Google Summer Of Code – 2023

Improving Polynomial GCD

Abhishek Patidar

Abstract

The project aims to add new algorithms for computing the greatest common divisor (GCD) of polynomials in the sparse representation in order to improve the speed of many parts of sympy such as matrices, solvers, integration, and simplification functions. Currently, the slow speed of polynomial GCD impacts many operations in SymPy, leading to heavy performance penalties and avoiding the use of domain elements in some algorithms. By improving the speed of polynomial GCD algorithms, many parts of SymPy could become faster, and the use of GCD could become more widespread. This project could significantly enhance the overall performance and usability of SymPy.

Table of Contents

1. About Me

1.1 Name and Contact Info

- **Name**: Abhishek Patidar
- **University**: Shri G. S. Institute of Technology & Science (SGSITS), Indore
- **Degree**: Bachelor of Technology
- **G-Mail**: 1e9abhi1e10@gmail.com
- **GitHub Handle**: 1e9abhi1e10
- **Blog**:
- **Time Zone**: IST (UTC +5:30)

1.2 Personal Background & Programming Experience

I am Abhishek Patidar, and I am a second-year undergraduate student. I have been programming in Python for the last two years, and I also have experience in other programming languages such as C++, C, Java, and JavaScript. I prefer using git for version control, and I usually work on Windows Operating System. My primary code editor is Visual Studio Code, which I find easy to use and have been using since my early programming days. Its numerous advanced features, including Intellisense, make it my preferred editor. Additionally, I have background knowledge in Differential Calculus, Linear Algebra, and Abstract Algebra. As a Competitive Programmer, I possess adequate knowledge of Data Structures and Algorithms in Python

1.3 My Motivation

I've always been fascinated by solving math problems accurately and quickly, especially algebraic equations. That's why I was interested in SymPy, a library that helps solve these problems. I found the Polys module of SymPy very powerful and impressive because of its advanced techniques and algorithms. So, I decided to explore it further by checking out its solver's module, which helped me better understand and appreciate all the features Polys has to offer. The Poly module in SymPy provides powerful tools for working with polynomial expressions, including polynomial arithmetic, factorization, and solving polynomial equations. The module allows users to perform a range of operations on polynomials, such as finding the degree, roots, and coefficients of a polynomial expression. Additionally, the Poly module includes various advanced algorithms and techniques for working with polynomials, such as the Fast Fourier Transform algorithm for multiplication and the Extended Euclidean Algorithm for computing greatest common divisors. Overall, the Poly module in SymPy provides a comprehensive and sophisticated set of tools for working with polynomial expressions, making it an essential component of the SymPy library.

My favorite feature in SymPy is the cancel function in the polys module. It simplifies polynomial expressions by factoring out common terms and canceling out common factors in the numerator and denominator. This feature has been extremely useful in my work with SymPy. Here's an example to illustrate how it works:

```
>>> import sympy as sp

>>> x, y = sp.symbols('x y')

>>> frac = (x**2 - 1)/(x**2 + 2*x + 1)

>>> simplified_frac = sp.cancel(frac)

>>> print("Original fraction: ", frac)
Original fraction:  (x**2 - 1)/(x**2 + 2*x + 1)

>>> print("Simplified fraction: ", simplified_frac)
Simplified fraction:  (x - 1)/(x + 1)

>>> sp.pprint(simplified_frac)

x - 1
```

```
-------

x + 1
```

1.4 Contributions

I started using SymPy and studying the code base in November and made my first contribution in Mid-November. I have been constantly contributing and learning new skills from the community. I have created many pull request, many of which got merged. It has been a beautiful Journey till now and willing to make it more fruitful. I am excited to continue contributing to Sympy and to help make a positive impact on the open-source community.

Here are the list of merged/open/closed **Pull Requests**, and open/closed **Issues**.

The list has been arranged in order that the*y* were created.

1.5 Pull Requests

Merged

- #24325 **:** Numerical error on Conversion of coulomb to statcoulomb
- #24370 **:** Fix Floor division with sympy.Integer
- #24401 **:** fixes puiseux polynomails doen not work with algebraic extensions
- #24414 **:** Fixes the boundary of "LessThan" becomes open in sets()
- #24435 **:** Correct documentation for interpolating_spline
- #24457 **:** Corrected the template to give better bug example
- #24463 **:** Adding eletron_rest_mass unit in physics.units
- #24488 **:** Improves the docstring of rigid body
- #24503 **:** Improves typos in doctstring

- #24562 **:** fixes rational calc value error
- #24573 **:** Added test case for syntax error on lambdify
- #24602 **:** Added test in test_solvers
- #24630 **:** Added angstrom unit in physics.units
- #24654 **:** Added test for float('inf') in test_simplify
- #24460 **:** Added tests in test test_refine to simplify piecewise function
- #24661 **:** Fixes the evaluate=False for relational in parse_expr
- #24666 **:** Added test for simplified Piecewise is missing conditions
- #24683 **:** Added test for Jordan_form() in test_eigen
- #24688 **:** Added test for numpy.array is commutative
- #24750 **:** Test case added for numerical evaluation in test_simplify
- #24756 **:** Added test for Nested Add with negative values
- #24800 **:** Fixes nsolve for inequality issue
- #24867 : Fixes diff and Derivative different results in matrix expressions

Open

- #24399 **:** Fixes invert on a symbol returns 0
- #24450 **:** Adding transverse_magnification for mirror and lens in physics.optics
- #24512 **:** Replaces "sympifyable" with "sympifiable"
- #24532 **:** Printing: Fixes UnevaluatedExpr.is_commutative is None
- #24621 **:** Improved discrepancies in find_unit

- #24710 **:** Fixes wrong result of a Matrix with quantum operators
- #24716 : Fixes is_tangent in ellipse.py
- #24760 : Splitted out the secondary license into second file
- #24768 : Add new method orient_from_dcm i.e. more intuitive than orient_explicit()
- #24771 : Replaces "==" with "is" in tests
- #24810 : Fixes RecursionError for unevaluated expression in latex

Closed

- #24295 : Fix evaluate=False parameter to parse_expr is ignored for relationals
- #24297 **:** The evaluate=False parameter to parse_expr is ignored for relational
- #24425 **:** Added angstrom unit in physics.unit
- #24467 **:** Fixed typos in sympy codebase
- #24809 **:** Fixes RecursionError for unevaluated expression

1.6 Reviewed Pull Requests

- #24329 **:** Fix printing of tensors with imaginary expressions
- #24351 **:** Issue #24304 fix
- #24362 **:** corrected the list of strictly increasing integer in bsplines to list of real values corresponding to x
- #24423 **:** corrected the documentation
- #24441 **:** fixed the importing of get_all_known_facts and get_known_fact
- #24468 **:** Fixed the typos in the codebase
- #24551 : Fix typo (paramatrization -> parametrization)
- #24556 **:** Issue #24304 fix (2nd try with pull request)

- **#24558 :** added bilinear() to tf in control lti
- **#24576 :** gh-24140: Fixed some broken Wikipedia links to anchor tags
- **#24615 :** better introductory documentation for units module
- **#24638 :** Fix MatMul(x, OneMatrix).diff(x) RecursionError
- **#24680 :** changed year in licenses to 2023
- **#24772 :** Fix as_coeff_Mul for Float(0.0)
- **#24794 :** Use explicit keyword argument in intersection._new_
- **#24869 :** Update mul.py
- **#24876 :** Add my name and email address to mailmap

1.7 Issues

Open

- #24407 **:** Unrelated problem in physics.units
- #24438 **:** Lens or Mirror not mentioned in transverse_magnification in physics.optics
- #24487 **:** Use of n(refractive index) in waves
- #24671 **:** Producing wrong result of a Matrix with quantum operators on applying transpose

Closed

- #24262 **:** Not showing sin(360) = 0, showing -2.44929359829471e-16.
- #24309 **:** Getting wrong result on finding the Laplace transform of sin(t)
- #24319 **:** Numerical error on conversion of coulomb to statcoulomb
- #24381 **:** Numerical error in conversion of volt and statvolt
- #24405 **:** sympy.physics.units doesn't support Temperature scales
- #24462 **:** Missing electron rest mass unit in physics.units

1.8 Discussions

- #24479 : Use of n(refractive index) in waves
- #24489 : Difference between "sympifiable" and "Sympifyable" in docstring
- #24655 : Express the results in sympy.core.relaional.StrictLessThan
- #24672 : wrong results in test_commonmatrix

2 The Project

In this section I will explain my project details and its vision. I expect this structure to be considerably enhanced under the guidance of my mentor.

2.1 Project Outline and Motivation

This project aims to improve the performance of many operations in SymPy by adding new algorithms for computing the greatest common divisor (GCD) of polynomials in the sparse representation.

Motivation: The current sparse polynomial GCD algorithm in SymPy is slow, particularly when there are many generators. This slow speed of polynomial GCD impacts many operations in SymPy, such as domain matrix calculations, solvers, integration, and simplification functions. In some cases, the heurisch integration algorithm avoids using domain elements to avoid the cost of slow sparse GCD and instead pays a heavy performance penalty to switch back and forth between the domain element representation and the Expr representation. By improving the speed of polynomial GCD algorithms, many parts of SymPy could become faster, and the use of GCD could become more widespread.

Expected Outcome: The project is expected to significantly improve the performance of many operations in SymPy that use polynomial GCD. This could make SymPy more efficient and user-friendly, particularly for complex symbolic calculations. The new algorithms could also facilitate the wider use of GCD in SymPy, leading to more robust and sophisticated mathematical models.

What is Dense Polynomial representation?

A multivariate polynomial (a polynomial in multiple variables) can be represented as a polynomial wit coefficient that are themselves polynomials.

For example x**2*y + x**2 + x*y + y + 1 can be represented as polynomial in x where the coefficients are themselves polynomials in y i.e.: (y + 1)*x**2 + (y)*x + (y+1). Since we can represent a polynomial with a list of coefficients a multivariate polynomial can be represented with a list of lists of coefficients:

```
>>> from sympy import symbols

>>> x, y = symbols('x, y')

>>> p = Poly(x**2*y + x**2 + x*y + y + 1)

>>> p

Poly(x**2*y + x**2 + x*y + y + 1, x, y, domain='ZZ')

>>> p.rep.rep

[[1, 1], [1, 0], [1, 1]]
```

This list of lists of (lists of…) coefficients representation is known as the "dense multivariate polynomial" (DMP) representation

What is Sparse polynomial representation?

Instead of lists we can use a dict mapping nonzero monomial terms to their coefficients. This is known as the "sparse polynomial" representation. We can see what this would look like using the **as_dict()** method:

```
>>> Poly(7*x**20 + 8*x + 9).rep.rep

[7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 9]



>>> Poly(7*x**20 + 8*x + 9).as_dict()

{(0,): 9, (1,): 8, (20,): 7}as_dict()
```

The keys of this dict are the exponents of the powers of x and the values are the coefficients so e.g. 7*x**20 becomes (20,): 7 in the dict. The key is a tuple so that in the multivariate case something like 4*x**2*y**3 can be represented as (2, 3): 4.

The sparse representation can be more efficient as it avoids the need to store and manipulate the zero coefficients. With a large number of generators (variables) the dense representation becomes particularly inefficient and it is better to use the sparse representation:

```
>>> from sympy import prod
>>> gens = symbols('x:10')
>>> gens
(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9)
>>> p = Poly(prod(gens))
>>> p
Poly(x0*x1*x2*x3*x4*x5*x6*x7*x8*x9, x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, domain='ZZ')
>>> p.rep.rep
[[[[[[[[[[1, 0], []], [[]]], [[[]]]], [[[[]]]]], [[[[[]]]]]], [[[[[[]]]]]]], [[[[[[[]]]]]]]], [[[[[[[[]]]]]]]]], [[[[[[[[[]]]]]]]]]]
>>> p.as_dict()
{(1, 1, 1, 1, 1, 1, 1, 1, 1, 1): 1}
```

In SymPy, improving the speed of polynomial gcd algorithms would have a significant impact on the performance of many parts of the system. Currently, some parts of SymPy avoid using the gcd algorithm directly because it is too slow. However, if the gcd algorithm were faster, it would be possible to use it more widely throughout the system.

For instance, the Heurisch algorithm in SymPy deliberately avoids using domain elements because it is too costly to perform sparse gcd with them. Instead, it incurs a heavy performance penalty by switching back and forth between the domain element representation and the Expr representation. With a faster gcd algorithm, the Heurisch algorithm could use domain elements more frequently, resulting in a significant performance improvement.

Current Status of GCD of Sparse Polynomials in SymPy

In SymPy currently, It checks the type of coefficients the polynomials have and selects the appropriate GCD algorithm. If the coefficients are rational numbers, it converts them to integers and then computes the GCD using the integer GCD algorithm. If the coefficients are integers, it uses the HEUGCD algorithm to compute the GCD. If the coefficients are of some other type, it uses a dense representation algorithm to compute the GCD.

```
def _gcd(f, g):
    ring = f.ring
    if ring.domain.is_QQ:
        return f._gcd_QQ(g)
    elif ring.domain.is_ZZ:
        return f._gcd_ZZ(g)
    else: # TODO: don't use dense representation (port PRS algorithms)
        return ring.dmp_inner_gcd(f, g)
def _gcd_ZZ(f, g):
    return heugcd(f, g)
def _gcd_QQ(self, g):
    f = self
    ring = f.ring
    new_ring = ring.clone(domain=ring.domain.get_ring())
    cf, f = f.clear_denoms()
    cg, g = g.clear_denoms()
    f = f.set_ring(new_ring)
    g = g.set_ring(new_ring)
    h, cff, cfg = f._gcd_ZZ(g)
```

This code defines two functions _gcd and _gcd_QQ, which calculate the greatest common divisor (GCD) of two polynomials in different domains. The _gcd function determines the domain of the input polynomials and calls the appropriate GCD function based on that domain. If the domain is rational numbers (QQ), _gcd calls _gcd_QQ, which first converts the input polynomials to a new domain with integer coefficients (ZZ) and then calculates the GCD using _gcd_ZZ. The _gcd_ZZ function calculates the GCD of two polynomials with integer coefficients using the HEUGCD algorithm. If the input domain is not rational numbers or integer numbers, _gcd currently does not handle the calculation and prints a TODO message.

There are two issues with this implementation:

1. HEUGCD is probably not the best algorithm even if the domain is ZZ: The HEUGCD algorithm is an efficient algorithm for computing the GCD of two polynomials over a finite field. However, its efficiency decreases when it is applied over the ring of integers (ZZ) or over other rings of algebraic integers. The reason behind this inefficiency is that the algorithm is designed to work well when the degree of the input polynomials is much smaller than the characteristic of the underlying field. In contrast, over ZZ, the degree of the input polynomials can be arbitrarily large, which leads to the algorithm's suboptimal performance. Therefore, it is recommended to use other algorithms, such as the subresultant algorithm or the modular GCD algorithm, for computing the GCD of polynomials over the ring of integers.

2. The dense implementation of PRS is very inefficient if the polynomial ring has many generators: The polynomial remainder sequence (PRS) is a well-known algorithm for computing the GCD of polynomials over a field. However, the dense implementation of PRS can be very inefficient if the polynomial ring has many generators. In particular, the degree of the intermediate polynomials that are computed during the PRS computation can be very high, which makes the algorithm's running time exponential in the number of generators. Therefore, it is recommended to use a sparse implementation of PRS which exploit the sparsity structure of the input polynomials to reduce the number of intermediate polynomials that need to be computed. The sparse implementation of PRS can significantly improve the algorithm's efficiency, especially for polynomials with many generators

The GCDHEU algorithm is a polynomial GCD algorithm that operates by evaluating the polynomials at large integers rather than finite fields. The approach is less efficient than the finite field approach for handling large polynomial problems. However, well designed program can quickly detect a failing case, and it can switch to an alternative algorithm to avoid coefficient growth.

The GCDHEU algorithm operates by reducing the number of variables in a polynomial problem. It assigns values to each variable and generates a pair of large integers. The algorithm constructs an approximate polynomial GCD using rn-adic interpolations one variable at a time, starting from the GCD of the two large integers. The algorithm then either returns the GCD of two polynomials and its cofactors or signals a failure

One of the limitations of the GCDHEU algorithm is that it requires larger evaluation points, which inevitably leads to calculations in the bignum regime. As the degree of the inputs increases, the size of the evaluation point also increases, making GCDHEU less effective for large polynomials. Additionally, the algorithm is not efficient in handling large problems. Therefore, it is not suitable for use in applications where large polynomial problems need to be solved.

The GCDHEU algorithm's strength lies in its ability to detect a failing case quickly and switch to an alternative algorithm. This feature helps avoid coefficient growth, which is a common problem in polynomial GCD algorithms. By avoiding coefficient growth, the algorithm can provide an accurate GCD of two polynomials without the need for excessive computational resources.

In summary, the GCDHEU algorithm is a polynomial GCD algorithm that operates by evaluating the polynomials at large integers. While it is not efficient for handling large problems, it can quickly detect a failing case and switch to an alternative algorithm to avoid coefficient growth. Therefore, the algorithm is useful in applications where large polynomial problems need to be solved, and computational resources are limited.

Some cases where HEUGCD is faster than other algorithms:

1. Completely dense non-monic quadratic inputs with dense non-monic linear GCD’s
2. Quadratic non-monic GCD, F and G have other quadratic factors (for degree < 5)
3. Trivariate polynomials whose GCD has common factors with its cofactors (for small examples)

Some cases where HEUGCD is slower than other algorithms:

1. Linearly dense quartic inputs with quadratic GCDs
2. Sparse GCD and inputs where degree are proportional to the number of variables
3. Quadratic non-monic GCD, F and G have other quadratic factors (for degree > 4)

4. Sparse non-monic inputs with linear GCDs
5. Trivariate inputs with increasing order
6. Trivariate polynomials whose GCD has common factors with its cofactors (for large examples)

Here an example that what is slow:

```
from sympy import symbols

xs = symbols("x:50")

K = ZZ_I.poly_ring(*xs)

p1 = K.from_sympy(Add(*(xi * I ** ni for ni, xi in enumerate(xs))))

p2 = p1 ** 2

%time ok = p1.gcd(p2)
```

```
CPU times: total: 2.56 s

Wall time: 7.29 s
```

2.2 The Plan

2.2.1 Phase 1 (2 week)

- **Implementation of Subresultant PRS algorithm for Sparse representation**
- **Adding the documentation and tests**

2.2.2 Phase 2 (3 week)

- **Implementation of Zippel's Algorithm**
- **Adding the documentation and tests**

2.2.3 Phase 3 (3 week)

- **Implementation of SparseModGcd algorithm**
- **Adding the documentation and tests**

2.2.4 Phase 4 (4 week)

- **Explore and Add benchmark for above algorithms**

- **Implement a logic for different strategies**
- **Improving the speed matrix inverse in DomainMatrix**

2.2.1 Phase 1

Implementation of Subresultant PRS Algorithm for Sparse representation

The Subresultant Polynomial Remainder Sequences (PRS) algorithm is a method for computing the greatest common divisor (GCD) of two polynomials. This algorithm is based on the Euclidean algorithm, but it makes use of subresultant to avoid expensive polynomial divisions.

In this section, will introduce a recursive formula for computing the polynomial $ß_i$ such that $F_i = G_i$ for *i = 3 to k.* We will then verify and discuss the formula. Additionally, we will transform the formula into an algorithmic form, which we will refer to as the subresultant PRS algorithm.

Given primitive polynomial F_1, F_2 *in I[x]*. we can obtain the subresultant PRS,

$G_1 = F_1, G_2 = F_2, G_3, \ldots\ldots G_k$, by computing

$G_3 = (-1)^{\delta i+1}$ *prem* (G_1, G_2)

$G_i = (-1)^{\delta i+1}$ *prem* (G_{i-2}, G_{i-1}), $i = 4\ldots., k$

Where $g_i = lc(G_i)$ for $i = 1, \ldots.., k$ and

$h_2 = g^{\delta_1}{}_2$, $h_i = g^{\delta_{i-1}}{}_i\ h^{1-\delta_{1-1}}{}_{i-1}$, $i = 3\ldots, k$

The iteration stops because *prem* $(G_{k-1}, G_k) = 0$ then if $n_k > 0$, we have

$gcd(F_1, F_2) = pp\,(G_k)$, $res(F_1, F_2) = 0$

Where *pp* denotes the primitive part, while if $n_k = 0$, we have

$gcd(F_1, F_2) = 1$, $res(F_1, F_2) = h_k$

Let *f(x)* and *g(x)* be two sparse polynomials with coefficients in a field *K*. The degree of *f(x)* is *n* and the degree of *g(x)* is *m*, where *n* >= *m*. We want to find the GCD of *f(x)* and *g(x)*, denoted as gcd(*f(x), g(x)).*

The first step of the algorithm is to compute the PRS of $f(x)$ *and* $g(x)$, denoted as where $\{q_k(x)\}$ is a polynomial of degree of n-k, k=0, 1...,m-1. The PRS is defined as

$q_0(x) = f(x)$, $q_1(x) = g(x)$

$q_k+1(x) = -1k * (q_k-1(x) \% q_k(x))$, *for* k = 1, 2..., m-1

where % denotes the polynomial remainder operation. Note that the subresultant PRS algorithm only requires the remainder operation, which is much faster than the division operation.

The next step is to compute the content of the PRS denoted as c_k, which is the GCD of the coefficient of $q_k(x)$. The content can be computed recursively as:

$c_0 = content(f(x))$

$c_k = c_k-1 * content(q_k(x))$, for k=1,2,...,m-1

Finally, the GCD of $f(x)$ and $g(x)$ is given by the product of the content of the PRS and the last non-zero polynomial in the PRS.

$gcd(f(x), g(x)) = c_m-1 * q_m-1(x)$

Proof:

The correctness of the subresultant PRS algorithm can be proven by showing that the GCD of $f(x)$ and $g(x)$ is equal to the GCD of any two consecutive polynomials in the PRS

Let $r_k(x) = q_k-1(x) \% q_k(x)$ be the remainder polynomial in the *k-th* step of the PRS algorithm. We can show that $gcd(q_k(x), r_k(x)) = gcd(q_k-1(x), q_k(x))$, for k=1, 2..., m-1.

First, note that $gcd(q_0(x), q_1(x)) = gcd(f(x), g(x))$ by definition

Secondly, we can use induction to show that $cd(q_k(x), r_k(x)) = gcd(q_k-1(x), q_k(x))$. Assume that $gcd(q_k-1(x), q_k(x)) = gcd(q_k(x), r_k(x))$ for some *k.* We need to show that $gcd(q_k+1(x), r_k+1(x)) = gcd(q_k(x), r_k(x))$.

By definition, we have:

$q_k+1(x) = -1k * r_k(x)$

$r_k+1(x) = -1^(k+1) * (q_k(x) \% r_k(x))$

Let d be the GCD of $q_k(x)$ and $r_k(x)$. By the Euclidean algorithm, we have:

$q_k(x) = d * u(x)$

$r_k(x) = d * v(x)$

Where $u(x)$ and $v(x)$ are polynomials.

Substituting the above expression into $q_k+1(x)$, we get

$q_k+1(x) = -1k * r_k(x)$

An Example:

$$F(x) = x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5$$

$F(x) = 3x^6 + 5x^4 - 4x^2 + 9x - 5$

We first compute the primitive PRS $P_1 = F_1$, $P_2 = F_2$, $P_3 \ldots\ldots P_k$, defined by

$$P_i = R_i / ß_i$$

Where R_i = prem (P_{i-2}, P_{i-1})

$$ß_i = content(R_i)$$

I	n_i	$ß_i$	P_i
1	8		1, 0, 1, 0, -3, -3, 8, 2, -5
2	6		3, 0, 5, 0, -4, -9, 21
3	4	-3	5, 0, -1, 0, 3
4	2	-45	13, 25, -49
5	1	-50	4663, -6150
6	0	$7 \times 13^3 \times 9311$	1

We can say that

$prem(aA, bB) = ab^{\delta+1} prem(A, B)$

$\delta = \partial(A) - \partial(B)$, it is easy to compute the PRS that is determined from F_1

I	$G_i \ / P_i$	$H_i \ / P_i$]
1	1	-
2	1	3
3	3	5
4	5	13
5	2	2
6	$2^2 \times 7 \times 9311$	$2^2 \times 7 \times 9311$

Rough Algorithm for Subresultant

Algorithm: *Given two polynomials A and B with coefficients in a UFD R, this algorithm computes a GCD of A and B, using only operations in R. Assuming that we are equipped with algorithms for accurate division and GCD in R*

1.[*Initializations and reductions*]

If $deg(B) > deg(A)$, exchange A and B. b. If $B = 0$, output A and terminate the algorithm. Set a to the content of A, b to the content of B, d to the gcd(a, b), A to A/a, B to B/b, g to 1, h to 1, and c to 0.

2.[*Pseudo division*]

Set δ to $deg(A)$ - $deg(B)$. b. Using Algorithm 3.1.2, compute R such that $l(B)^{\delta+1}A = BQ + R$

l, where Q is a polynomial such that $deg(Q) = l$ and !(B) divides R. If $R = 0$, go to step 4. d. If $deg(R) = 0$, set B to 1 and go to step 4.

3. [*Reduce remainder*]

Set A to B, B to $R/(gh^{\delta})$and c to c+1. If c<10

set g to $l(A)$, h to $h^{\delta-1}g^{\delta}$ and go to step 2. Ohterwise set A to cont(A), B to cont(B) , g to 1 h to 1, , c to 0 and go to step 2. (Note that all the divisions which may occur in this step give a result in the ring R.)

4. [*Terminate*]

a. Output d times B/cont(B) and terminate the algorithm.

The current implementation of the PRS algorithm involves recursively applying it to each nested list, which is highly inefficient. However, using a sparse representation, we can first identify the symbols present in the polynomial and then only apply the PRS algorithm to those symbols. While it is possible to improve the dense representation for this case, it would be more advantageous to have a sparse implementation of the PRS algorithm. A rudimentary code for this implementation is available in #20874, which can potentially be refined and utilized for this scenario

```
def coeffs_sparse(p1, syms):
        p2 = _coeffs_sparse(p1, syms)
        return [p1.ring(pi) for pi in p2.values()]
def coeff_sparse(p1, sym, deg):
        p2 = _coeffs_sparse(p1, {sym})
        m = [0] * len(p1.ring.gens)
        m[sym] = deg
        m = tuple(m)
        return p1.ring(p2[m])
def gcd_sparse(p):
        R = p[0].ring
        K = R.domain
        # 1-term gcd?
        if any(len(pi) == 1 for pi in p):
        return gcd_terms(p, R, K)
        # Extract the monomial gcd
        p, d = gcd_monomial(p)
        # Eliminate nonshared symbols
        p, common = gcd_coeffs(p)
        # Use subresultant PRS
        g = p[0]
        for pi in p[1:]:
                g = gcd_prs_sparse(g, pi)
        if d is not None:
                g = g * d
        return g
```

The code includes an example that demonstrates how the function works in practice, using a list of polynomials defined over a field of rational numbers. This example shows that the gcd of the two input polynomials is 1, and that the set of symbols in the resulting simplified polynomials is empty. Overall, this function is an important optimization for computing gcd of multivariate polynomials, particularly in cases where the gcd is likely to be 1.

```
def gcd_coeffs(p):
#Example:
#   >>> x = symbols('x:10')
#   >>> K = QQ[x]
#   >>> p1 = K.from_sympy(sum(x[:8]))
#   >>> p2 = K.from_sympy(sum(x[2:]))
 #   >>> p1
#   x0 + x1 + x2 + x3 + x4 + x5 + x6 + x7
#   >>> p2
 #   x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9
#   >>> gcd_coeffs([p1, p2])
 #   ([1], set())
all_coeffs = p
        while True:
 # Quick exits are most efficient if we start from the simplest polys
         p = sorted(set(all_coeffs), key=len)
# Find the intersection of symbols for each poly:
```

```
          common = free_symbols_sparse(p[0])
           nsyms = len(common)
          allsame = True
          for pi in p[1:]:
        # Quick exit
              if not common:
                R = p[0].ring
               K = R.domain
               gcd = gcd_terms(p, R, K)
           return [gcd], None
syms = free_symbols_sparse(pi)
if allsame and syms != common:
           allsame = False
        common &= syms
 # The loop is complete if they all have the same symbols.
     if allsame:
        return p, common
     # Extract coefficients as polys containing only the common symbols.
     all_coeffs = []
     for i, pi in enumerate(p):
        coeffs_i = coeffs_sparse(pi, free_symbols_sparse(pi) - common)
        all_coeffs.extend(coeffs_i)
        # Quick exit:
        if any(len(c) == 1 for c in coeffs_i):
           R = p[0].ring
           K = R.domain
            gcd = gcd_terms(all_coeffs + p[i+1:], R, K)
           return [gcd], None
```

This code defines a function gcd_coeffs that computes the greatest common divisor of a list of polynomials. The function first initializes the list of all coefficients as the input list of polynomials. It then enters a loop that iteratively refines the list of coefficients by extracting the coefficients that contain only the common symbols of the polynomials in the list. This is done by finding the intersection of the symbols for each polynomial and keeping only the symbols that are common to all polynomials. The loop continues until all polynomials in the list have the same symbols. At this point, the function returns the list of polynomials along with the set of common symbols. If at any point during the loop, a polynomial is found to contain a single coefficient or no common symbols, the function quickly exits and returns the greatest common divisor of the remaining polynomials in the list.

```
def subresultants_sparse(f, g, x):
        n = f.degree(x)
        m = g.degree(x)
        if n < m:
           f, g = g, f
           n, m = m, n
       if f == 0:
           return 0, 0
       if g == 0:
           return f, 1
      R = [f, g]
      d = n - m
      b = (-1) ** (d + 1)
    h = prem_sparse(f, g, x)
    h = h * b
    lc = coeff_sparse(g, x, m)
    c = lc ** d
    S = [1, c]
```

```
        c = -c
        while h:
            k = h.degree(x)
            R.append(h)
        f, g, m, d = g, h, k, m-k
b = -lc * c**d
 h = prem_sparse(f, g, x)
       h = h.exquo(b)
       lc = coeff_sparse(g, x, k)
       if d > 1:
          p = (-lc) ** d
          q = c ** (d-1)
          c = p.exquo(q)
       else:
          c = -lc
       S.append(-c)
return R
```

This code defines a Python function called "subresultants_sparse" that computes the subresultant sequence of two sparse polynomials f and g with respect to a variable x. The subresultant sequence is a sequence of polynomials that plays an important role in polynomial factorization and related topics. The function takes three input arguments: f, g, and x. The function first determines the degrees of f and g with respect to x, and reorders the polynomials so that f has the higher degree. If f is zero, the function returns 0 and 0. If g is zero, the function returns f and 1.

The function then initializes some variables and starts a while loop. In each iteration of the loop, the function computes the polynomial h using the "prem_sparse" function (which computes the polynomial remainder of f and g with respect to x), and multiplies h by a coefficient b. The coefficient b depends on the degree difference between f and g, and is computed using the leading coefficient of g. The function then updates some variables and appends h to a list of polynomials R. The loop continues until h becomes zero. At each iteration, the function also computes a coefficient c, which depends on the leading coefficient of g and the degree difference between f and g. The function appends the coefficient -c to a list of coefficients S.

Finally, the function returns the list R, which contains the subresultant sequence of f and g, with the last element being the greatest common divisor of f and g, and the list S, which contains the coefficients of the subresultant sequence.

The code presented is a partial snippet extracted from the overall codebase related to the issue #20874, and has been provided by oscarbenjamin. However, there is a need for further improvement in its implementation to handle cases that may appear trivial. For instance, if there are no common symbols between the polynomials, if the GCD is linear, or if one or both polynomials only have a single term. These cases may require additional handling to ensure the correct calculation of the GCD.

Adding Documentation and Tests

```
def subresultants_sparse(f, g, x):
  """
Computes the subresultants of two polynomials in sparse representation with respect to the variable x.
  Arguments:
  - f: polynomial (in sparse representation) of degree n in variable x
 - g: polynomial (in sparse representation) of degree m in variable x, where m <=
  - x: variable with respect to which the degree of f and g are calculated


  Returns:
  - R: a list of subresultant polynomials in sparse representation such that R[i] is the (i+1)-th subresultant of f and g
  - If g is the zero polynomial, then returns (f, 1)
  - If f is the zero polynomial, then returns (0, 0)
  """
```

Tests

```
def test():

   x, y, z = symbols('x, y, z') # Added a new variable z

   K = QQ[x, y, z] # Update the ring to include z

    assert free_symbols_sparse(K(y)) == {1}

   assert coeffs_sparse(K(y**2), {0}) == [K(y**2)]

   assert coeffs_sparse(K(y**2), {1}) == [K(1)]

   assert coeffs_sparse(K(x + y), {0}) == [K(1), K(y)]

   assert coeffs_sparse(K(x + y), {1}) == [K(x), K(1)]

   assert gcd_coeffs([K(x + y), K(y)]) == ([K(1)], None)

   assert gcd_coeffs([K(x + y), K(y + 1)]) == ([K(1)], None)

   ps, syms = gcd_coeffs([K(x + y), K(x - y)])

   assert (set(ps), syms) == ({K(x - y), K(x + y)}, {0, 1})

  assert coeffs_sparse(K(x*y**2 + 2*x**2*z + y*z**2), {0, 2}) == [K(x*y**2 + y*z**2),     K(2*x**2*z)]

   assert coeffs_sparse(K(x*y**2 + 2*x**2*z + y*z**2), {1, 2}) == [K(x*y), K(0), K(y*z)]

   assert gcd_coeffs([K(x + y + z), K(x - y - z), K(x + z)]) == ([K(1)], None)
```

2.2.2 Phase 2

Implementation of Zippel's Algorithm

This algorithm computes the greatest common divisors (GCDs) of polynomials using a combination of dense and sparse interpolation. It starts by computing a "skeleton" of the GCD in the main variable using the Chinese Remainder Algorithm, and then uses the fast sparse interpolation algorithm to determine the correct coefficients. These coefficients are then recursively interpolated into polynomials involving the second variable and so on. The algorithm is fast but probabilistic and not efficient for dense GCDs.

The project involves studying sparse multivariate polynomial interpolation and GCD computation algorithms. The GCD is interpolated modulo several primes, and the integer coefficients are recovered using the Chinese Remainder Algorithm. The polynomial to be interpolated is treated as a black box function, and various approaches are used, such as Zippel's probabilistic interpolation algorithm and the deterministic interpolation algorithm based on the Berlekamp-Massey algorithm. A combination of Newton interpolation and the univariate version of Ben-Or/Tiwari is used to interpolate multivariate polynomials using Zippel's approach. A parallel algorithm that uses Kronecker substitution and approaches of Zippel and Ben-Or/Tiwari gives extremely fast benchmark results.

The performance of the GCD algorithms can be improved in two situations: when the modular GCD algorithm generates a monic GCD, which needs to be scaled appropriately to find the correct modular image, and when the GCD is non-monic, it is difficult to reconstruct the GCD from modular images, known as the normalization problem

Algorithm for Zippel Algorithm

```
***Input***: $a, b \in Z_p[x_1,....., x_n]$, a prime $p$, and degree bounds
$d_x$ on the gcd in $x_1,...., xn$.
***Output***: $g = GCD(a, b) \in Z_p(x_2,...., x_n)[x1]$ or Fail.
**0.** If the gcd of the inputs has content in $x_n$ return Fail.
**1.** Compute the scaling factor:
$\gamma = gcd(lc_{x1,....x_{n-1}}(a); lc_{x1,....x_{n-1}} \in Z_p [xn]$.
If $\gamma = 1$ then set $RR = False$ else set $RR = True$.
**2.** Choose $v \in Z_p$ {0} at random such that $\gamma$ mod$(x_n-v) \neq 0$
 Set $av = a \bmod (x_n - v)$, $bv = b \bmod (x_n - v)$
then compute $gv = gcd(av, bv) \in Z_p(x_2,....,x_{n-1})[x_1]$
with a recursive call to Zippel $P$ $(n > 2)$ or via
the Euclidean algorithm $(n = 2)$. If for $n > 2$ the algo-
rithm returns Fail or for $n = 2$ we have $deg_{x1}(gv) >$
$d_{x1}$ then return Fail, otherwise set $d_{x1} = deg_{x1}(gv)$
and continue.
```

```
3. Assume $gv$ has no missing terms, and that the evaluation is not unlucky. We call the assumed form $gf$. For each coefficient of $x_1$ in $gf$, count the number of terms in the numerator $nt$ and the number of terms in the denominator $dt$. Take the maximum sum $nt + dt$ over all coefficients and set $nx = nt + dt - 1$. The $-1$ is because we normalize the leading coefficients of the denominators to be $1$.
4. Set $g$□ $= gv$, $m = x$□ $- v$, and $Ni = 1$.
5. Repeat
    5.1. Choose a new random $v \in Z$□ $\backslash\{0\}$ such that $\gamma \bmod (x$□ $- v)$ and set $av = a \bmod (x$□ $- v)$, $bv = b \bmod (x$□ $- v)$.
    5.2. Set $S = \phi$, $ni = 0$.
    5.3. Repeat.
        5.3.1 Choose $\alpha 2,,\ldots\ldots,\alpha_{n-1} \in Z_p$ $\{0\}$ at random such that $deg_{x1}(av \bmod I) = deg_{x1}(a)$ and $deg_{x1}(bv \bmod I) = deg_{x1}(b)$ where $I = (x_2 - \alpha_2,\ldots,x_{n-1} - \alpha_{n-1})$ Set $a_1 = av \bmod I$, $b_1 = bv \bmod I$.
        5.3.2 Compute $g_1 = gcd(a_1, b_1)$.
        5.3.3 If $deg_{x1}(g_1) < d_{x1}$ then our original image and form $gf$ and degree bounds were unlucky, so set $d_{x1} = deg_{x1}(g_1)$ and goto step 2.
        5.3.4 If $deg_{x1}(g_1) > d_{x1}$ then our current image $g_1$ is unlucky, so goto 5.3.1, unless the number of failures $> min(1. ni)$, in which case assume $x_n = v$ is unlucky and goto 5.1.
        5.3.5 Add the equations obtained from equating coefficient of $g_1$ and the evaluation of $gf \bmod I$ to $S$, and set $ni = ni + 1$.
    Until $ni \geq n_x$.
```

```
5.4 We should now have a su cient number of equations in S to solve for all unknowns in gf mod p so attempt this now, calling the result gv.
5.5 If the system is inconsistent our original image is incorrect (missing terms or unlucky), so goto step 2.
5.6 If the system is under-determined, then record the degrees of freedom, and if this has occurred twice before with the same degrees of freedom then assume the content problem was introduced by the evaluation of x⍰ so go to 5.1. Otherwise, we need more images so goto 5.3.1.
5.7. The system is consistent and determined, so we have a new image gv.
Solve f ≡ g⍰ mod m(x⍰) and f ≡ gv mod (x⍰ - v) using the Chinese remainder algorithm for f ϵ Z⍰[x⍰](x₂,…. x⍰₋₁)[x₁] mod m(x⍰) × (x⍰ - v). Set g⍰ = f, m = m(x⍰)× (x⍰ - v), and Ni = Ni + 1.
Until Ni ≥ dₓ⍰ +1 and RR = False or Ni ≥ 3
6. Reconstruct
6.1. If RR = True then apply rational function reconstruction in x⍰ and assign the result to gc. If it fails then we need more points, goto step 5.1. For n > 2, clear the rational function denominators of gc ϵ Z⍰(x⍰)(x₂,…..x⍰₋₁)[x₁] to obtain gc ϵ Z⍰(x₂, x⍰)[x₁].
6.2. If RR = False then set gc = g⍰.
```

Below, I provide rough code snippet for the dense interpolation and sparse interpolation

```
def dense_interpolation(p, m):

    x = symbols('x')

    f = m[0] * x ** 0

    q = x - p[0]

    for i in range(1, len(p)):

        temp = q.subs(x, p[i])

        qpi_inv = q.subs(x, p[i]) ** (-1)

        f += qpi_inv * q * (m[i] - f.subs(x, p[i]))

        q = (x - p[i]) * q

    if f == nan:

         return sympify(1)

else:

        return expand(f)

"""'sparse interpolation algorithm '''

def sparse_interpolation(x_var, a, d):

               S = [sympify(1)]

               p0 = sympify(oracle(a))

               L = []

      for i in range(0, len(x_var)):

                 r = []

                 X = x_var[i]

                 P = []

               for j in range(0, d):

                    temp = random.randint(1, 10000)

               r.append(temp)

                     L = []

                     t = len(S)
        skeletal = sympify(0)

        G = symbols('g0:%d'%(t+1))

        for k in range(0, t):

           skeletal += G[k] * S[k]

        for k in range(0, t):

      # for i=0, there is no need to solve system of linear equations we append the output of oracle [F]\

    #in the list 'P'\ otherwise, we need to create a list 'L' of 't' linear equations

                if i == 0:

              oracle_out = oracle([r[j]] + a[1:len(a)])

              P.append(oracle_out)

                              sub_list = [(x_var[tem], A[tem]) for tem in range(i)]

      L.append(skeletal.subs(sub_list) - oracle_out

           else:

              A = random.sample(xrange(10000), i)

      or_lis = A + [r[j]] + a[i+1:len(a)]

              oracle_out = oracle(or_lis)

              sub_list = [(x_var[tem], A[tem]) for tem in range(i)]

                  L.append(skeletal.subs(sub_list)-oracle_out)

             if i != 0:

           sol1 = linsolve(L, G)

    solution = next(iter(sol1))

    temp_p = sympify(0)

                for mon in range(len(S)):

              temp_p += S[mon]*solution[mon]

    P.append(temp_p)

          if i == 0:

        p0 = sympify(dense_interpolation([a[0]] + r, [p0] + P).subs(symbols('x'), x_var[0]))

        S = p0.as_coefficients_dict().keys()

         else:
        P = [p0] + P

        r = [a[i]]+r

        p0 = sympify(0)

    for mon in range(len(S)):

    p_i_temp = []

    for pol in range(len(P)):

              p_i_temp += [P[pol].as_coefficients_dict()[S[mon]]]

           interp_temp =  dense_interpolation(r,p_i_temp).subs(symbols('x'), x_var[i])

           p0 += S[mon]*interp_temp

        p0 = expand(p0)

        S = p0.as_coefficients_dict().keys()

    return p0
```

Chinese Remainder Theorem

```
def poly_gcd(f, g):

   """

   Computes the GCD of two polynomials f and g using the Chinese Remainder Theorem algorithm.

   """

   # Step 1: Compute h1(x) = gcd(f(x), g(x))

   h1 = gcd(f, g)

# Step 2: Compute r(x) = f(x) / h1(x) and s(x) = g(x) / h1(x)

   r = f // h1

   s = g // h1

   # Step 3: Compute q(x) = gcd(r(x), s(x)) using the Extended Euclidean Algorithm

   q, _ = xgcd(r, s)

   # Step 4: Compute h(x) = h1(x) * q(x)

   h = h1 * z

   return h  # Return the GCD of f and g
```

Adding the Documentation for Zippel Algorithm and Tests

```
def Zippel(a, b, p, dx):

        """

        Compute gcd(a, b) using RatZip algorithm with degree bounds dx.

        Parameters:

        a, b: sympy polynomials in Zp[x1, ..., xn]

        p: prime number

        dx: degree bounds for gcd in x1, ..., xn

        Returns:

         g: gcd(a, b) in Zp(x2, ..., xn)[x1]

        None: if the algorithm fails

        """
```

Tests

```
def test_ratzip_p():

# Define input parameters

  p = 17

  n = 3

  dx = [2, 2, 2]

  a = Poly(x*y*z + x*z, domain=FF(p))

  b = Poly(x**2*y**2 + x*y + x, domain=FF(p))

  # Run the algorithm

  try:

    g = ratzip_p(a, b, p, dx)

  except ValueError:

    assert False, "Unexpected ValueError raised"
```

```python
# Define evaluation points for probabilistic division test
    alpha = [sympy.randprime(1, p-1) for i in range(n)]
    # Evaluate g at alpha to obtain a list of values
    values = [g.subs(x, alpha[i]).eval(x, modulus=p) for i in range(n)]
    # Check that g divides a and b at the evaluation points
    for i in range(n):
        a_mod = a.eval(x, z0=alpha[i], modulus=p)
        b_mod = b.eval(x, z0=alpha[i], modulus=p)
        assert a_mod % values[i] == 0, "gcd did not divide a at evaluation point"
        assert b_mod % values[i] == 0, "gcd did not divide b at evaluation point"
    # Check that g is the correct result
    assert g == Poly(x + 1, domain=FF(p)), "Incorrect gcd result"
# define the input polynomials and parameters
p = 13
dx = [2, 2]
a_dict = {(0, 1): 1, (1, 0): 1, (1, 1): 1, (2, 0): 2}
b_dict = {(0, 0): 1, (1, 1): 1, (1, 2): 1, (2, 2): 2}
a = sparse_poly_from_dict(a_dict, gens=('x', 'y'))
b = sparse_poly_from_dict(b_dict, gens=('x', 'y'))
# run the algorithm
from sympy_algolib.polynomial.gcd_algorithms import ratzip_p
g = ratzip_p(a, b, p, dx)
# check the result
assert str(g) == 'x + 1'
```

2.2.3 Phase 3

Implementation of Sparse Modular GCD Algorithm

In the given below there is a **SparseModGcd** algorithm. This modular GCD algorithm first calls **Subroutine M** which computes the gcd in $L[x]$ from a number of images in $L_p[x]$. **Subroutine P** which is called by subroutine M computes the gcd in $L_p[x]$ using both dense and sparse interpolations. Finally, **Subroutine S**, which stands for Sparse Interpolation and is called by **Subroutine P**, does the sparse interpolation.

```
Algorithm SparseModGcd
Input: $f_1, f_2 \in L[x]$ and $m_1, \ldots, m_r \in F[z_1, \ldots, z_r]$ where
$F = Q(t_1, \ldots, t_k)$ s.t. $cont_x(g) = 1$.
Output: $\check{g}$ where $g$ is the monic gcd of $f_1$ and $f_2$ in $L[x]$.
    1. Call Subroutine M with input $\check{f}_1$, $\check{f}_2$ and $\check{m}_1, \ldots, \check{m}_r$.

Subroutine M
Input: $f_1, f_2 \in D[z_1, \ldots, z_r]/(m_1, \ldots, m_r)[x]$ and $m_1, \ldots, m_r$
$\in D[z_1, \ldots, z_r]$ where $D = Z[t_1, \ldots, t_k]$.
Output: $\check{g}$, where $g$ is the monic gcd of $f_1$ and $f_2$.
1. Set $n = 1$, $G = 0$, and the assumed form $g_f = 0$.
2. Take a new prime $p$ that is not bad.
3. Let $g_n$ be the output of subroutine P applied to $p$,
$(f_1, f_2) \bmod p$, $g_f$, and $(m_1, \ldots, m_r) \bmod p$.
4. If $g_n$ = "ZeroDivisor" or $g_n$ = "Unlucky" or $g_n$ =
"UnluckyContent" then go back to step 2.
5. If $g_n$ = "BadForm" then go back to step 1.
6. If $g_n = 1$ then return 1.
7. If $G = 0$ then set $G = g_n$, $M = p$ and go to step 9.
8. Set $M = M \times p$ and combine $g_n$ with $\{g_1, \ldots, g_{n-1})\}$
using Chinese remaindering to obtain $G \bmod M$.
9. Set $g_f = G$, $n = n + 1$.
10. Apply integer rational reconstruction to obtain $h$ satisfying
$h \equiv G \bmod M$. If this fails then go back to
step 2.
11. Clear fractions in Q: Set $h = \check{h}$.
12. Trial division: if $h | f_1$ and $h | f_2$ then return h, otherwise,
go back to step 2.
```

```
Subroutine P
Input: $p, f_1, f_2, g_f \in$ D⍰[$z_1, \ldots, z_r$]/$m_1, \ldots, m_r$) [$x$] and
$m_1, \ldots, m_r$ epsilon D⍰[$z_1, \ldots, z_r$].
Output: Either ˇ$g$, the primitive associate of the monic gcd
of $f_1$ and $f_2$, or "ZeroDivisor" or "Unlucky" or "BadForm"
or "UnluckyContent."
1. If $k$ (the number of parameters) = 0 then output the result
   of the monic Euclidean algorithm applied to $f_1, f_2$
   modulo ($m_1, \ldots, m_r, p$). If a zero divisor is encountered
   then output "ZeroDivisor".
2. If the assumed form $g_f = 0$ then go to step 4.
3. Sparse Interpolation: (we already know the form
   gf of the gcd from subroutine M.)
   Return g⍰ $\in$ D⍰[$z_1, \ldots, z_r, x$], the output of subroutine
   S applied to $p$, $f_1, f_2, m_1, \ldots, m_r$ and $g_f$ .
4. Choose $\alpha_1$ at random from Z⍰ that is not bad.
5. Let $g_1 \in$ Z⍰[$t_1, \ldots,$ t⍰$_{-1}$][[$z_1, \ldots, z_r, x$] be the output of
   subroutine P applied to $p, f_1, f_2, m_1, \ldots, m_r$ at t⍰ = $\alpha_1$
   and assumed form $g_f = 0$
6. If $g_1 = 1$ then return 1.
7. If $g_1 \in$ { "BadForm", "Unlucky", "UnluckyContent",
   "ZeroDivisor" } then return $g_1$.
8. Set $g_f = g_1$, $G = g_1$, $M$ = (t⍰ − $\alpha_1$), $n = 2$, $c = 1$, $d =$
   $1$, $u = 1$.
9. Main Loop: Take a new evaluation point α⍰ at random
   from Z⍰ that is not bad.
10. Let g⍰ $\in$ Z⍰[$t_1, \ldots,$ t⍰$_{-1}$][$z_1, \ldots, z_r, x$] be the output of
    subroutine S applied to $p, f_1, f_2, m_1, \ldots, m_r$ evaluated
    at t⍰ = α⍰ and $g_f$ .
```

```
11. If g□ = "BadForm" then return "BadForm".
12. If g□ = "UnluckyContent" then set $c = c + 1$ and if
    $c > n$ return "UnluckyContent" else go to main loop.
13. If g□ = "Unlucky" then set $u = u + 1$ and if $u > n$
    then return "Unlucky", else go back to main loop.
14. If g□ = "ZeroDivisor" then set $d = d + 1$ and if $d > n$
    then return "ZeroDivisor", else go back to main loop.
15. If g□ = 1 then return 1.
16. Set $M = M \times$(t□ −α□) and Chinese remainder g□ with {$g_1$, . . . , g□₋₁} to obtain $G$
    mod $M$(t□).
17. Set $n = n + 1$.
18. Apply rational function reconstruction to coefficients
    of $G$ to obtain $h \in$ Z□(t□) [$t_1$, . . . , t□₋₁][$z_1$, . . . , $zr$, $x$]
    s.t. $h \equiv G$ mod $M$(t□). If this fails, go back to main loop.
19. Clear fractions in Z□(t□): Set $h = \check{h}$.
20. Trial division: if $h|f_1$ and $h|f_2$ then return $h$, otherwise , go back to main
    loop
```

Subroutine S

```
Input: $p, f_1, f_2, gf \in$ D□[$z_1$, . . . , $zr$]/ ($m_1$, . . . ,$mr$) [$x$] and
    $m_1$, . . . ,$mr \in$ D□[$z_1$, . . . , $zr$] where D□ = Z□[$t_1$, . . . , t□].
Output: Either $\check{g}$, the primitive associate of the monic gcd
    of $f_1$ and $f_2$, or "BadForm" or "ZeroDivisor" or "Unlucky"
    or "UnluckyContent."
1. If $k$ (the number of parameters) = 0 then call the monic
    Euclidean algorithm on $f_1$,$f_2$ and output the result.
2. Suppose the assumed form $gf = \sum i$
     $CiTi$ where $Ti$
    is a monomial in ($x$, $z_1$, . . . , $zr$) and $Ci$ =
    $\sum j\ cijSj$
    where $Sj$ is a monomial in parameters $t_1$, . . . , t□ with
    unknown $cij$ .
```

```
3. Set U to be the minimum number of images needed −
see below. ( The algorithm uses one more image than U to detect a wrong
assumed form gf . )
4. Set $d = 1, u = 1, n = 1$.
5. While $n <\leq U + 1$ do
        5.1. Take a new random evaluation point
        $α⍰ = (t_1 = a_1, . . . , t⍰ = a⍰)$ in $Z^k⍰$ which is not bad.
        5.2. Let g⍰ be the output of the monic Euclidean algorithm
        applied to $f1(α⍰), f2(α⍰)$ modulo
        $(m1(α⍰),..., mr(α⍰), p)$.
        5.3. If g⍰ = “ZeroDivisor” then set $d = d + 1$ and if
        $d > n$ then return “ZeroDivisor” else go back to
        step 5.1.
        5.4. If $deg_x(g⍰) > deg_x(gf)$ then set $u = u + 1$ and if
        $u > n$ then return “Unlucky” else go back to step 5.1.
5.5. If $deg_x(g⍰) < deg_x(gf)$ return “BadForm”.
5.6. If g⍰ has terms in $(x, z_1, . . . , zr)$ not present in the
assumed form gf then return “BadForm”.
5.7. Set $n = n + 1$.
6. Construct the system of $U + 1$ linear equations by
equating $gf(α⍰) = μ⍰g⍰$ with μ⍰ unknown. Solve the
linear system with $μ_1 = 1$ to determine the cij s.
7. If the system is inconsistent, return “BadForm”.
8. If the system is under determined, return “Unlucky-
Content”.
9. Set g⍰ =
∑i∑jcijSj)T_i
10. Make $lc_{x,⍰_1,...,⍰⍰}(g⍰) = 1$ and return g⍰.
```

```
def is_bad(p, n):
  # check if n is a bad point in Zp
  return False  # replace with implementation


def is_zero_divisor(g):

  # check if g is a zero divisor

  return False  # replace with implementation


def is_unlucky(gn, c, n):

  # check if gn is unlucky

  return False  # replace with implementation


def is_bad_form(gn):

  # check if gn is in bad form

  return False  # replace with implementation


def is_unlucky_content(gn):

  # check if gn has unlucky content

  return False  # replace with implementation


def euclidean_algorithm(p, f1, f2, m):

  # implementation of monic Euclidean algorithm

  # applied to f1 and f2 modulo m

  return gcd(f1, f2) % m


def sparse_interpolation(p, f1, f2, m1, m2, gf):

  # implementation of sparse interpolation subroutine

  return None  # replace with implementation


def polynomial_P(p, f1, f2, m1, m2, t_val, gf):

  # implementation of polynomial P subroutine

  return None  # replace with implementation


def chinese_remainder(g, G, M):

# implementation of Chinese remainder theorem
```

```
   return None  # replace with implementation


def rational_function_reconstruction(p, t_val, G, M):

    # implementation of rational function reconstruction

    return None  # replace with implementation

def clear_fractions(p, t_val, h):

    # implementation of clearing fractions in Zp(tk)

    return None  # replace with implementation


def trial_division(p, f1, f2, h):

    # implementation of trial division

    return None  # replace with implementation
```

```
import random

def Subroutine_P(p, f1, f2, gf, m1, m2):

    """

    Compute the primitive associate of the monic GCD of f1 and f2

    using the sparse interpolation algorithm.

    """

    # Step 1: Apply the monic Euclidean algorithm

    gcd_mod_h = poly_euclidean(p, f1, f2, m1, m2)

    if gcd_mod_h == "ZeroDivisor":

        return "ZeroDivisor"

    # Step 2: Check if gf is zero

    if gf == 0:

        tk = random_good_element(p)

        gf = poly_pseudo_division(p, f1, f2, m1, m2, tk)

        if gf in ["BadForm", "Unlucky", "UnluckyContent", "ZeroDivisor"]:

            return gf

    # Step 3: Sparse interpolation

    gn = poly_sparse_interpolation(p, f1, f2, m1, m2, gf)

    # Step 4: Choose a random element that is not bad

    tk = random_good_element(p)

    # Step 5: Compute the output of P

    g1 = poly_pseudo_division(p, f1, f2, m1, m2, tk, gf)

    if g1 == 1:

        return 1

    if g1 in ["BadForm", "Unlucky", "UnluckyContent", "ZeroDivisor"]:

        return g1

    # Step 8: Initialize variables

    G = g1

    M = tk – 1

    n = 2
```

```
  c = 1
  d = 1
  u = 1
 while True:
     # Step 9: Choose a random evaluation point
     tn = random_good_element(p)
     # Step 10: Compute the output of S
     gn = poly_sparse_interpolation(p, f1, f2, m1, m2, gf, tn)
     if gn == "BadForm":
        return "BadForm"
# Step 12: Handle UnluckyContent
     if gn == "UnluckyContent":
        c += 1
        if c > n:
           return "UnluckyContent"
        else:
           continue
     # Step 13: Handle Unlucky
     if gn == "Unlucky":
        u += 1
        if u > n:
           return "Unlucky"
        else:
           continue
     # Step 14: Handle ZeroDivisor
     if gn == "ZeroDivisor":
        d += 1
        if d > n:
           return "ZeroDivisor"
        else:
           continue
# Step 15: Check if gn is 1
     if gn == 1:
        return 1
     # Step 16: Update M and G
     M *= (tk - tn)
     G = poly_chinese_remainder(p, G, gn, M, tk)
     # Step 17: Update n
     n += 1
     # Step 18: Rational function reconstruction
     h = poly_reconstruction(p, G, M, tk)
     if h is None:
        continue
     # Step 19: Clear fractions
     h = poly_clear_fractions(p, h, tk)
     # Step 20: Trial division
     if poly_divides(p, h, f1) and poly_divides(p, h, f2):
        return poly_primitive_associ
```

```
#Subroutine M
def M(f1, f2, m_list):
        n = 1
        G = 0
        gf = 0
        # Helper function for prime selection
        def select_prime():
            while True:
                p = randint(2, 1000)
                if p in bad_primes:
                    continue
                else:
                    return p
        while True:
            p = select_prime()
            g_n = P(p, f1 % p, f2 % p, gf % p, [m % p for m in m_list])
            if g_n == "ZeroDivisor" or g_n == "Unlucky" or g_n == "UnluckyContent":
                continue
            elif g_n == "BadForm":
                n = 1
                G = 0
                Continue
            elif g_n == 1:
                return 1
            elif G == 0:
                G = g_n
                M = p
                n += 1
                continue
            else:
                M = lcm(M, p)
                c = [G] + [g_n] * (n-1)
                G = chinese_remainder(c, [M] * n)
                n += 1
                h = rational_reconstruction(G, M)
                if h is None:
                    continue
                h = QQ.clear_denoms(h)
                if gcd(f1, h) == h and gcd(f2, h) == h:
                    return h
```

```
#Subroutine P
def S(p, f1, f2, gf, h, m, r):

  # Step 1

  if len(gf.variables()) == 1:

    return f1.gcd(f2)

  # Step 2

  T = [var for var in gf.variables() if var != 'x']

  C = [polynomial({(0,)*len(T): 1}, domain=GF(p)) for _ in range(len(r))]

  for i in range(len(r)):

    for j in range(len(C[i])):

      Sj = prod([T[k]**exp for k, exp in enumerate(C[i][j].exponents())])

      C[i][j] = r[i].subs({var: Sj for var in T})

  # Step 3

  U = len(T)*2

  # Step 4

  d, u, n = 1, 1, 1

  # Step 5

  while n <= U+1:

    # Step 5.1

    eval_point = {T[i]: randint(0, p-1) for i in range(len(T))}

    if any(h[i].subs(eval_point) == 0 for i in range(len(r))):

      continue

    # Step 5.2

    gn = f1.eval(eval_point).gcd(f2.eval(eval_point))

    for i in range(len(r)):

      gn = gn.quo_rem(h[i].eval(eval_point))[1][0]

      if gn.is_zero():

        d += 1

        if d > n:

          return "ZeroDivisor"
```

```
            else:
                continue
        # Step 5.3
        if gn.degree('x') > gf.degree('x'):
            u += 1
            if u > n:
                return "Unlucky"
            else:
                continue
        # Step 5.4
        if gn.degree('x') < gf.degree('x'):
            return "BadForm"
        # Step 5.5
        gf_terms = set(gf.terms())
        gn_terms = set(gn.terms())
        if gn_terms - gf_terms:
            return "BadForm"
        # Step 5.6
        n += 1
    # Step 6
    A = matrix(GF(p), U+1, len(C))
    for i in range(U+1):
        eval_point = {T[j]: i if j == 0 else randint(0, p-1) for j in range(len(T))}
        A[i, :] = vector([C[k][j].eval(eval_point) for k in range(len(C))])
    # Step 7
    if A.rank() < U+1:
        return "BadForm"
    # Step 8
    if A.rank() < len(C):
        return "UnluckyContent"

    # Step 9

    c = A.solve_right(vector([1] + [0]*U))

    gp = sum([sum([c[j][i]*C[j][k] for j in range(len(C))]) * prod([T[l]**exp for l, exp in enumerate(k)]) for i, k in
enumerate(gf.monoms())])

    # Step 10

    lc = gp.lc('x')

    gp = gp.monic('x')

    for var in T:

        gp = gp.monic(var)

    return gp
```

Adding the documentation and Tests

```
def SparseModgcd(f, g, p):

            """

            Compute the gcd of two sparse polynomials f and g modulo p using SparseModgcd algorithm.
            Parameters:

             -----------

            f : sympy.Poly

            The first sparse polynomial.

            g : sympy.Poly

            The second sparse polynomial.

            p : sympy.Poly

            The modulus.

            Returns:

             --------

            gcd : sympy.Poly

            The gcd of f and g modulo p.

            """
```

Tests

```
def test_SparseModgcd():

  # Test case 1: Two sparse polynomials of degree 0.

  f = Poly(2, x)

  g = Poly(3, x)

  p = Poly(5, x)

  assert SparseModgcd(f, g, p) == Poly(1, x)

  # Test case 2: Two sparse polynomials of degree 1.

  f = Poly(2*x + 3, x)

  g = Poly(3*x + 4, x)

  p = Poly(5, x)

  assert SparseModgcd(f, g, p) == Poly(1, x)

  # Test case 3: Two sparse polynomials of degree 2.

  f = Poly(2*x**2 + 3*x + 1, x)

  g = Poly(3*x**2 + 4*x + 1, x)

  p = Poly(5, x)

  assert SparseModgcd(f, g, p) == Poly(x + 1, x)\

    # Test case 4: Two sparse polynomials of different degrees.

 f = Poly(2*x**3 + 3*x + 1, x)

  g = Poly(3*x**2 + 4*x + 1, x)

  p = Poly(5, x)

  assert SparseModgcd(f, g, p) == Poly(1, x)

  # Test case 5: Two sparse polynomials that have a common factor.

  f = Poly(2*x**2 + 6*x + 4, x)

  g = Poly(3*x**2 + 9*x + 6, x)

  p = Poly(5, x)

  assert SparseModgcd(f, g, p) == Poly(x**2 + 3*x + 2, x)



  # Test case 6: Two sparse polynomials that have no common factor.
```

```
f = Poly(2*x**2 + 6*x + 4, x)

g = Poly(3*x**2 + 7*x + 4, x)

p = Poly(5, x)

assert SparseModgcd(f, g, p) == Poly(x + 1, x)
```

2.2.4 Phase 4

Explore and Add benchmark for above algorithms

In this section, I will implement all four polynomial factorization algorithms: Zippel's algorithm, Subresultant PRS, SparseModGcd, and HEUGCD. We will then compare the running time of each algorithm using different types of inputs. Specifically, we will use the input examples given in Liao's paper to ensure that our benchmarking is consistent with previous work in the field.

To begin, we will describe the input examples provided by Liao. These examples consist of polynomials with varying degrees and coefficients. Some of the polynomials have a large number of variables, while others have only one variable. The coefficients of the polynomials are randomly generated, but their magnitudes are bounded to ensure that the polynomials are well-behaved.

We will use these input examples to benchmark each algorithm's performance. For each input example, we will record the running time of each algorithm and compare the results. This will allow us to determine which algorithm performs best on which types of inputs.

By benchmarking each algorithm on a variety of inputs and optimizing their implementations, we can provide a comprehensive comparison of the performance of these polynomial factorization algorithms.

We have varied different parameters such as degrees, bitlength of coefficients, and density of polynomials (number of non-zero terms).

Case 1. GCD = 1

$F = (1 + x + \sum_{i=1}^{v} y_i)(2 + x + \sum_{i=1}^{v} y_i)$

$G = (1 + x^2 + \sum_{i=1}^{v} y^2{}_i)(-3 + y_1 x^2 + y^2{}_1)$

Case 2. Linearly dense quartic inputs with quadratic GCDs.

$D = (1 + x + \sum_{i=1}^{v} y_i)^2$

$F = D.(-2 + x - \sum_{i=1}^{v} y_i)^2$

$G = F = D.(2 + x + \sum_{i=1}^{v} y_i)^2$

Case 3. Sparse GCD and inputs where degree are proportional to the number of variables

$D = 1 + x^{v+1} + \sum_{i=1}^{v} y^{v+1}{}_i$

$F = D.(-2 + x^{v+1} + \sum_{i=1}^{v} y^{v+1}{}_i)$

$G = D.(2 + x^{v+1} + \sum_{i=1}^{v} y^{v+1}{}_i)$

Case 4. Quadratic non-monic GCD, F and G have other quadratic factors

$D = 1 + x^2 y^2{}_1 + \sum_{i=2}^{v} y^2{}_i$

$F = D.\ (-1 + x^2 - y^2{}_1 + \sum_{i=2}^{v} y^2{}_i)$

$G = D.\ (2 + y_1 x + \sum_{i=2}^{v} y_i)^2$

Case 5. Sparse non-monic quadratic inputs with linear GCDs

$D = -1 + x.\prod_{i=1}^{v} y_i$

$F = D.\ (3 + x.\prod_{i=1}^{v} y_i)$

$G = D.\ (-3 + x.\prod_{i=1}^{v} y_i)$

Degree *(v)*	1	2	3	4	5	6	7	8	9	10
Zippel										
Sparse Mod GCD										
Subresultant										
HEUGCD										

Implement a logic for different strategies

When we want to compute the greatest common divisor (gcd) of two integers or polynomials, there are different algorithms that we can use. Some of them, such as Subresultant PRS and Zippel algorithm, are based on polynomial factorization, while others, like SparseModGcd algorithm, use modular arithmetic.

To determine which algorithm is the fastest for a particular case, we can implement each of them and time how long they take to compute the gcd of a pair of integers or polynomials. For example, if we want to compute the gcd of two polynomials with degree 1000, we can implement Subresultant PRS, Zippel algorithm, and SparseModGcd algorithm and measure the time taken by each one. Based on the timing results, we can then determine which algorithm is the most efficient for that specific case. We might choose SparseModGcd algorithm, which is designed for sparse polynomials and is faster than other algorithms like Subresultant PRS or Zippel algorithm for such cases.

Finally, some gcd algorithms can be interrupted if they take too long to compute the gcd. For example, in Subresultant PRS algorithm, we can interrupt the computation if the size of the subresultant matrices becomes too large. In Zippel algorithm, we can interrupt the computation if the degree of the polynomials becomes too large. By doing so, we can save computational resources and try another algorithm that might be more suitable for that particular case.

After benchmarking the algorithms, we can implement conditional statements like the ones mentioned above to choose the most efficient algorithm for a given set of input data. The examples provided are general and may need to be modified based on the results of the benchmarks. By using benchmarks to test the performance of different algorithms, we can create more specific and accurate conditional statements for different input scenarios.

If the polynomials are dense and have a high degree:

```
if degree of polynomials > threshold_degree:
  use SparseModGcd algorithm
else:
  use Subresultant PRS or Zippel algorithm
```

In this case, we use SparseModGcd algorithm only if the degree of the polynomials is higher than the threshold_degree, because Subresultant PRS and Zippel algorithm are more efficient than SparseModGcd algorithm for dense polynomials with a lower degree.

If the polynomials are sparse and non-monic

```
if number of non-zero coefficients in polynomials > threshold_sparsity:
   use Subresultant PRS or Zippel algorithm
else:
  use SparseModGcd algorithm
```

In this case, we use Subresultant PRS or Zippel algorithm only if the polynomials are sparse and non-monic, because SparseModGcd algorithm is more efficient than Subresultant PRS or Zippel algorithm for sparse and monic polynomials.

if the polynomials have a low degree and are monic:

```
use Subresultant PRS or Zippel algorithm
```

In this case, we use Subresultant PRS or Zippel algorithm, because they are generally more efficient than SparseModGcd algorithm for polynomials with a low degree and are also suitable for monic polynomials.

If the polynomials have a small degree and are dense:

```
if number of non-zero coefficients in polynomials > threshold_sparsity:
  use Subresultant PRS or Zippel algorithm
else:
  use SparseModGcd algorithm
```

In this case, we use Subresultant PRS or Zippel algorithm only if the polynomials are dense and have a low degree, because SparseModGcd algorithm is more efficient for sparse polynomials with a low degree.

Improving the speed matrix inverse in DomainMatrix

The code mentions that the algorithms for computing inverse of DomainMatrix can be improved. Currently, they involve polynomial greatest common divisor (gcd) computations which can become a bottleneck for larger examples. This means that as the size of the matrix and the degree of the polynomials involved increase, the gcd computations take longer to perform and can slow down the overall computation.

```
def inv(self):
    if not self.domain.is_Field:
        raise DMNotAField('Not a field')
    m, n = self.shape
    if m != n:
        raise DMNonSquareMatrixError
    inv = self.rep.inv()
    return self.from_rep(inv)
```

All arithmetic operations in ZZ(x, y) involve computing the greatest common divisor (gcd) of polynomials. For example, to add two fractions in ZZ(x, y), we first find a common denominator by taking the lcm (least common multiple) of the denominators, then we multiply each fraction by the appropriate factor to obtain a common denominator. Finally, we add the numerators and simplify the resulting fraction by computing the gcd of the numerator and denominator. Similarly, to multiply two fractions, we multiply the numerators and denominators separately and simplify the result by computing the gcd of the resulting polynomial.

3. Proposed Timeline

Given below is the timeline, pointing out the exact time-frame I will require to complete each subproject.

3.1 Commitments during GSoC

I will have end semester exams from May 1st to May 20th, I will complete all necessary prerequisites before the coding period begins. After 20th May Since I have no other commitments in my summer, giving more than 40-50 hours per week for the project will not going to be hard for me during summer vacations. My next semester's classes will commence from 17th July. However, during the initial month of the semester, courses are just introduced, hence, I have a very little load during that phase. Hence, I would be able to work without hindrance, during the final phase of the project for around 40 hours per week. I will be writing blogs of what all discussions have taken place and also what and how I have implemented. If in case, due to some reasons, I lag behind the projected timeline, I will devote extra time to the project to cover the delay.

3.2 Community Bonding Period (4 May – 28 May)

During this period, my schedule includes meeting with my mentors to discuss my project and thoroughly examining the codebase, particularly the modules relevant to my project. As a long-time contributor to SymPy, this task should not require a significant amount of time. Additionally, I plan to actively contribute by addressing issues in the Polys module and other modules of SymPy. If my mentor meetings conclude early, I intend to begin working on my project during this time.

3.3 Coding Period

Phase 1:

- **Week 1 – Week 2 (May 29 – June 11)**
  - Implementation of Subresultant PRS algorithm for Sparse representation.
  - Adding the documentation and tests.

Phase 2:

- **Week 3 - Week 4 (June 12 – June 25)**
  - Implementation of Zippel's Algorithm.

- **Week 5 (June 26 – July 2)**
  - Adding the documentation and tests.
  - Fixing minor issues related to Polys module.

Phase 3:

- **Week 6 - Week 7 (July 3 – July 16)**
  - Implementation of SparseModGcd algorithm.
- **Week 8 (July 17 – July 23)**
  - Adding the documentation and tests.
  - Finish pending work and issues raised in previous implementations.

Phase 4:

- **Week 9 - Week 10 (July 24 – August 6)**
  - Explore and Add benchmark for above algorithms.
- **Week 10 (August 7 – August 13)**
  - Implement a logic for different strategies.
- **Week 11 (August 14 – August 20)**
  - Improving the speed matrix inverse in DomainMatrix.

Final Week (August 21 – August 28)

- Finishing above work if it is remaining.
- Discuss the future opportunities in the Polys module.

3.4 Post GSoC

In the event that there are any incomplete tasks after the conclusion of GSoC, my plan is to continue working on them. Additionally, I intend to maintain my contributions to sympy and assist new contributors as well. My dedication to the library extends beyond the duration of the GSoC program, and I will prioritize completing any unfinished work so that future additions to the physics module can be more advanced. SymPy has been instrumental in helping me gain experience in the open-source community, and I am eager to continue contributing as much as possible. The guidance and support of key organization members such as Aaron Meurer (asmeurer), Oscar Benjamin (oscarbenjamin), S.Y. Lee (sylee957), Francesco Bonazzi (Upabjojr), Jason K. Moore (moorepants), Sam Brokie (broksam), Oscar Gustafsson (oscargus), Christopher Smith (smichr), Timo Stienstra (TJStienstra) has been invaluable in my contributions to the library, and I look forward to remaining connected with the organization in the future. In fact, I aspire to become a mentor for future GSoC programs with SymPy.

4. References

[1] Oscar Benjamin Issue on Improving the sparse polynomial gcd.
https://github.com/sympy/sympy/issues/23131

[2] Oscar Benjamin Issue on Port the PRS algorithm to the sparse polynomial implementation.
https://github.com/sympy/sympy/issues/20874

[3] Harshit Yadav PR on port PRS for sparse poly.
https://github.com/sympy/sympy/pull/21477

[4] Previous years proposals.
https://github.com/sympy/sympy/wiki/_pages

[5] Poly module in SymPy
https://docs.sympy.org/latest/modules/polys/domainsintro.html

[6] A course in Computational Algebraic Number Theory
https://books.google.co.in/books?id=5TP6CAAAQBAJ&pg=PA108&source=gbs_toc_r&cad=4#v=onepage&q&f=false

[7] Zippel, Richard. "Probabilistic algorithms for sparse polynomials."
http://www.cecm.sfu.ca/~mmonagan/teaching/TopicsinCA15/zippel79.pdf

[8] W. S. Brown. 1978. The Subresultant PRS Algorithm
https://dl.acm.org/doi/pdf/10.1145/355791.355795

[9] [Liao95] Hsin-Chao Liao, R. Fateman, Evaluation of the heuristic polynomial GCD
https://people.eecs.berkeley.edu/~fateman/282/readings/liao.pdf

[10] Computer algebra symbolic and algebraic computation B Buchberger, GE Collins, R Loos
https://books.google.co.in/books?id=yUmqCAAAQBAJ&printsec=frontcover&redir_esc=y#v=onepage&q&f=false