOM PARA ENCONTRAR CAMPOS DE RETAS INVARIANTES POR GRUPOS INFINITOS

RIO DE JANEIRO

2018

FILIPE RAMOS FERREIRA

BIBLIOTECA DO AXIOM PARA ENCONTRAR CAMPOS DE RETAS INVARIANTES POR GRUPOS INFINITOS

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Severino Collier Coutinho, D.Sc.

RIO DE JANEIRO

2018

CIP - Catalogação na Publicação

Ferreira, Filipe Ramos
F383b Biblioteca do Axiom para Encontrar Campos de Retas Invariantes por Grupos Infinitos / Filipe Ramos Ferreira. -- Rio de Janeiro, 2018.
73 f.

Orientador: Severino Collier Coutinho.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2018.

1. 1-formas. 2. Axiom. 3. Literate Programming. I. Coutinho, Severino Collier, orient. II. Titulo.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a).

FILIPE RAMOS FERREIRA

BIBLIOTECA DO AXIOM PARA ENCONTRAR CAMPOS DE RETAS INVARIANTES POR GRUPOS INFINITOS

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em _____ de _____ de _____ .

BANCA EXAMINADORA:

_____

Prof. Severino Collier Coutinho, D.Sc.

_____

Prof. Amaury Alvarez Cruz, D.Sc.

_____

Prof. Juliana Vianna Valério, D.Sc.

AGRADECIMENTOS

Gostaria de agradecer inicialmente a minha esposa, Jullyana, que me deu todo o suporte necessário para continuar a perseverar dentro da Universidade, mesmo quando eu só tinha vontade de desistir de tudo, assim como a outros amigos e conhecidos que fiz aqui, e levarei para o resto da vida.

Também quero agradecer ao professor Collier, que me deu a oportunidade de uma bolsa de Iniciação Científica, e que acabou me levando tão longe, se transformando neste trabalho que você etá lendo agora.

Por fim, gostaria de agradecer a banca que lerá este trabalho, e espero que consiga convencê-los dos benefícios do Literate Programming, e que possam levar isso para mais alunos.

RESUMO

Formas diferenciais são um objeto matemático que possui as mais diversas aplicações em geometria, topologia e física. Este trabalho, utilizando-se do paradigma de Literate Programming, implementou uma biblioteca para o sistema Axiom com o objetivo de auxiliar a encontrar 1-formas invariantes por grupos de transformações lineares.

Palavras-chave: 1-formas. Axiom. Literate Programming.

ABSTRACT

Differential forms are a mathematical object that have several applications in geometry, topology and physics. This project, following the Literate Programming paradigm, implemented a library for the Axiom system, with the purpose of helping to find 1-forms that are invariant by groups of linear transformations.

Keywords: 1-form. Axiom. Literate Programming.

SUMÁRIO

1 INTRODUÇÃO

1.1 LITERATE PROGRAMMING

O paradigma de *Literate Programming* foi introduzido pela primeira vez por Donald Knuth, um cientista da computação americano, e professor na Universidade de Stanford. Seu objetivo era mostrar um novo modo de programar, que ao invés de focar em dar instruções para o computador executar, se dedica a explicar para humanos o que queremos que o computador faça. [5]

Um programa seguindo esse paradigma é mais semelhante a uma redação do que a um código usual. A preocupação é muito maior em introduzir conceitos seguindo uma lógica de fluxo de pensamentos, ao invés da ordem que o computador requer. [6]

Através dos processos de *tangle* e *weave*, é possível criar um código-fonte para ser lido pelo computador, ou um arquivo de texto, para ser lido por humanos. Knuth acreditou que esse novo paradigma acarretaria em programas melhor escritos, facilitando também o entendimento do código por outros programadores que eventualmente precisassem entendê-lo. [7]

1.2 AXIOM

O Axiom é um sistema de computação algébrica, originalmente chamado de Scratchpad, foi desenvolvido em 1965 por James Greismer, um funcionário da IBM, mas nunca chegou a ser lançado. Seu sucessor, o Scratchpad II, começou a ser desenvolvido em 1977, sob a direção de Richard D. Jenks. [1] Em torno de 1990, foi renomeado Axiom para o seu lançamento comercial, até 2001 quando foi retirado do mercado, e relançado sob uma licença open-source.

Atualmente é mantido por Tim Daly, que é um grande defensor das ideias de Knuth sobre *Literate Programming*. De acordo com ele, em um comentário na rede social de notícias *Hacker News*: “Imagine um livro de Física que fosse “apenas as

equações" sem nenhum texto. Esse é o jeito que escrevemos código hoje em dia. É verdade que as equações são a essência, mas sem o texto em volta, elas são bastante opacas." [3]

Devido a esse pensamento, todo o sistema do Axiom foi re-escrito para utilizar o *Literate Programming*, com a esperança de que programadores do futuro possam entender, manter, e construir sobre o sistema de modo fácil. [4]

1.3 A BIBLIOTECA

A biblioteca apresentada aqui, escrita para o sistema Axiom, começou como um trabalho de Iniciação Científica no final de 2014, com a intenção de encontrar campos de retas que sejam deixados invariantes por grupos cíclicos. Campos de retas com essas propriedades são bem comuns na Física, aparecendo em diversos problemas, e esperávamos conseguir encontrar soluções mais fáceis para eles ao estudá-los.

Eventualmente, fomos motivados pelo trabalho de D. Cervaeu et. al [2], que encontrou e classificou as 1-formas de grau 2 com 1 singularidade, para tentar fazer o mesmo com as 1-formas de grau 3, tentando encontrar um padrão, na esperança de conseguir generalizar para 1-formas de qualquer grau.

Apesar de não ter sido originalmente escrito utilizando-se do paradigma de *Literate Programming*, esta biblioteca foi re-escrita desse modo, seguindo o ideal de Tim Daly para as bibliotecas do Axiom. Dessa forma, o próprio código pode ser transformado no texto desse Projeto de Conclusão de Curso, além de facilitar o trabalho de futuros alunos de Iniciação Científica, que possam vir a expandir sobre esse trabalho.

1.4 FUNÇÕES UTILIZADAS

Para a confecção da biblioteca, foram utilizadas algumas funções da biblioteca padrão do Axiom, que não foram implementadas por mim. Para que se saiba exatamente o que elas fazem, estão explicadas abaixo:

1.4.1 Normal Form

A função `normalForm` recebe como parâmetros de entrada um polinômio, e uma lista de polinômios, realizando a divisão do primeiro por todos os polinômios da lista e retornando o resto.

Para os efeitos deste trabalho, o que está efetivamente sendo feito, é aplicar as restrições da lista no polinômio. Suponha que tenhamos um polinômio P e uma lista de polinômios L, tais que:

$$P = A_5y^2 + A_4xy + A_3x^2 + A_2y + A_1x + A_0$$

$$L = [A_5 - 2A_2, A_3 - 7A_0, A_1]$$

Os polinômios de L podem ser considerados como as igualdades $A_5 = 2A_2$, $A_3 = 7A_0$ e $A_1 = 0$, que são aplicadas ao polinômio P, resultando em:

$$P' = 2A_2y^2 + A_4xy + 7A_0 + A_2y + A_0$$

1.4.2 Groebner Factorize

A função `groebnerFactorize` recebe uma lista G de polinômios, da seguinte forma:

$$G = [P_1, P_2, P_3, ..., P_n]$$

Então, para cada P_i, é feita a diferença com todos os outros polinômios de P_{i+1} até P_n, após igualar os seus maiores monômios, considerando a ordem lexicográfica. Considere por exemplo que temos P_1 e P_2 tais que:

$$P_1 = xy^2 + y^4 + 1,$$

$$P_2 = x^3 + xy^5 + xy - y.$$

Para igualar os primeiros monômios, xy^2 e x^3, devemos multiplicar o primeiro por x^2 e o segundo por y^2, de forma que teremos:

$$S(P_1, P_2) = x^2 P_1 - y^2 P_2.$$

Em seguida, $S(P_1, P_2)$ é dividido por todos os polinômios em G, para ter certeza de que não conterá nenhum polinômio que já está na lista, e então, ele é adicionado ao final da lista G.

A segunda parte consiste em fatorar os polinômios da lista resultante da primeira parte. Considere que após calcular todos os pares $S(P_i, P_j)$, terminamos com a seguinte lista:

$$L = [x^2 - 1, xy - y]$$

O polinômio $x^2 - 1$ pode ser fatorado em $x - 1$ e $x + 1$, assim como o polinômio $xy - y$ pode ser fatorado em $x - 1$ e y. Portanto, abrindo em uma árvore, teremos quatro listas diferentes:

$$L_1 = [x - 1, x - 1]$$

$$L_2 = [x - 1, y]$$

$$L_3 = [x + 1, x - 1]$$

$$L_4 = [x + 1, y]$$

Para cada uma dessas listas, é aplicado novamente o algoritmo, parando apenas quando não for possível fatorar mais nada. Ao final, todas elas são retornadas em uma lista de listas.

1.5 PLANO PROJETIVO

O plano projetivo, representado pelo símbolo $\mathbb{P}^2$, é semelhante ao conceito do plano Euclidiano, com a adição da reta no infinito. É interessante notar que não

existem retas paralelas em um plano projetivo, pois elas devem sempre se encontrar em algum ponto. Retas que são paralelas em um plano Euclidiano, se encontram em um ponto no infinito no plano projetivo.

Um ponto em $\mathbb{P}^2$ é uma reta que passa pela origem em $\mathbb{C}^3$. Assim a um vetor não nulo $v = (x, y, z) \in \mathbb{C}^3$, associamos o ponto:

$$[v] = [x : y : z] = \{\lambda v \mid \lambda \in \mathbb{C}\}.$$

Neste caso dizemos que x, y, z são as coordenadas homogêneas de $[v]$. Note que as coordenadas homogêneas de um ponto não são únicas. Se $[v] = [x : y : z]$ e $\lambda \neq 0$ é um número complexo, então:

$$[v] = [\lambda x : \lambda y : \lambda z].$$

Podemos identificar o subconjunto

$$\mathbb{U} = \{[x : y : z] \in \mathbb{P}^2 \mid z \neq 0\} \subseteq \mathbb{P}^2$$

com o espaço $\mathbb{C}^3$ usando a aplicação:

$$\Pi : \begin{array}{ccc} \mathbb{U} & \to & \mathbb{C}^2 \\ [x:y:z] & \mapsto & (\frac{x}{z}, \frac{y}{z}) \end{array}.$$

2 PULLBACK E 1-FORMAS

2.1 1-FORMA

Uma 1-forma polinomial em $\mathbb{C}^3$ é uma aplicação $\omega : \mathbb{C}^3 \times \mathbb{C}^3 \to \mathbb{C}$ que satisfaz as seguintes propriedades:

- Para cada $p_0 \in \mathbb{C}^3$, $\omega(p_0, u)$ é uma transformação linear de $\mathbb{C}^3$ em $\mathbb{C}^3$.
- Para cada $u_0 \in \mathbb{C}^3$, $\omega(p, u_0)$ é um polinômio em três variáveis.

Seja $p_0 \in \mathbb{C}^3$ e $u = a_1e_1 + a_2e_2 + a_3e_3$, temos que:

$$\omega(p_0, u) = a_1\omega(p_0, e_1) + a_2\omega(p_0, e_2) + a_3\omega(p_0, e_3)$$

Podemos definir dx, dy e dz como sendo as tranformações lineares $\mathbb{C}^3 \to \mathbb{C}$ que satisfazem:

$$dx(u) = a_1,$$

$$dy(u) = a_2,$$

$$dy(u) = a_3.$$

Estas 1-formas são definidas no Axiom usando a seguinte construção:

—— * ——

```
coef:=Integer
lv:List Symbol:=[x,y,z]
der:=DERHAM(coef,lv)
R:=Expression coef

dx:der:=generator(1)
```

```
dy:der:=generator(2)
dz:der:=generator(3)
```

———-

Com isso, podemos reescrever $\omega(p_0, u)$ como:

$$\omega(p_0, u) = \omega(p_0, e_1)dx(u) + \omega(p_0, e_2)dy(u) + \omega(p_0, e_3)dz(u)$$

Por outro lado, segue da segunda propriedade da 1-forma polinomial descrita anteriormente que $\omega((x, y, z), e_i)$ é um polinômio em x, y e z para $i = 1, 2, 3$. Logo, se definindo:

$$A_i = \omega((x, y, z), e_i),$$

podemos com isso escrever a 1-forma ω como:

$$\omega = A_1dx + A_2dy + A_3dz.$$

Podemos definir uma 1-forma em $\mathbb{C}^2$ de maneira análoga, mas envolvendo apenas as variáveis x e y. É possível converter uma 1-forma em $\mathbb{C}^3$ para uma 1-forma em $\mathbb{C}^2$ e vice-versa. Utilizaremos desse conceito para "homogeneizar" e "desomogeneizar" 1-formas, assim como é feito com as coordenadas do plano projetivo. Para mais detalhes veja a seção 2.6.

Na maior parte, utilizaremos 1-formas desomogeneizadas, ou seja em $\mathbb{C}^2$, pois os cálculos são mais rápidos. Porém, em alguns momentos precisaremos trabalhar com as 1-formas em $\mathbb{C}^3$, para não perder informações sobre o que acontece com elas nos pontos do infinito no plano projetivo.

2.2 CAMPOS DE RETAS

Um campo de retas em $\mathbb{P}^2$ é semelhante a um campo vetorial em $\mathbb{C}^3$, porém, temos uma reta associada a cada ponto ao invés de um vetor. Uma reta do plano

projetivo $\mathbb{P}^2$ pode ser definida por uma equação da forma $ax + by + cz = 0$. Em outras palavras, esse é o núcleo da transformação linear:

$$\varphi : \begin{matrix}\mathbb{C}^3 \\ (x,y,z)\end{matrix} \begin{matrix}\to \\ \mapsto\end{matrix} \begin{matrix}\mathbb{C} \\ ax+by+cz\end{matrix}.$$

Para qualquer vetor v em $\mathbb{C}^3$, temos um ponto correspondente $[v]$ em $\mathbb{P}^2$. Mesmo que v seja multiplicado por qualquer constante $\lambda \in \mathbb{C}$, o seu ponto $[v]$ correspondente em $\mathbb{P}^2$ continua inalterado. Sabendo disso, podemos escrever uma reta r como:

$$r = \{[v] \mid \varphi(v) = 0\}.$$

Definindo $v = \alpha_1 e_1 + \alpha_2 e_2 + \alpha_3 e_3$, teremos:

$$\varphi(v) = \alpha_1\varphi(e_1) + \alpha_2\varphi(e_2) + \alpha_3\varphi(e_3)$$

Utilizando-se da definição para dx, dy e dz feita na seção anterior, podemos reescrever φ como:

$$\varphi = \varphi(e_1)dx + \varphi(e_2)dy + \varphi(e_3)dz,$$

$$\varphi(e_1), \varphi(e_2), \varphi(e_3) \in \mathbb{C}.$$

É possível notar que φ se parece muito com as 1-formas que vimos na seção anterior, mas os seus coeficientes são constantes em $\mathbb{C}$ e não polinômios. Se utilizarmos coeficientes polinomiais ao invés de constantes, φ passa a corresponder não apenas a uma única reta, mas a todo o campo de retas.

Suponha que tenhamos polinômios homogêneos de mesmo grau, A, B e C, com coeficientes complexos. Desde que $Ax + By + Cz = 0$, podemos definir o campo de retas Ω como:

$$\Omega = Adx + Bdy + Cdz$$

A condição $Ax + By + Cz = 0$ é necessária para garantir que, para cada ponto $p = [x_0 : y_0 : z_0]$, a reta $A(p)x + B(p)y + C(p)z = 0$ passa no ponto p.

É importante dizer que o campo de retas não está definido no ponto na origem, e que assim como v e λv correspondem a mesma reta em $\mathbb{P}^2$, Ω e $\lambda\Omega$ correspondem ao mesmo campo de retas em $\mathbb{P}^2$, desde que λ seja uma constante complexa diferente de zero.

2.3 TRANSFORMAÇÕES

As transformações no escopo deste trabalho são transformações lineares bijetivas definidas como:

$$\sigma\colon \mathbb{C}^2 \to \mathbb{C}^2.$$

Elas podem ser escritas facilmente na forma de matriz, um exemplo sendo a seguinte transformação σ':

$$\sigma' = \begin{bmatrix} 10 & 6 & 0 \\ 0 & 4 & 2 \\ 0 & 0 & 2 \end{bmatrix}.$$

Nesta biblioteca porém, foi escolhida a representação vetorial, por ser mais fácil de realizar os cálculos. Para passar da representação de matriz para vetor, simplesmente se multiplica a matriz σ pelo vetor $[x, y, z]$:

$$\sigma'(x, y, z) = \begin{bmatrix} 10 & 6 & 0 \\ 0 & 4 & 2 \\ 0 & 0 & 2 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 10x + 6y \\ 4y + 2z \\ 2z \end{bmatrix}.$$

Como estamos trabalhando em $\mathbb{P}^2$, é necessário tomar um cuidado com as transformações, pois a reta $z = 0$ foi definida como a reta no infinito. Portanto, é necessário fazer a desomogeneização da transformação por z, para obter uma transformação correspondente que seja possível aplicar nas nossas 1-formas em $\mathbb{C}^2$.

Isso é feito dividindo a matriz pelo valor no canto inferior direito, que corresponde ao z, de modo que naquele canto o valor fique igual a 1, para efetivamente remover a variável z, levando a transformação de $\mathbb{C}^3$ para $\mathbb{C}^2$. Fazendo isso com σ' obtemos:

$$\sigma'' = \begin{bmatrix} 5 & 3 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 5x + 3y \\ 2y + 1 \\ 1 \end{bmatrix}.$$

Como, agora, a terceira parte da transformação é uma constante igual a 1 ela não causará nenhuma mudança quando for aplicada e podemos apenas ignorá-la, resultando na transformação de $\mathbb{C}^2 \to \mathbb{C}^2$ definida por:

$$\sigma'' = [5x + 3y, 2y + 1]$$

Na próxima seção, veremos como essa transformação pode ser aplicada nas nossas 1-formas.

2.4 PULLBACK

De um modo simplificado, o pullback pode ser visto como uma composição de funções. Suponha que exista uma função f que depende de x. Por sua vez, x é uma função de y, de modo que temos $f(x(y))$. Se for feito o pullback de f por x, teremos uma função equivalente g, que depende apenas de y:

$$f(x(y)) \equiv g(y).$$

O pullback define uma transformação linear no espaço vetorial das 1-formas. No escopo das 1-formas deste trabalho, os coeficientes de dx e dy serão funções polinomiais de x e y.

A função `pullback` foi implementada recebendo como entrada uma transformação σ e uma 1-forma ω tal que:

$$\sigma = [u_1(x,y), u_2(x,y)]$$

$$\omega = a(x,y)\ dx + b(x,y)\ dy$$

— * —

```
pullback(sigma:List(R), w:der):der ==
```

————-

São, então, feitas as substituições nos coeficientes de dx e dy, tendo como resultado novos coeficientes do tipo:

$$a'(u_1,u_2) = a(x(u_1), y(u_2))$$

$$b'(u_1,u_2) = b(x(u_1), y(u_2))$$

Como definimos a função `pullback` para funcionar tanto em $\mathbb{C}^2$ como $\mathbb{C}^3$, precisamos verificar se a lista σ da transformação contém dois ou três elementos, de modo a não ter nenhuma referência inválida durante a execução do código.

— * —

```
  coefList := extractCoefficients(w, 1)
  if #sigma = 2 then
    coefList := eval(coefList,[x=sigma.1, y=sigma.2])
  if #sigma = 3 then
    coefList := eval(coefList,[x=sigma.1, y=sigma.2, z=sigma.3])
```

————-

É necessário porém, tomar cuidado com os diferenciais. Ao fazer a substituição de x por u_1, é necessário lembrar que precisamos também fazer a substituição de dx por du_1. A mesma coisa para y e u_2.

Para isso a função `totalDifferential` é aplicada em u_1 e u_2. O que essa função fará é uma espécie de gradiente com u_1 e u_2, que são funções em x, y. Assim:

$$du_1 = \frac{\partial u_1}{\partial x}dx + \frac{\partial u_1}{\partial y}dy,$$

$$du_2 = \frac{\partial u_2}{\partial x}dx + \frac{\partial u_2}{\partial y}dy.$$

Em seguida, multiplicando-os pelos coeficientes anteriormente calculados para obter uma nova forma com o pullback aplicado:

$$\omega' = a'(u_1, u_2)\ du_1 + b'(u_1, u_2)\ du_2.$$

E a nova forma ω' é o que a função `pullback` retornará.

— * —

```
w := coefList.1 * totalDifferential(sigma.1)
w := w + coefList.2 * totalDifferential(sigma.2)
if #sigma = 3 then
  w := w + coefList.3 * totalDifferential(sigma.3)
return(w)
```

————-

2.5 GRAU DE UM CAMPO DE RETAS

A função `degreeLineField` calcula o grau do campo de retas definido pela 1-forma, nas variáveis dadas como parâmetro. Ela é importante para fazer a homogeneização da 1-forma, pois para isso é necessário saber o grau do campo de retas definida por ela.

— * —

```
degreeLineField(w:der, lv:List(Symbol)):NNI ==
```

———————-

Inicialmente, são calculados os graus dos coeficientes da 1-forma. Considere por exemplo a 1-forma:

$$\omega = (-B_1y^2 + (B_1 - 2A_0)y + 2B_1x + 2B_0)\ dy + 2A_0\ dx$$

Verificamos que o coeficiente de dy possui grau 2, e o coeficiente de dx possui grau 0.

—— * ——

```
a := coefficient(w, dx)::POLY(INT)
b := coefficient(w, dy)::POLY(INT)
adegw := max(totalDegree(a, lv), totalDegree(b, lv))
```

———————-

A partir do maior dos graus dos coeficientes calculados anteriormente, encontramos a componente homogênea deste grau máximo de cada um deles, ou seja, apenas a parte do polinômio que tem grau igual ao encontrado. A componente homogênea de grau máximo h_b do coeficiente de dy no exemplo será:

$$h_b = -B_1y^2.$$

Como o coeficiente de dx não tem nenhuma parte com grau igual a 2, a sua componente homogênea de grau máximo h_a será 0.

—— * ——

```
hca := homogeneousComponent(a, adegw, lv)
hcb := homogeneousComponent(b, adegw, lv)
```

———-

Sabendo as componentes homogêneas de grau máximo, podemos calcular um δ da forma:

$$\delta = h_a x + h_b y$$

Seguindo do nosso exemplo:

$$\delta = 0x - B_1 y^3$$

Sendo δ diferente de 0, o grau do campo de retas corresponde ao grau dos componentes homogêneos, e é igual ao grau da 1-forma. Se δ for igual a zero, o grau do campo de retas é o grau dos componentes homogêneos subtraído de 1, e não é igual ao grau da 1-forma.

As 1-formas estudadas no escopo deste trabalho, têm sempre grau igual ao do campo de retas correspondente.

—— * ——

```
delta := x * hca + y * hcb
if ~zero?(delta) then
  return (adegw)
else
  return (adegw-1)
```

———-

2.6 HOMOGENEIZAÇÃO DE 1-FORMAS

Como mencionado anteriormente, é possível converter 1-formas de $\mathbb{C}^2$ em $\mathbb{C}^3$ e vice-versa. Para isso, usaremos as funções `homogenizeForm` e `dehomogenize`, ilustradas nesta seção.

A função `homogenizeForm` receberá como parâmetro uma 1-forma ω em $\mathbb{C}^2$, ou seja, que envolve apenas x e y, e retornará uma 1-forma ω_h em $\mathbb{C}^3$, que está definida em x, y e z. Também será entrada da função o grau do campo de retas associado a 1-forma.

— * —

```
homogenizeForm(w:der, degree:NNI):der ==
```

———-

Começamos definindo a transformação:

$$\sigma = [x/z, y/z],$$

que será aplicada a ω utilizando-se da função `pullback`. Com isso, toda ocorrência de x e y na 1-forma será dividida por z.

Essa ideia vem da definição dos pontos no plano projetivo $\mathbb{P}^2$, no qual o ponto em coordenadas homogêneas $[x : y : z]$ poderia ser escrito em coordenadas não homogêneas como $(x/z, y/z)$. É como se na 1-forma em $\mathbb{C}^2$, houvesse uma definição de que $z = 1$, e retiramos essa restrição para encontrar a 1-forma em $\mathbb{C}^3$.

Supondo que temos a 1-forma:

$$\omega = 2A_0\ dx + (-B_1y^2 + (B_1 - 2A_0)y + 2B_1x + 2B_0)\ dy,$$

ao fazer o pullback por σ, teremos:

$$\omega'_h = \frac{2A_0}{z}dx + \frac{(-B_1y^2+(B_1-2A_0)y+2B_1x+2B_0)}{z^3}dy$$
$$+\frac{(-2B_0y-2A_0x)z^2+((-B_1+2A_0)y^2-2B_1xy)z+B_1y^3}{z^4}dz$$

— * —

```
sigma:List(R) := [x/z, y/z]
w := pullback(sigma, w)
```

———————-

Tendo ω'_h, multiplicamos toda a forma por z elevado ao grau do campo de retas definido por ω, acrescido de dois, para finalmente obter ω_h.

$$z^4 * \omega'_h = \omega_h = 2A_0z^3dx + (2B_0z^3 + ((B_1 - 2A_0)y + 2B_1x)z^2 - B_1y^2z)dy$$
$$+((-2B_0y - 2A_0x)z^2 + ((-B_1 + 2A_0)y^2 - 2B_1xy)z + B_1y^3)dz$$

—— * ——

```
coefList := extractCoefficients(w, 1)
w := z^(degree + 2) * (coefList.1 * dx + coefList.2 * dy + coefList.3 * dz)
return(w)
```

———————-

É interessante notar que os coeficientes da 1-forma original ω possuiam grau máximo 2, enquanto os de ω_h possuem todos grau 3. Como dito na seção anterior, as formas que trabalhamos com tem sempre grau máximo dos coeficientes igual ao grau do seu campo de retas, mas isso não é verdade para todas as 1-formas.

Podemos usar a função `dehomogenize` para desomogeneizar uma 1-forma, levando-a de $\mathbb{C}^3$ para $\mathbb{C}^2$. A função receberá a 1-forma homogênea e a variável pela qual queremos desomogeneiza-la, que pode ser x, y ou z.

—— * ——

```
dehomogenize(w:der, variable:Symbol):der ==
```

———————-

Se utilizarmos como parâmetro a 1-forma ω_h obtida anteriormente, fazendo a desomogeneização por z, obteremos novamente a 1-forma original ω, comprovando que as funções são consistentes entre si. O que acontece, porém, se escolhermos desomogeneizar por x, por exemplo?

O algoritmo é bem simples, sendo simplesmente necessário aplicar a restrição $x = 1$ sobre a 1-forma, e cortar o seu termo em dx, o que resulta em:

$$\omega' = (2B_0z^3 + ((B_1 - 2A_0)y + 2B_1)z^2 - B_1y^2z)dy + ((-2B_0y - 2A_0)z^2 \\ +((-B_1 + 2A_0)y^2 - 2B_1y)z + B_1y^3)dz.$$

O processo é o mesmo para desomogeneizar por y ou z, mudando apenas qual a variável que será igualada a 1, e qual termo será cortado.

— * —

```
coefList := extractCoefficients(w, 1)
if (variable = x) then
  coefList := eval(coefList, [x = 1])
  w := coefList.2 * dy + coefList.3 * dz
if (variable = y) then
  coefList := eval(coefList, [y = 1])
  w := coefList.1 * dx + coefList.3 * dz
if (variable = z) then
  coefList := eval(coefList, [z = 1])
  w := coefList.1 * dx + coefList.2 * dy
return(w)
```

————-

É possível ver que a 1-forma desomogeneizada por x ainda tem o grau máximo dos coeficientes igual a 3, assim como ω_h. Portanto, não necessariamente o grau da 1-forma é mantido após homogeneizações e desomogeneizações.

3 ENCONTRANDO 1-FORMAS INVARIANTES

3.1 OBJETIVO

O objetivo da função `possibleForms` é encontrar, a partir de uma transformação σ, quais 1-formas ω são deixadas invariantes por essa transformação, a menos de multiplicação por constante não nula. Ou seja, quando feito o pullback dessa forma pela transformação, teremos como resultado a mesma forma, multiplicada por uma constante diferente de zero:

$$\sigma^*\omega = \lambda\omega \ (\lambda \neq 0).$$

Ela é dividida em duas partes principais: as funções `collectCoefficients` e `createFormList`. A primeira fará o trabalho de encontrar os coeficientes de uma 1-forma genérica na qual foi aplicada a transformação σ, e a segunda utilizará isso para montar as 1-formas invariantes, juntamente com a lista de restrições que deve ser aplicada à 1-forma genérica para chegar naquela 1-forma.

3.2 COLLECT COEFFICIENTS

A função `collectCoefficients` fará o trabalho de aplicar a transformação em uma 1-forma genérica ω (uma 1-forma com todos os coeficientes indeterminados) e retornar uma lista de seus coeficientes.

As entradas da função são um inteiro N, correspondente ao grau dos coeficientes da 1-forma ω, uma transformação σ e um autovalor λ da transformação definida pelo pullback de σ no espaço de todas as 1-formas de grau N.

— * —

```
collectCoefficients(degree:NNI, sigma:List(R), lambda:R):List(POLY(INT)) ==
  coefList:List(POLY(INT)) := []
```

———-

A forma genérica ω é criada em duas etapas. Primeiro, criam-se dois polinômios genéricos, ou seja, que têm todos os seus coeficientes indeterminados, utilizando a função `genericPolynomial`. Considerando uma entrada com $N = 2$, os polinômios terão a forma:

$$A = A_5y^2 + A_4xy + A_3x^2 + A_2y + A_1x + A_0,$$

$$B = B_5y^2 + B_4xy + B_3x^2 + B_2y + B_1x + B_0.$$

—— * ——

```
poly1 := genericPolynomial(degree, A)
poly2 := genericPolynomial(degree, B)
```

———-

Para efetivamente criar a forma genérica, cada um dos polinômios é multiplicado por um diferencial, resultando na forma:

$$\omega = A\ dx + B\ dy.$$

Sendo A e B os polinômios criados anteriormente, essa é uma 1-forma de grau 2 com todos os seus coeficientes indeterminados, que é o que chamamos de forma genérica.

—— * ——

```
w:der := poly1 * dx + poly2 * dy
```

———-

Então é feito o pullback da transformação σ em ω, tomando cuidado para adicionar a restrição que é necessária para que essa forma seja invariante, ou seja, queremos que:

$$\sigma^*\omega = \lambda\omega,$$

que é equivalente a:

$$\sigma^*\omega - \lambda\omega = 0.$$

— * —

```
w := pullback(sigma, w) - lambda*w
```

———-

Chamando de ω' a forma resultante do cálculo $\sigma^*\omega - \lambda\omega$, devemos garantir que ω' seja igual a 0. Como sabemos que ω' é uma forma do tipo $a\ dx + b\ dy$, o único modo de fazer com que seja igual a zero, é se $a = b = 0$.

Como a e b são polinômios, isto só vai ocorrer se seus monômios em x e y tiverem coeficientes nulos. Para podermos impor estas condições aos coeficientes de a e b, precisamos listá-los.

— * —

```
dxCoef := coefficient(w, dx)::POLY(INT)
dyCoef := coefficient(w, dy)::POLY(INT)
for i in 0..degree repeat
  for j in i..0 by -1 repeat
    k:NNI := i - j
    coefList := cons(coefficient(dxCoef, [x, y], [j, k]), coefList)
    coefList := cons(coefficient(dyCoef, [x, y], [j, k]), coefList)
```

———-

Com a lista dos coeficientes pronta, a função irá retorná-la, para ser usada pela `possibleForms`.

—— * ——

```
  return(coefList)
```

———-

3.3 CREATE FORM LIST

A segunda parte da `possibleForms` é a função `createFormList`. Ela receberá o grau N da forma e uma lista que foi derivada da saída da função `collectCoefficient` vista anteriormente.

O tratamento exato dado à saída da função anterior para chegar a essa lista será explicado durante o algoritmo de `possibleForms`; por enquanto, é necessário apenas saber que essa lista corresponde às restrições que devem ser aplicadas aos coeficientes da forma ω' calculada em `collectCoefficients`, para que esta seja igual a zero.

—— * ——

```
createFormList(degree:NNI, groebnerList:List(List(POLY(INT)))):pfList ==
  formList:pfList := []
```

———-

Assim como na função anterior, são utilizados polinômios gerados pela função `genericPolynomial` para criar uma 1-forma. Dessa vez porém, serão aplicadas res-

trições aos polinômios gerados, para podermos construir corretamente os coeficientes da 1-forma.

```
                    — * —

poly1 := genericPolynomial(degree, A)::POLY(INT)
poly2 := genericPolynomial(degree, B)::POLY(INT)

                    ————————-
```

Utilizando a lista de restrições recebidas como entrada, será feita a divisão dos polinômios genéricos pelos da lista de restrições, utilizando a função `normalForm`, para efetivamente aplicar essas restrições nos polinômios. Por exemplo, suponhamos que temos o seguinte polinômio P e a lista de restrições R:

$$P = A_5y^2 + A_4xy + A_3x^2 + A_2y + A_1x + A_0,$$

$$R = [A_5 = 3A_2,\ A_4 = 0,\ A_3 = \tfrac{1}{2}A_0,\ A_1 = 0].$$

Uma vez aplicadas as restrições, teremos o polinômio P':

$$P' = A_2(3y^2 + y) + A_0(\tfrac{1}{2}x^2 + 1)$$

Tendo sido aplicadas as restrições aos polinômios genéricos `poly1` e `poly2`, é possível montar a 1-forma ω correspondente, multiplicando-os por dx e dy.

```
                    — * —

for i in 1..#groebnerList repeat
  form1 := normalForm(poly1, groebnerList.i)
  form2 := normalForm(poly2, groebnerList.i)
  w := form1 * dx + form2 * dy
```

———-

Caso ω seja diferente de zero, encontramos uma 1-forma invariante pela transformação σ. Dois cuidados são tomados antes de adicionar essa 1-forma à lista que será retornada por esta função:

- Primeiro, a lista de restrições é limpa para remover as restrições que simplesmente igualam uma variável a 0. Na lista exemplo R, isso significaria remover as restrições $A_4 = 0$ e $A_1 = 0$, resultando na lista $R' = [A_5 = 3A_2,\ A_3 = \frac{1}{2}A_0]$.

  Isso é feito apenas para deixar a lista menor e mais legível na saída da função, pois é fácil, apenas olhando para a 1-forma, ver quais coeficientes foram igualados a zero pelas restrições. Porém, não é tão fácil de ver as equivalências entre os coeficientes, e apenas por isso a lista é mantida.

- Verifica-se também se o grau da 1-forma é correspondente ao grau que queremos. Caso contrário, temos uma 1-forma degenerada, que não nos interessa. Isso é feito verificando o grau dos polinômios nos coeficientes de dx e dy na 1-forma. Supondo que temos $\omega = A\ dx + B\ dy$, queremos que o grau de A ou o grau de B seja igual ao grau N recebido como entrada.

As formas que passam pela verificação, são adicionadas à lista que é retornada ao final da execução da função.

—— * ——

```
  if w ~= 0 then
    eqList := removeSymbols(degree, groebnerList.i)
    temp:Record(form:der, equations:List(POLY(INT))) := [w, eqList]
    if (getMaxDegree(temp.form) = degree) then
      formList := cons(temp, formList)
return(formList)
```

———-

3.4 POSSIBLE FORMS

Já tendo visto os dois principais componentes da função `possibleForms`, podemos agora ver como se juntam para encontrar as 1-formas invariantes. A entrada de `possibleForms` é composta de três itens:

- O grau N para as 1-formas invariantes que se deseja encontrar.
- A transformação σ para a qual se quer encontrar as formas invariantes.
- Um autovalor λ correspondente ao pullback de sigma considerado como operador linear no espaço vetorial de todas as formas de grau N, tendo como base formas do tipo $x^i y^j dx$ e $x^i y^j dy$, com $i+j \leq N$. Este autovalor irá corresponder à constante que multiplicará à forma invariante.

— * —

```
pfList ==> List(Record((form:der), equations:List(POLY(INT))))
possibleForms(degree:NNI, sigma:List(R), C:R): pfList ==
```

———-

Como visto anteriormente, inicia-se utilizando a função `collectCoefficients`. Dos coeficientes polinomiais de uma 1-forma genérica onde foi aplicada a transformação σ, é gerada uma lista com os seus monômios.

A essa lista é aplicado o algoritmo `groebnerFactorize`, assim como descrito na introdução deste trabalho, para simplificar as equações, tornando mais fácil observar as restrições impostas pelo sistema. É possível que, após a fatoração, existam resultados triviais, que não impõem restrição alguma. Esses resultados são filtrados pela função `removeOnes`.

Também é possível que existam duas ou mais listas de restrições nas quais uma está contida na outra. Isso é filtrado através da função `removeRedundancy`, o que

deixará apenas a lista de restrição mais geral nesses casos, ou seja, a que impõe menos restrições.

Para fins de exemplificação, suponha que escolhemos $N = 2$, $\sigma = [x + y, y + 1]$, e $\lambda = 1$. Após a execução de `collectCoefficients`, `groebnerFactorize` e as filtragens, teremos a seguinte lista:

$$L = [[2B_5 + B_1, B_4, B_3, 2B_2 - B_1 + 2A_0, A_5, A_4, A_3, A_2, A_1]]$$

Podemos verificar que, após toda a execução, apenas uma lista de restrições restou em L, vamos chamá-la de L_0. Cada um dos polinômios que aparece deve ser igual a zero, ou seja, $2B_5 + B_1 = 0$, $B_4 = 0$, ..., $A_1 = 0$. Essa lista corresponde às restrições que uma 1-forma genérica ω deve respeitar para ser invariante pela transformação σ.

— * —

```
coefList := collectCoefficients(degree, sigma, C)
groebnerList := groebnerFactorize(coefList)
groebnerList := removeOnes(groebnerList)
groebnerList := removeRedundancy(groebnerList)
```

————-

A segunda parte do trabalho é feita pela função `createFormList`, que irá aplicar as restrições indicadas na lista L a uma 1-forma genérica ω, resultando em uma ou mais ω' que é deixada invariante por σ.

Continuando o exemplo com os valores que definimos anteriormente, ao aplicar as restrições de L na 1-forma genérica, teremos:

$$\omega' = \frac{-B_1y^2+(B_1-2A_0)y+2B_1x+2B_0}{2}dy + A_0dx,$$

$$L_0 = [2B_2 - B_1 + 2A_0, 2B_5 + B_1].$$

Correspondentes à 1-forma invariante por σ e à lista de restrições L_0 imposta para chegar a esta 1-forma.

```
__ * __

formList := createFormList(degree, groebnerList)
return(formList)

_______-
```

Fazendo isso, `possibleForms` é capaz de encontrar 1-formas invariantes por qualquer transformação σ no espaço projetivo. Porém, também seria interessante para o trabalho que fosse possível fazer o caminho inverso, e a partir de uma 1-forma ω qualquer, encontrar todas transformações σ que a deixam invariante. As funções apresentadas nos próximos capítulos lidarão com esse caminho inverso.

4 ENCONTRANDO TRANSFORMAÇÕES

4.1 OBJETIVO

O objetivo da função `groupSystem` é encontrar as restrições que uma transformação σ necessita obedecer para deixar uma dada 1-forma ω invariante, a menos de multiplicação por constante, exatamente como foi visto no capítulo anterior:

$$\sigma^*\omega = \lambda\omega \ (\lambda \neq 0).$$

Assim como `possibleForms`, a função `groupSystem` é dividida em duas partes principais: as funções `formSystem` e `extractSubs`.

4.2 FORM SYSTEM

O primeiro passo de `groupSystem` é a função `formSystem`. A partir de uma 1-forma ω recebida como entrada e do seu grau, ela encontrará um sistema que representa as restrições que devem ser aplicadas a uma 1-forma genérica, para se obter ω.

No capítulo anterior foi vista a função `possibleForms`, cuja saída continha uma lista de 1-formas e suas restrições. Se uma dessas 1-formas for usada como a entrada de `formSystem`, a saída deve coincidir com a lista de restrições associada a ela. Isto é, com as equações que o subspaço definido parametricamente por uma dessas 1-formas satisfaz.

—— * ——

```
formSystem(w:der, degree:NNI):List(POLY(INT)) ==
```

————-

Inicialmente são criados dois polinômios genéricos P_1 e P_2 de grau N recebido como entrada. Supondo que $N = 2$, teremos:

$$P_1 = C_5y^2 + C_4xy + C_3x^2 + C_2y + C_1x + C_0,$$

$$P_2 = E_5y^2 + E_4xy + E_3x^2 + E_2y + E_1x + E_0.$$

— * —

```
poly1 := genericPolynomial(degree,C)::POLY(INT)
poly2 := genericPolynomial(degree,E)::POLY(INT)
numberOfConstants := quo((degree + 1) * (degree + 2), 2)
variableListC := [C[i] for i in 0..numberOfConstants]
variableListE := [E[i] for i in 0..numberOfConstants]
```

————-

Não temos garantia de que os coeficientes da 1-forma serão polinômios com coeficientes inteiros, e isso pode deixar os cálculos bem inconvenientes. Portanto, primeiro multiplicamos a 1-forma pelo mmc de seus coeficientes. Então, supondo que temos a 1-forma:

$$\omega' = \frac{-B_1y^2+(B_1-2A_0)y+2B_1x+2B_0}{2}\ dy + A_0\ dx,$$

verificamos que o mmc de seus denominadores é 2, e toda a 1-forma será multiplicada por este valor, para obter:

$$\omega = -B_1y^2 + (B_1 - 2A_0)y + 2B_1x + 2B_0\ dy + 2A_0\ dx.$$

Não há nenhum problema em fazer isso, pois como visto anteriormente, o campo de retas definido pela 1-forma não varia por multiplicação de constante não nula.

— * —

```
multiplier := lcm(denominator(coefficient(w, dx)),_
denominator(coefficient(w, dy)))
dxCoef := (coefficient(w,dx) * multiplier)::POLY(INT)
dyCoef := (coefficient(w,dy) * multiplier)::POLY(INT)
```

———-

Para começar a construir um subspaço do problema, precisamos descobrir quais são os coeficientes não-nulos em P_1 e P_2. Para isso, fazemos o produto interno entre os coeficientes de ω e estes polinômios, utilizando-se da função `innerProduct`.

Deve-se lembrar que o produto interno é uma operação entre dois vetores, mas P_1 e P_2 são polinômios. Porém, a função `innerProduct` foi definida de modo a utilizar polinômios, tratando-os como se fossem vetores na base formada pelos monômios daquele polinômio, o que permite que a operação seja feita sem problemas, e evita o trabalho de ter que realmente construir os vetores.

$$P_1' = 2A_0C_0$$

$$P_2' = -B_1E_5 + (B_1 - 2A_0)E_2 + 2B_1E_1 + 2B_0E_0$$

—— * ——

```
dxPoly:POLY(INT) := innerProduct(dxCoef,poly1)
dyPoly:POLY(INT) := innerProduct(dyCoef,poly2)
d0Poly:POLY(INT) := dxPoly + dyPoly
```

———-

Cria-se então uma lista com todos os coeficientes livres na 1-forma. Para ω, essa lista será:

$$L = [A_0, B_0, B_1]$$

Em seguida, é criada outra lista que contém os coeficientes de P_1' e P_2' em relação a cada uma das variáveis em L.

$$P_{A_0} = -2E_2 + 2C_0$$

$$P_{B_0} = 2E_0$$

$$P_{B_1} = -E_5 + E_2 + 2E_1$$

$$L' = [P_{A_0}, P_{B_0}, P_{B_1}]$$

— * —

```
formVariableList := getVariables(w)
finalList:List(POLY(INT)) := []
for i in 1..#formVariableList repeat
   polyCoefficient := coefficient(d0Poly, formVariableList.i, 1)
   finalList := append(finalList, [polyCoefficient])
```

————-

A lista L' define agora uma parametrização do complemento ortogonal do conjunto solução do sistema. Utilizamos a função `groebner`, que funciona semelhantemente à Eliminação Gaussiana, para encontrar uma base para este complemento ortogonal, dando origem a uma nova lista:

$$G = [E_5 - 2E_1 - C_0,\ E_2 - C_0,\ E_0]$$

Então, as restrições de G são aplicadas aos polinômios genéricos P_1 e P_2, utilizando-se da função `normalForm`, resultando em dois polinômios:

$$Q_1 = C_5y^2 + (C_4x + C_2)y + C_3x^2 + C_1x + C_0$$

$$Q_2 = (2E_1 + C_0)y^2 + (E_4x + C_0)y + E_3x^2 + E_1x$$

— * —

```
finalbasis := groebner(finalList)
poly1 := normalForm(poly1,finalbasis)::POLY(INT)
poly2 := normalForm(poly2,finalbasis)::POLY(INT)
```

———-

Os polinômios Q_1 e Q_2 representam então elementos do complemento ortogonal gerado pela base G. Com a ajuda da função `innerProduct`, podemos fazer o produto interno deles com polinômios genéricos criados pela função `genericPolynomial`, iguais a P_1 e P_2.

O vetor resultante desse produto interno, representado nos coeficientes dos polinômios resultantes Q'_1 e Q'_2, nos ajudarão a encontrar o sistema de restrições que queremos:

$$Q'_1 = A'_5C_5 + A'_4C_4 + A'_3C_3 + A'_2C_2 + A'_1C_1 + A'_0C_0$$

$$Q'_2 = B'_4E_4 + B'_3E_3 + (2B'_5 + B'_1)E_1 + (B'_5 + B'2)C_0$$

— * —

```
poly3 := genericPolynomial(degree,AA)::POLY(INT)
poly4 := genericPolynomial(degree,BB)::POLY(INT)
eq1:= innerProduct(poly1,poly3)
eq2:= innerProduct(poly2,poly4)
eq:= eq1+eq2
```

———-

Podemos listar todos os coeficientes das variáveis C_i e E_i nos polinômios Q'_1 e Q'_2. Cada um desses coeficientes será uma equação do sistema que representa as

restrições que devem ser aplicadas a uma 1-forma genérica para obter a 1-forma original ω. Utilizamos a função `groebner` mais uma vez, para simplificar a lista de restrições, obtendo:

$$E = [2B'_5 + B'_1, B'_4, B'_3, 2B'_2 - B'_1 + 2A'_0, A'_5, A'_4, A'_3, A'_2, A'_1].$$

Prestando atenção, vemos que essa lista de restrições é igual à que obtemos no final de `possibleForms` no capítulo anterior, mas parametrizada em A' e B' ao invés de A e B. Isso não é nenhuma coincidência, pois utilizamos a forma obtida por ela como entrada em `formSystem`, portanto, esse era o resultado esperado, e mostra que o algoritmo está funcionando corretamente.

— * —

```
eqs1:=[coefficient(eq,var,1) for var in variableListC]
eqs2:=[coefficient(eq,var,1) for var in variableListE]
eqs:= append(eqs1,eqs2)
eqs:= groebner(eqs)
return (eqs)
```

————-

4.3 EXTRACT SUBS

A segunda parte de `groupSystem` é feita com a função `extractSubs`. Ela recebe uma 1-forma, e o seu grau correspondente, gerando relações de igualdade entre os coeficientes dos monômios da 1-forma e a parametrização em A' e B'.

— * —

```
extractSubs(w:der, degree:NNI):List(EQ(POLY(INT))) ==
```

———-

São gerados dois polinômios genéricos com os coeficientes A'_i e B'_i, iguais aos da forma parametrizada criada em `formSystem`. Esses polinômios serão usados para encontrar as relações entre os coeficientes de mesmos monômios entre as 1-formas em A' e B' e as 1-formas em A e B.

— * —

```
poly1 := genericPolynomial(degree,AA)::POLY(INT)
poly2 := genericPolynomial(degree,BB)::POLY(INT)
```

———-

Novamente, assim como em `formSystem`, a forma é multiplicada pelo mmc de seus denominadores, de forma a não ter parte fracionária, tornando nossos cálculos mais convenientes.

— * —

```
tdxCoef := coefficient(w, dx)
tdyCoef := coefficient(w, dy)
multiplier := lcm(denominator(tdxCoef), denominator(tdyCoef))
dxCoef := (multiplier * tdxCoef)::POLY(INT)
dyCoef := (multiplier * tdyCoef)::POLY(INT)
```

———-

A lista de restrições então é criada, igualando os coeficientes de $x^i y^j$ dos coeficientes da 1-forma com o coeficiente de $x^i y^j$ dos polinômios genéricos. Para ilustrar, suponha que temos:

$$\omega = 2A_0\ dx + (-B_1 y^2 + (B_1 - 2A_0)y + 2B_1 x + 2B_0)\ dy$$

$$P_1 = A'_5y^2 + A'_4xy + A'_3x^2 + A'_2y + A'_1x + A'_0$$

$$P_2 = B'_5y^2 + B'_4xy + B'_3x^2 + B'_2y + B'_1x + B'_0$$

O resultado seria a lista:

$$L = [A'_0 = 2A_0, B'_0 = 2B_0, A'_1 = 0, B'_1 = 2B_1, A'_2 = 0, B'_2 = B_1 - 2A_0, A'_3 = 0, B'_3 = 9, A'_4 = 0, B'_4 = 0, A'_5 = 0, B'_5 = -B_1]$$

— * —

```
L:List(EQ(POLY(INT))) := []
for i in 0..degree repeat
  for j in 0..degree repeat
    if i+j <= degree then
      eqx := coefficient(poly1,[x,y],[i,j]) = coefficient(dxCoef,[x,y],[i,j])
      eqy := coefficient(poly2,[x,y],[i,j]) = coefficient(dyCoef,[x,y],[i,j])
      L := append(L,[eqx,eqy])
return(L)
```

———-

Já temos, agora, tudo que é necessário para apresentar a função principal deste capítulo, `groupSystem`

4.4 GROUP SYSTEM

Como dito no início deste capítulo, a função `groupSystem` é o primeiro passo em fazer o caminho inverso da função `possibleForms`, recebendo uma 1-forma ω e encontrando quais restrições uma transformação σ deve obedecer para manter ω invariante.

É importante notar aqui que o resultado da função acaba sendo grande demais para ilustrar com um exemplo numérico, como tem sido feito até aqui. Portanto,

todo o algoritmo será descrito, dentro do possível com exemplos, mas em sua maior parte apenas indicaremos as variáveis pelos seus nomes, sem representar os seus valores.

— * —

```
groupSystem(w:der):List(POLY(INT)) ==
```

———-

Inicialmente precisamos encontrar o grau de ω. Utilizamos a função definida anteriormente, `degreeLineField`, para fazer isso. Em seguida, é construída uma matriz 3×3 da seguinte forma:

$$M = \begin{bmatrix} I_1 & J_1 & K_1 \\ I_2 & J_2 & K_2 \\ I_3 & J_3 & K_3 \end{bmatrix}.$$

Essa matriz é uma representação de uma transformação genérica σ, com todos os coeficientes indeterminados, em forma de matriz. A primeira restrição da transformação é que o determinante dessa matriz deve ser diferente de zero, pois deve ser inversível. Para isso, adicionamos na lista que será retornada no final da função a restrição:

$$\det(M) = 1,$$

— * —

```
degree := degreeLineField(w, [x,y])
M := matrix[[I[1], J[1], K[1]],[I[2], J[2], K[2]],[I[3], J[3], K[3]]]
finalS:List(POLY(INT)):=[determinant(M)-1]
```

———-

É usada agora a função `formSystem`, como definida anteriormente nesse capítulo, recebendo o ω da entrada e o grau que foi calculado por `degreeLineField`, e obtendo o sistema S cujo conjunto solução é a 1-forma parametrizada.

—— * ——

```
S := formSystem(w, degree)
```

———-

Para o próximo cálculo, é necessário que a 1-forma seja escrita com as coordenadas homogêneas do plano projetivo, pois não queremos ainda nesse momento impor nenhuma restrição sobre os parâmetros da matriz M.

Para homogeneizar ω, chamamos a função `homogenizeForm`, resultando em uma nova forma homogeneizada ω_h. Nesta forma, podemos fazer o pullback por M sem nos preocupar com perder nenhuma informação sobre o campo de retas, resultando na forma ω'_h.

É necessário então desomogeneizar a 1-forma obtida, para continuar a execução do algoritmo. Utilizando a função `dehomogenize` em ω'_h, obtemos uma 1-forma apenas em x e y. Esta 1-forma resultante ω', corresponde ao resultado do pullback de ω pela transformação representada por M, mas não foi necessário impor nenhuma restrição sobre suas variáveis.

—— * ——

```
homoW := homogenizeForm(w, degree)
sigma := [I[a]*x + J[a]*y + K[a]*z for a in 1..3]
listSymbol := [I[1], J[1], K[1], I[2], J[2], K[2], I[3], J[3], K[3]]
homoW := pullback(sigma, homoW)
transformedW := dehomogenize(homoW, z)
```

———-

Precisamos agora encontrar uma relação entre as parametrizações de ω' e do sistema S, para poder escrever os dois utilizando a mesma parametrização, tornando mais fácil observar as relações entre os dois. Isso é feito com a função `extractSubs` descrita anteriormente, que irá gerar uma lista L com todas as relações necessárias para levar uma parametrização na outra.

Aplicamos no sistema S as relações descritas em L, tendo agora um novo sistema S', que descreve as restrições da 1-forma ω' em sua própria parametrização.

—— * ——

```
L := extractSubs(transformedW, degree)
S := eval(S, L)
```

———-

Por fim, é utilizada a função `collectVariables`, que irá encontrar no sistema S, todas as variáveis que aparecem mas não são variáveis de M, retornando-as em uma lista V. Elas correspondem às variáveis A_i e B_i de ω'.

Então, para cada uma dessas variáveis em V, será obtido o seu coeficiente, que é adicionado a mesma lista de restrições na qual colocamos o determinante de M. Essa lista corresponderá às restrições que M deve satisfazer para deixar ω invariante.

—— * ——

```
varList := collectVariables(S, listSymbol)
for i in 1..#S repeat
  finalS := append(finalS, [coefficient(S.i,var,1) for var in varList])
return (finalS)
```

———-

A lista retornada será tratada com a função `groebnerFactorize`, antes de ser utilizada pelo próximo passo no caminho, a função `getTransporterGroup`, que nos ajudará a visualizar as restrições de modo mais simples.

5 GRUPOS DE TRANSPORTE E ESTABILIZADORES

5.1 GROUP?

Os sistemas de restrições que encontramos com `getTransporterGroup` nos ajudam a encontrar um conjunto de transformações que leva uma certa 1-forma a uma outra que possui a mesma parametrização. Porém, esses conjuntos apenas nos são interessantes quando representam grupos. Então, para filtrar o resultado da função `getTransporterGroup`, que será apresentada nesse capítulo, precisamos conseguir descobrir se uma certo conjunto de transformações é ou não um grupo.

A função `group?`, que será usada para este fim, recebe uma família de transformações σ em sua forma de matriz, e um sistema S, que indica as restrições os coeficientes desta esta transformação satisfazem.

— * —

```
group?(G:transGroup):Boolean ==
  M1:= copy(G.matrix)
  M2:= copy(G.matrix)
  M2:= eval(M2,[I[1]=II[1], I[2]=II[2], I[3]=II[3], J[1]=JJ[1], J[2]=JJ[2],_
   J[3]=JJ[3], K[1]=KK[1], K[2]=KK[2],K[3]=KK[3]])
```

———-

Considere σ e S tais que:

$$\sigma = \begin{bmatrix} I_1 & J_1 & K_1 \\ 0 & 1 & K_2 \\ 0 & 0 & K_3 \end{bmatrix}$$

$$S = [J_2 - 1,\ 2I_1K_2 - 2J_1 + I_1 - 1,\ I_1K_3 - 1,\ (2J_1 + 1)K_3 - 2K_2 - 1]$$

Um dos requisitos para que o conjunto seja um grupo, é que a matriz identidade esteja contida nele. É utilizada a função `contains?` para verificar se a identidade está contida no conjunto recebido. Caso não esteja, a função retornará falso imediatamente, caso contrário irá prosseguir.

Para nosso exemplo, queremos que $I_1 = K_3 = 1$ e $J_1 = K_1 = K_2 = 0$. Podemos fazer essas substituições nas equações do sistema e verificar se a igualdade é mantida:

$$2I_1K_2 - 2J_1 + I_1 - 1 = 2 * 1 * 0 - 2 * 0 + 1 - 1 = 0$$

$$I_1K_3 - 1 = 1 * 1 - 1 = 0$$

$$(2J_1 + 1)K_3 - 2K_2 - 1 = (2 * 0 + 1) * 1 - 2 * 0 - 1 = 0$$

Portanto, a matriz identidade está contida no nosso conjunto, e podemos prosseguir ao próximo passo.

— * —

```
Id:= diagonalMatrix([1,1,1])
if ~contains?(G, Id) then
  return false
```

————-

Outro requisito para ser um grupo, é que a matriz inversa de uma matriz do conjunto deve estar contida no conjunto, usando novamente a função `contains?` para fazer essa verificação.

Podemos calcular a matriz inversa de σ, que é:

$$\sigma^{-1} = \begin{bmatrix} K_3 & -J_1K_3 & J_1K_2 - K_1 \\ 0 & I_1K_3 & -I_1K_2 \\ 0 & 0 & I_1 \end{bmatrix}$$

As contas acabam ficando grandes demais para mostrar aqui, mas é possível verificar que a matriz inversa também está contida no conjunto.

```
        __ * __

M1i:= M1^(-1)
dt := determinant(M1)
M1i := dt*M1i
if ~contains?(G, M1i) then
  return false

        _______-
```

Para finalmente poder dizer que é um grupo, devemos nos certificar que o produto de duas matrizes do conjunto resulta em uma matriz do conjunto. Mais uma vez a função `contains?` faz a verificação.

Podemos calcular o produto entre σ e um σ' que também está contido no conjunto, que resulta em:

$$\sigma \cdot \sigma' = \begin{bmatrix} I_1 & J_1 & K_1 \\ 0 & 1 & K_2 \\ 0 & 0 & K_3 \end{bmatrix} \cdot \begin{bmatrix} I'_1 & J'_1 & K'_1 \\ 0 & 1 & K'_2 \\ 0 & 0 & K'_3 \end{bmatrix}$$

Podemos verificar pelas restrições de S que a matriz resultante deste produto faz parte do conjunto. Portanto, podemos afirmar que σ e S são um grupo, e a função `group?` retornará verdadeiro.

```
        __ * __

M3:= M1*M2
if ~contains?(G, M3) then
  return false
return true
```

———-

Agora que conseguimos identificar um grupo, podemos ver como funcionará a função `getTransporterGroup`.

5.2 GET TRANSPORTER GROUP

Vimos anteriormente como a função `groupSystem` encontra uma lista de sistemas de restrições para uma transformação σ geral, que leva uma certa 1-forma ω a uma outra que satisfaz a mesma parametrização. Porém, a saída dessa função é de difícil interpretação por nós humanos. Para formatar esse sistema de um modo que deixe mais fácil de ser entendido, é usada a função `getTransporterGroup`.

Considere L a lista de sistemas que representam as restrições de σ, saída de `groupSystem`. Nele, é aplicado a função `groebnerFactorize`, simplificando-a para um lista L', que será a entrada de `getTransporterGroup`.

— * —

```
transGroup ==> Record(matrix:Matrix(EXPR(INT)), equations:List(POLY(INT)))
getTransporterGroup(groupList:List(List(POLY(INT)))): List(transGroup) ==
```

———-

Suponha que a lista L' contenha dois sistemas: S_1 e S_2, tais que:

$$S_1 = [(J_2 + 2J_1)K_3 - 2J_2K_2 + J_2 + 1,$$
$$(4J_1 - 2J_1 + 1)K_3 + (-4J_1J_2 - 2)K_2 + (2J_1 - 1)J_2 + 2J_1,\ I_1K_3 + J_2 + 1,$$
$$2I_1K_2 + (-2J_1 + I_1 + 1)J_2 + 1,\ J_3,\ J_2 + J_2 + 1,\ I_3,\ I_2]$$

$$S_2 = [(2J_1 + 1)K_3 - 2K_2 - 1,\ I_1K_3 - 1,\ 2I_1K_2 - 2J_1 + I_1 - 1,\ J_3,\ J_2 - 1,\ I_3,\ I_2]$$

Perceba o quanto é difícil observar quais realmente são as restrições que os sistemas aplicam, pois suas equações são longas e complicadas. Para facilitar, podemos

aplicar as restrições na transformação, tendo uma visualização da matriz juntamente com o sistema simplificado, para auxiliar a interpretação.

Primeiro é criada a transformação genérica σ, na qual iremos trabalhar:

$$\sigma = [I_1x + J_1y + K_1z,\ I_2x + J_2y + K_2z,\ I_3x + J_3y + K_3z]$$

— * —

```
matrixRecord:List(transGroup) := []
sigma := [I[a]*x + J[a]*y + K[a]*z for a in 1..3]
```

———-

Os passos a seguir serão repetidos para cada sistema S_i em L'. Para cada uma das equações de σ, é aplicada a função `normalForm` pelo sistema S_i, efetivamente aplicando as restrições do sistemas a elas. Em seguida, a função `sigmaToMatrix` reescreve a transformação em forma de matriz, facilitando a visualização.

Considerando a transformação σ_i como sendo a transformação na qual foi aplicada o sistema S_i, temos:

$$\sigma_1 = \begin{bmatrix} I_1 & J_1 & K_1 \\ 0 & J_2 & K_2 \\ 0 & 0 & K_3 \end{bmatrix} \text{ e } \sigma_2 = \begin{bmatrix} I_1 & J_1 & K_1 \\ 0 & 1 & K_2 \\ 0 & 0 & K_3 \end{bmatrix}$$

— * —

```
for i in 1..#groupList repeat
  newTransform := [normalForm(sigma.j, groupList.i) for j in 1..3]
  mtx:=sigmaToMatrix(newTransform)
```

———-

Caso o determinante dessa matriz seja diferente de zero, `getTransporterGroup` irá então enxugar um pouco dos sistemas, removendo deles as restrições que foram aplicadas, em seguida usando a função `group?` para testar se o conjunto composto pela matriz mais as restrições é um grupo.

Como só queremos grupos, resultados que não passem pelo filtro de `group?` são descartados, no final sendo retornados apenas os resultados que queremos. No caso de nosso exemplo, apenas o par σ_2 e S_2 representa um grupo, portanto, teremos no resultado final:

$$\sigma_2 = \begin{bmatrix} I_1 & J_1 & K_1 \\ 0 & 1 & K_2 \\ 0 & 0 & K_3 \end{bmatrix}$$

$$S_1 = [J_2 - 1,\ 2I_1K_2 - 2J_1 + I_1 - 1,\ I_1K_3 - 1,\ (2J_1 + 1)K_3 - 2K_2 - 1]$$

—— * ——

```
if determinant(mtx) ~= 0 then
  noSymbolList := removeSymbolsFromList(groupList.i,_
  [I[1], J[1], K[1], I[2], J[2], K[2], I[3], J[3], K[3]])
  temp:transGroup := [mtx, noSymbolList]
  if group?(temp) then
    matrixRecord := cons(temp, matrixRecord)
return (matrixRecord)
```

————-

É possível agora observar a forma da transformação e ter alguma ideia de como ela é. O sistema também está menor, e mais fácil de ser resolvido. Caso queiramos encontrar uma solução específica, só é necessário definir o valor para uma das variáveis livres, e é possível encontrar o valor de todas as outras utilizando-se das restrições impostas pelo sistema.

5.3 GET STAB

Com `getTransporterGroup`, conseguimos encontrar um grupo de transformações σ que leva uma 1-forma ω parametrizada a uma outra 1-forma $\sigma^*(\omega)$ que satisfaz a mesma parametrização. Podemos usar isso para encontrar o estabilizador, que é o grupo das transformações que leva ω em $\lambda\omega$, para algum número complexo $\lambda \neq 0$. Para isso, a função recebe inicialmente a 1-forma ω e um grupo de transporte G, encontrado com `getTransporterGroup`.

— * —

```
stabGroup ==> Record(matrix:Matrix(EXPR(INT)), equations:List(List(POLY(INT))))
getStab(form:der, normalizerList:List(transGroup)):List(stabGroup) ==
  finalRecord:List(stabGroup) := []
```

———-

A função `degreeLineField` é usada novamente aqui, para encontrar o grau do campo de retas associado a ω, que neste caso será o próprio grau de ω. Começaremos, então, a encontrar um estabilizador para o grupo de transporte G.

— * —

```
  degree := degreeLineField(form, [x,y])
  for i in 1..#normalizerList repeat
```

———-

Para os exemplos desta seção, utilizaremos os seguintes resultados obtidos anteriormente para σ, S e ω:

$$\sigma = \begin{bmatrix} I_1 & J_1 & K_1 \\ 0 & 1 & K_2 \\ 0 & 0 & K_3 \end{bmatrix}$$

$$S = [J_2 - 1,\ 2I_1K_2 - 2J_1 + I_1 - 1,\ I_1K_3 - 1,\ (2J_1 + 1)K_3 - 2K_2 - 1]$$

$$\omega = (-B_1y^2 + (B_1 - 2A_0)y + 2B_1x + 2B_0)\ dy + 2A_0\ dx$$

Como visto, nosso grupo de transporte é composto de uma transformação σ e um sistema de restrições S. O primeiro passo para encontrar o estabilizador é calcular o determinante da matriz σ, adicionando às restrições S o fato de que esse determinante não pode ser zero.

O determinante de σ é I_1K_3, portanto, é adicionado ao sistema S a expressão $I_1K_3t - 1$, representando o fato do determinante precisar ser diferente de zero, pois como já explicado anteriormente, precisamos que a matriz seja inversível.

Com a nova restrição adicionada, simplificamos o novo sistema com a função `groebnerFactorize`, e limpamos o resultado com `removeOnes`, ambas já explicadas anteriormente. Com isso, teremos o sistema S':

$$S' = [J_2 - 1,\ 2I_1K_2 - 2J_1 + I_1 - 1,\ I_1K_3 - 1,\ (2J_1 + 1)K_3 - 2K_2 - 1, t - 1]$$

— * —

```
eqList := normalizerList.i.equations
det := determinant(normalizerList.i.matrix)
eqList := cons(t*det-1, eqList)
groebEqList := groebnerFactorize(eqList)
groebEqList := removeOnes(groebEqList)
temp:transGroup := [normalizerList.i.matrix, eqList]
```

————-

Utiliza-se a função `matrixToSigma` para transformar σ da forma de matriz para lista. E novamente, como não estamos querendo impor nenhum tipo de restrição na transformação além das impostas pelo sistema S', precisamos homogeneizar ω, antes de realizar o seu pullback por σ, evitando perder informação sobre o que acontece com o campo de retas na reta do infinito.

$$\sigma = [K_1 z + J_1 y + I_1 x, K_2 z + y, K_3 z]$$

$$\omega = ((-2B_0 y - 2A_0 x)z^2 + ((-B_1 + 2A_0)y^2 - 2B_1 xy)z + B_1 y^3)\ dz + (2B_0 z^3 + ((B_1 - 2A_0)y + 2B_1 x)z^2 - B_1 y^2 z)\ dy + 2A_0 z^3\ dx$$

Infelizmente, a partir de agora, as 1-formas acabam sendo grandes demais para valer a pena mostrar aqui, mas continuemos. Após o pullback aplicado, obtemos a forma ω', e como estamos procurando uma transformação que deixe ω invariante, podemos calcular uma nova forma ω_0, tal que:

$$\omega_0 = \omega' - \lambda\omega$$

Logo, queremos que o valor de ω_0 seja igual a zero.

—— * ——

```
transform := matrixToSigma(temp.matrix)
homoForm := homogenizeForm(form, degree)
pullForm := pullback(transform, homoForm)
pmf := pullForm - lambda*homoForm
```

————-

A função `extractCoefficients` é então usada para extrair os coeficientes de dx, dy e dz de ω_0, usando em seguida a função `getCoefficients` para obter os coeficientes dos monômios de cada um desses coeficientes, adicionando-os às restrições de S'. É necessário também adicionar às restrições de S' o fato de que $\lambda \neq 0$.

Em seguida, mais uma vez simplificamos a lista de restrições utilizando a função `groebnerFactorize`, além de `removeOnes` e `removeExtraRestrictions`, que removerão os resultados triviais e restrições que imponham condições extras aos coeficientes da 1-forma, já que queremos que as restrições se apliquem apenas a σ.

É possível então retornar a matriz da transformação juntamente com o sistema de todas as suas restrições, tendo certeza agora de que temos um estabilizador da 1-forma ω que queríamos.

— * —

```
    formCoefs := extractCoefficients(pmf, 1)
    coefficientList := reduce(concat,_
    [getCoefficients(degree+1, formCoefs.j, true) for j in 1..3])
    finalEq := append(temp.equations, coefficientList)
    finalEq := cons(lambda*u-1, finalEq)
    grobFinalEq := groebnerFactorize(finalEq)
    grobFinalEq := removeOnes(grobFinalEq)
    grobFinalEq := removeExtraRestrictions(grobFinalEq, degree)
    temp2:stabGroup := [temp.matrix, grobFinalEq]
    finalRecord := cons(temp2, finalRecord)
  return(finalRecord)
```

———-

Ao final, podemos verificar a matriz σ_{stab} e o seu sistema de restrições S_{stab}:

$$\sigma_{stab} = \begin{bmatrix} I_1 & J_1 & K_1 \\ 0 & 1 & K_2 \\ 0 & 0 & K_3 \end{bmatrix}$$

$$S_{stab} = [u-1, t-1, \lambda-1, K_3-1, K_2-J_1, 2K_1-J_1^2+J_1, J_2-1, I_1-1]$$

Podemos aplicar as restrições de S_{stab} a σ_{stab}. Algumas que são fáceis de ver pelo

sistema são $I_1 = J_2 = K_3 = 1$, e $J_1 = K_2$, e $K_1 = \frac{J_1^2 - J_1}{2}$. Isso nos deixa com a matriz:

$$\sigma_{stab} = \begin{bmatrix} 1 & C & \frac{C^2-C}{2} \\ 0 & 1 & C \\ 0 & 0 & 1 \end{bmatrix}$$

A 1-forma que utilizamos para encontrar essa transformação, foi a mesma que encontramos no capítulo 1, usando a função `possibleForms` com a transformação:

$$\sigma = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Podemos verificar que se $C = 1$, σ e σ_{stab} são iguais. Portanto, é possível confirmar os resultados e verificar que realmente achamos um grupo estabilizador para a 1-forma ω.

6 CONCLUSÃO

A biblioteca conseguiu cumprir o que foi proposto inicialmente, sendo possível encontrar 1-formas invariantes por certa transformação, ou um grupo de transformações que deixam certa 1-forma invariante, com resultados internamente consistentes quando são comparados, indicando a sua corretude.

O paradigma de Literate Programming utilizado permitiu que a própria biblioteca fosse o texto apresentado durante essa dissertação, apresentado de maneira a ter um código fonte que é fácil de entender e construir sobre, sem precisar ler e decifrar linhas de código.

Como originalmente o trabalho não foi feito utilizando-se do Literate Programming, houveram diversas dificuldades em "traduzi-lo" para este paradigma, que tiveram que ser contornados durante a confecção do texto. Acredito que para trabalhos futuros, iniciar o código já utilizando-se do paradigma seria a melhor solução para obter um trabalho com a melhor qualidade possível.

Vejo o Literate Programming como sendo bastante útil para o meio acadêmico, promovendo a criação de código mais bem escrito, e espero que este trabalho possa ser uma inspiração para que mais alunos e docentes passem a utilizá-lo.

REFERÊNCIAS

[1] Caviness, B., Trager, B., e Gianni, P. Richard dimick jenks axiom developer and computer algebra pioneer biographical information. https://www.eecis.udel.edu/~caviness/jenks/jenksbio/. Acessado em 21/09/2018.

[2] Cerveau, D., Desérti, J., e et al. Géométrie classique de certains feuilletages de degré deux. *Bulletin of the Brazilian Mathematical Society 41*, 2 (2010), 161–198.

[3] Daly, T. Comentário em "ask hn: Why did literate programming not catch on?". https://news.ycombinator.com/item?id=10071207, Agosto 2015. Acessado em 21/09/2018.

[4] Daly, T. Readme do github do axiom. https://github.com/daly/axiom, Junho 2017. Acessado em 21/09/2018.

[5] Knuth, D. E. Literate programming. *The Computer Journal 27*, 2 (1984), 97–111.

[6] Knuth, D. E. *Literate Programming*. Center for the Study of Language and Inf, 1992.

[7] Wallace, M. The art of don e. knuth. https://www.salon.com/1999/09/16/knuth/, Setembro 1999. Acessado em 21/09/2018.

ANEXOS

ANEXO FUNÇÕES AUXILIARES – Necessárias para o funcionamento da biblioteca

- `Build Symbol List`

  Dado como entrada um grau, constroi uma lista de símbolos genéricos, correspondentes aos coeficientes dos monômios de dois polinômios genéricos do grau dado.

  – Entrada:

  $N = 2$

  – Saída:

  $L = [B_5, A_5, B_4, A_4, B_3, A_3, B_2, A_2, B_1, A_1, B_0, A_0]$

  — * —

```
buildSymbolList(degree:NNI):List(Symbol) ==
  symbolList:List(Symbol) := []
  quantity := (degree + 2) * (degree + 1) / 2

  i := 0
  while i < quantity repeat
    symbolList := cons(A[i], symbolList)
    symbolList := cons(B[i], symbolList)
    i := i + 1

  return(symbolList)
```

  ————-

- `Collect Variables`

Recebe uma lista de equações e uma lista de variáveis, retornando uma lista de variáveis que estão presentes nas equações, mas não estão contidas na lista recebida como parâmetro.

- Entrada:

  $L_1 = [a + b + c, b + c]$ $L_2 = [b]$

- Saída:

  $L = [a, c]$

—— * ——

```
collectVariables(L:List(POLY(INT)), V:List(Symbol)):List(Symbol) ==
  tempVars:List(Symbol):= []
  finalVars:List(Symbol):= []

  for i in 1..#L repeat
    tempVars:= append(tempVars, variables(L.i))
  tempVars:= removeDuplicates(tempVars)

  for i in 1..#tempVars repeat
    if ~member?(tempVars.i, V) then
      finalVars := append(finalVars, [tempVars.i])

  return(finalVars)
```

———-

- `Contains?`

  Dado um conjunto de matriz M_1 e equações S, e uma segunda matriz M_2, a função `contains?` irá verificar se M_2 está contida no conjunto identificado por M_1 e S.

  - Entrada:

$$M_1 = \begin{bmatrix} I_1 & J_1 & K_1 \\ 0 & 1 & K_2 \\ 0 & 0 & K_3 \end{bmatrix}$$

$$S = [J_2 - 1,\ 2I_1K_2 - 2J_1 + I_1 - 1,\ I_1K_3 - 1,\ (2J_1 + 1)K_3 - 2K_2 - 1]$$

$$M_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

– Saída:

$B = true$

— * —

```
contains?(S:transGroup, A:Matrix(EXPR(INT))):Boolean ==
  M:= S.matrix
  H1:List(EXPR(INT)) := [M(1,1)-A(1,1),M(2,1)-A(2,1),M(3,1)-A(3,1),_
  M(1,2)-A(1,2),M(2,2)-A(2,2),M(3,2)-A(3,2),M(1,3)-A(1,3),_
  M(2,3)-A(2,3),M(3,3)-A(3,3)]
  H2:= [elto::EXPR(FRAC(INT)) for elto in H1]
  H:= [elto::POLY(FRAC(INT)) for elto in H2]
  H:= append(H,S.equations)
  G:= groebner(H)
  if member?(1,G) then
    return(false)
  else
    return(true)
```

————-

- `Extract Coefficients`

  Recebendo uma 1-forma ou 2-forma ω, retorna os seus coeficientes. Para uma 1-forma:

  – Entrada:

  $\omega = a\ dx + b\ dy + c\ dz$

  - Saída:

    $L = [a, b, c]$

  É importante notar que nem sempre haverá um coeficiente para dz, neste caso, a lista de saída terá $c = 0$. Para uma 2-forma:

  - Entrada:

    $\omega = a\ (dx \wedge dy)$

  - Saída:

    $L = [a]$

```
          —— * ——
```

```
extractCoefficients(w:der, type:Integer):List(R) ==
  if type = 1 then
    dxCoef := coefficient(w, dx)
    dyCoef := coefficient(w, dy)
    dzCoef := coefficient(w, dz)
    coefList := [dxCoef, dyCoef, dzCoef]

  if type = 2 then
    dxdyCoef := coefficient(w, dx*dy)
    coefList := [dxdyCoef]

  return(coefList)
```

```
          ————————-
```

- `Generic Polynomial`

  Recebe como entrada um inteiro que representará o grau do polinômio, e um símbolo que servirá para representar os coeficientes das variáveis, retornando um polinômio com todos os coeficientes indeterminados.

- Entrada:

  N = 2

  S = A

- Saída:

  $P = A_5y^2 + A_4xy + A_3x^2 + A_2y + A_1x + A_0$

—— * ——

```
genericPolynomial(degree:NNI, variable:Symbol):POLY(INT) ==
  poly:POLY(INT) := 0
  coefNum := 0
  for i in 0..degree repeat
    for j in i..0 by -1 repeat
      poly := poly + (variable[coefNum] * x^j * y^((i-j)::NNI))
      coefNum := coefNum + 1
  return(poly)
```

————-

- `Get Max Degree`

  Retorna o grau do maior dos coeficientes de uma 1-forma recebida como entrada.

  - Entrada:

    $\omega = 3x^2\ dx + y\ dy$

  - Saída:

    $N = 2$

- `Get Coefficients`

  Recebe um polinômio, seu respectivo grau, e um booleano que indica se o polinômio é em (x, y), valor $false$, ou (x, y, z), valor $true$, retornando uma lista com todos os seus coeficientes de seus monômios.

– Entrada:

$P = A_5y^2 + (A_4x + A_2)y + A_3x^2 + A_1 + A_0\ N = 2\ B = false$

– Saída:

$L = [A_0, A_2, A_5, A_1, A_4, A_3]$

—— * ——

```
getCoefficients(degree:NNI, polynomial:POLY(INT),_
homogenized:Boolean):List(R) ==
  coefList:List(R) := []

  if homogenized then
    for i in 0..degree repeat
      for j in 0..degree while i+j <= degree repeat
        for k in 0..degree while i+j+k <= degree repeat
          coefList:=cons(coefficient(polynomial, [x,y,z], [i,j,k]),_
          coefList)
  else
    for i in 0..degree repeat
      for j in 0..degree while i+j <= degree repeat
        coefList:=cons(coefficient(polynomial, [x,y], [i,j]), coefList)

  coefList := reverse(coefList)
  return(coefList)
```

———————

—— * ——

```
getMaxDegree(w:der):NNI ==
  dxCoef := coefficient(w, dx)
  dxCoef := denominator(dxCoef) * dxCoef
  degree := totalDegree(dxCoef::POLY(INT), [x, y])
```

```
  dyCoef := coefficient(w, dy)
  dyCoef := denominator(dyCoef) * dyCoef
  if totalDegree(dyCoef::POLY(INT), [x, y]) > degree then
    degree := totalDegree(dyCoef::POLY(INT), [x, y])
  return(degree)
```

———————-

- `Get Variables`

  Recebe uma 1-forma e retorna todos os seus coeficientes indeterminados.

  – Entrada:

  $\omega = -B_1y^2 + (B_1 - 2A_0)y + 2B_1x + 2B_0\ dy + 2A_0\ dx$

  – Saída:

  $L = [A_0, B_0, B_1]$

———— * ————

```
getVariables(w:der):List(Symbol) ==
  poly1:=coefficient(w, dx)
  poly2:=coefficient(w, dy)
  variableList := variables(poly1)
  variableList := concat(variableList, variables(poly2))
  variableList := removeDuplicates(variableList)
  if member?(x, variableList) then
    variableList := delete!(variableList, position(x, variableList))
  if member?(y, variableList) then
    variableList := delete!(variableList, position(y, variableList))
  return (variableList)
```

———————-

- `Homogeneous Component`

Recebe como entrada um polinômio, um grau e uma lista de variáveis, retornando apenas a parte do polinômio que tem o grau dado nas variáveis escolhidas.

- Entrada:

  $P = A_5y^2 + A_4xy + A_3x^2 + A_2y + A_1x + A_0$

  N = 2

  V = $[x, y]$

- Saída:

  $P = A_5y^2 + A_4xy + A_3x^2$

—— * ——

```
homogeneousComponent(f:POLY(INT), n:NNI, lv:List(Symbol)):POLY(INT) ==
  comp:POLY(INT) := 0

  for i in 0..n repeat
   for j in 0..n | i+j = n repeat
     comp := coefficient(f,lv,[i,j]) * (lv.1)^i * (lv.2)^j + comp

  return(comp)
```

————-

- `Inner Product`

  Recebe dois polinômios, e tratando-os como vetores na base desses polinômios, faz o produto interno entre os dois, retornando um novo polinômio.

  - Entrada:

    $P_1 = -B_1y^2 + (B_1 - 2A_0)y + 2B_1x + 2B_0$

    $P_2 = A_5y^2 + A_4xy + A_3x^2 + A_2y + A_1x + A_0$

  - Saída:

    $P = -B_1A_5 + (B_1 - 2A_0)A_2 + 2B_1A_1 + 2B_0A_0$

```
—— * ——

innerProduct(poly1:POLY(INT), poly2:POLY(INT)):POLY(INT) ==
  grau:= max(totalDegree(poly1,[x,y]),totalDegree(poly2,[x,y]))
  dxPoly:POLY(INT) := 0
  i := 0
  while i <= grau repeat
    j := 0
    while j <= (grau - i) repeat
      dxPoly := dxPoly + coefficient(poly1, [x,y], [i,j]) * coefficient(poly2,_
      [x,y], [i,j])
      j := j+1
    i := i+1
  return(dxPoly)

————————-
```

- `Matrix to Sigma`

  Recebe uma transformação σ em formato de matriz, retornando a mesma transformação em forma de lista.

  – Entrada:

  $$\sigma_M = \begin{bmatrix} 10 & 6 & 0 \\ 0 & 4 & 2 \\ 0 & 0 & 2 \end{bmatrix}$$

  – Saída:

  $$\sigma_L = [10x + 6y, 4y + 2z, 2z]$$

```
—— * ——

matrixToSigma(matrix:Matrix(R)):List(R) ==
  sigma:List(R) := []
```

```
  for i in 1..3 repeat
    thisRow := row(matrix, i)
    thisMember := thisRow.1 * x + thisRow.2 * y + thisRow.3 * z
    sigma := cons(thisMember, sigma)

  sigma := reverse(sigma)
  return(sigma)
```

———-

- `Remove Extra Restrictions`

  Recebe uma lista de sistemas de equações e um grau. Se algum dos sistemas conter restrições em A_i e B_j, ele é removido da lista.

  – Entrada:

  $L = [[I_1 - K_2, I_3 + J_1, A_1], [I_2 + J_1]]\ N = 2$

  – Saída:

  $L' = [[I_2 + J_1]]$

—— * ——

```
removeExtraRestrictions(eqList:List(List(POLY(INT))),_
degree:NNI):List(List(POLY(INT))) ==
  L := []
  L2 := []

  poly1 := genericPolynomial(degree, A)
  poly2 := genericPolynomial(degree, B)

  poly1coef := getCoefficients(degree, poly1, false)
  poly2coef := getCoefficients(degree, poly2, false)

  for i in 1..#poly1coef repeat
```

```
    if poly1coef.i ~= 0 then
      L := cons(poly1coef.i, L)
    if poly2coef.i ~= 0 then
      L := cons(poly2coef.i, L)

  for i in 1..#eqList repeat
    test := true
    for j in 1..#L repeat
      if member?(L.j, eqList.i) then
        test := false
    if test = true then
      L2 := cons(eqList.i, L2)

  return L2
```

———-

- `Remove Ones`

  A função removeOnes tem como entrada uma lista de listas, removendo qualquer lista interna que contenha apenas o elemento [1], pois estas correspondem a sistemas sem solução.

  – Entrada:

  $L = [[A_0, A_1 + A_2], [1], [A_5]$

  – Saída:

  $L' = [[A_0, A_1 + A_2], [A_5]]$

—— * ——

```
removeOnes(list:List(List(POLY(INT)))):List(List(POLY(INT))) ==
  newList:List(List(POLY(INT))) := []

  for i in 1..#list repeat
```

```
    if list.i ~= [1] then
      newList := cons(list.i, newList)

  return(newList)
```

- `Remove Redundancy`

  Recebe uma lista de listas, cada uma representando um sistema. As listas são comparadas, encontrando sistemas que estejam contidos em outro. São removidos todos, exceto o mais geral, ou seja, o que impõe menos restrições.

  – Entrada:

  $L = [[A_1 + A_2, A_3], [A_1 + A_2, A_3, A_4], [B_1]]$

  – Saída:

  $L' = [[A_1 + A_2, A_3], [B_1]]$

```
removeRedundancy(list:List(List(POLY(INT)))):List(List(POLY(INT))) ==
  newList:List(List(POLY(INT))) := []

  for i in 1..#list repeat
    fail := false

    for j in 1..#list repeat
      test := true

      if ~fail and i ~= j then
        normalList := [normalForm(l, list.i) for l in list.j]

        for k in 1..#normalList repeat
          if normalList.k ~= 0 then
```

```
            test := false
          if test = true and k = #normalList then
            fail := true

    if ~fail then
      newList := cons(list.i, newList)

  return(newList)
```

———————-

- Remove Symbols

  A função removeSymbols recebe uma lista de polinômios, removendo todos os que são compostos por apenas um monômio.

  – Entrada:

    $L = [A_0 + A_2,\ A_1,\ A_3 - A_5]$

  – Saída:

    $L' = [A_0 + A_2,\ A_3 - A_5]$

———— * ————

```
removeSymbols(degree:NNI, equationsList:List(POLY(INT))):List(POLY(INT)) ==
  symbolsList := buildSymbolList(degree)
  newEquationsList:List(POLY(INT)) := []

  for i in 1..#equationsList repeat
    if ~member?(equationsList.i, symbolsList) then
      newEquationsList := cons(equationsList.i, newEquationsList)

  return(newEquationsList)
```

———————-

- `Remove Symbols From List`

  Recebe uma lista de equações L e uma lista de símbolos S, removendo de L todos os membros de S.

  - Entrada:

    $L = [A, B, C, D]$

    $S = [A, C]$

  - Saída:

    $L' = [B, D]$

—— * ——

```
removeSymbolsFromList(equationsList:List(POLY(INT)),_
symbolsList:List(Symbol)):List(POLY(INT)) ==
  newEquationsList:List(POLY(INT)) := []
  i := 1
  while i <= #equationsList repeat
    if member?(equationsList.i, symbolsList) = false then
      newEquationsList := cons(equationsList.i, newEquationsList)
    i := i + 1
  return(newEquationsList)
```

————-

- `Sigma to Matrix`

  Recebe uma transformação σ em forma de lista e retorna a mesma transformação em forma de matriz.

  - Entrada:

    $\sigma_L = [10x + 6y, 4y + 2z, 2z]$

  - Saída:

    $$\sigma_M = \begin{bmatrix} 10 & 6 & 0 \\ 0 & 4 & 2 \\ 0 & 0 & 2 \end{bmatrix}$$

—— * ——

```
sigmaToMatrix(sigma:List(R)):Matrix(R) ==
  size := #sigma
  sigMatrix:Matrix(R) := new (3, 3, 0)
  setColumn!(sigMatrix, 3, [0, 0, 1])

  for i in 1..size repeat
    poly1 := sigma.i::EXPR(FRAC(INT))
    poly := poly1::POLY(FRAC(INT))

    xCoef := coefficient(poly, [x], [1])
    yCoef := coefficient(poly, [y], [1])
    zCoef := coefficient(poly, [x,y,z], [0,0,0])
    if size = 3 then
      zCoef := coefficient(poly, [z], [1])
    setColumn!(sigMatrix, i, [xCoef, yCoef, zCoef])

  sigMatrix := transpose(sigMatrix)
  return(sigMatrix)
```

———-