Developing Polynomial Manipulation Module in SymEngine and implementing Sets and Solvers for the same

Personal Details

Name: Akash Trehan
University: Indian Institute of Technology, Bombay
Email: akash.trehan123@gmail.com
Github: CodeMaxx
Other Contact Methods: Google Hangouts, Gitter Chat
Blog: codemaxx.github.io
Time-zone: UTC+5:30

Personal background

I am a first year B.Tech. undergraduate at Indian Institute of Technology, Bombay. I am pursuing **Computer Science and Engineering** as my Major. I love coding. I'm just addicted to it. I have a keen interest in Mathematics and Theoretical Physics. I like reading books on theoretical physics, such as Feynman Lectures.

Other than this I like tinkering with electronic circuits and making small robotic vehicles. I also like dancing, listening to music and performing Card magic tricks. I'll like learning and trying out new things.

Programming Details

Platform and Text Editor Used

I use both **Mac OS X** and **Ubuntu 14.04 LTS** as my work machines. I use **Sublime Text 3** as my primary text editor because of its functionality, user-friendly nature and the useful packages which can be installed on it. Sometimes I also use vim.

Programming Experience

I have been using **C** and **C++** since the past one year and **python** since past 5 months.

Apart from these I also do web programming so I have used **Javascript, PHP** with **SQL** for databases. I've also used PHP with **Laravel** framework. I contribute to **mozilla**'s open source project 'wptview' which is a web app for displaying results of web-platform-tests.

I'm also good at using python for web crawling and scraping. I also know some **Django**. Recently won a Hackathon at IIT Bombay in which we created a web app with Django backend.

While working with SymEngine I also learnt about '**cmake**' and 'CMake files' - how to create them and how they function to create 'Makefiles'.

I use **git** and **Github** everyday and I am well acquainted with how to use them for version control.

Among all the programming languages, I'm most comfortable with C++ and use it in most of my projects. I developed a game of Checkers, with Artificial Intelligence, for one player , using C++ for my class project.

Contributions to SymEngine

I was introduced to SymEngine in mid-December 2015 and I have been contributing ever since. It has been an enriching experience and I look forward to continue contributing to it. A list of my contributions is as follows:

1. PR #678(**Merged**) - Wrote a python script to check for the presence of Trailing Whitespace in the code during continuous integration in Travis. From this I learnt how to use Travis for Continuous Integration.

2. PR #715(**Merged**) - Moved dependencies(Catch and Teuchos) to utilities directory so that all the dependencies are together in a single subfolder.

3. PR #727(**Merged**) - Added an additional compiling test for Travis with the latest gcc and g++ compiler(version 5.2 for both when I did this) so that new warnings coming up from these can be fixed. I learned more about Travis in this and also came to know about different kinds of compilers.

4. PR #736(**Merged**) - Adds five new functions, for finding quadratic residues, for checking if a number is a quadratic residue of another ,for checking if a number is a nth power residue of the other and added two helper functions to improve performance. Added tests for them. Also fixed variable names in another function which were causing ambiguity. This led to an issue for checking Integer overflows. This was my first PR in which I had to read about algorithms I hadn't heard of before. It taught me how to teach myself completely new stuff.

5. PR #758(**Merged**) - Removed some redundant code left by another Developer's PR.

6. PR #761(**Merged**) - Fixed a bug in *eval_mpfr.cpp* and *eval_mpc.cpp* where Euler's constant was being instead of Euler's Number.Added Tests for the same. This helped me learn about new libraries such as mpc, mpfr, mpz and how to use them - the syntax and functions used.

7. PR #767(**Merged**) - Code for Whitespace Check shifted to the end of Travis file so that other more important errors can be handled first, before the program exits due to trailing whitespaces.

8. PR #792(**Merged**) - Added complete functionality for functions sech, csch and acsch and their derivatives, printing and tests. This was my first major contributions. I added code to a lot of files. It helped me understand how all of these files communicate and works together.

9. PR #795(**Closed**) - This meant to prevent the use of 'Arb' when the user disables 'Flint'. It was closed since the discussion proved the corresponding Issue #788 to be Invalid. This helped me learn a lot of new things about cmake and how it works.

10. PR #807(**Merged**) - Improved and fixed functions of Polygamma class and added tests for the same.

11. PR #815(**Unmerged**) - Add a cmake switch to prevent Catch from catching exceptions so that stacktraces can be obtained from Teuchos. This helped me learn about how Catch works.

12. PR #835(**Merged**) - Improves the abs functions so that it can handle Complex. I also tried to implement functionality of abs(x-y) being treated the same as abs(y-x). This helped me gain a lot of insight into the core structure of SymEngine i.e. Add,Mul and the functions such as could_extract_minus.

- I also try to review some less complicated Pull Request and follow the discussions on others. I also follow discussion on SymEngine's Gitter Chat. This has helped me learn a lot of new stuff.

Contributions to SymPy

1. PR #10714(**Unmerged**) - Added complete functionality for Inverse hyperbolic function acsch. Added Tests.

2. PR #10721(**Merged**) - Fixed typo in documentation of asech function.

3. PR #10908(**Unmerged**) - Added tests for bugs found in evalf(). Closes Issue #5412

4. Issue #10717(**Open**) - Issue about missing acsch function which I fix in PR #10714.

The Project

What excites me

I've been working with SymEngine community from quite some time and want to remain associated with it for long. Making a significant contribution to it by taking up this

project would be the best method to achieve this and to develop better understanding with the mentors. Maybe I can be a mentor myself some day.

I have always liked Maths and a projects such as this would help me improve my skills. Solvers would expose me to diverse areas of mathematics such as interesting Polynomial manipulation algorithms, Set theory and help me learn new things in each.

Also being a programming and open source enthusiast, I've always wanted to take part in Google Summer of Code.

I also want to meet other people working in this area and learn from them. Open Source indeed has a lot of talented people.

Qualification

In this project I will be working with Polynomial Manipulation, Sets and Solvers of Polynomials.

The related courses I've taken are - Calculus, Linear Algebra, Differential equations. As regard the programming skills I've done courses on - Computer Programming and Utilisation, Introduction to Computer Science, Abstractions and Paradigms for Programming. I also have knowledge about Algorithms and Data Structures. I think I am well suited to work on this project.

Also since I've worked with SymEngine, I've got comfortable with its code. I recently audited SymEngine's code for Univariate Polynomials around which my whole project will be based.

I have also started auditing Sympy's code for Polynomial Manipulation, Set ,Solveset module and the Module and would complete that in the Pre-GSoC period.

I have done a lot of research for this proposal and I have tried to understand the algorithms I am porting. So the project will be intuitive for me and I'll make less algorithmic errors during porting.

The preparation of this proposal has given me a very good insight into the working of SymPy's Polynomial and Sets.

I also already have a pretty good bonding with the community.

Previous Implementations

Currently SymEngine has a UnivariatePolynomial class which has support for basic algebraic computations such as addition, subtraction and multiplication of Polynomials. It's just at the beginning stage and there is still a long way to go before the module is complete. Regarding Sets there is only a basic implementation of Intervals which got added very recently with this PR in which I helped in reviewing.

SymPy on the other hand has a very robust Polynomial Manipulation Module and a flexible Sets module. These would act as a good reference since these are very well documented. For Infinity class, there is a reference of Pynac's Infinity class. For understanding Polynomial Manipulation algorithms my best friend will be the Modern Computer Algebra Book by Gathen and Gerard.

Time during Summers

I have no other commitments this summer. So I'll be able to give a full 50 hours or more per week.

My summer break starts from 24th April so I can start working full time from that day on. I'll not be taking any vacations.

My classes start around 20th July. That might reduce the time given during weekdays but I'll compensate on weekends. I'll not be having any exams during that time so everything would be manageable.

What I want to Achieve

My aim through this project is to take the initial steps towards the implementation of Solvers Module for SymEngine. I basically want to set the ground so as to facilitate the porting of Solvers Module from SymPy.

The most important solvers in SymPy are the Polynomial Solvers. Hence it would naturally be the first thing to implement in SymEngine. This would involve improving the Polynomial module by implementing Polynomial Manipulation algorithms such as those for

- Division
- GCD
- Square Free Decomposition
- Polynomial factorisation
- Galois field and arithmetics in them
- other algorithms for finding roots of Polynomials.

Polynomial Solvers do a lot of heavy lifting in Sympy. Hence it is necessary to port them to SymEngine. I would like to focus on implementing Division, GCD, LCM for Univariate Polynomials, factoring Univariate Polynomials in Galois fields(finite fields) and also on implementing root finding algorithms for polynomials up to degree 5(which can be used to find roots of many higher degree polynomials by first factoring them and then using these algorithms on the factors).

Other thing required is a structure to represent solutions of Polynomials. For this I aim to implement some basic functionality to represent the roots of Polynomial. Sets is what would provide this functionality.

Infinity class is also missing in Symengine and it is required for the complete representation of Sets. It would also be helpful in the Calculus module soon to be implemented.

Solvers are a necessary part of any Computer Algebra System and since the final aim is to use SymEngine as a core for SymPy, we definitely need this functionality(especially Polynomial Solvers).

I also have an optional objective of getting Flint C++ polynomial wrappers to work with SymEngine for very fast polynomial manipulations.

Why I aim at Polynomials?

SymEngine aims to be very fast core for SymPy. The most robust solver in SymPy is the polynomial solver hence my main focus is on it. Everything else in solveset works by converting the equation to polynomial in one way or the other. Thus a lot of code depends on it. It does a lot of heavy lifting. The corresponding code in SymPy is settled, there are no major bug reports for code from a long time and we aren't expecting any api or structural changes. So it is a good time to port Polynomials to SymEngine.

Discussion

- For the design of Infinity class there was a confusion between following Sympy's design and Pynac's design. Discussion with Isuru Fernando and Ralf Stephan led to finalising on Pynac's design. This is because Pynac's design can be easily extended to handle any direction in Argand Plane while Sympy's design has Infinity, NegativeInfinity and ComplexInfinity as separate Singleton classes and the only way to accommodate more directions is by creating new classes.

- Sympy is currently trying to replace their Solve module with Solveset module which returns solution in the form of sets and also has other useful functionality such as checking if all solution have been found or not. This obviously led to the decision of porting Solveset, rather than Solve, to SymEngine. This requires a Sets module to be implemented(at least some basic functionality) before starting with the implementation of Solveset.

- As regard the polynomial factoring algorithm to be implemented for polynomials, there was a choice among two algorithms - **Berlekamp** and **Cantor-Zassenhaus**.

A fundamental difference between the two algorithms is that the Berlekamp algorithm is deterministic. This means that given any polynomial it will provide the unique factorisation eventually, regardless of how many steps it takes. The Cantor-Zassenhaus algorithm on the other hand is probabilistic, in that it can effectively fail to factor a polynomial completely. The tradeoff is that probabilistic factoring algorithms tend to utilise tricks that enable them to perform tasks quicker if they do succeed.

- Discussion also led me to believe that Sets such as ImageSet requiring non-trivial implementation are not required in SymEngine. They are already present in SymPy and don't take a lot of computational time. They are just for representing the solution and it's better to spend the same time on the implementation of Polynomial algorithms.

The Plan

1. Regarding the Infinity class, I've audited Sympy's Infinity and also taken a look at how Pynac(backend for symbolic expression in Sage) implements it. Pynac's

implementation is pretty interesting and I would be implementing SymEngine's Infinity along its lines.

2. Learning from Sympy, I propose to return Set as a solution of Solvers since Sets can be used to represent all types of solutions.

Since we don't have much support for Sets, these would have to be implemented first. The output of Solve would make heavy use of a lot of functionalities of Sets in the future. As of now we don't require the more fancy features.

The Set capabilities we need to have are

- Empty Sets - To Represent the solution of equation with No solutions.(x**2 + 1 = 0)
- Finite Sets - These would be required to represent discrete solutions such as those of Polynomials.(x**2 = 1)
- Interval - To Represents a real interval as a set. These are required for the representations such as range of solutions, domain of the independent variables.(floor(x) = 0)
- Universal Set - When we talk of complement of a set, it requires some kind of Universe for context.
- Union
- Intersection
- Complement - Represents the set difference of a set with another set.A - B = {x: x in A but not in B}
- Symmetric Difference - {x: x in A or x in B but not in both}

I would also implement some special and commonly required sets such as

- Naturals
- Whole Numbers
- Integers

3.Now moving to the most important part - Polynomial Manipulation Module. I'll now try to break the procedure of implementing Solvers into small parts. Since we don't have general solutions for polynomials of arbitrary degree, given a polynomial the first thing we want to do is to break it up into product of irreducible polynomials with smaller degrees such that we know how to solve each of these separately. The breaking up process is requires a factorisation algorithms for Polynomials. Currently we have nothing on this in SymEngine. This will be the first major task.

Since the factoring algorithm I plan to implement require the implementation of Finite Field arithmetics. I'll first implement Finite Field(Galois Field).

Galois Field

In mathematics, a finite field or Galois field (so-named in honour of Évariste Galois) is a field that contains a finite number of elements. As with any field, a finite field is a set

on which the operations of multiplication, addition, subtraction and division are defined and satisfy certain basic rules. The most common examples of finite fields are given by the integers mod p when p is a prime number.

Representation of Polynomials in finite fields implies that all the coefficients of the Polynomials lie in the field.

Factoring Polynomials(Cantor-Zassenhaus Algorithm)

In computational algebra, the Cantor–Zassenhaus algorithm is a well known method for factoring polynomials over Galois fields. The algorithm consists mainly of exponentiation and polynomial GCD computations. It was invented by David G. Cantor and Hans Zassenhaus in 1981. It is arguably the dominant algorithm for solving the problem, having replaced the earlier Berlekamp's algorithm of 1967.

As already mentioned above, the Cantor-Zassenhaus algorithm is a probabilistic algorithm, in that it can effectively fail to factor a polynomial completely. But it is faster(than Berlekamp's algorithm) if it succeeds.

One reason for implementing it first is the speed and the other is that it's the default algorithm used in Sympy.

Also note that the probability of the algorithm failing is low. Let F be a field of q=p**n elements, where p is prime.It uses d*log(q)(base2) random bits and fails to find a complete factorisation with probability no more than (1/q)**((1−e)*d/4) where d is the degree of the Polynomial and e is a fixed parameter between 0 and 1 which can be decided by the implementer. The failure probability is exponentially small in the number of random bits used.

The algorithm is not very trivial hence it is necessary to break it into smaller less complex parts.

The three stages of factorisation of Univariate Polynomials include

1. Square-Free Factorisation
2. Distinct Degree Factorisation
3. Equal Degree Factorisation

Square-Free Factorisation

Given a polynomial f, square–free decomposition(factorisation)[2] of f gives a list of polynomials (factors) f_1, f_2... f_n, such that all pairs of polynomials (f_i, f_j), for i not equal to j, are co–prime, and f = f_1*f_2^2*...*f_n^n. Thus each f_i has no repeated roots. Note that square–free decomposition does not give a true factorisation into irreducibles, although is a very important step in any factorisation algorithm.

Square Free Decomposition itself involves succession of Greatest Common Divisor computation and exact Divisions of Polynomials. Currently SymEngine has no GCD or division algorithm for Univariate Polynomials. These would have to be implemented before we go on to implement an algorithm for Square-Free Decomposition.

Division with Remainder

(For Polynomials with Integer coefficients as well as those with **Symbolic** coefficients)

Division between two Univariate polynomials f, g gives quotient q and remainder r such that f = g*q + r is satisfied with degree of r is less than q. I read up on the Euclidean algorithm for polynomial division. It is essentially the long division method but it works well for computers as well. But the classical division algorithm for polynomials requires O(n^2) operations for inputs of size n. Using reversal technique and Newton iteration, it can be improved to O(M(n)), where M is a multiplication time. Hence I propose to implement **Newton iteration method** Before implementing this I'll see how SymPy optimises it and then change the implementation process accordingly.

GCD and LCM

(For Polynomials with Integer and Rational coefficients)

With division, GCD and LCM of polynomials can be computed. GCD is a part of a lot of algorithms involving polynomials. An example is GCD of x^2 + 7x + 6 and x^2 − 5x − 6.

```
x^2 + 7x + 6 = (x + 1)(x + 6)
x^2 - 5x - 6 = (x + 1)(x - 6)
```

Thus, their GCD is x + 1 and LCM is (x + 6)(x + 1)(x - 6)

Distinct Degree Factorisation

The distinct-degree decomposition of a non-constant polynomial f is the sequence (g1,...,gs) of polynomials, where gi is the product of all monic irreducible polynomials of degree i that divide f and gs != 1 (but some gi for i < s may be 1). In other words this algorithm splits a square-free polynomial into a product of polynomials whose irreducible factors all have the same degree.

Equal Degree Factorisation

The remaining task is equal-degree factorisation: to factor each of the polynomials that are produced by the previous distinct-degree factorisation. This involves separating factors which have equal degrees. We want to factor a monic polynomial f with deg f = n, and have a divisor d ∈ N of n so that each irreducible factor of f has degree d. There are r = n/d(since this is equal-degree factorisation) such factors, and we can write f = f1 ⋯ fr with distinct monic irreducible f1,..., fr.

Any more details that I need to understand about this algorithm will be done alongside or some time prior to implementation.

This completes the factorisation process for polynomials. The final result we get is a set of polynomial factors. Since the degree of the resulting polynomials will be less than the original polynomial, each of these can be separately solved to find their roots. Each of the roots of these factor polynomials will be a root of the original polynomial.

- I've done enough study on this algorithm and I'm pretty confident that I'll be able to tackle any problem that come my way during its implementation.

Finding Roots

(For Polynomials with Integer as well as Symbolic coefficients)

Finally to solve the irreducibles we get from the above process, I want to implement algorithms for finding roots of some standard polynomials such as

- Linear(Degree = 1)
- Quadratic(Degree = 2)
- Cubic(Degree = 3)
- Binomial(Polynomial containing two terms only. For e.g. x**6 + x)
- Quartic(Degree = 4)
- Quintic(Degree = 5)
- Cyclotomic(The nth cyclotomic polynomial, for any positive integer n, is the unique irreducible polynomial which is a divisor of x**n-1 and is not a divisor of x**k-1 for any k < n. Its roots are the nth primitive roots of unity e^(2i*pi*k/n), where k runs over the integers lower than n and coprime to n.)

These are properly implemented in SymPy and porting the code would not be much of a problem since most of the algorithms used are trivial.

Why am I including two not so related projects(namely Sets and Solver for Polynomials) in proposal?

As you will see in the Timeline the actual focus of my proposal is Polynomials. This was actually a part of my strategy for handling the complexity of the implementation process. This will be the first time I will be adding a large chunk of code to SymEngine and I believe the following method would be the best way to tackle it. Sets and Infinity are non-algorithmic. Hence I feel they will not be very hard to implement. The thing I want to learn from these is how to make design decision, how to write significant amount of new code which fits into the previously present machinery and also write such code which is maintainable in the future and can be scaled if necessary. This might also help me to learn more about the programming language. I will have 11 weeks of time after implementing this on which I can focus solely on Polynomial algorithms and their implementations not being worried about the "syntax" of programming in SymEngine. I will then be able to able to read up more and polish up my knowledge on the algorithms.

Why I am not implementing all other Set functionality SymPy has?

The aim of my project is to develop solvers for Polynomials. My reason to implement sets is to be able to represent all solutions to polynomials. Some of the polynomial algorithms are known from centuries and the output is also uncontroversially a finite set of

numbers. Finite Set will thus be used to represent the roots of the polynomial. Intervals will be used for Root Isolation(Finding intervals rather than the number of roots with each interval containing one root). I'm not including ImageSet since it is not required for Polynomial solvers though it would be required when more solvers are implemented. It can be implemented then according to need.

Implementation Details

Infinity

The constructor would look something like

```
Infinity::Infinity(const Integer & _direction)
```

There will be other overloaded constructors taking input from other datatypes such as int but the behaviour will essentially be the same.

When _direction is positive, it behaves as Positive Infinity; when negative as Negative Infinity and when zero(0) it behave as Infinity with unknown direction(Complex Infinity).

Various other properties of Infinity will be defined such as:

```
1. Infinity(1)(or -1) + Infinity(1)(or -1) = Infinity(1)(or -1)
2. Infinity(0) + Infinity(0) = NaN
3. Finite + Infinity(1)(or -1)(or 0)  = Infinity(1)(or -1) (or 0)
4. Infinity(1)(or -1)(or 0)  - Infinity(1)(or -1)(or 0)  = NaN
5. PositiveFinite * Infinity(1)(or -1) = Infinity(1)(or -1)
6. NegativeFinite * Infinity(1)(or -1) = Infinity(-1)(or 1)
7. Infinity(1)(or -1)(or 0)  * Infinity(1)(or -1)(or 0)  = Infinity(1)(or -1)(or 0)
8. Finite / Infinity(1)(or -1)(or 0)  = Zero
9. Infinity(1)(or -1)(or 0)  / Infinity(1)(or -1)(or 0)  = Nan
10. Infinity(1)(or -1)(or 0) **Zero = Nan
11. Infinity(1)(or -1)**Infinity(1)(or -1) = Infinity(1)(or -1)
12. Infinity(0)**Infinity(0) = NaN
```

NaN class would follow the Singleton Pattern with the following properties:

```
1. NaN + 1 = NaN

2. Eq(NaN, NaN)   ==  false  // mathematical equality

3. (NaN == NaN)   ==  true  //structural equality
```

Any more properties required for some specific functions in symengine will also be implemented as required.

Sets

Using the concept of inheritance and keeping the Sets class as the parent class, other classes would be implemented as derived classes.

Constructors

- `FiniteSet(const std::vector<RCP<const Basic>>)`
- `Interval(const Basic start, const Basic end, bool left = false, bool right = false)`

  By default the interval will be closed as done in Sympy. The user will have to pass additional boolean arguments to get an open interval.
- `Union(const std::vector<const Set>)`

  This class is used to represent union of intervals such as Union((2,3), (4,5)) is (2,3) U (4,5). This cannot be directly represented with any of the sets we talked about earlier. Also this can compute Union for more than two sets.
- `Intersection(const std::vector<const Set>)`

  This class is used to represent Intersection of Sets. It can evaluate the intersection of more than two sets.
- `Complement(const std::vector<const Set>)`

  Computes complement of first set with respect to all others i.e. it forms a set from those elements which are present in the first set of the vector passed but are not present in any of the other Sets
- `Symmetric Difference(const std::vector<const Set>)`

  This also takes a vector of Sets as an input. It gives a set containing elements which are not present in the intersection of all the sets, but is present in at least one the Sets.

All this will be done along the lines of SymPy.

There would be many member functions for each of these classes. Some of them are:

- `set_union()`
- `set_intersection()`
- `is_disjoint()`
- `is_proper_subset()`
- `is_proper_superset()`
- `power_set()`

  This involves having sets of sets. Since the input type is Basic and Set is inherited from Basic this would be possible.
- `complement()`

  This involves complement with respect to the Universal Set.(i.e U - A)

Although I think this much functionality is more than enough, but any other functionality can be developed according to needs.

Singleton Pattern The Singleton Design pattern is used, where only one instance of an object is needed throughout the lifetime of an application. The Singleton class is instantiated at the time of first access and same instance is used thereafter till the application quits.

- EmptySet recently got implemented in SymEngine but it does not follow the singleton pattern of implementation. I plan to modify the implementation so that EmptySet becomes a Singleton.

- Universal Set - This would also follow the Singleton pattern. It is usually not directly used but rather used while taking intersections and complements. Some properties would then be defined accordingly. For e.g.

```
A Universal Set U
A Finite Set F
then
F.set_intersection(U) = F
F.set_union(U) = U
F.complement() = U - F
```

There are some special Sets to be implemented:

- Natural
- Whole Numbers
- Integer

These too will be developed as Singletons.

- Printing also has to be implemented for these Sets. This would be done so as the outputs can be used as direct inputs to Sympy.

Polynomial Manipulation Module

Exponentiation

When we have to do exponentiation i.e f**n where f is a Polynomial and n is an integer constant we apply the repeated squaring algorithm that has the complexity O(log(n)). This can be done iteratively or recursively as desired. We already have the function for multiplication of two polynomials. I will also see if this can be improved somehow for Squaring of polynomials.

Galois Field

Sympy has a special file for Galois field (*galoistools.py*) which contains arithmetics for algebraic computations over a Galois field. Some of these functions will have to be ported to SymEngine. The functions to be ported include:

- gf_from_int_poly - Create a GF(p)[x] polynomial from Z[x](i.e. It represent the polynomial in the Galois field) The declaration would be

```
gf_from_int_poly(const UnivariateIntPoly &a, const Integer p)
```

We can also have a function like

```
gf_from_vector(const std::vector<int> &a, const Integer p)
```

Now we require functions for addition, subtraction, multiplication, division, power, negation, squaring(can be used for exponentiation over the finite field) etc.

Most of the algorithms involve simple modular arithmetics. The corresponding functions for the required computations are

```
UnivariateIntPolynomial gf_add(const UnivariateIntPolynomial &f, const UnivariateIntPolynomial
    &g, const Integer p)
```

Add polynomials in GF(p)[x].

```
UnivariateIntPolynomial gf_sub(const UnivariateIntPolynomial &f, const UnivariateIntPolynomial
    &g, const Integer p)
```
Subtracts polynomials in GF(p)[x].

```
UnivariateIntPolynomial gf_neg(const UnivariateIntPolynomial &f, const Integer p)
```
Negate a polynomial in GF(p)[x].

```
UnivariateIntPolynomial gf_mul(const UnivariateIntPolynomial &f, const UnivariateIntPolynomial
    &g, const Integer p)
```
Multiplies polynomials in GF(p)[x].

```
UnivariateIntPolynomial gf_sqr(const UnivariateIntPolynomial &f, const Integer p)
```
Squares polynomials in GF(p)[x].

```
UnivariateIntPolynomial gf_div(const UnivariateIntPolynomial &f, const UnivariateIntPolynomial
    &g, const Integer p)
```
Division with remainder in GF(p)[x].

```
UnivariateIntPolynomial gf_pow(const UnivariateIntPolynomial &f,const Integer n, const Integer
    p)
```
Compute f**n in GF(p)[x] by repeated squaring.

```
UnivariateIntPolynomial gf_gcd(const UnivariateIntPolynomial &f, const UnivariateIntPolynomial
    &g, const Integer p)
```
Computes GCD of polynomials in GF(p)[x] by Euclidean-Algorithm(explained under GCD)

```
UnivariateIntPolynomial gf_lcm(const UnivariateIntPolynomial &f, const UnivariateIntPolynomial
    &g, const Integer p)
```
Computes LCM in GF(p)[x]. This function is dependent on the GCD, multiplication and division functions.

```
UnivariateIntPolynomial gf_diff(const UnivariateIntPolynomial &f, const Integer p)
```
Differentiate a polynomial in GF(p)[x]. This would be required in Square-Free decomposition of the polynomial which is a part of the factorisation process.

```
UnivariateIntPolynomial gf_random(const Integer n, const Integer p)
```
Creates a random polynomial in GF(p)[x] of degree n. This is required for Equal-Degree factorisation which is a part of Cantor-Zassenhaus algorithm for factoring.

```
UnivariateIntPolynomial gf_irreducible_p(const UnivariateIntPolynomial &f, const Integer p)
```
This will be used to check if the final result of factorisation is irreducible in GF(p)[x]. It uses **Ben-Or's** and **Rabin's Polynomial irreducibility test**.

```
UnivariateIntPolynomial gf_sqf_p(const UnivariateIntPolynomial &f, const Integer p)
```

This would be used to check if a polynomial is square-free in GF(p)[x].

Division

For every pair of polynomials (A, B) such that B ≠ 0, polynomial division provides a quotient Q and a remainder R such that A = BQ + R and either R=0 or degree(R) < degree(B). Moreover (Q, R) is the unique pair of polynomials having this property.

The process of getting the uniquely defined polynomials Q and R from A and B is called **Euclidean division** (sometimes division transformation). Polynomial long division is thus an algorithm for Euclidean division.

The pseudo-code is as follows

```
q = 0;
r = A;
r = degree(b);
c = LeadingCoefficient(b);
while(degree(r) >= d)
{
    s = LeadingCoefficient(r)*(x^(degree(r) - d))/c;
    q = q+s;
    r = r-sb;
}
return(q, r);
```

As already mentioned this is an O(n^2) algorithm. Hence we move to **Newton's Iteration method**. This method uses the reversal of polynomials to calculate the quotient and remainder polynomials.

The details of Newton's Iteration Method can be seen this handout from University of Waterloo.

GCD

A very simple algorithm for finding GCD of Polynomials is the **Euclidean Algorithm**. Starting from two polynomials a and b, Euclid's algorithm consists of recursively replacing the pair (a, b) by (b, rem(a, b)) (where "rem(a, b)" denotes the remainder of the Euclidean division, computed by the algorithm of the preceding section), until b = 0. The GCD is the last non-zero remainder. Recursively saying

```
gcd(a,b):= if b=0 then a else gcd(b, rem(a,b))
```

This is the algorithm currently used by SymPy for GCD over Galois field.

Some other algorithms for exploration are:

EEZ–GCD algorithm of Wang - This is a fast algorithm specifically for sparse polynomials and it computes GCDs of sparse multivariate polynomials over integers and rationals. Implementation of EEZ–GCD algorithm will require the variable–by–variable Hensel lifting algorithm.

Heuristic GCD algorithm - This transforms the problem of computing GCD of polynomials to integer GCD problem. This requires a fast kernel hence currently used in Maple only and SymPy also has an implementation Although the algorithm is heuristic, the same algorithm is implemented in SymPy and with current parametrization it never failed in SymPy. Because sophisticated sparse algorithm are relatively slow on small problems and emerge superior in large problems, this algorithm is chosen. It is faster than most algorithms for low degree polynomials of upto ten variables.

I'll have to find out how fast these algorithms are for Univariate Polynomials since the fast algorithms for Multivariate Polynomials usually turn out to be slower than the normal algorithms for Univariate algorithms. The optimal algorithm will be decided and implemented.

Square-Free Decomposition

For Square-Free factorisation I propose to use the fast algorithm of Yun.

Reference Yun's Algorithm

The cost of computing square–free decomposition is equivalent to the computation of the greatest common divisor of f and its derivative. Thus this is dependent on the GCD algorithm.

Input: A monic polynomial f

Output: Square-free factorisation of f

Working Principle

```
a_0 = gcd(f, f'); // f' is derivative of f
b_1 = f/a_0;
c_1 = f'/a_0;
d_1 = c_1 - b_1'; // b_1' is derivative of b_1
i = 1;

iterate
a_i = gcd(b_i, d_i);
b_(i+1) = b_i/a_i;
c_(i+1) = d_i/a_i;
i = i + 1;
d_i = c_i - b_(i+1)';
until b = 1
return a_1, a_2, ... ,a_(i-1)
```

Distinct Degree Factorisation

This will use the outputs of the Square-free factorisation process.

Input: A square-free monic polynomial f of degree n > 0.
Output: The distinct-degree decomposition (g1,...,gs) of f .

The pseudo-code to the algorithm is:

```
h0 = x;
f0 = f;
i = 0;

repeat
    i = i+1;
    Call the repeated squaring algorithm in R = Fq[x]/<f>
    to compute h(i) = h(i-1)^q rem f

    g(i) = gcd(h(i) - x, f(i-1);
    f(i) = f(i-1)/gi;

until fi=1

s=i;
return (g(1),g(2),g(3),g(4),...,g(s));
```

Thus we are finding one factor in each iteration.

Equal Degree Factorisation

This is the where the heart of Cantor-Zassenhaus algorithm lies. This is the step which makes the algorithm probabilistic in the sense that it depends on a randomly chosen polynomial.

Input: A squarefree monic polynomial f ∈ Fq[x] of degree n > 0, where q is an odd prime power, and a divisor d < n of n, so that all irreducible factors of f have degree d.
Output: A proper monic factor g ∈ Fq[x] of f , or “failure”.

The pseudo code for the algorithm is as follows:

```
1. choose a ∈ Fq[x] with degree of a < n at random

   if a ∈ Fq then return "failure"

2. g1 = gcd(a, f)
```

```
   if g1 != 1 then return g1

3. call the repeated squaring algorithm in R = Fq[x]/<f> to compute b = a^((q^d-1)/2) rem f

4. g2 = gcd(b-1, f)

if g2 != 1 and g2 != f then return g2 else return "failure"
```

Finding Roots of Lower Degree Polynomial

Here I will focus on implementing the algorithms for finding roots of the polynomials whose general solutions exist. I will mostly be doing this along the lines of SymPy.

- **Linear**(Degree = 1) The algorithm is very direct. ax + b is solved to give x = -b/a. Any simplification if possible is done on -b/a.

- **Quadratic**(Degree = 2) Finding roots of quadratic polynomials simply uses the Quadratic Formula.

- **Cubic**(Degree = 3) The solutions for cubics can also be found using a general formula. That's exactly how sympy does it and how I'm planning to implement it. For reference see General Formula for Roots of Cubic

- **Quartic**(Degree = 4) Quartics are relatively hard to solve and have harder algorithms as compared to the polynomials above. For quartics I plan to implement the famous Descartes-Euler algorithm. The steps involved are the following

1. Reduction to an incomplete equation by change of variable
2. Find roots of incomplete equation using the Cubic resolvent of the equation
3. Using these roots to get the roots of the original equation

   For reference see Quartic equation of general form

- **Quintic**(Degree = 5) It is a well known theorem that there can be no known solution for a Quintic polynomial. We can only solve for some specific kind of quintics. The irreducible quintic polynomials whose roots may be expressed in terms of radicals are called solvable quintics.

1. To recognise which quintic is solvable we use Cayley's Criterion.
2. After we know that the quintic is solvable we use Tschirnhaus transformation to depress the quintic.(that means removes the term of degree four)
3. Finding its roots is a more difficult problem, which consists of expressing the roots in terms of the coefficients of the quintic. Though this has been implemented in SymPy, currently I'm not very thorough with the algorithm as of now but I have a number of references for it and will be going through them during the GSoC period.

Here is the wiki page which has the algorithm and also reference to other pages which can be helpful.

- **Binomial** Binomials are the simplest polynomials after monomials. The roots can easily be expressed in terms of the coefficients of the binomial. This will be useful to all the above algorithms and hence I will be implementing this after Linear and before Quadratic polynomials.

  For e.g. The polynomial `ax^5 + bx` has solutions as `x=0` and the solutions of `x^4 = -b/a` . `x**n = constant` equations can be easily solved to find real and complex roots and require no specific named algorithm.

- **Cyclotomic Polynomials** For a given polynomial which is nth cyclotomic, first of all we need to know what n is for the given polynomial. This requires finding the inverse totient of the degree of the polynomial. Since Euler's Totient is not a one-one function the inverse is an interval rather than a single number. For calculating this interval a _inverse_totient_estimate() function will be defined. Then we loop over the interval comparing the original polynomial to each cyclotomic polynomial in that interval. After this is done, calculating the roots is straightforward.

```
Each root is exp(k*d) where d=2*pi*I/n and 1 <= k <= n and n and k are co-prime.
```

- All these would require some common helper functions which will have to be implemented.

Usage

The Set Module

Basic Functionality

The user will be allowed to declare different types of sets and perform operations of Union, Intersection, Complement, Symmetric Difference and many others.

Initialisation

```
RCP<const Basic> i1 = integer(1);
RCP<const Basic> i2 = integer(2);
RCP<const Basic> i3 = integer(3);
RCP<const Basic> i4 = integer(4);
RCP<const Basic> i5 = integer(5);
RCP<const Basic> i6 = integer(6);

std::vector<RCP<const Basic>> v1 = {i1,i2,i3,i4};
std::vector<RCP<const Basic>> v2 = {i1, i3, i5, i6};
```

```
//Infinity
RCP<const Basic> inf = infinity(1);

//Finite Sets
RCP<const Set> fs1 = finiteSet(v1);
RCP<const Set> fs2 = finiteSet(v2);

//Intervals
RCP<const Set> in1 = interval(i1, i5); // [1, 5]
RCP<const Set> in2 = interval(i3, i5,true,true); // (3, 5)
RCP<const Set> inf_set = interval(i3, inf); [3, oo)

//Empty Set
RCP<const Set> es1 = emptySet();

//Universal Set
RCP<const Set> us = universalSet();
```

Union

```
RCP<const Set> in3 = union({in1, in2});
RCP<const Set> fs3 = union({fs1, fs2});
RCP<const Set> in4 = union({fs2, in1});
```

Intersection

```
RCP<const Set> fs4 = intersection({fs1, fs2});
RCP<const Set> in5 = intersection({in1, in2});
RCP<const Set> es2 = intersection({in1, es1});
```

Complement

```
RCP<const Set> in6 = complement({in1, in2});
RCP<const Set> fs5 = complement({fs1, in2});
```

Symmetric Difference

```
RCP<const Set> fs6 = symmetricDifference({fs1, fs2});
```

Other Functionality

```
bool a = fs1.is_subset(in1);
bool b = in1.is_disjoint(in2);
```

After the above operations,

```
in3 = [1, 5]
fs3 = {1, 2, 3, 4, 5, 6}
in4 = [1, 5] U {6}
fs4 = {1, 3}
in5 = (3,5)
es2 = {}
in6 = [1, 3] U {5}
```

```
fs5 = {1, 2, 3}
fs6 = {2, 4, 5, 6}
a = true
b = false
```

Polynomial Manipulation Module

UnivariatePolynomial(Division) ***and*** **UnivariateIntPolynomial**(Division, GCD and LCM)

```
RCP<const Symbol> x  = symbol("x");
Expression a(symbol("a"));
Expression b(symbol("b"));
Expression c(symbol("c"));

RCP<UnivariatePolynomial> PExp = univariate_polynomial(x, 2,{{0, c}, {1, b}, {2, a}});
//ax**2 + bx + c
RCP<UnivariatePolynomial> QExp = UnivariatePolynomial::create(x,{2,3}); // 3x + 2

std::vector<UnivariatePolynomial> RExp = PExp->div_poly(QExp);
// RExp = {a*x/3 + b/3 - 2*a/9, 4*a/9 - 2*b/3 + c}


RCP<UnivariateIntPolynomial> PInt = univariate_int_polynomial(x, {{0, 1_z}, {1, 1_z}, {3,
    1_z}});
RCP<UnivariateIntPolynomial> QInt = univariate_int_polynomial(x, {{1, 1_z}, {2, 1_z}});

std::vector<UnivariateIntPolynomial> R = PInt->div_poly(QInt);
// R = {x + 1, 1} // x + 1 is the quotient. 1 is the remainder.

PInt = univariate_int_polynomial(x, {{0, 2_z}, {1, 3_z}, {2, 1_z}});
QInt = univariate_int_polynomial(x, {{0, -1_z}, {2, 1_z}});

UnivariateIntPolynomial K = PInt->gcd(QInt);
// K = x + 1;

UnivariateIntPolynomial L = PInt->lcm(QInt);
// L = (x+1)(x-1)(x+2)
```

Galois Field*(For UnivariateIntPolynomial)*

```
RCP<const Symbol> x  = symbol("x");
RCP<const UnivariateIntPolynomial> P = univariate_int_polynomial(x, {{0, 5_z}, {1, 2_z}, {2,
    3_z}});
const UnivariateIntPolynomial fa = gf_from_int_poly(P, 5); // fa = 3x**2 + 2x

const std::vector<integer_class> v = {integer_class(5), integer_class(3), integer_class(2)};
const UnivariateIntPolynomial fb = gf_from_vec(v, 5); // fb = 3x**2 + 2x
```

```
const UnivariateIntPolynomial f1 = gf_add(3x**2 + 2x + 4, 2x**2 + 2x + 2, 5); // f1 = 4x + 1
const UnivariateIntPolynomial f2 = gf_sub(3x**2 + 2x + 4, 2x**2 + 2x + 2, 5); // f2 = x**2 + 2

const UnivariateIntPolynomial f3 = gf_mul(x + 2, 2x + 3, 5); // f3 = 2x**2 + 2x + 1

const UnivariateIntPolynomial f4 = gf_sqr(3x + 4, 5); // f4 = 4x**2 + 4x + 1

const UnivariateIntPolynomial f5 = gf_pow(3x**2 + 2x + 4, 3, 5)
// f5 = 2x**6 + 4x**5 + 4x**4 + 2x**3 + 2x**2 + x + 4

const std::vector<UnivariateIntPolynomial> f6 = gf_div(x**3 + x + 1, x**2 + x, 2) // f6={x +
    1, 1}  q=x+1  r=1

const UnivariateIntPolynomial f6 = gf_gcd(3x**2 + 2x + 4,2x**2 + 2x + 3, 5) f6 = x + 3

const UnivariateIntPolynomial f7 = gf_lcm(3x**2 + 2x + 4, 3, 5) // f7 = x**3 + 2x**2 + 4

const UnivariateIntPolynomial f8 = gf_random(4, 5) // f8 = 2x**4 + 3x**2 + x + 4 (could have
    been any fourth degree polynomial belonging to the field)

const UnivariateIntPolynomial f9 = gf_diff(3x**2 + 2x + 4, 3, 5) // f9 = x + 2

bool b1 = gf_sqf_p(x + 1, 3) // b1 = true
bool b2 = gf_sql_p(x**2 + 2x + 1, 3) // b2 = false (Polynomial can be written as (x+1)**2 and
    x+1 belongs to the finite field)

bool b3 = gf_irreducible_p(x+1, 3) // b3 = true(x + 1 can't be factored further)
```

Factorisation*(For UnivariateIntPolynomial)*

```
RCP<const Symbol> x  = symbol("x");
RCP<const UnivariateIntPolynomial> P = univariate_int_polynomial(x, {{0, 1_z}, {1, 1_z}, {2,
    1_z}, {3, 1_z}});
std::vector<const UnivariateIntPolynomial> V = P.factor(1, 5); // V = {x + 1, x + 2, x + 3}
```

Finding Roots*(For UnivariatePolynomial and UnivariateIntPolynomial)*

```
RCP<const Symbol> x  = symbol("x");
Expression a(symbol("a"));
Expression b(symbol("b"));
Expression c(symbol("c"));

RCP<UnivariatePolynomial> PExp1 = univariate_polynomial(x, 1,{{0, b}, {1, a}});
//ax + b
RCP<const Set> fs_exp1 = PExp1.roots(); // fs_exp1 = {-b/a} // A FiniteSet

RCP<UnivariatePolynomial> PExp2 = univariate_polynomial(x, 2,{{0, c}, {1, b}, {2,a}});
//ax + b
RCP<const Set> fs_exp2 = PExp2.roots();
// fs_exp2 = {(-b + sqrt(-4*a*c + b**2))/(2*a), -(b + sqrt(-4*a*c + b**2))/(2*a)}
// A FiniteSet
```

```
RCP<UnivariatePolynomial> PExp3 = univariate_polynomial(x, 3,{{1, c}, {3, b}});
//bx**3 + cx
RCP<const Set> fs_exp3 = PExp3.roots();
// fs_exp3 = {0, +sqrt(-c/b), -sqrt(-c/b)}

RCP<const UnivariateIntPolynomial> P1 = univariate_int_polynomial(x, {{0, 2_z}, {1, 1_z}}); //
     Linear
RCP<const Set> fs1 = P1.roots(); // fs1 = {-2} // A FiniteSet

RCP<const UnivariateIntPolynomial> P2 = univariate_int_polynomial(x, {{0, 6_z}, {1, -5_z}, {2,
     1_z}}); //Quadratic
RCP<const Set> fs2 = P2.roots(); // fs2 = {2, 3} // A FiniteSet

RCP<const UnivariateIntPolynomial> P3 = univariate_int_polynomial(x, {{0, -6_z}, {1, 11_z},
     {2, -6_z}, {3, 1_z}}); // Cubic
RCP<const Set> fs3 = P3.roots(); // fs3 = {1,2, 3} // A FiniteSet

RCP<const UnivariateIntPolynomial> P4 = univariate_int_polynomial(x, {{0, 6_z}, {2, 1_z}}); //
     Binomial
RCP<const Set> fs4 = P4.roots(); // fs4 = {sqrt(6)I, -sqrt(6)I} // A FiniteSet

RCP<const UnivariateIntPolynomial> P5 = univariate_int_polynomial(x, {{0, 1_z}, {1, -1_z}, {2,
     1_z}});
//6th Cyclotomic polynomial
RCP<const Set> fs5 = P5.roots(); // fs5 = {e^(2*pi*I/6), e^(2*pi*5*I/6)} //Since 1,5 are
     co-prime to 6
// A FiniteSet

RCP<const UnivariateIntPolynomial> P6 = univariate_int_polynomial(x, {{0, 20_z}, {1, -26_z},
     {2, 17_z},
{3, -6_z}, {4, 1_z}}); //Quartic
RCP<const Set> fs6 = P6.roots(); // fs6 = {2 + I, 2 - I, 1 + sqrt(3)I, 1 - sqrt(3)I} // A
     FiniteSet
```

Timeline

Pre GSoC

- Before the official period of GSoC begins, I would do a rigorous audit of sympy's Sets and Solve, Solveset for algebraic equations especially UnivariatePolynomials, simultaneously helping out in solving bugs found in those modules. I would also go over Sympy's implementation of Polynomial factoring and root finding algorithms. In fact I've already got started with this task.
- Develop a bonding with Sympy community through the above task.
- Submit a wiki on the possible algorithms for various implementation and their details(Though I have already mentioned most of them in my proposal).
- Start with the implementation of Infinity and NaN(Not a Number) class and complete most of it.

Community Bonding Period

- Complete the Infinity and NaN class, send in a PR and get it merged during the first week of this period.
- Rest of the three weeks would be spent in implementing the following classes:
  - Finite Set
  - Empty Set
  - Intervals
  - Universal Set

  and their diverse functionalities.

Week 1

- Implement:
  - Union
  - Intersection
  - Complement

Week 2

- Symmetric Difference
- Natural Set
- Whole Number Set
- Integer Set
- Send in a PR
- Implement Division for Univariate Polynomials

Week 3

- Decide on the GCD algorithm - Euclidean or EEZ-GCD or Heuristic
- Implement GCD algorithm for Univariate Polynomials
- Benchmarking for GCD algorithm
- Send in a PR

Week 4

- Get started with Finite Fields(Read up on any specific topics in Galois Field if necessary)
- Implement:
  - gf_from_int_poly()
  - gf_from_vec()
  - gf_add()
  - gf_sub()
  - gf_mul()

- gf_sqr()
- gf_pow()

Week 5

- Implement:
  - gf_div()
  - gf_gcd()
  - gf_lcm()
  - gf_random()
  - gf_diff()
- Send in a PR

Week 6

- Implement Square-free decomposition
- gf_sqf_p() - For checking if a polynomial is square-free over the finite field.
- Send in a PR

Week 7

- Distinct-Degree factorisation
- gf_irreducible_p() - For checking if a polynomial is irreducible over the finite field.

Week 8

- Equal-Degree factorisation
- Linking the whole Cantor-Zassenhaus process and getting it to work.
- Send in a PR

Week 9,10

- Getting all helper functions ready for root finding process.
- Implement Solver for:
  - Linear Polynomial
  - Binomials
  - Quadratic Polynomial
  - Cubic Polynomial
- Send in a PR

Week 11,12

- Implement Solver for
  - Quartic Polynomial
  - Quintic Polynomial
- Send in a PR

- Implement inverse_totient_function
- Implement Solver for
    - Cyclotomic Polynomials -Send in a PR

Week 13

- Buffer Time for TODOs, for writing more test to achieve 100% coverage for my code and for proper documentation of my code. Try to get the PRs merged.

Note: The tests for the methods implemented will be written simultaneously with the methods, this is not specifically mentioned in the timeline. I've tried to structure the timeline so that there is no week which is purely for coding or purely for learning.

Optional Stuff

I'll be doing this if I have extra time left out or if at some point of time in future the community comes to believe that this has a higher priority than some of the above mentioned material.

Wrapping FLINT polys and make them usable in SymEngine

FLINT

FLINT is a fast library for number theoretic computations, written in C. That is to say, it implements arbitrary-precision data types, and arithmetic and higher operations on them. For example, it provides arbitrary precision integers (so no matter how many times you multiply by two, you never get an overflow; although the required memory will increase linearly). Basically, anything you might want to ask about particular elements of the ring of integers, FLINT should be able to do.

FLINT also has similar support for the rational numbers, **finite fields**, and **polynomial** and matrix rings over these. Thats where it becomes useful to my project and to SymEngine.

FLINT(flint2) Repository Link

Since FLINT is in C, it needs be wrapper in C++ for use in SymEngine. Tom Bachmann wrote C++ wrappers for FLINT during Google Summer of Code 2013. Now what remains is to modify the polynomial wrappers so as to work with SymEngine. Since FLINT is written in C it provides field for very fast manipulation of polynomials.

Also from what I get from the discussion here(Issue #877) , FLINT might soon be a dependency for SymEngine and then these wrappers will be all the more necessary.

I don't have a lot of experience of writing wrappers for C code but I've been trying to learn by looking at the C++ wrappers for FLINT(present in the flint2 repository itself) as well as those for ARB (named ARB++) and how they relate to the actual code. In case we do this

during the summers I'll read up more on the conventions and best designs for the task and then I can contribute toward its implementation.

Post-GSoC

I would continue contributing to the community. I plan to take GSoC project as a platform to be one of the strong contributors of SymEngine. I have a lot of things in mind to keep me engaged after the GSoC period. There are many other things that would be left to do in Polynomials. These will be developed in the coming year. I would like to be a major contributor to the same. I hoping to myself become a mentor in the coming years. SymEngine aims to be a very fast core for SymPy. I want to be a part of the team when Sympy starts using SymEngine as its core. Knowing that millions of people will be directly or indirectly using my code gives a unique satisfaction.

Blog

I'll be maintaining a blog at codemaxx.github.io to record my progress each week so that the community can keep track of my work and also give their opinion on it.

References

1. Gitter Chat with @isuruf @rwst @Sumith1896
2. Discussion on Mailing list - SymEngine
3. Discussion on mailing list SymPy
4. Modern Computer Algebra Book by Gathen and Gerhard
5. Square-free Polynomial - Yun's Algorithm
6. Cantor-Zassenhaus Algorithm
7. Berlekamp Algorithm
8. Sympy's code for Sets and polynomials
9. Mateusz Paprocki's Master Thesis for understanding code structure
10. THE EEZ-GCD ALGORITHM, Paul S. Wang
11. Sympy's code for Heuristic GCD
12. Evaluation of Heuristic Polynomial GCD, Liao, Fateman
13. Polynomial division
14. Galois Field
15. Sympy's implementation of Finite Fields
16. Finding roots of Quintic Equation
17. Sympy's Documentation for Sets
18. Sympy's Documentation for Solveset
19. Euclidean algorithm for GCD
20. Sympy's code for solvers of lower degree polynomials
21. Distinct-Degree factorisation
22. Hensel's Lifting