Computación Simbólica

Half-GCD vs. Euclides

Ángel Ríos San Nicolás

31 de diciembre de 2020

1 Introducción

Vamos a comparar la eficacia en la práctica de dos algoritmos para el cálculo del máximo común divisor de polinomios con coeficientes en un cuerpo: el algoritmo de Euclides y un método que aprovecha el Half-GCD. Sabemos que en $\mathbb{Q}[X]$ estos algoritmos no evitan que crezcan los coeficientes de los polinomios en los cálculos intermedios. Como estamos interesados en compararlos en número de operaciones, tomaremos los polinomios con coeficientes en un cuerpo finito, donde no crecen los coeficientes, el coste de las operaciones es el mismo y podemos compararlos fijándonos en el tiempo de ejecución. En particular, fijaremos el anillo $\mathbb{F}_{101}[X]$ de polinomios con coeficientes en el cuerpo finito de 101 elementos. El código lo hemos programado en Sage y está recogido en el anexo.

2 Algoritmos del máximo común divisor de polinomios

2.1 Algoritmo recursivo mediante Half-GCD

Si denotamos $\mathbf{div}(f, g)$ al cociente de la división euclídea de f por g, podemos escribir el algoritmo del Half-GCD como sigue.

Entrada: $f, g \in \mathbb{K}[X]$ con $\deg(f) > \deg(g)$

Salida: $\mathrm{HGCD}(f, g) = R \in \mathscr{M}_2(R[X])$ tal que $R \begin{pmatrix} f \\ g \end{pmatrix} = \begin{pmatrix} f' \\ g' \end{pmatrix}$ con

$$\deg(f') > \left\lceil \frac{\deg(f)}{2} \right\rceil > \deg(g')$$

Inicialización: $m = \deg(f)$
si $\deg(g) < \left\lceil \frac{m}{2} \right\rceil$ **entonces**
  devolver I
fin
$a_0 = \mathbf{div}(f, x^m)$; $b_0 = \mathbf{div}(g, x^m)$
$R = \mathrm{HGCD}(a_0, b_0)$
$\begin{pmatrix} a' \\ b' \end{pmatrix} = R \begin{pmatrix} f \\ g \end{pmatrix}$
si $\deg(b') < m$ **entonces**
  devolver R
fin
$q = \mathbf{div}(a', b')$
$d = a' \mod b'$
$c = b'$
$l = \deg(c)$; $k = 2m - l$
$c_0 = \mathbf{div}(c', x^k)$; $d_0 = \mathbf{div}(d', x^k)$
$S = \mathrm{HGCD}(c_0, d_0)$
devolver $R \cdot \begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix} \cdot S$

El Half-GCD permite, multiplicando por la inversa, calcular dos restos consecutivos del algoritmo de Euclides y además con la propiedad de calculamos el primer resto cuyo grado es menor que la mitad del de f. Aprovechando el algoritmo anterior, calculamos el máximo común divisor.

Entrada: $f, g \in \mathbb{K}[X]$
Salida: $\gcd(f, g)$
si $f \mod g = 0$ **entonces**
  devolver g
fin
$R = \mathrm{HGCD}(f, g)$
$\begin{pmatrix} b_0 \\ b_1 \end{pmatrix} = R^{-1} \begin{pmatrix} f \\ g \end{pmatrix}$
si $b_0 \mod b_1 \neq 0$ **entonces**
  devolver b_1
fin
en otro caso
  $\gcd(b_1, b_0 \mod b_1)$
fin

2.2 Algoritmo de Euclides

El algoritmo de Euclides clásico para el cálculo del máximo común divisor es el siguiente.

Entrada: $f, g \in \mathbb{K}[X]$
Salida: $\gcd(f, g)$
Inicialización: $r_0 = f$; $r_1 = g$
mientras $r_1 \neq 0$ **hacer**
  $r_0 = r_1$
  $r_1 = r_0 \mod r_1$
fin
devolver r_0

3 Comparación de los métodos en $\mathbb{F}_{101}[X]$

Como en el Half-GCD tenemos $\deg(f) > \deg(g)$, para probar los algoritmos fijamos un grado para f y vamos aumentando el de g.

3.1 Polinomios pseudoaleatorios

Si escogemos f y g de grados fijos aleatoriamente con distribución uniforme en $\mathbb{F}_{101}[X]$, es de esperar que sean coprimos. Vamos a ver que, en la práctica el algoritmo del Half-GCD es eficiente en la práctica para grados muy grandes.

Empezamos con grado 100 para f y aumentamos el grado de g. Observamos que tarda más en terminar el algoritmo de del Half-GCD.

```
K.<x> = PolynomialRing(GF(101))
f = K.random_element(100)
g = K.random_element(20)
%time print(euclid_gcd(f,g))
%time print(MIGCD(f,g))

4
CPU times: user 243 µs, sys: 0 ns, total: 243 µs
Wall time: 209 µs
4
CPU times: user 2.61 ms, sys: 0 ns, total: 2.61 ms
Wall time: 2.58 ms
```

```
K.<x> = PolynomialRing(GF(101))
f = K.random_element(100)
g = K.random_element(80)
%time print(euclid_gcd(f,g))
%time print(MIGCD(f,g))

8
CPU times: user 290 µs, sys: 70 µs, total: 360 µs
Wall time: 364 µs
8
CPU times: user 11.9 ms, sys: 0 ns, total: 11.9 ms
Wall time: 11.3 ms

K.<x> = PolynomialRing(GF(101))
f = K.random_element(100)
g = K.random_element(99)
%time print(euclid_gcd(f,g))
%time print(MIGCD(f,g))

89
CPU times: user 262 µs, sys: 1e+03 ns, total: 263 µs
Wall time: 266 µs
89
CPU times: user 12.8 ms, sys: 2 µs, total: 12.8 ms
Wall time: 12.7 ms
```

Aumentamos el grado de f a 1000 y seguimos teniendo la misma tendencia, el algoritmo de Euclides es mucho más eficiente y menos sensible al cambio.

```
K.<x> = PolynomialRing(GF(101))
f = K.random_element(1000)
g = K.random_element(20)
%time print(euclid_gcd(f,g))
%time print(MIGCD(f,g))

87
CPU times: user 280 µs, sys: 68 µs, total: 348 µs
Wall time: 260 µs
87
CPU times: user 2.47 ms, sys: 0 ns, total: 2.47 ms
Wall time: 2.47 ms

K.<x> = PolynomialRing(GF(101))
f = K.random_element(1000)
g = K.random_element(800)
%time print(euclid_gcd(f,g))
%time print(MIGCD(f,g))

49
CPU times: user 3.01 ms, sys: 0 ns, total: 3.01 ms
Wall time: 2.98 ms
49
CPU times: user 108 ms, sys: 0 ns, total: 108 ms
Wall time: 106 ms
```

```
K.<x> = PolynomialRing(GF(101))
f = K.random_element(1000)
g = K.random_element(999)
%time print(euclid_gcd(f,g))
%time print(MIGCD(f,g))

39
CPU times: user 4.99 ms, sys: 2 µs, total: 4.99 ms
Wall time: 4.81 ms
39
CPU times: user 130 ms, sys: 3 µs, total: 130 ms
Wall time: 130 ms
```

Si el grado de f es 10^5, si el grado de g es, por ejemplo 9000 o $5 \cdot 10^4$, sigue siendo mejor el algoritmo de Euclides, pero si el grado es $6 \cdot 10^4$ el algoritmo que utiliza el Half-GCD es más rápido.

```
K.<x> = PolynomialRing(GF(101))
f = K.random_element(100000)
g = K.random_element(50000)
%time print(euclid_gcd(f,g))
%time print(MIGCD(f,g))

52
CPU times: user 6.26 s, sys: 0 ns, total: 6.26 s
Wall time: 6.26 s
52
CPU times: user 6.72 s, sys: 0 ns, total: 6.72 s
Wall time: 6.72 s

K.<x> = PolynomialRing(GF(101))
f = K.random_element(100000)
g = K.random_element(60000)
%time print(euclid_gcd(f,g))
%time print(MIGCD(f,g))

71
CPU times: user 9.09 s, sys: 0 ns, total: 9.09 s
Wall time: 9.09 s
71
CPU times: user 8.1 s, sys: 0 ns, total: 8.1 s
Wall time: 8.09 s

K.<x> = PolynomialRing(GF(101))
f = K.random_element(100000)
g = K.random_element(99999)
%time print(euclid_gcd(f,g))
%time print(MIGCD(f,g))

58
CPU times: user 25.6 s, sys: 0 ns, total: 25.6 s
Wall time: 25.6 s
58
CPU times: user 13.9 s, sys: 0 ns, total: 13.9 s
Wall time: 13.9 s
```

Con esto observamos que, en este caso, con esta implementación, el algoritmo que utiliza el Half-GCD solo sería más eficiente para grados muy grandes, mayores que 10^5 para f y $5 \cdot 10^4$ para g, con los que las ejecuciones pueden tardan más de 8 segundos. Observamos también que se acentúa la mejoría con el grado ya que pasamos de una diferencia de 1 segundo a una de 12 segundos.

3.2 Máximo común divisor de grado al menos $\left\lceil \frac{\deg(f)}{2} \right\rceil$

Si tomamos $f, g \in \mathbb{F}_{101}[X]$ con $\deg(f) = n$ de manera que $\deg(g) < n$ y $\gcd(f, g) \geq \lceil \frac{n}{2} \rceil$, entonces, claramente no son coprimos. Es de esperar que en este caso el Half-GCD sea más eficiente en la práctica porque precisamente está diseñado para calcular de manera más rápida hasta el resto de grado menor que $\frac{n}{2}$ recursivamente y verificar que es el máximo común divisor.

4 Mejora: Generalización a grados cualesquiera

Una mejora funcional que se puede hacer al algoritmo del GCD con el Half-GCD permite que se pueda aplicar para polinomios f y g del mismo grado. La idea es hacer una división euclídea con la que construimos los polinomios q y r tales que

$$f = gq + r \qquad \text{con } \deg(r) < \deg(g)$$

y, como $\gcd(f, g) = \gcd(g, r)$, podemos aplicar el algoritmo a g y $r = f \mod g$. El método quedaría entonces:

Entrada: $f, g \in \mathbb{K}[X]$
Salida: $\gcd(f, g)$
si $\deg(f) = \deg(g)$ **entonces**
  $r = f \mod g$
  devolver $\gcd^*(g, r)$
fin
si $\deg(g) > \deg(f)$ **entonces**
  devolver $\gcd^*(g, f)$
fin
devolver $\gcd^*(f, g)$

donde $\gcd^*$ es el máximo común divisor calculado con el algoritmo anterior que aprovecha el Half-GCD. La implementación del algoritmo es trivial.

ANEXO: Código en SAGE

Producto de matrices 2×2 como listas

```
def prod(A, B):  # Calcula el producto de dos matrices 2x2 como lista.
    A = list(A)  # Si A y B son matrices, las pasamos a listas.
    B = list(B)
    return [(A[0][0] * B[0][0] + A[0][1] * B[1][0],
             A[0][0] * B[0][1] + A[0][1] * B[1][1]),
            (A[1][0] * B[0][0] + A[1][1] * B[1][0],
             A[1][0] * B[0][1] + A[1][1] * B[1][1])]
```

Half-GCD

```
def HGCD(a, b):  # Calcula el Half-GCD de los polinomios a, b.
    K = a.parent()  # Anillo de polinomios de a y b.
    dega = a.degree() # Grados de a y b.
    degb = b.degree()
    m = (dega / 2).ceil()
    if degb < m:  # Si el grado de b es menor que m, es la identidad.
        return matrix(K, 2, [1, 0, 0, 1])
    a0 = a.shift(-m)  # Cocientes de a y b por x^m.
    b0 = b.shift(-m)
    R = HGCD(a0, b0)  # Calculamos el HGCD recursivamente.
    r11 = R[0][0]  # Guardamos las componentes de la matriz R.
    r12 = R[0][1]
    r21 = R[1][0]
    r22 = R[1][1]
    d = (r11 * r22 - r12 * r21)  # R^-1*(a,b)'.
    a1 = (d * (a * r22 - b * r12))
    b1 = (d * (b * r11 - a * r21))
    if b1.degree() < m:  # Si el grado de b1 es menor que m, devuelve R.
        return R
    qt, d = a1.quo_rem(b1)  # Si no, hacemos un paso de Euclides.
    c = b1
    l = c.degree()
    k = 2 * m - l  # Tomamos k para que la recursion funcione.
    c0 = c.shift(-k)
    d0 = d.shift(-k)
    S = HGCD(c0, d0)  # Calculamos el HGCD recurisvamente.
    RM = matrix(K, 2, prod(R, matrix(K, 2, [qt, K(1), K(1), K(0)])))
    Q = matrix(K, 2, prod(RM, S))  # Matriz R * [qt, 1, 1, 0] * S.
    return Q
```

GCD con Half-GCD

```
def MIGCD(f,g):  # Calcula el gcd(f,g) aprovechando el HGCD.
    K = f.parent()
    a = f  # No queremos que modifique la entrada.
    b = g
    if a % b == 0:  # Si f es divisible por g, GCD(f,g) = g.
        return b
    R = HGCD(a, b)  # Matriz R del HGCD.
    r11 = R[0][0]  # Guardamos las componentes de la matriz R.
    r12 = R[0][1]
    r21 = R[1][0]
    r22 = R[1][1]
    d = (r11 * r22 - r12 * r21)  # det(R)
    b0 = (d * (a * r22 - b * r12))  # R^-1*(f,g)'
    b1 = (d * (b * r11 - a * r21))
    rem = b0 % b1  # Resto del cociente de b0 entre b1.
    if rem == 0:  # Si b0 es divisible por b1, GCD(f,g) = b1.
        return b1
    else:  # Si no, calculamos recursivamente el GCD con el resto.
        return MIGCD(b1, rem)
```

Algoritmo de Euclides

```
def euclid_gcd(f, g):  # Calcula gcd(f,g) con el algoritmo de Euclides.
    r0 = f
    r1 = g
    while r1 != 0:
        r0, r1 = r1, r0%r1
    return r0
```

Bibliografía

[1] THULL, Klaus, CHEE K. Yap, 1990. *A unified approach to HGCD Algorithms for Polynomials and integers.*