

SymPy

Improving Series Expansions and Limit Computations

for Google Summer of Code 22'

Anutosh Surendra Bhat

IIT MADRAS (2019 - 2024)

Potential Project Mentors:

[@jksuom](#), [@0sidharth](#)

Contents

- [Personal Information](#)
 1. Points of contact and other relevant links
 2. Mathematics Background
 3. Programming Background
 4. Motivation behind participation in Gsoc 22'
- [Contributions to sympy](#)
- [Overview of the Project/Module](#)
- [Proposed Improvements and Implementations](#)
 1. Improve leading term & series method for functions involving branch cuts/ branch points
 2. Adding nseries, aseries and leading term support to elementary/special functions
 3. Fixing limit errors through gruntz
 4. Refactor Order code and order.py to handle cases better
 5. Miscellaneous
- [Timeline to be followed](#)
- [References and Links](#)

Personal Information

- **Name** - Anutosh Surendra Bhat
- **University** - Indian Institute of Technology Madras (IIT M)
- **Degree** - Integrated Dual Degree (Bachelors and Masters)
- **Major** - Data Science and Engineering
- **Residence** - Madras/Chennai, India
- **Timezone** - Indian Standard Time (UTC + 5:30)
- **Github Account** - [GitHub](#)
- **Email** -
 1. be19b037@smail.iitm.ac.in (Institute mail)
 2. andersonbhat491@gmail.com (Github linked mail)
 3. anutosh.bhat.21@gmail.com (Personal use)
- **Language spoken** - English

I am [Anutosh](#) , a third year undergraduate pursuing an integrated dual degree in the Data Science and Computational Biology Department of IIT Madras . I have been fortunate enough to study at [IIT Madras](#) since 2019 which holds the national ranking of **1**, currently in India which is a country producing 15 lakh engineering graduates every year through lakhs of institutions nationwide.

My hobbies include playing football , beatboxing , cycling on campus and watching documentaries whenever I find some free time . I end up attempting around 5-8 above average coding problems on platforms such as LeetCode or Codechef on a daily basis to keep my coding skills sharp and improve my logic. I have also started contributing to other open source libraries recently to explore my math based interest on other domains like Networkx for graph theory.

Mathematical Background

I have been interested and motivated to learn mathematics since my childhood as I used to take part in various state and national level olympiads under the guidance of my father who used to train me almost on a daily basis back then.

Math also had a major role to play eventually in my decision to take up engineering . Apart from this some courses which I've taken at my institute are [Multivariate Calculus](#), [Series and Matrices](#), [Graph Theory](#), [Probability](#), [Stochastic Process and Statistics](#), [Applied Statistics](#) and a couple more .

Programming Background

- **General Programming Background** - My first interaction with programming took place back in my 9th grade . The school I was studying in had made it compulsory to introduce everyone to programming , just to teach the power programming holds. Hence I had prior experience with C++ and Java , even before joining my institute.

My first year was filled with exploring various domains and tech stacks that coding had to offer in general like web and android app development, Data Science and Machine Learning etc .

It was only in my second year when I got to know about python and I'll be sharing my experience below .

- **Python experience** - As I wrote above , I was introduced to python back in August of 2020 when I had registered for the following course on [Programming, computing and graphics using Python](#) at University . The course really showed me the power python carries and all its advantages over other low level languages like C or C++ .Debugging became quite easy using the [pdb module](#) and the [ipython console](#).Though there are some things I didn't like about python , I remember being in awe of functionalities such a list comprehension and comparison operator chaining back then . I ended up learning a lot from the course and also secured a complete score in the course project linked to it .

I have been working in python ever since, I have undertaken courses based heavily upon python at university such as [Data Science- Theory and Practice](#), [Algorithmic approach to Computational Biology](#) to name a few . These courses were accompanied with weekly labs and assignments , which helped me polish my python further.

Though python has quite advanced features like generators and decorators and some really useful modules like itertools , functools and the collections module , my favorite python feature is the packing/unpacking operator. Although some other languages also have implementations of the packing/unpacking operator , I never really understood the concept behind it until I read about how it has been implemented in python .

- **Git and Github experience** - I have been using git for a year now and I am quite familiar with all workflows involved with git and github . I understood the more tedious and error prone stuff like rebasing on other branches, hard and soft reset etc through making a decent number of mistakes on my Pr's while contributing to sympy
- **Platform Details** - I use Ubuntu 20.04 as my operating system on WSL 2, Windows 10 along with Spyder as my primary source code editor whenever I am working on a python project. Spyder helps me a lot while working on my course assignments and I find that Spyder has almost all necessary benefits a comprehensive development tool should have along with competences for scientific packages like Numpy, Matplotlib, Pandas and also SymPy. Spyder offers an inbuilt isympy console too which was also my first point of interaction with sympy .
- **Favorite sympy feature** - Coming to sympy, my favorite feature is definitely the pretty printing module. The implementation is too good and makes contributing to sympy even much easier .I remember struggling to read expressions when I started out contributing as I had no idea about the `pprint` method back then

An example to demonstrate it would be as follows

```

>>> n = Symbol('n')
>>> expr = (-1 + (n**2*cos(1/n) + n**2 + 1)/(n**2*cos(1/n) + 1))
>>> pprint(expr)
      2      (1)
      n  cos| - | + n  + 1
      (n)
-1 + -----
      2      (1)
      n  cos| - | + 1
      (n)

>>> expr.simplify()
n**2/(n**2*cos(1/n) + 1)
>>> pprint(_)
      2
      n
-----
      2      (1)
      n  cos| - | + 1
      (n)

```

This literally shows how easy pprint makes visualizing simplifications and expressions in general in sympy.

Motivation behind participating in GSOC 22' -

My motivation behind taking part in Gsoc 22' especially as a contributor to sympy is based quite heavily on my interest in mathematics , logic and programming in general . If someone told me a year back or two that there exists a library solving complex differential equations or something like engineering mechanics systems and other niche math and physics problems , I wouldn't have believed them . I am astonished till date when I think about what sympy has achieved solely through the help of contributors like me . I don't think I would have applied to any organization at least for this year other than some Math/Science based organizations like Sympy, Sage, NumFocus, Networkx etc and out of these Sympy was the best match.

The open source community overall has taught me a lot of things which I wouldn't be aware of otherwise like communication ethicets , writing up to standard code , building and testing before review and other things too . Having the opportunity to contribute to large codebases and maintain them too was really something that I was looking forward too and participating in Gsoc 22' gives me a perfect chance to fulfill my desires.

Contributions to sympy

I was somewhat familiar with a few domains/modules of sympy even before I had started contributing as I had used it in a course project of mine at University. I started out contributing to sympy last year in the month of September, when I was quite new to open source and was contributing to any open source library for the first time. Hence I was only able to contribute good meaningful code by the end of November or first half of December when I had developed a decent amount of familiarity with open source and sympy in particular. The following is a list of Pr's and issues opened by me so far ...

I'll start with those which have been **Merged /Closed**

(Merged) #22230 - Appropriate tests added for `limit(x**n - x**(n-k), x, oo)` which had been fixed on master.

(Closed) #22264 - The Pr was meant to fix `simplify/doi` methods for `Summations` and `Products` and was working fine but it was pointed out that the implementation might affect sympy's performance, hence was closed for not being the best approach for solving the issue.

(Merged) #22403 - Added Tests for `symbolic limits` at `infinity` that had been fixed on master

(Merged) #22491 - Fixed `limit(1/asin(x), x, 0)` which was returning wrong outputs along both directions

(Merged) #22614 - Fixed bug(system of 2nd order Ode failing) in `solve_ics` method

(Merged) #22870 - Fixes series for expressions of type Pow having unspecified exponents for eg `series(x**(I + 1), x)`

(Merged) #22905 - Improved consistency between basic limits , fixed leading term method for `tan(1/x)` , `cot(1/x)` , `csc(1/x)` , `sec(1/x)` and implemented `_eval_as_leading_term` method for `csc(x)`

(Merged) #22934 - **Polys/Domains**: Fixes domain containment check for the Fraction fields class

(Merged) #23127 - Fixes leading term for **exponential** functions involving **AccumBounds** based arguments for eg leading terms for such expressions was fixed `exp(2*x*(log(tan(1/x)**2))).as_leading_term(x)`

(Merged) #23241 - Fixed **leading term** method inconsistency for expressions involving variables in powers

(Merged) #23258 - Series/limits : Fixes limit computations where `arg0` would evaluate to S.NaN for the **ceiling** and **floor** functions

Now I'll be listing down my Pr's which are currently **Unmerged**. These Pr's form a decent part of my Gsoc proposal/proposed work and will be explained thoroughly in the implementation section. I plan to finish work on all of these by the end of the coding period of GSOC 22'. I would also like to specify the work left on these using labels as **Looks_Good_To_Go**, **Could_Be_Improved**, **Needs_Decision**, **Needs_Work** and **Needs_Review** .

(Needs_Review/ Could_be_Improved) #22193 - Fixed the `Sum.doit()` cases where eventually after splitting the components we would end up in an oo-oo form . For eg fixes `Sum(sqrt(x) - 1/x, (x, 1, oo)).doit()` to return oo

(Looks_Good_To_Go) #22200 - Fixes Ratio-Test which skips checking whether the limit ratio was S.NaN or not , leading to wrong results from basic Sums

(Needs Work) #22247 - Fixed methods like `extract_leading_term` , `Order` and `is_convergent` for the expression `(1 - cos(1/x))` , which weren't functioning as required

(Looks_Good_To_Go/Needs_Review) #22339 - Fixes Subs where the old and the new expression both have an `Order` term

(Looks_Good_To_Go) #22422 - Addresses multiple minor bugs/modifications in the series module (Docstring changes , adding tests etc)

(Needs_Work) #22521 - Added `_eval_nseries`, `_eval_as_leading_term` methods for `asech` and `acsch` functions. (Works for all points except couple special points like branch cuts at ± 1)

(Could_Be_Improved/ Needs_Work) #22639 - Modified code in `Fourier.py` to obtain fourier coefficients

(Needs_Review) #22804 - Fixed `taylor_term` method for multivariate and other cases

(Looks_Good_To_Go) #23044 - Added `_eval_nseries`, `_eval_as_leading_term` method for `frac` function

(Needs_Work) #22995 - Fixes limits for basic inverse ,hyperbolic functions .(Corner cases at special points need to be observed carefully)

(Needs_Work) #23190 - Fixes limits for special functions like `lowergamma` and `uppergamma` functions. (`_eval_rewrite_as_tractable` methods implemented for these currently are buggy and leading term methods must be implemented)

(Looks_Good_To_Go) #22806 - Fixes differentiation and series for multivariate cases (more than 2 variables). Current Implementation was quite buggy

(Needs_Work) #22555 - Fixes basic bugs encountered while evaluating limits of Piecewise Functions (Works for all cases but code for bi-directional cases needs to be improved)

(Looks_Good_To_Go/Needs_Review/Could_Be_Improved)#22463-
Implemented function `return_piecewise` for evaluating limit when parameter `z0` is passed as `Symbol`. Implementation example

I am most proud of the last three contributions mentioned above since I started contributing to sympy in the series module.

Issue opened - Most of these would also be addressed during Gsoc 22'

(Open) #22149 - Sympy should be supporting these basic **Summation** (Convergence /Divergence) properties

(Open) #22178 - Improvements in **Sum.doit()** for functions having the \pm operator

(Open) [#22252](#) - Deprecation of `extract_leading_order()` of `core/add.py`

(Open) [#22334](#) - Wrong answer returned while calculating symbolic limits for different arrangements of the same expression

(Closed) [#22533](#) - `Integral(1/abs(x), x).doit()` returns an incomplete answer as output

(Open) [#22590](#) - Regarding Dirichlet's test of convergence in `summations.py`

(Open) [#22795](#) - `taylor_term` method not working as expected for multivariate series expansions

(Open) [#22836](#) - Possible improvements for Order of expressions involving factorials

(Open) [#22982](#) - `limit((log(E + 1/x) - 1)**(1 - sqrt(E + 1/x)), x, oo)` returns 0 instead of oo

Besides the above work, I have also helped fellow contributors by reviewing PRs in the **series** and **calculus** based modules, and have also helped users and contributors in the Gitter chat room.

Overview of the Project/Module

- [What has been achieved](#) - Sympy currently has some great modules which have been developed to a large extent. This also includes the series module. The series module has been very fortunate to Google Summer of Code as a good chunk of the series module has been developed under guidance of mentors during the Gsoc coding period.

It currently allows users to generate various types of series which include Power, Taylor, Laurent and Puiseux. Other series like Fourier series, Formal Power series and Ring series, were introduced to sympy or heavily worked upon during the Gsoc coding period.

Sympy also currently provides a decent functionality of asymptotic expansions through the series module which has been used for approximation of functions . This also helps sympy to solve limits for expressions which might be basic or even complex involving special functions. Sympy uses the Gruntz Algorithm internally for computing the above limits .

The Ring series module which is outside the series module has also been developed to some extent based on efficient operations and manipulation of sparse polynomials. This currently provides the users with a good amount of leverage in speed to the standard series method which was introduced earlier.

- **What needs to be achieved** - Although the series module has been implemented to a good extent , it is not yet complete . The amount of bugs reported under the **series**, **limits** and **ring_series** labels overall are only second to the integrals, polys and core module.

These bugs mainly consist of limit failures and incorrect series expansions.

Shortcomings in series expansions range from incorrect Multivariate Taylor series expansions where either terms returned are wrong or some terms have been skipped to incorrect asymptotic expansions which either return mathematically wrong results or haven't been Implemented yet.

The limit failures might be due to incorrect results through the gruntz algorithm where a wrong asymptotic series expansions might be returned or the algorithm is not able to cater to different assumptions on symbols for calculating symbolic limits. Limit failures can also arise due to specific cases not being handled correctly in the leading term method of functions and also due to the leading term method not being implemented for a good amount of special functions.

The ring series module also is far from complete and hasn't been worked upon seriously since 2015 . Our sister CAS which has a decent

implementation of the ring series for e.g. Sage didn't start too long back but currently have a full fledged implementation of ring series methods for univariate/multivariate power, laurent, puiseux and lazy formal power series.

A Formal power series when denoted as an infinite series can be given combinatorial meaning, which makes them an indispensable tool in discrete mathematics. Generating functions are the most powerful tool available to combinatorial enumerators and there are some CAS which already have decent implementations for ordinary and exponential generating functions.

There are also important factors like computing Order of an expression or the AccumBounds method that play a good role in computing series and limits and these too have a good scope of improvement as I'll be pointing out below.

Hence a decent amount of work is required in the series module. This work required is extensive enough that it would need contributions by new contributors in future even after I complete the Gsoc project. The sole reason for this is that there are a huge number of basic (elementary, trigonometric and hyperbolic) and special functions and infinite combinations can be made out of them and there are infinite points where a series expansion or a limit can be computed. Hence we could always land ourselves a wrong series expansion or limit computation in the future through sympy.

Proposed Improvements and Implementations

- Improve leading term & series method for functions involving branch cuts/ branch points

A **branch cut** is a curve in the complex plane across which an analytical multivalued function is discontinuous. The purpose of having

a branch cut is to define a single analytical branch of a multivalued function on a plane correctly. Branch cuts are usually though not always, taken between pairs of **branch points**.

Series and limit computations on or across branch cuts of functions have always been a concern for sympy since the time the series and leading term method for such functions were introduced . We can also find issues back from 2008 regarding this topic ([#3663](#)) and has been neglected ever since.

This idea would involve refactoring the series and leading term method of almost all **inverse** functions in **trigonometric.py**, **hyperbolic.py** . I'll address some known issues here first

- [#3663](#), [#10868](#) - Currently sympy lacks series expansion support for **asec**, **acsc**, **asech**, **acsch** . The latter two also lack any sort of implementation for the **_eval_as_leading_term** method .

```
>>> acsc(x).series(x)
nan
>>> asec(x).series(x)
nan
>>> acsch(x).series(x)
nan
>>> asech(x).series(x)
Nan
>>> acsch(x).as_leading_term(x)
Traceback (most recent call last):
...
NotImplementedError: acsch has no _eval_as_leading_term
routine
>>> asech(x).as_leading_term(x)
Traceback (most recent call last):
...
NotImplementedError: asech has no _eval_as_leading_term
routine
```

This is because these functions lack the **taylor_term** method, and the **_eval_nseries** method which has only been

implemented for `asec/acsc` functions hasn't been tested enough . This needs to be checked .

The second part of this issue goes as follows .I had implemented the `taylor_term` and the `_eval_as_leading_term` method for functions `asech` and `acsch` through [#22521](#) and had solved one of the issues(the second one) mentioned above .

The **proposed code** goes as follows for the `acsch` case

```
@staticmethod
@cacheit
def taylor_term(n, x, *previous_terms):
    if n == 0:
        return log(2 / x)
    elif n < 0 or n % 2 == 1:
        return S.Zero
    else:
        x = sympify(x)
        if len(previous_terms) > 2 and n > 2:
            p = previous_terms[-2]
            return p * (n - 1)**2 // (n // 2)**2 * x**2 / 4
        else:
            k = n // 2
            R = RisingFactorial(S.Half, k) * n
            F = factorial(k) * n // 2 * n // 2
            return S.NegativeOne**(k + 1) * R / F * x**n / 4

def _eval_as_leading_term(self, x, logx=None, cdir=0):
    from sympy.series.order import Order
    arg = self.args[0].as_leading_term(x)
    if x in arg.free_symbols and Order(1, x).contains(arg):
        return -1*log(arg)
    else:
        return self.func(arg)
```

This comfortably addressed all bugs pointed out in the issue and also implements a basic series.

```

def test_issue_10868():
    assert limit(log(x)+asech(x),x,0, dir = '+') == log(2)
    assert limit(log(x)+asech(x),x,0, dir = '-') == log(2) + 2*I*pi
    assert limit(log(x)+acsch(x),x,0, dir = '+') == log(2)
    assert limit(log(x)+acsch(x),x,0, dir = '-') == -oo
    assert asech(x).as_leading_term(x) == -log(x)
    assert acsch(x).as_leading_term(x) == -log(x)

def test_acsch_series():
    x = Symbol('x')
    assert acsch(x).series(x, 0, 8) == \
        log(2) - log(x) + x**2/4 - 3*x**4/32 + 5*x**6/96 + O(x**8)
    t6 = acsch(x).taylor_term(6, x)
    assert t6 == 5*x**6/96
    assert acsch(x).taylor_term(10, x, t6, 0) == 63*x**10/2560

```

But [Kalevi's 1st comment](#) on the issue correctly pointed out that branch cuts haven't been taken care of and the leading term and series methods would need special handling for points on branch cuts, and even more special treatment for endpoints like 0, -1, 1 and maybe points representing infinite values in some sense like ∞ , ∞ etc. The committed code wasn't able to address cases on $\pm 1/-1$ for asech, though it could for acsch. The straightforward reason being acsch has branch points located along the complex axis, hence it fails for $\pm 1/-1$.

```

>>> asech(x).series(x, 1)
Traceback (most recent call last):
.....
      raise PoleError("Cannot expand %s around 0" % (self))
sympy.core.function.PoleError: Cannot expand asech(_x + 1) around 0
>>>
>>> asech(x).series(x, -1)
Traceback (most recent call last):
.....
      raise PoleError("Cannot expand %s around 0" % (self))
sympy.core.function.PoleError: Cannot expand asech(_x - 1) around 0
>>>
>>> acsch(x).series(x, I)
Traceback (most recent call last):
.....
      raise PoleError("Cannot expand %s around 0" % (self))
sympy.core.function.PoleError: Cannot expand acsch(_x + I) around 0
>>> acsch(x).series(x, -I)
Traceback (most recent call last):
.....
      raise PoleError("Cannot expand %s around 0" % (self))
sympy.core.function.PoleError: Cannot expand acsch(_x - I) around 0

```

Hence after investigating series and limits for all functions having branch cuts, **I realized support for series and limits at functions having branch cuts is almost nil as of now on master.** Consider the following table from [wolfram](#) .

function name	function	branch cut(s)
inverse cosecant	$\csc^{-1} z$	$(-1, 1)$
inverse cosine	$\cos^{-1} z$	$(-\infty, -1)$ and $(1, \infty)$
inverse cotangent	$\cot^{-1} z$	$(-i, i)$
inverse hyperbolic cosecant	csch^{-1}	$(-i, i)$
inverse hyperbolic cosine	\cosh^{-1}	$(-\infty, 1)$
inverse hyperbolic cotangent	\coth^{-1}	$[-1, 1]$
inverse hyperbolic secant	sech^{-1}	$(-\infty, 0]$ and $(1, \infty)$
inverse hyperbolic sine	\sinh^{-1}	$(-i\infty, -i)$ and $(i, i\infty)$
inverse hyperbolic tangent	\tanh^{-1}	$(-\infty, -1]$ and $[1, \infty)$
inverse secant	$\sec^{-1} z$	$(-1, 1)$
inverse sine	$\sin^{-1} z$	$(-\infty, -1)$ and $(1, \infty)$
inverse tangent	$\tan^{-1} z$	$(-i\infty, -i]$ and $[i, i\infty)$

Currently the series for no function which has branch cuts along the imaginary axis works although a couple having branch cuts along the real axis do work . The couple which work are **acos** and **asin**. The results go as follows

```

>>> asin(x).series(x, 1, 3)
pi/2 - sqrt(2)*I*sqrt(x - 1) + sqrt(2)*I*(x - 1)**(3/2)/12 -
3*sqrt(2)*I*(x - 1)**(5/2)/160 + O((x - 1)**3, (x, 1))
>>>
>>> asin(x).series(x, -1, 3)
-pi/2 + sqrt(2)*sqrt(x + 1) + sqrt(2)*(x + 1)**(3/2)/12 + 3*sqrt(2)*(x +
1)**(5/2)/160 + O((x + 1)**3, (x, -1))
>>>
>>> acos(x).series(x, 1, 3)
sqrt(2)*I*sqrt(x - 1) - sqrt(2)*I*(x - 1)**(3/2)/12 + 3*sqrt(2)*I*(x -
1)**(5/2)/160 + O((x - 1)**3, (x, 1))
>>>
>>> acos(x).series(x, -1, 3)
pi - sqrt(2)*sqrt(x + 1) - sqrt(2)*(x + 1)**(3/2)/12 - 3*sqrt(2)*(x +
1)**(5/2)/160 + O((x + 1)**3, (x, -1))

```

This is because there have been careful log based rewrites here, in the `_eval_nseries` blocks of these functions for handling series at `+1/-1`. These considerations haven't been taken care of for other functions and should be addressed too.

```

if arg0 is S.One:
    t = Dummy('t', positive=True)
    ser = asin(S.One - t**2).rewrite(log).nseries(t, 0, 2*n)
    arg1 = S.One - self.args[0]
    f = arg1.as_leading_term(x)
    g = (arg1 - f)/f
    if not g.is_meromorphic(x, 0): # cannot be expanded
        return O(1) if n == 0 else S.Pi/2 + O(sqrt(x))
    res1 = sqrt(S.One + g)._eval_nseries(x, n=n, logx=logx)
    res = (res1.removeO()*sqrt(f)).expand()
    return ser.removeO().subs(t, res).expand().powsimp() + O(x**n, x)

if arg0 is S.NegativeOne:
    t = Dummy('t', positive=True)
    ser = asin(S.NegativeOne + t**2).rewrite(log).nseries(t, 0, 2*n)
    arg1 = S.One + self.args[0]
    f = arg1.as_leading_term(x)
    g = (arg1 - f)/f
    if not g.is_meromorphic(x, 0): # cannot be expanded
        return O(1) if n == 0 else -S.Pi/2 + O(sqrt(x))
    res1 = sqrt(S.One + g)._eval_nseries(x, n=n, logx=logx)
    res = (res1.removeO()*sqrt(f)).expand()
    return ser.removeO().subs(t, res).expand().powsimp() + O(x**n, x)

```

Some results which fail are the following

```

>>> atan(x).series(x, 1)
>>> atanh(x).series(x, 1)
>>> asinh(x).series(x, -1)
>>> acosh(x).series(x, 1)
>>> acot(x).series(x, -1)
>>>
>>> # All these give a similar error
Traceback (most recent call last):
  raise PoleError("Cannot expand %s around 0" % (self))
sympy.core.function.PoleError: Cannot expand atanh(x + 1) around 0

```

These results above show a **huge potential** for improvement in the `series` module as all these cases need to be addressed . I have currently only spoken about the series side of things here but it is quite easy to find `limits` failing here too . Hence this would require a good amount of testing and careful observation of how series and limits behave at branch points for all the inverse `trigonometric` and `hyperbolic` functions mentioned above.

Theory and Proposed Implementation

[Kalevi's 2nd comment](#) on this discussion somewhat shows the way to go about this . As we know that `Inverse hyperbolic` (and `trigonometric`) functions can be expressed as `logarithms` of algebraic expressions, we need to improve methods dealing with series and limits for the log function at `origin/infinities` . A comment based on which this improvement should be made is [this one](#) . Hence it makes sense to address all issues related to series and limits of log , then use log based rewrites for implementing `as_leading_term`, `nseries` and `aseries` methods for functions with branch points . The final proposed implementations for leading term and series methods for such functions will be shown at the end of this section.

Some Improvements for the log function

1. [Comment on 22521](#) , [Comment on 22453](#) - These comments deal with the leading term method of log function . The leading term method needs to be fixed especially for branch points at **0** and **-∞** . It also seems that logarithms could be considered as constants while using them in leading terms as this is also expected by gruntz.

The code for handling the **0** based branch cut cases goes as follows

```

    if x0 == 1:
        return (arg0 - S.One).as_leading_term(x)
+   if x0 is S.Zero:
+       c, e = arg.as_coeff_exponent(x)
+       arg1 = (arg0 - arg).as_leading_term(x, cdir=cdir)
+       c1, e1 = arg1.as_coeff_exponent(x)
+       if c > 0:
+           if re(cdir) >= 0:
+               return log(c) + e*log(x)
+           else:
+               if im(c1) >= 0:
+                   return log(c) + I*pi + e*log(-x)
+               return log(c) - I*pi + e*log(-x)
+       if c < 0:
+           if re(cdir) >= 0:
+               if im(c1) >= 0:
+                   return log(-c) + I*pi + e*log(x)
+               return log(-c) - I*pi + e*log(x)
+           else:
+               return log(-c) + e*log(-x)

```

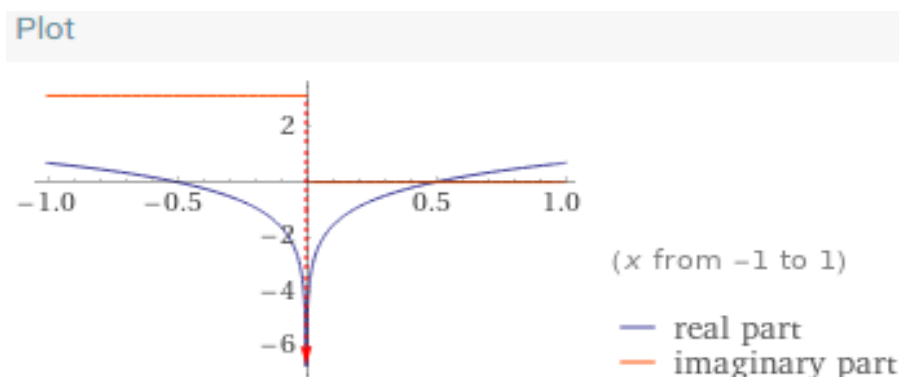
Some results obtained

```

>>> log(2*x).as_leading_term(x, cdir = 1)
log(x) + log(2)
>>>
>>> log(2*x).as_leading_term(x, cdir = -1)
log(-x) + log(2) + I*pi
>>>
>>> log(x**2).as_leading_term(x, cdir = 1)
2*log(x)
>>>
>>> log(x**2).as_leading_term(x, cdir = -1)
2*log(-x) + I*pi

```

We can confirm the result for the first one (**log(2*x)**) graphically



This would also improve all buggy limit results for log involving the branch cut at origin. For eg

```
>>> limit(log(-2*x + (3 - I)*x**2) - log(x), x, 0, "+")
log(2) - I*pi          # on master log(2) + I*pi
>>>
>>> limit(log(-2*x + (3 - I)*x**2) - log(x), x, 0, "-")
log(2) - I*pi          # on master log(2) - I*pi
>>>
>>> limit(log(2*x + (3 - I)*x**2) - log(x), x, 0, "+")
log(2)                  # on master log(2)
>>>
>>> limit(log(2*x + (3 - I)*x**2) - log(x), x, 0, "-")
log(2) - 2*I*pi        # on master log(2)
```

The branch cut at negative infinity could in most cases be answered through [this comment on 22220](#). Maybe something like this would do the job .

```
+   if x0 is S.ComplexInfinity:
+       res = -self.subs(x, 1/x).as_leading_term(x, cdir=cdir)
+       if re(cdir) >= 0:
+           return res
+       return res + 2*I*S.Pi
```

2. [Comment-1 on 22220](#), [Comment-2 on 22220](#) - These comments are related to computation on limits based on Gruntz algorithm, which involves generating series expansions for logarithmic functions at branch cuts . Hence even the **nseries** method for log needs improvement.

I plan to use this definition of log to deal with this as shown in [this link from Wolfram](#).

$$\log(z) = \log(a) + \log\left(\frac{z}{a}\right) + 2i\pi \left\lfloor \frac{\arg(z-a)}{2\pi} \right\rfloor ; a < 0$$

The reason behind using the following definition is because it is based on the leading term's coefficient being negative , which is where the error lies currently

The Code for fixing this is as follows

```
-         if a.is_real and a.is_negative and im(cdir) < 0:
-             res -= 2*I*S.Pi
-         return res + Order(x**n, x)
-
+         if p != 0:
+             res += Order(x**n, x)
+             # branch handling
+             if a.is_real and a.is_negative:
+                 from sympy import floor, arg
+                 l = floor(arg(p.removeO()*a)/(2*S.Pi)).limit(x, 0)
+                 if l.is_finite:
+                     res += 2*I*S.Pi*l
+         return res
```

We can see that we need to make a decision based on a being negative and the complex argument of `p` as prompted by the above formula. We make decisions based on `im(cdir)` currently but support for complex `cdir` is anyways very minimal as of now in sympy , hence using that approach might not be the best way to address all cases currently. This fixes issues pointed out on both comments mentioned above and the original issue

```

# Comment 1
>>> x = Symbol('x')
>>> expr=log(x-I)-log(-x-I)
>>> expr2=logcombine(expr,force=True)
>>>
>>> limit(expr, x, oo)      # -I*pi on master
I*pi
>>> limit(expr2, x, oo)
I*pi
-----
# Comment 2
>>> limit(-log(-cot(x/2) - I) + log(-cot(x/2) + I), x, 0)  # 0 on master
2*I*pi
>>>
>>> expr = log(-cot(x/2) - I)
>>> expr.series(x, cdir = 1)      # first term is I*pi on master
-I*pi + log(2) - log(x) + I*x/2 + x**2/24 + x**4/2880 + O(x**6)
>>>
-----
# Original Issue
>>> e1 = sqrt(30)*atan(sqrt(30)*tan(x/2)/6)/30
>>> e2 = sqrt(30)*I*(-log(sqrt(2)*tan(x/2) - 2*sqrt(15)*I/5) +
log(sqrt(2)*tan(x/2) + 2*sqrt(15)*I/5))/60
>>>
>>> limit(e1, x, -pi)
-sqrt(30)*pi/60
>>> limit(e2, x, -pi)      # 0 on master which is fixed now
-sqrt(30)*pi/30
>>> limit(e1, x, -pi, '-')
sqrt(30)*pi/60
>>> limit(e2, x, -pi, '-')
0

```

3. We can also see the improvements that `log` based rewrites would offer. Now this is somewhat helping us without making any changes on master currently, but I can't see the same happening for those functions with branch cuts along the imaginary axis (results shown below), hence this surely needs some improvement.

```

>>> asech(x).series(x, 1)
sympy.core.function.PoleError: Cannot expand asech(_x + 1)
around 0
>>>
>>> asech(x).rewrite(log).series(x, 1, 4)
sqrt(2)*I*sqrt(x - 1) - 5*sqrt(2)*I*(x - 1)**(3/2)/12 +
43*sqrt(2)*I*(x - 1)**(5/2)/160 - 177*sqrt(2)*I*(x -
1)**(7/2)/896 + O((x - 1)**4, (x, 1))

```

```

>>> acsch(x).rewrite(log).series(x, I)
-I*pi/2 + (-3/2 - sqrt(2)*I*sqrt(-I)*(-3/4 + 3*I/4))*(x - I)**2 + (x - I)**3*((16/5)*(-1)*I +
sqrt(2)*I*sqrt(-I)*(-2*sqrt(2)*I*sqrt(-I) + 2*I*(-3/4 + 3*I/4))/3 +
I*(2*sqrt(2)*I*sqrt(-I)*(-3/4 + 3*I/4) - 1)/3 - sqrt(2)*sqrt(-I)*(-3/4 + 3*I/4)/6 - (-3/4 +
3*I/4)**2/2 + 673*I/240 - sqrt(2)*I*sqrt(-I)*(sqrt(2)*I*sqrt(-I)*(2*sqrt(2)*I*sqrt(-I)*(-3/4 +
3*I/4) - 1) - 2*sqrt(2)*I*sqrt(-I) + 2*I*(-3/4 + 3*I/4))/4) + (x -
I)**4*(-sqrt(2)*I*sqrt(-I)*3 +
sqrt(2)*I*sqrt(-I)*(sqrt(2)*I*sqrt(-I)*(sqrt(2)*I*sqrt(-I)*(2*sqrt(2)*I*sqrt(-I)*(-3/4 + 3*I/4)
- 1) - 2*sqrt(2)*I*sqrt(-I) + 2*I*(-3/4 + 3*I/4)) - 6*I - 2*sqrt(2)*sqrt(-I)*(-3/4 + 3*I/4)) +
8*sqrt(2)*sqrt(-I) - 3*I)/6 - sqrt(2)*I*sqrt(-I)*(sqrt(2)*I*sqrt(-I)*(-39*I/8 + (-3/4 +
3*I/4)**2) + I*(-2*sqrt(2)*I*sqrt(-I) + 2*I*(-3/4 + 3*I/4)) + 55*sqrt(2)*sqrt(-I)/16 + (-3/4 +
3*I/4)*(2*sqrt(2)*I*sqrt(-I)*(-3/4 + 3*I/4) - 1))/4 -
I*(sqrt(2)*I*sqrt(-I)*(-2*sqrt(2)*I*sqrt(-I) + 2*I*(-3/4 + 3*I/4)) +
I*(2*sqrt(2)*I*sqrt(-I)*(-3/4 + 3*I/4) - 1) - 2*sqrt(2)*sqrt(-I)*(-3/4 + 3*I/4) - 2*I)/4 -
I*(16*(-1)*I - 4*I)/6 - (-3/4 +
# just showing 5% of the material here and I had to get rid of 95% to show the change
# this fills up the entire console
# and takes up around 15/20 seconds
I*(sqrt(2)*I*sqrt(-I)*(sqrt(2)*I*sqrt(-I)*(2*sqrt(2)*I*sqrt(-I)*(-3/4 + 3*I/4) - 1) -
2*sqrt(2)*I*sqrt(-I) + 2*I*(-3/4 + 3*I/4)) - 6*I - 2*sqrt(2)*sqrt(-I)*(-3/4 + 3*I/4))/5 +
sqrt(2)*I*sqrt(-I)*(sqrt(2)*I*sqrt(-I)*(16*(-1)*I - 4*I) + 4*sqrt(2)*sqrt(-I))/7 +
sqrt(2)*I*sqrt(-I)*(9/2 + sqrt(2)*I*sqrt(-I)*(87*sqrt(2)*I*sqrt(-I)/8 + (-3/4 +
3*I/4)*(sqrt(2)*I*sqrt(-I)*(sqrt(2)*I*sqrt(-I)*(2*sqrt(2)*I*sqrt(-I)*(-3/4 + 3*I/4) - 1) -
2*sqrt(2)*I*sqrt(-I) + 2*I*(-3/4 + 3*I/4)) - 6*I - 2*sqrt(2)*sqrt(-I)*(-3/4 + 3*I/4)) + I*(9/2
+ sqrt(2)*I*sqrt(-I)*(sqrt(2)*I*sqrt(-I)*(-2*sqrt(2)*I*sqrt(-I) + 2*I*(-3/4 + 3*I/4)) +
I*(2*sqrt(2)*I*sqrt(-I)*(-3/4 + 3*I/4) - 1) - 2*sqrt(2)*sqrt(-I)*(-3/4 + 3*I/4) - 2*I) +
I*(sqrt(2)*I*sqrt(-I)*(2*sqrt(2)*I*sqrt(-I)*(-3/4 + 3*I/4) - 1) - 2*sqrt(2)*I*sqrt(-I) +
2*I*(-3/4 + 3*I/4)) - 9*I/2 + 2*sqrt(2)*sqrt(-I)) + sqrt(2)*I*sqrt(-I)*(71/8 + (-3/4 +
3*I/4)*(sqrt(2)*I*sqrt(-I)*(2*sqrt(2)*I*sqrt(-I)*(-3/4 + 3*I/4) - 1) - 2*sqrt(2)*I*sqrt(-I) +
2*I*(-3/4 + 3*I/4)) + I*(sqrt(2)*I*sqrt(-I)*(-2*sqrt(2)*I*sqrt(-I) + 2*I*(-3/4 + 3*I/4)) +
I*(2*sqrt(2)*I*sqrt(-I)*(-3/4 + 3*I/4) - 1) - 2*sqrt(2)*sqrt(-I)*(-3/4 + 3*I/4) - 2*I) +
sqrt(2)*I*sqrt(-I)*(sqrt(2)*I*sqrt(-I)*(-39*I/8 + (-3/4 + 3*I/4)**2) + I*(-2*sqrt(2)*I*sqrt(-I)
+ 2*I*(-3/4 + 3*I/4)) + 55*sqrt(2)*sqrt(-I)/16 + (-3/4 + 3*I/4)*(2*sqrt(2)*I*sqrt(-I)*(-3/4 +
3*I/4) - 1))))/7) + O((x - I)**6, (x, I))

```

This seems to be a simplification problem rather than a code error or a logical error. The console fills up due to unsimplified expressions. The block which needs simplification is this one

```

@@ -1013,6 +1013,7 @@ def mul(d1, d2):
    ex = e1 + e2
    if ex < n:
        res[ex] = res.get(ex, S.Zero) + d1[e1]*d2[e2]
+   res[ex] = res[ex].nsimplify()
    return res

>>> # Result after this
>>> acsch(x + I).rewrite(log).series(x, 0, 4)
-I*pi/2 + sqrt(2)*I*sqrt(x)*sqrt(-I) + x**(3/2)*(-5/12 + 5*I/12) +
x**(5/2)*(9/20 - 23*sqrt(2)*I*sqrt(-I)/32 + 9*I/20) + x**(7/2)*(177/896 -
177*I/896) + O(x**4)
>>>
>>> # nsimplify does a good job but we may be able to do better

```


We also need to be quite careful about the mathematical correctness while using rewrites here, simply using a rewrite currently won't do the job. For eg

```
>>> acsch(x).rewrite(log).as_leading_term(x)
Traceback (most recent call last):
sympy.core.function.PoleError: Cannot expand log(sqrt(1 + x**(-2))
+ 1/x) around 0
>>> acsch(x).rewrite(log)
log(sqrt(1 + x**(-2)) + 1/x)
>>> # somewhat rearrangement of the expression works though
>>> # taking a common denominator of both terms in log works
>>>
>>> log((sqrt(1 + x**2) + 1)/x).as_leading_term(x)
log(2/x)      # correct answer of expected !
```

- We would have accomplished quite a good amount of series and limit computations once we fix the above issue. Hence, Once we have the log rewrite working perfectly, we could move on to implementing `_eval_nseries` and `_eval_aseries` method for all inverse hyperbolic functions (currently none of these functions have these methods).

Consider the following issue [#22986](#)

```
>>> x = Symbol('x')
>>> limit(acosh(1 + 1/x)*sqrt(x), x, oo)
0 ---> correct answer is sqrt(2)
```

Now that we have the `log` rewrites working perfectly, we could just base these methods on the `log` function. A glimpse of how this would look like will be the following

```

diff --git a/sympy/functions/elementary/hyperbolic.py
b/sympy/functions/elementary/hyperbolic.py
index bfc7589b5..4af3520aa3 100644
--- a/sympy/functions/elementary/hyperbolic.py
+++ b/sympy/functions/elementary/hyperbolic.py
@@ -1368,6 +1368,8 @@ def _eval_as_leading_term(self, x, logx=None, cdir=0):
     if arg0.is_zero:
         return S.ImaginaryUnit*S.Pi/2
     elif arg0.is_finite:
+         if arg0 in (-S.One, S.One):
+             return self.rewrite(log)._eval_as_leading_term(x, logx=logx,
+ cdir=cdir)
         return self.func(arg0)
     else:
         return self
@@ -1381,6 +1383,13 @@ def inverse(self, argindex=1):
     """
     return cosh

+ def _eval_nseries(self, x, n, logx, cdir=0):
+     x0 = self.args[0].limit(x, 0)
+     if x0 in (-1, 1):
+         return self._eval_rewrite_as_log(self.args[0])._eval_nseries(x, n,
+ logx, cdir=0)
+     else:
+         return super()._eval_nseries(x, n, logx, cdir=0)
+

```

We can see that this would also end up working currently on master because log rewrite works correctly for `acosh(x)` (this might and will not happen for all functions correctly as of now of master . As shown above easily something like `acsch(x)` could go wrong). The results are as follows ([results](#) from wolfram for reference)

```

>>> # Currently on master
>>> acosh(x).rewrite(log).series(x, 1)
sqrt(2)*sqrt(x - 1) - sqrt(2)*(x - 1)**(3/2)/12 + 3*sqrt(2)*(x -
1)**(5/2)/160 - 5*sqrt(2)*(x - 1)**(7/2)/896 + 35*sqrt(2)*(x -
1)**(9/2)/18432 - 63*sqrt(2)*(x - 1)**(11/2)/90112 + O((x - 1)**6, (x, 1))
>>>
>>> acosh(x).rewrite(log).series(x, -1)
I*pi - sqrt(2)*I*sqrt(x + 1) - sqrt(2)*I*(x + 1)**(3/2)/12 - 3*sqrt(2)*I*(x +
1)**(5/2)/160 - 5*sqrt(2)*I*(x + 1)**(7/2)/896 - 35*sqrt(2)*I*(x +
1)**(9/2)/18432 - 63*sqrt(2)*I*(x + 1)**(11/2)/90112 + O((x + 1)**6, (x, -1))
>>>
>>> acosh(x + 1).rewrite(log).as_leading_term(x)
sqrt(2)*sqrt(x)
>>>
>>> acosh(x - 1).rewrite(log).as_leading_term(x)
I*pi
>>>
>>> limit(acosh(1 + 1/x).rewrite(log)*sqrt(x), x, oo)
sqrt(2)

```

Hence we could address all `_eval_nseries` methods like the above shown diff. We could easily see `_eval_aseries`, being implemented similarly .

```

>>> acosh(x).series(x, oo)
Traceback (most recent call last):
  File "C:\Users\anuto\sympy\sympy\sympy\core\expr.py", line 2977, in series
    s = self.subs(x, sgn/x).series(x, n=n, dir='+', cdir=cdir)
  File "C:\Users\anuto\sympy\sympy\sympy\core\expr.py", line 3006, in series
    rv = self.subs(x, xpos).series(xpos, x0, n, dir, logx=logx, cdir=cdir)
  File "C:\Users\anuto\sympy\sympy\sympy\core\expr.py", line 3014, in series
    s1 = self._eval_nseries(x, n=n, logx=logx, cdir=cdir)
  File "C:\Users\anuto\sympy\sympy\sympy\core\function.py", line 702, in
_eval_nseries
    return self._eval_aseries(n, args0, x, logx)
  File "C:\Users\anuto\sympy\sympy\sympy\core\function.py", line 663, in
_eval_aseries
    raise PoleError(filldedent('''
sympy.core.function.PoleError:
Asymptotic expansion of acosh around [oo] is not implemented.
>>>
>>> acosh(x).rewrite(log).series(x, oo)
-3/(32*x**4) - 1/(4*x**2) + log(2) - log(1/x) + O(x**(-6), (x, oo))
>>> acosh(x).rewrite(log).series(x, -oo)
-3/(32*x**4) - 1/(4*x**2) + I*pi + log(2) - log(-1/x) + O(x**(-6), (x, -oo))

```

- These were all the improvements related to series expansions of functions having branch cuts/branch points . Talking about limits, the limit method also performs poorly for functions involving branch cuts/points on master.

```

>>> limit(atan(x), x, I) # expected I*oo
Limit(atan(x), x, I)
>>> limit(atan(x), x, -I) # expected -I*oo
Limit(atan(x), x, -I)
>>> limit(acot(x), x, I) # expected -I*oo
Limit(acot(x), x, I)
>>> limit(acot(x), x, -I) # expected I*oo
Limit(acot(x), x, -I)
>>> limit(atanh(x), x, 1) # expected oo
Limit(atanh(x), x, 1)
>>> limit(atanh(x), x, -1) # expected -oo
Limit(atanh(x), x, -1)
>>> limit(acoth(x), x, 1) # expected oo
Limit(acoth(x), x, 1)
>>> limit(acoth(x), x, -1) # expected -oo
Limit(acoth(x), x, -1)
>>> limit(asec(x), x, 0) # expected I*oo
zoo
>>> limit(asec(x), x, 0, '-') # expected -I*oo
zoo
>>> limit(acsc(x), x, 0, '+') # expected -I*oo
zoo
>>> limit(acsc(x), x, 0, '-') # expected I*oo
zoo
>>> limit(acsch(x), x, 0, '+') # expected oo
zoo

```

As correctly pointed out by [Kalevi's Comment](#), the `_eval_as_leading_term` methods for all these functions need to be completely rewritten.

Proposed Solutions

- 1) Those which return **self** need correct leading terms being implemented for the respective points . These are missing currently. Once that is done we could get all the correct answers through the heuristics block . For eg

```

# After adding the following block for atanh

    if arg0 is S.NaN:
        arg0 = arg.limit(x, 0, dir='-' if cdir.is_negative
        else '+')
+ if abs(arg0) is S.One:
+     return -log(-x)/2 if arg0 > 0 else log(x)/2

-----
>>> limit(atanh(x), x, 1)
oo
>>> limit(atanh(x), x, -1)
-oo

```

- 2) Those which return **zoo** , need a **taylor_term** method implementation which will be done as pseudocode for this is provided on page 13.

Hence the final proposed implementations for such functions involving branch cuts and branch points goes as follows .
 Demonstrating code using function **asinh**

```

+ def _eval_nseries(self, x, n, logx, cdir=0): # asinh
+     from sympy import Dummy, im, O
+     arg0 = self.args[0].subs(x, 0)
+     if arg0 is S.ImaginaryUnit:
+         t = Dummy('t', positive=True)
+         ser = asinh(I + t**2).rewrite(log).nseries(t, 0, 2*n)
+         # code goes here
+         # .....
+         # .....
+
+     if arg0 is S.NegativeOne:
+         t = Dummy('t', positive=True)
+         ser = asinh(-I - t**2).rewrite(log).nseries(t, 0, 2*n)
+         # code goes here
+         # .....
+         # .....
+
+     res = Function._eval_nseries(self, x, n=n, logx=logx)
+     if arg0 is S.ComplexInfinity:
+         return res
+     # Handling points between Imaginary branches (-I*oo, -I).union(I, I*oo)
+     if cdir != 0:
+         cdir = self.args[0].dir(x, cdir)
+         if im(cdir) < 0 and arg0.is_real and im(arg0) < S.NegativeOne:
+             # return respective result
+         elif im(cdir) > 0 and arg0.is_real and im(arg0) > S.One:
+             # return respective result
+     return res

```

```

+ def _eval_as_leading_term(self, x, logx=None, cdir=0):
+     arg = self.args[0]
+     arg0 = arg.subs(x, 0).cancel()
+     if arg0 is S.NaN:
+         arg0 = arg.limit(x, 0, dir='-' if cdir.is_negative else '+')
+     if arg0.is_zero:
+         return arg
+     elif arg0.is_finite:
+         return self.func(arg0)
+     elif arg0 is S.ComplexInfinity:
+         return log(sqrt(x**2) + 1/x)
+     # Handling points between
+     # Imaginary branche points (-I*oo, -I).union(I, I*oo)
+     if cdir != 0:
+         cdir = arg.dir(x, cdir)
+     if im(cdir) < 0 and x0.is_real and im(x0) < S.NegativeOne:
+         # return respective result
+     elif im(cdir) > 0 and x0.is_real and im(x0) > S.One:
+         # return respective result
+     return self

```

The implementation for the `aseries` method would be quite similar to that of the `nseries` method. Though for some special cases like `atanh` and `acoth` are interconvertible through various methods hence we could do the following for `atanh`

```

>>> # atanh(1/x) is same as acoth(x)
>>> # acoth(1/x) is same as atanh(x)
-----
+ def _eval_aseries(self, n, args0, x, logx):
+     if args0[0] is S.Infinity:
+         return (I*S.Pi/2 + atanh(1/self.args[0]))._eval_nseries(x, n, logx)
+     elif args0[0] is S.NegativeInfinity:
+         return (-I*S.Pi/2 + atanh(1/self.args[0]))._eval_nseries(x, n, logx)
+     else:
+         return super()._eval_aseries(n, args0, x, logx)
+

```

- Improving `nseries`, `aseries` and leading term support for elementary/special functions

Sachin Agarwal and Sidharth Mundra worked quite heavily on leading term and `nseries` methods of core and special function classes through their Gsoc projects in [2020](#) and [2021](#). Sympy's series module is also greatly dependent on these leading term methods. These leading term methods extract the first/leading/the most useful term when it comes to

computing limits and are very useful while calculating series expansions too . Hence it is important to address buggy leading term methods or in some cases implement them from scratch.

Asymptotic series expansions were introduced primarily in sympy through Avichal Dayal Gsoc project in [2014](#) and Argihna Chakrabarthy then built on this work in [2019](#) to further improve asymptotic expansions . But since then, a good number of special functions have been added to sympy and there is almost no support for asymptotic expansions for these functions . Currently only the **error functions** and few functions(4 in number) based on the **gamma function** such as **uppergamma**, **lowergamma**, **polygamma** etc have aseries support. Hence as we lack series expansions for such functions we aren't able to compute quite some limits through gruntz as it is the last step required for the algorithm to return the limit .

Hence under this header I plan to improve leading term , nseries and aseries methods for elementary as well as special functions

1) `_eval_nseries` and `_eval_as_leading_term` for `frac` function

We currently have the following errors related to the `frac` function on master

```

>>> x =Symbol('x')
>>> limit(frac(x), x, 0, '+')
frac(_w)
>>> limit(frac(x), x, 0, '-')
0
>>> limit(frac(x + 1), x, 0, '-')
0
>>> limit(frac(-x), x, 0, '+')
0
>>> limit(frac(-x), x, 0, '-')
frac(-_w)
>>> limit(frac(-2*x + 1), x, 0, '+')
0
>>> limit(frac(x), x, 0, '+-')
Traceback (most recent call last):
  raise ValueError("The limit does not exist since "
ValueError: The limit does not exist since left hand limit = 0 and
right hand limit = frac(_w)

```

These results are clearly wrong and need `_eval_nseries` and `_eval_as_leading_term` implementations for correcting these. There is a **proposed implementation** in my Pr [#23044](#) which has been reviewed to a basic extent, it fixes all the above issues. I've also taken care of other error's that could arise. This would be reviewed and checked more carefully for any errors.

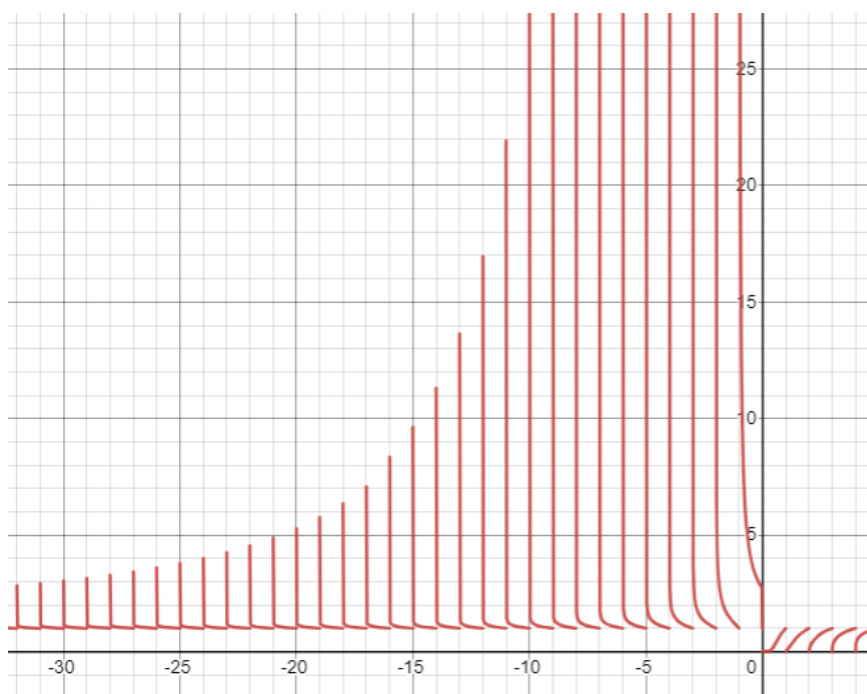
Another thing the pr does is improve `_eval_as_leading_term`, of the **exponential function** to handle some good corner cases. This was something I had discovered while implementing a particular limit for `frac`. Consider the following

```

>>> limit(frac(x)**(1/x), x, -oo)
1      # but this expected ans could be anything from 1 to oo

```

The graph goes like this



Hence the Pr also addresses this corner case which would also lead to improvements of similar corner cases overall . Hence the tests added there are

```
assert limit(frac(x)**x, x, oo) == AccumBounds(0, oo)
assert limit((cos(x) + 2)**(1/x), x, -oo) == 1
assert limit((cos(x) + 1)**(1/x), x, -oo) == AccumBounds(1, oo)
assert limit((tan(x)**2)**(2/x), x, -oo) == AccumBounds(0, oo)
assert limit((sin(x)**2)**(1/x), x, -oo) == AccumBounds(1, oo)
```

All these results could be checked with a graphing tool for confirming mathematical correctness.

2) Improving leading term and nseries methods for bessel and modified bessel functions

Work on bessel functions was a part of Sidharth Mundra's project last year. Although, there are still some bugs which need to be

addressed and new implementations which should be taken care of (for the modified bessel functions).

- Some basic errors on current master (related to `bessely`)

```
>>> limit(bessely(1, x), x, 0)      # expected -oo
0
>>> gruntz(bessely(1, x), x, 0)    # expected -oo
0
>>> limit(bessely(-1, x), x, 0)     # expected oo
0
>>> gruntz(bessely(-1, x), x, 0)    # expected oo
0
```

Now we see that even the gruntz method ends up returning wrong results here which indicates bugs in series implementations too. Although it works for other cases, which tells us that the series implementation would be missing some terms here and there. It goes as follows

```
>>> limit(bessely(3, x), x, 0)
oo
>>> limit(bessely(4, x), x, 0)
oo
>>> gruntz(bessely(3, x), x, 0) # Correct
-oo
>>> gruntz(bessely(4, x), x, 0) # Correct
-oo
>> limit(bessely(-2, x), x, 0)
oo
>>> gruntz(bessely(-2, x), x, 0) # Correct
-oo
>>> limit(bessely(-3, x), x, 0)
-oo
>>> gruntz(bessely(-3, x), x, 0) # Correct
oo
```

These limits aren't too tough to fix as the signs and a specific case haven't been considered properly. The main issue here are the missing terms. As gruntz mainly deals with the first term in the

series expansion in the last step of the algorithm, it goes out of notice that the series expansion is missing out on terms.

```
>>> bessely(1, x).series(x, 0, 3)
x*(log(x)/pi - log(2)/pi - (1 - 2*EulerGamma)/(2*pi)) +
O(x**3*log(x))
>>>
>>> # first term -2/(pi*x) is missing
>>>
>>> bessely(2, x).series(x, 0, 3)
-4/(pi*x**2) + x**2*(log(x)/(4*pi) - log(2)/(4*pi) - (3/2 -
2*EulerGamma)/(8*pi)) + O(x**3*log(x))
>>>
>>> # Second term of the expansion -1/pi is missing
>>>
>>> bessely(3, x).series(x, 0, 3)
-16/(pi*x**3) - 4/(pi*x) + O(x**3*log(x))
>>>
>>> # missing term -2/(pi*x) and wrongly returned term -4/(pi*x)
>>> # The correct return instead is -x/(4*pi)
>>>
>>> # Hence the leading term methods are also incorrect here
>>> bessely(1, x).as_leading_term(x)
(x*log(x) - x*log(2))/pi
```

The block responsible for missing terms is the following and the loop implementation which calculates terms recursively should be checked correctly .

```
if nu > S.One:
    term = r**(-nu)*factorial(nu - 1)/pi
    b.append(term)
    for k in range(1, nu - 1):
        term *= t*(nu - k - 1)/k
        term = (_mexpand(term) + o).removeO()
        b.append(term)
```

- New Implementations under this header (related to modified bessel functions)

Currently Sympy lacks nseries and leading term method support for modified bessel functions. For the `besseli` function only the

leading term implementation would suffice but for `besselk` both methods would need to be implemented.

We currently have

```
>>> limit(besseli(0, x), x, 0)
Traceback (most recent call last):
.....
NotImplementedError: besseli has no _eval_as_leading_term routine
>>> limit(besseli(1, x), x, 0)
Traceback (most recent call last):
.....
NotImplementedError: besseli has no _eval_as_leading_term routine
```

A **proposed implementation** for this would be the following which ends up returning correct answers for all cases. This is mostly quite related to leading term method of `besseli`

```
def _eval_as_leading_term(self, x, logx=None, cdir=0):
    nu, z = self.args
    arg = z.as_leading_term(x)
    if x in arg.free_symbols:
        return arg**nu/(2**nu*gamma(nu + 1))
    else:
        return self.func(nu, z.subs(x, 0))
```

Talking about the `besselk` function now. There was an issue reported for this recently on the github issue page ([#22911](#)). The issue goes as follows.

```

>>> limit(x*besselk(1,x) , x, 0)    # expected 1
Traceback (most recent call last):
.....
NotImplementedError: besselk has no _eval_as_leading_term routine
>>>
>>> limit(x**3*besselk(1,x) , x, 0)    # expected 0
Traceback (most recent call last):
.....
NotImplementedError: besselk has no _eval_as_leading_term routine
>>>
>>> besselk(0, x).series(x)
Traceback (most recent call last):
.....
sympy.core.function.PoleError: Cannot expand besselk(0, _x) around 0

```

The general series which should be followed here is the one given by [Wolfram](#) . It goes as follows

$$K_n(z) = (-1)^{n-1} \log\left(\frac{z}{2}\right) I_n(z) + \frac{1}{2} \sum_{k=0}^{n-1} \frac{(-1)^k (n-k-1)!}{k!} \left(\frac{z}{2}\right)^{2k-n} + \frac{(-1)^n}{2} \sum_{k=0}^{\infty} \frac{\psi(k+1) + \psi(k+n+1)}{k! (k+n)!} \left(\frac{z}{2}\right)^{2k+n} ; n \in \mathbb{N}$$

As we can see couple summations in the series above , the leading term and series implementation here would end up taking a similar structure to that of `bessely` where we split the series in three individual parts, compute them separately and add them up eventually. The **proposed implementation** is as follows.

```

def _eval_as_leading_term(self, x, logx=None, cdir=0):
    nu, z = self.args
    term_one = ((-1)**(nu-1)*log(z/2)*besseli(nu, z))
    term_two = (z/2)**(-nu)*factorial(nu-1)/2 if (nu-1).is_positive
    else S.Zero
    term_three = (-1)**nu*(z/2)**nu/(2*factorial(nu))*(digamma(nu+1) -
    S.EulerGamma)
    arg = Add(*[term_one, term_two, term_three]).as_leading_term(x)
    if x in arg.free_symbols:
        return arg
    else:
        return self.func(nu, z.subs(x, 0).cancel())

```

Similarly for `_eval_nseries` we have the following

```

def _eval_nseries(self, x, n, logx, cdir=0):
    from sympy.series.order import Order
    nu, z = self.args

    # In case of powers less than 1, number of terms need to be computed
    # separately to avoid repeated callings of _eval_nseries with wrong n
    try:
        _, exp = z.leadterm(x)
    except (ValueError, NotImplementedError):
        return self

    if exp.is_positive and nu.is_integer:
        # FIRST PART OF THE SERIES
        newn = ceiling(n/exp)
        bi = besseli(nu, z)
        a = ((-1)**(nu - 1)*log(z/2)*bi)._eval_nseries(x, n, logx, cdir)

        b, c = [], []
        o = Order(x**n, x)
        r = (z/2)._eval_nseries(x, n, logx, cdir).removeO()
        if r is S.Zero:
            return o
        t = (_mexpand(r**2) + o).removeO()

```

```

    # SECOND PART OF THE SERIES
    if nu > S.One:
        term = r**(-nu)*factorial(nu - 1)/2
        b.append(term)
        for k in range(1, nu - 1):
            term *= (-1)**k*t*(nu - k - 1)/k
            term = (_mexpand(term) + o).removeO()
            b.append(term)

    # THIRD PART OF THE SERIES
    # CODE CAN BE FORMULATED SIMILARLY AS DONE IN THE SECOND PART
    # .....
    # .....
    for k in range(1, (newn + 1)//2):
        # CODE GOES HERE
    return a + Add(*b) + Add(*c) # Order term comes from a

return super(besseli, self)._eval_nseries(x, n, logx, cdir)

```

3) Implement `_eval_as_leading_term` methods for `lowergamma`, `uppergamma`, `expint`, `Si` functions

One of my Pr's [#23190](#), was trying to add a special case while evaluating the `lowergamma` function (cases where a will be 0). Some comments on the Pr pointed out that quite a good number of

limits for lowergamma and uppergamma aren't working as expected. Those are the following.

```
>>> limit(lowergamma(a, x), a, 0) # expected oo
lowergamma(0, x)
>>> limit(lowergamma(a, x), a, 0, '-') # expected -oo
lowergamma(0, x)
>>> limit(lowergamma(-a, x), a, 0) # expected -oo
lowergamma(0, x)
>>> limit(lowergamma(I*a, x), a, 0, '-') # expected I*oo
lowergamma(0, x)
>>> limit(lowergamma(I*a, x), a, 0) # expected -I*oo
lowergamma(0, x)
```

Though I ended up fixing these by adding an `_eval_rewrite_as_tractable` method.

```
--- a/sympy/functions/special/gamma_functions.py
+++ b/sympy/functions/special/gamma_functions.py
@@ -396,6 +396,9 @@ def _eval_is_zero(self):
     if x.is_zero:
         return True

+    def _eval_rewrite_as_tractable(self, s, x, **kwargs):
+        return self.rewrite(uppergamma)
+
```

[Kalevi's Comment](#), correctly pointed out that this isn't the natural fix for the issue. Though this would help us sometimes, it will still fail rather many times. For eg

```
>>> # After including the fix
>>> limit(lowergamma(x, 1), x, 0)
NotImplementedError: lowergamma has no _eval_as_leading_term routine
```

But then I realized that most of the limits which are being addressed for the `uppergamma` function are also being addressed through a similar `_eval_rewrite_as_tractable` method.

```
def _eval_rewrite_as_tractable(self, s, x, **kwargs):
    return exp(loggamma(s)) - lowergamma(s, x)
```

And hence I realized that this would be buggy too !

```
>>> # Hence if we keep the rewrite as tractable
>>> # method for uppergamma, we end up making these mistakes
>>> limit(uppergamma(a, x), a, 0) # expected expint(1, x)
oo
>>> limit(uppergamma(a, x), a, 0, '-') # expected expint(1, x)
Traceback (most recent call last):
  NotImplementedError: loggamma has no _eval_as_leading_term routine
>>>
>>> # These mistakes are removed once we comment out the method
>>> limit(uppergamma(a, x), a, 0)
expint(1, x)
>>> limit(uppergamma(a, x), a, 0, '-')
expint(1, x)
>>>
>>> # But now that we have commented out the method
>>> # The issues it was solving fail to give correct results
>>> limit(uppergamma(x, 1) / gamma(x), x, oo) # expected 1
0
```

This says that having the method or not having the method both would lead to different mistakes , hence this calls for a dedicated `_eval_as_leading_term` method for both these functions .

Now `lowergamma` can be expressed in terms of `uppergamma` and sequentially `uppergamma` can be expressed in terms of the `expint` function (which also lacks leading term method implementation) . Hence adding a leading term method for the `expint` function would do the job here .

```
>>> lowergamma(a, x).rewrite(uppergamma)
gamma(a) - uppergamma(a, x)
>>> uppergamma(a, x).rewrite(expint)
x**a*expint(1 - a, x)
```

The **proposed implementation** which addresses all failing limits above has been shown below . As eventually the leading term method of `uppergamma` and `lowergamma` would end up

depending on this , we still need to check for all corner/special cases if any and hence the code still could have decent improvements under guidance of the mentor. Otherwise I see this catering comfortably to all the general cases .

```
def _eval_as_leading_term(self, x, logx=None, cdir=0):
    nu = self.args[0]
    arg = self.args[1].as_leading_term(x, logx=logx, cdir=cdir)
    arg0 = arg.subs(x, 0)

    if arg0 is S.ComplexInfinity:
        arg0 = arg.limit(x, 0, dir='-' if cdir == -1 else '+')
    if arg0 in (S.Infinity, S.NegativeInfinity):
        raise PoleError()
    if nu.is_Number:
        if nu == 1:
            if arg0.is_zero:
                c, e = arg.as_coeff_exponent(x)
                return -EulerGamma -log(c) -e*log(x)
            f = self._eval_rewrite_as_Si(*self.args)
            return f.subs(x, 0)
        elif nu.is_Integer and nu > 1:
            f = self._eval_rewrite_as_Ei(*self.args)
            return f._eval_as_leading_term(x, logx, cdir).as_real_imag()[0]
    return self.func(nu, arg0)

diff --git a/sympy/functions/special/gamma_functions.py
b/sympy/functions/special/gamma_functions.py

def _eval_as_leading_term(self, x, logx=None, cdir=0):
    from sympy.functions.special.error_functions import expint
    return self.rewrite(expint).as_leading_term(x, logx = logx, cdir = cdir)
```

This fixes all the incorrectly operating limits above. A case that needs to be addressed here is when `nu=1` . This needs to be addressed through a rewrite involving the `Si` function (Sine integral). This case would end up calling for a leading term implementation for the `Si` function which is also missing currently .The above block tries to explicitly take care of the leading term based on `Si` , which shouldn't be done and something like the leading term for the following function `self._eval_rewrite_as_Si(*self.args)` should be used here. Currently this is missing

```
>>> Si(x).as_leading_term(x)
Traceback (most recent call last):
  raise NotImplementedError(
NotImplementedError Si has no _eval_as_leading_term routine
```

The **proposed Implementation** for this would be

```
def _eval_as_leading_term(self, x, logx=None, cdir=0):
    arg = self.args[0].as_leading_term(x, logx=logx, cdir=cdir)
    arg0 = arg.subs(x, 0)

    if arg0 is S.NaN:
        arg0 = arg.limit(x, 0, dir='-' if re(cdir).is_negative else '+')
    if arg0.is_zero:
        return arg
    elif not arg0.is_infinite:
        return self.func(arg0)
    else:
        return self
```

4) Improvements affecting formal power series

This section builds upon work done by [@leosartaj](#) during Gsoc 2015. The algorithm implemented for calculating formal power series (based on paper [Formal Power Series by Dominik Gruntz](#)) works perfectly fine and I don't really see any error here except that some of the `FormalPowerSeries` examples in the paper don't work correctly in sympy. These examples are as follows.

```
> FormalPowerSeries(exp(x)*Ei(-x) + exp(-x)*Ei(x), x=infinity);
```

$$2 \sum_{k=0}^{\infty} \frac{(1+2k)!}{(1/x)^{(2k+2)}}$$

```
> FormalPowerSeries(exp(x)*(1-erf(sqrt(x))), x=infinity);
```

$$\sum_{k=0}^{\infty} \frac{(-1)^k (2k)! 4^{(k)} (1/x)^{(1/2+k)}}{k!}$$

$\frac{1}{\sqrt{\pi}}$

The actual errors behind these are shown below and the algorithm should be able to comfortably address these once these limits are fixed.

```
>>> # First error
>>> limit(exp(1/x)*Ei(-1/x), x, 0) # expected 0
Traceback (most recent call last):
.....
TypeError: super(type, obj): obj must be an instance or subtype of
type
>>>
>>>
>>> # Second error
>>> limit(exp(1/x)*(1 - erf(sqrt(1/x))), x, 0) # expected 0
>>> # Infinitely hangs on master
```

The first one doesn't work as expected because , asymptotic expansions for `Ei` and the helper function `_eis` , both operate quite similarly at `oo` and `-oo` . So a **proposed solution** for this would be

```
Change for Ei function

@@ -1231,7 +1231,7 @@ def _eval_aseries(self, n, args0, x, logx):
    from sympy.series.order import Order
    point = args0[0]

-    if point is S.Infinity:
+    if abs(point) is S.Infinity:

Change for _eis function
@@ -2696,7 +2696,7 @@ class _eis(Function):

    def _eval_aseries(self, n, args0, x, logx):
        from sympy.series.order import Order
-        if args0[0] != S.Infinity:
+        if abs(args0[0]) != S.Infinity:
            return super(_erfs, self)._eval_aseries(n, args0, x,
logx)
```

The second fail is because the following hangs infinitely on master.

```
>>> (1 - erf(sqrt(1/x))).as_leading_term(x) # expected 0
# hangs infinitely
```

The error taking place here through `Add._eval_as_leading_term()` method is the following

```
>>> erf(sqrt(1/x))._eval_nseries(x, 6, None)
erf(sqrt(1/x))
>>> # there isn't really a series expansion here so should return self
>>> # expected answer below would be -erf(sqrt(1/x))
>>> -erf(sqrt(1/x))._eval_nseries(x, 6, None)
-1 + O(x**6)
```

Hence this will be looked into and addressed in this section.

5) Improving leading term methods for trig, inverse and hyperbolic functions

- Some basic errors on master

```
>>> limit(atan(x), x, I*oo) # expected pi/2
-pi/2
>>> limit(asinh(x), x, I*oo) # expected oo
-oo
```

`I*oo`, is a branch point for quite some functions that have branch cuts along the complex direction. Quite some times it is sufficient enough to consider evaluations based on `I*oo` as we do for `oo`. If we use the `abs` function at a couple places to see how this is justified, we see that we would get correct results.

```

--- a/sympy/series/limits.py
+++ b/sympy/series/limits.py
@@ -77,7 +77,7 @@ def heuristics(e, z, z0, dir):

    rv = None
    if abs(z0) is S.Infinity:
-        rv = limit(e.subs(z, 1/z), z, S.Zero, "+" if z0 is S.Infinity
else "-")
+        rv = limit(e.subs(z, 1/z), z, S.Zero, "+" if abs(z0) is
S.Infinity else "-")
        if isinstance(rv, Limit):
            return
        elif e.is_Mul or e.is_Add or e.is_Pow or e.is_Function:
@@ -148,9 +148,9 @@ def __new__(cls, e, z, z0, dir="+"):
    z = sympify(z)
    z0 = sympify(z0)

-    if z0 is S.Infinity:
+    if abs(z0) is S.Infinity:
        dir = "-"
-    elif z0 is S.NegativeInfinity:
+    elif abs(z0) is S.NegativeInfinity:
        dir = "+"

```

This diff is just showing how this could be approached , we need to be considering infinities in the complex plane separately though.

- Improving limits, series for functions `acoth` and `acosh` -

One of my Pr's [#22995](#), was converted into draft recently. The Pr was a try to address limit failures for basic inverse trigonometric/hyperbolic functions (**at points which are other than branch cuts/ branch points**) . Though I realized that a lot of cases were going wrong and this would need me to carefully examine the limits to be fixed and the tests to be added so that we would not end up missing any important case . Hence I had converted it into a draft.... to take it up later . Some basic issues the Pr tries to fix

[illegible]

As [Kalevi's Comment](#) on this Pr correctly pointed out there should be more significant tests, addressing more significant cases and currently I'm not fully sure if all cases are being handled correctly but this would be addressed and all relevant errors will be fixed here. This would also be addressed while refactoring the leading term methods of `inverse trigonometric/hyperbolic` functions as mentioned above. The `acoth` case is quite easily fixable as done in the first commit of the pr but the `acosh` case might need a log rewrite for the same.

- Bi-directional limits for functions acot and acoth -

```
>>> limit(acot(x), x, 0, '+') # correct
pi/2
>>> limit(acot(x), x, 0, '-') # correct
-pi/2
>>> limit(acot(x), x, 0, '+-') # expected ValueError
pi/2
>>>
>>> limit(acoth(x), x, 0, '+-') # expected ValueError
I*pi/2
```

Another set of errors were reported in the Pr shown above . This is based on the structure of `limits.py` for handling bi-directional limits. I had explained what's going wrong in a [comment](#) on the pr. We need to deal with this as some functions (those where we can see significant differences at origin for e.g. floor) do .They use a block as they are bound to make the same error here(returning the answer from right for bi-directional limit) based on `cdir`.

```
def _eval_as_leading_term(self, x, logx=None, cdir=0):
    arg = self.args[0]
    arg0 = arg.subs(x, 0)
    r = self.subs(x, 0)
    if arg0.is_finite:
        if arg0 == r:
            if cdir == 0:
                ndirl = arg.dir(x, cdir=-1)
                ndir = arg.dir(x, cdir=1)
                if ndir != ndirl:
                    raise ValueError("Two sided limit of %s around 0"
                                     "does not exist" % self)
            else:
                ndir = arg.dir(x, cdir=cdir)
                return r - 1 if ndir.is_negative else r
        else:
            return r
    return arg.as_leading_term(x, logx=logx, cdir=cdir)
```

This clearly shows that it considered the `cdir = 0` to be either the default case from right **or** a false alarm of the default case and actually a bi-directional case in particular.

Hence we have the following

```
>>> floor(x).as_leading_term(x)
Traceback (most recent call last):
.....
    raise ValueError("Two sided limit of %s around 0"
ValueError: Two sided limit of floor(x) around 0does not exist
>>> floor(x).as_leading_term(x, cdir = 1)
0
>>> floor(x).as_leading_term(x, cdir = -1)
-1
```

Which means that the original author had kept this in mind that by default we would end up calling the `cdir = 1` case always and the `cdir = 0`, would be called when we have bi-directional cases .

Now I have two options in mind to deal with such things

1. Either change the `cdir = 0` default case to a `cdir = 1` default case , so that it doesn't end up disturbing the bi-directional limit.
2. Or use the following diff , something which is done by floor and include this in all functions where we would have a disparity of leading terms about origin . Out of all inverse trigonometric functions only this one would be involving such a case

```
@@ -2895,6 +2895,9 @@ def _eval_as_leading_term(self, x, logx=None, cdir=0):
     if x0.is_zero:
         if re(cdir) < 0:
             return self.func(x0) - S.Pi
+         if re(cdir) == 0:
+             if self._eval_as_leading_term(x, logx=None, cdir=1) !=
self._eval_as_leading_term(x, logx=None, cdir=-1):
+                 raise ValueError("Leading term from either sides is not
equal")
             return self.func(x0)
         if re(cdir) > 0 and re(x0).is_zero and im(x0) > S.Zero and im(x0) <
S.One:
             return self.func(x0) + S.Pi
```

After including such a diff across all similar functions we would end up with the following, which are all the answers we would expect


```

>>> from sympy import *
>>> x = Symbol('x')

>>> limit(acot(x), x, 0, '+-')
Traceback (most recent call last):
ValueError: The limit does not exist since left hand limit = -pi/2
and right hand limit = pi/2
>>> limit(acot(x), x, 0, '+')
pi/2
>>> limit(acot(x), x, 0, '-')
-pi/2
>>> acot(x).as_leading_term(x) # default case considering cdir = 1 or
direction from right
pi/2
>>> acot(x).as_leading_term(x, cdir = 0)
Traceback (most recent call last):
ValueError: Leading term from either sides is not equal

>>> acot(x).as_leading_term(x, cdir = 1) # default case
pi/2
>>> acot(x).as_leading_term(x, cdir = -1)
-pi/2

```

6) Fixing leading term methods for Piecewise Functions

This would involve finishing my work on [#22555](#) which fixes basic bugs encountered while evaluating limits of Piecewise Functions.

This pr has been implemented to some extent but it needs reviews and work to better handle bi-directional limit cases. Currently a block for `_eval_is_meromorphic` and `_eval_as_leading_term` have been implemented here and we are able to address most of the basic limits as shown below . The code seems to work perfectly until an error could be spotted.

```

>>> expr = Piecewise((x - 3, x <= 0), ((x + 1)**2, x < 10), (5, True))
>>> limit(expr, x, 0, dir = '+')
-3
>>> limit(expr, x, 10, dir = '-')
5
>>> limit(expr, x, oo)
Traceback (most recent call last):
  TypeError: Invalid comparison of non-real zoo
>>> limit(expr, x, -oo)
Traceback (most recent call last):
  TypeError: Invalid comparison of non-real zoo
>>>
>>> # On Committed Branch
>>> limit(expr, x, 0, dir = '+')
1
>>> limit(expr, x, 10, dir = '-')
121
>>> limit(expr, x, oo)
5
>>> limit(expr, x, -oo)
-oo

```

Also issue [#18492](#), could be fixed simultaneously. Although once the above Pr goes in, we wouldn't have to compute limits for the Piecewise function separately through `gruntz`. But it would always be nice to have support for calculating the set of most rapidly varying expressions (mrv) for Piecewise functions. Currently we lack such a code block for this support but it wouldn't be too complex to add a block for the same.

7) Miscellaneous bugs and issues related to series and leading term methods

- The Inverse Error function (`erfinv`) and the Inverse Complementary Error function (`erfcinv`) lacks Taylor term method implementation. Therefore currently on master we have

```
>>> erfinv(x).series(x, 1)
Traceback (most recent call last):
  sympy.core.function.PoleError: Cannot expand erfinv(_x + 1) around 0
>>>
>>> erfcinv(x).series(x)
nan
```

Both series expansions are quite similar

$$\operatorname{erf}^{-1}(x) = \sqrt{\pi} \left(\frac{1}{2} x + \frac{1}{24} \pi x^3 + \frac{7}{960} \pi^2 x^5 + \frac{127}{80640} \pi^3 x^7 + \dots \right)$$

Where the n th coefficient is given by

$$a_n = \frac{c_n}{2n+1},$$

where c_n is given by the recurrence equation

$$c_n = \sum_{k=0}^{n-1} \frac{c_k c_{n-1-k}}{(k+1)(2k+1)}$$

with initial condition $c_0 = 1$.

The **proposed implementation** for this would be as follows

```
@staticmethod
@cacheit
def taylor_term(n, x, *previous_terms):
    t = sqrt(S.Pi)/2*x
    if n < 0 or n % 2 == 0:
        return S.Zero
    else:
        x = sympify(x)
        # Code goes here
        # .....
        # .....
        return _c(n)/(2*n + 1) * t**n

def _c(n):
    # Helper function for calculating series expansion
    # of the inverse error function.
    if n is S.Zero:
        # base case
        return 1
    # recursive case
    return Sum(_c(k)*_c(n-k-1)/((k+1) * (2*k + 1)), (k, 0, n-1))
```

Obviously the solution can be optimized through memoization and other techniques and this is a very rough code to show what can be done here. Once this has been achieved, I could say that series implementations for almost all error functions have been taken care of.

- Implement leading term method for **loggamma** function :

After finishing implementation of **lowergamma** and **uppergamma** function as shown above , the only important function which would lack a leading term method implementation is the **loggamma** function . A **basic structure of the implementation** would be

```
def _eval_as_leading_term(self, x, logx=None, cdir=0):
    arg = self.args[0]
    z = arg.as_leading_term(x, logx=logx, cdir=cdir)
    z0 = z.subs(x, 0)
    # handling poles for gamma function
    if z.is_integer and z <= S.Zero:
        return log(gamma(arg))._eval_as_leading_term(x, logx=logx, cdir=cdir)
    # handling cases where z0 is infinite
    if z0.is_infinite:
        if logx is not None:
            res = -z*(-log(z) + 1)
            # Code to replace logarithms
            # .....
            return res
        return z*(-log(z))
    return log(gamma(z))._eval_as_leading_term(x, logx=logx, cdir=cdir)
```

- Improve leading term and series implementation for **Abs**:
Consider the following issue

```

>>> expr = x/abs(sqrt(x**2 - 1))
>>> pprint(expr)
      x
-----
|  \ 2  |
|  /  x  - 1  |
|  \ 2  |
|  /  x  - 1  |
>>>
>>> limit(expr, x, 1, '+')      # correct
oo
>>> limit(expr, x, -1, '-')      # correct
-oo
>>> limit(expr, x, -1, '+')      # expected -oo
oo
>>> limit(expr, x, 1, '-')      # expected oo
-oo
>>>
>>> x = Symbol('x')
>>> e1 = 1/abs(sqrt(1 - (x - 1)**2))
>>> limit(e1, x, -oo)
0
>>> limit(e1, x, oo)      # expected 0
Traceback (most recent call last):
.....
File "C:\Users\anuto\sympy\sympy\sympy\core\power.py", line
1742, in _eval_nseries
    raise NotImplementedError()
NotImplementedError

```

Not only for limits , also for series , we can see that series based methods are not perfect . Check the following

```

>>> x = Symbol('x', negative=True)
>>> 1/Abs(sqrt(1 - (-1 + 1/x)**2))
1/Abs(sqrt(1 - (-1 + 1/x)**2))
>>> _.series(x)
-x - x**2 - 3*x**3/2 - 5*x**4/2 - 35*x**5/8 + O(x**6)
>>>
>>> x = Symbol('x', positive=True)
>>> _.series(x)
File "C:\Users\anuto\sympy\sympy\sympy\core\power.py", line
1742, in _eval_nseries
    raise NotImplementedError()
NotImplementedError

```

There are two branches of the above series but for us only the negative part works correctly.

Series expansion at x=0

$$\begin{cases} -\frac{63x^6}{8} - \frac{35x^5}{8} - \frac{5x^4}{2} - \frac{3x^3}{2} - x^2 - x & x \leq 0 \\ \frac{63x^6}{8} + \frac{35x^5}{8} + \frac{5x^4}{2} + \frac{3x^3}{2} + x^2 + x & \text{(otherwise)} \end{cases}$$

(assuming x is real valued)

The Abs class currently lacks a leading term method which could be implemented to address the limits but overall just by having correct nseries, we could get both the series and also the limits through gruntz

The **proposed solution** here would be the following. This fixes all three cases mentioned above.

```
--- a/sympy/functions/elementary/complexes.py
+++ b/sympy/functions/elementary/complexes.py
@@ -664,7 +664,12 @@ def _eval_nseries(self, x, n, logx, cdir=0):
     if direction.has(log(x)):
         direction = direction.subs(log(x), logx)
         s = self.args[0]._eval_nseries(x, n=n, logx=logx)
-        return (sign(direction)*s).expand()
+        eq = Eq(direction, 0)
+        lim = direction.limit(x, 0, cdir=cdir)
+        if not lim is S.Zero:
+            return (s/sign(lim)).expand()
+        else:
+            return Piecewise((lim, eq), (s/sign(direction),
+True))
```

Although this would suffice in all cases, if there is some need for the leading term method, I would be adding that too.

8) Implementing aseries for gamma and factorial function

Asymptotic series for the gamma functions has been talked about since some time now in sympy. It would be great to have aseries

for the gamma function as it is also mentioned many times in the Gruntz Thesis.

```
>>> gamma(x).series(x, oo)
Traceback (most recent call last):
.....
sympy.core.function.PoleError:
Asymptotic expansion of gamma around [oo] is not
implemented.
>>> factorial(x).series(x, oo)
Traceback (most recent call last):
.....
sympy.core.function.PoleError:
Asymptotic expansion of factorial around [oo] is not
implemented.
```

The asymptotic series for the gamma function is given by the following

$$\Gamma(z) \sim e^{-z} z^{z-1/2} \sqrt{2\pi} \left(1 + \frac{1}{12z} + \frac{1}{288z^2} - \frac{139}{51840z^3} - \frac{571}{2488320z^4} + \dots \right)$$

There are two methods we could take to address this based on what is currently implemented in sympy.

1.

The coefficient a_n of z^{-n} can given explicitly by

$$a_n = \sum_{k=1}^{2n} (-1)^k \frac{d_3(2n+2k, k)}{2^{n+k} (n+k)!},$$

Where $d_3(n, k)$ stands for the number of permutations of n with k [permutation cycles](#) all of which are greater than equal to 3 and as our combinatorics module can something like permutation cycles implemented ([sympy.combinatorics.permutations.Cycle](#)), we can create this series

2. We could also explicitly implement the sterling series which is nothing but the expansion of `log(gamma(x))`, and after having a sterling series we could just exponentiate this for a series of the gamma/factorial function.

$$\ln \Gamma(z) = \frac{1}{2} \ln(2\pi) + \left(z - \frac{1}{2}\right) \ln z - z + \sum_{n=1}^{\infty} \frac{B_{2n}}{2n(2n-1)z^{2n-1}}$$

$$= \frac{1}{2} \ln(2\pi) + \left(z - \frac{1}{2}\right) \ln z - z + \frac{1}{12z} - \frac{1}{360z^3} + \frac{1}{1260z^5} - \dots$$

This is a simple analytic expression, where `Bn` stands for the Bernoulli number which is also implemented in sympy currently `combinatorial.number.bernoulli`. I would like to go with this method unless I hear a better approach from my mentors.

The **proposed implementation** for this would be

```
def _sterling_series(self, n, args0, x, logx):
    # Helper function for aseries of gamma function
    # Recreating Sterling Series or asymptotic expansion for ln(Gamma)
    from sympy.series.order import Order
    if args0[1] != oo or not \
        (self.args[0].is_Integer and self.args[0].is_nonnegative):
        return super()._eval_aseries(n, args0, x, logx)
    z = self.args[1]
    r = (z - 1/2)*log(z) - z + log(2*S.Pi)/2
    o = None
    if n < 2:
        o = Order(1/z, x)
    else:
        m = ceiling((n + 1)//2)
        l = [bernoulli(2*k) / (2*k*(2*k - 1)*z**(2*k - 1)) for k in range(1, m)]
        r += Add(*l)
        o = Order(1/z**n, x)
    return r._eval_nseries(x, n, logx) + o
```

This not only gives us the sterling series but after exponentiating the obtained series, we would also obtain the correct asymptotic series. I am not fully sure regarding the use of the already implemented `loggamma` series because if

we are keen on using that we could also get this through the following implementation

```
def _eval_aseries(self, n, args0, x, logx):
    from sympy import Order
    z = self.args[0].as_leading_term(x)
    r = loggamma(z)._eval_aseries(n + 1, args0, x, logx=None).removeO()
    return exp(r)
```

9) Implementing aseries for modified bessel functions

The TODO from `bessel.py` says they should probably be implemented but since the bessel type functions are oscillatory they are not actually tractable at infinity. I understand that many bessel based functions like Hankel, Kelvin, Spherical Bessel functions would fall under this category though the modified bessel ones wouldn't.

`Besseli` grows exponentially whereas `Besselk` exponentially decays. The issue which requires this implementation is [#19154](#).

Aseries for modified bessel functions is given by the following

1) Besseli

$$I_{\nu}(z) \sim \frac{e^z}{(2\pi z)^{\frac{1}{2}}} \sum_{k=0}^{\infty} (-1)^k \frac{a_k(\nu)}{z^k},$$

2) Besselk

$$\begin{aligned} K_{\nu}(z) &\approx z e^{-z} \sum_{n=0}^{\infty} \frac{a_{2n+1}}{\nu} \int_0^{\infty} u e^{-zu^2/2} \left(\frac{u}{2}\right)^{2n+1} du \\ &= e^{-z} \sum_{n=0}^{\infty} \frac{a_{2n+1}}{\nu} \sqrt{\frac{\pi}{2z}} \frac{1}{2^{2n+1}} \frac{(2n+1)!!}{z^n}, \end{aligned}$$

Where the coefficient `an` is given by

$$a_n(\nu) = (-1)^n \frac{\cos(\pi\nu)}{\pi} \frac{\Gamma\left(n + \frac{1}{2} + \nu\right) \Gamma\left(n + \frac{1}{2} - \nu\right)}{2^n \Gamma(n+1)}$$

$$= \frac{(4\nu^2 - 1^2)(4\nu^2 - 3^2) \cdots (4\nu^2 - (2n-1)^2)}{8^n \Gamma(n+1)},$$

The **proposed implementation** for `besseli` would be as follows and asymptotic series for `besselk` would also be addressed similarly

```
def _eval_aseries(self, n, args0, x, logx):
    from sympy.series.order import Order
    point = args0[1]
    nu, z = self.args

    # Expansion at oo
    if point is S.Infinity:
        coeff = exp(z)/sqrt(2*S.Pi*z)
        # Code for a(nu, z) in terms of gamma and cos
        # will be framed here
        s = [S.NegativeOne**k * a(nu, z) / z**k for k in range(0, n)] +
    [Order(1/z**n, x)]
    return coeff * Add(*s)

    # All other points may not be handled as of now
    return super(self)._eval_aseries(n, args0, x, logx)
```

10) Implementing aseries for Chi and Shi functions

Consider the following

```
>>> n = Symbol('n')
>>> expint(n, x).series(x, oo).subs(n, 1)
(-120/x**5 + 24/x**4 - 6/x**3 + 2/x**2 - 1/x + 1 + O(x**(-6)), (x,
oo))*exp(-x)/x
>>>
>>> # but using any integer initially doesn't work
>>> expint(1, x).series(x, oo)
Traceback (most recent call last):
.....
Asymptotic expansion of Chi around [oo] is not implemented.
>>>
```

The series given by [Wolfram](#) is as follows

$$\left(\left(\frac{1}{x}\right)^2 + \frac{6}{x^4} + \frac{120}{x^6} + O\left(\left(\frac{1}{x}\right)^7\right)\right) \cosh\left(x + O\left(\left(\frac{1}{x}\right)^7\right)\right) + \left(\frac{1}{x} + \frac{2}{x^3} + \frac{24}{x^5} + O\left(\left(\frac{1}{x}\right)^7\right)\right) \sinh\left(x + O\left(\left(\frac{1}{x}\right)^7\right)\right) + \left(-\frac{i\pi}{2} + O\left(\left(\frac{1}{x}\right)^7\right)\right)$$

The **proposed implementation** for the aseries method of Chi would be the following

```
def _eval_aseries(self, n, args0, x, logx):
    from sympy.series.order import Order
    point = args0[0]

    # Expansion at oo
    if point is S.Infinity:
        z = self.args[0]
        p = [factorial(2*k) / z**(2*k) for k in range(0, int((n - 1)/2))]
        + [Order(1/z**n, x)]
        q = [factorial(2*k + 1) / z**(2*k + 1) for k in range(0, int(n/2) - 1)]
        + [Order(1/z**n, x)]
        res = (sinh(z)/z + Order(1/z**n, x))*Add(*p) +
              (cosh(z)/z + Order(1/z**n, x))*Add(*q) +
              (-S.ImaginaryUnit*S.Pi/2 + Order(1/z**(n + 1), x))
        return res

    # All other points may not be handled as of now
    return super(Chi, self)._eval_aseries(n, args0, x, logx)
```

We get a result which is identical to what we expect and implementation for Shi function would be done similarly.

```
>>> Chi(x).series(x, oo)
-I*pi/2 + (sinh(x)/x + O(x**(-7), (x, oo)))*(120/x**6 + 24/x**4 + 2/x**2 + 1
+ O(x**(-7), (x, oo))) + (cosh(x)/x + O(x**(-7), (x, oo)))*(24/x**5 + 6/x**3
+ 1/x + O(x**(-7), (x, oo))) + O(x**(-6), (x, oo))
```

11) Finish implementing leading term and series methods from [@Osidharth](#) proposal

Sidharth Mundra worked on a good amount of leading term and series methods for special functions, though series for quite some special functions (specifically asymptotic) couldn't be covered during his coding period. Those special functions are

- a. Zeta Functions
- b. Lerchphi Functions
- c. Polygamma Functions
- d. Beta Functions
- e. Elliptic Functions

I would like to work on this because we are fortunate enough to see what goes wrong or isn't implemented correctly or lacks implementation through his proposal (specially the issues mentioned from page 15 - 20)

A general **proposed implementation** for addressing nseries and aseries for these functions would be.

```
def _eval_nseries(self, x, n, logx, cdir=0):
    from sympy.series.order import Order

    """
    Initialize z (argument of special_func) and other variables
    using self.args
    """
    # In case of powers less than 1, number of terms need to be computed
    # separately to avoid repeated callings of _eval_nseries with wrong n
    try:
        _, exp = z.leadterm(x)
    except (ValueError, NotImplementedError):
        return self
    # Other conditions specific to use case of the series
    # expansion would also be checked here
    if exp.is_positive:
        newn = ceiling(n/exp)
        o = Order(x**n, x)
        r = z._eval_nseries(x, n, logx, cdir).removeO()
        if r is S.Zero:
            return o

    """
    Iteratively calculate and return the series expansion
    Calculate each term individually if it isn't possible iteratively
    """

    return super(special_func, self)._eval_nseries(x, n, logx, cdir)
```

`_eval_aseries` methods can also be dealt with similarly . This template was put to good use by [@Osidharth](#) , hence I would say this template is good enough to tackle the remaining cases too.

The errors and `NotImplementedError`'s seen in his proposal still prevail for these functions and need to be addressed or rather implemented.

- Fixing symbolic limit errors through gruntz and other gruntz errors

Cherry Picking issues which are seemingly related to gruntz - These are a group of issues which have been collected from

1. The github issue page and the mailing list
2. The gruntz thesis
3. Already implemented tests (for eg a test where limits works at a point and doesn't work at a related/contemporary point

These are the following

1. [#10382](#) - From github issues page

The issue is claimed to be fixed on master but while conducting a basic survey and cherry picking issues , it seems that this still needs to be fixed

```
>>> n = Symbol('n', positive=True)
>>> limit(fibonacci(n + 1)/fibonacci(n), n, oo)
1      # Expected GoldenRatio
```

A **proposed solution** here might be through an `_eval_rewrite_as_tractable` method which should be implemented. Changing the `_eval_rewrite_as_sqrt` method would also do the job here .

```
def _eval_rewrite_as_sqrt(self, n, sym=None):
    from sympy.functions import sqrt
    if sym is None:
        return (S.GoldenRatio**n - cos(S.Pi*n)/S.GoldenRatio**n)/sqrt(5)
    _eval_rewrite_as_tractable = _eval_rewrite_as_sqrt
```

2. From Gruntz thesis -

```
>>> limit(exp((log(2) + 1)*x)*(zeta(x + exp(-x)) - zeta(x)), x, oo)
Traceback (most recent call last):
NotImplementedError: Don't know how to calculate the mrv of
'Subs(Derivative(zeta(_xi_1), _xi_1), _xi_1, _p)'
```

Zeta function currently misses all implementation for series which includes `_eval_nseries` / `_eval_aseries` / `_eval_as_leading_term` and as stated above (in the second sub-topic), all these methods for the zeta function would be implemented by then after which this issue could be addressed correctly.

3. [#6682](#) - From github issues page

This seems to be fixed on master but returns different answer through the limit call and through the gruntz call

```
>>> limit( exp(2*Ei(-z))/z**2, z, 0)
oo
>>> gruntz( exp(2*Ei(-z))/z**2, z, 0)
exp(2*EulerGamma)
```

This to me indicates an error through the `leading_term` method of the function `Ei`, and obviously we shouldn't be expecting our users on the other side to type out `gruntz` as most of the users might not even know that sympy uses the gruntz's algo for solving a good chunk of limits, hence this should be addressed.

A **possible fix** for this could be the following

```

# Changes in leading term of Ei
def _eval_as_leading_term(self, x, logx=None, cdir=0):
+   from sympy import re
+   x0 = self.args[0].limit(x, 0)
+   arg = self.args[0].as_leading_term(x, cdir=cdir)
+   cdir = arg.dir(x, cdir)
+   if x0.is_zero:
+       f = self._eval_rewrite_as_Si(*self.args)
+       return f._eval_as_leading_term(x, logx=logx, cdir=cdir)
+   if logx is not None:
+       return logx + S.EulerGamma - (
+       S.ImaginaryUnit*pi if re(cdir).is_negative else S.Zero)
+   return log(-arg) + S.EulerGamma if re(cdir).is_negative else
log(arg)
return super()._eval_as_leading_term(x, logx=logx, cdir=cdir)

# Changes in leading term of Chi
@@ -2253,7 +2273,10 @@ def _eval_as_leading_term(self, x, logx=None, cdir=0):
+   if arg0 is S.NaN:
+       arg0 = arg.limit(x, 0, dir='-') if re(cdir).is_negative else '+'
+   if arg0.is_zero:
+       return S.EulerGamma
+   if logx is not None:
+       return S.EulerGamma + logx
+   c, e = arg.as_coeff_exponent()
+   return log(c) + e*log(x)

```

This passes all tests and comfortably addresses the relevant issues mentioned above.

4. [#10804](#) - From github issue page

```

>>> limit(2*airyai(x)*root(x, 4) * exp(2*x**Rational(3,
2)/3), x, oo)
Limit(2*x**(1/4)*exp(2*x**(3/2)/3)*airyai(x), x, oo,
dir='-')
>>> # Expected answer for both cases as shown
>>> # by wolfram is 1/sqrt(pi)
>>> limit(airybi(x)*root(x, 4) * exp(-2*x**Rational(3,
2)/3), x, oo)
0

```

Although `airyai` and `airybi` functions lack leading term methods, that might not be the correct method to address this issue. As the above mentioned functions won't be tractable at infinity, it seems that an

`_eval_rewrite_as_tractable` method for these special functions should be implemented. This would also require us to write a helper class which will be semi tractable at infinity , so that we could use the rewrite as tractable method.

The **proposed fix** here

```

-- a/sympy/functions/special/bessel.py
++ b/sympy/functions/special/bessel.py
@@ -1296,6 +1296,9 @@
+ def _eval_rewrite_as_tractable(self, z):
+     return exp(-Rational(2, 3)*z**Rational(3, 2))*sqrt(pi*sqrt(z))/2*_airyais(z)
+
+
+class _airyais(Function):
+
+    def _eval_rewrite_as_intractable(self, x):
+        return 2*_airyai(x)*exp(Rational(2, 3)*x**Rational(3,2))/sqrt(8.Pi*sqrt(x))
+
+    def _eval_aseries(self, n, args0, x, logx):
+        # code goes here
+        # .....
+        # .....
+        return super()._eval_aseries(n, args0, x, logx)

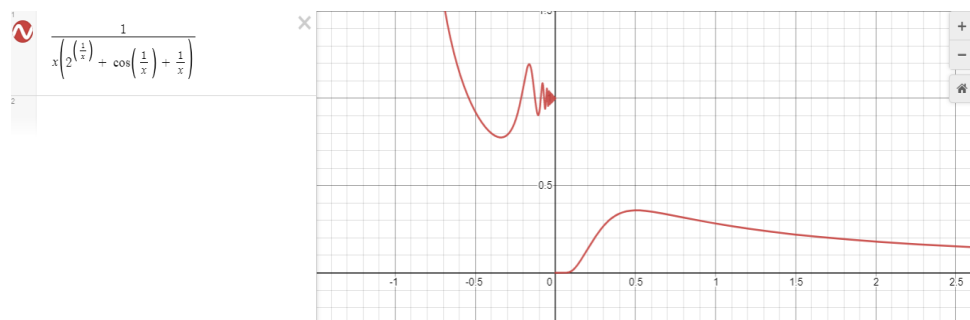
```

5. From Gruntz Thesis

```

>>> limit(x / (x + 2**x + cos(x)), x, oo)
0
>>> limit(x / (x + 2**x + cos(x)), x, -oo)
Limit(x/(2**x + x + cos(x)), x, -oo) # expected 1
>>>
>>> gruntz(x / (x + 2**x + cos(x)), x, -oo)
sympy.core.function.PoleError: Cannot expand cos(1/_w) around 0

```



As shown in the figure above , the first answer is correct 0, but the second one doesn't work as expected . Now this seems somewhat weird because the series method and the leading term method from both sides (cdir = +1/ -1) both are

the same. It seems to me that using the `calculate_series` method and then calculating the leadterm based on that might not be required in a general case . Just using the leadterm directly should also suffice as we can see that calculate series method is quite buggy and could be deprecated. Hence we can **propose an implementation** like the following

```

@@ -544,18 +544,17 @@ def mrv_leadterm(e, x):
    #
    w = Dummy("w", positive=True)
    f, logw = rewrite(exps, Omega, x, w)
    series = calculate_series(f, w, logx=logw)
    try:
    -     lt = series.leadterm(w, logx=logw)
    - except (ValueError, PoleError):
    -     lt = f.as_coeff_exponent(w)
    -     # as_coeff_exponent won't always split in required form. It may simply
    -     # return (f, 0) when a better form may be obtained. Example (-x)**(-pi)
    +     lt = f.leadterm(w, logx=logw)
    + except (NotImplementedError, PoleError, ValueError):
    +     base = f.as_base_exp()[0].as_coeff_exponent(w)
    +     ex = f.as_base_exp()[1]
    +     lt = (base[0]**ex, base[1]*ex)
    +     # can be written as (-1**(-pi), -pi) which as_coeff_exponent does not
    return
    if lt[0].has(w):
    -     base = f.as_base_exp()[0].as_coeff_exponent(w)
    -     ex = f.as_base_exp()[1]
    -     lt = (base[0]**ex, base[1]*ex)
    +     f = f.rewrite('tractable', deep=True, limitvar=w)
    +     series = f.nseries(w, n=1, logx=logw).removeO()
    +     lt = series.leadterm(w, logx=logw)
    return (lt[0].subs(log(w), logw), lt[1])

```

The result after introducing this change is as follows

```

>>> gruntz(x / (x + 2**x + cos(x)), x, -oo)
1
>>> gruntz(x / (x + 2**x + cos(x)), x, oo)
0

```

Though there is some room for improvement as exactly a couple, not too tedious tests fail after this change but that would be taken care of .

- Refactor Order code and order.py to handle cases better

- [#22836](#) - Possible improvements for Order of expressions involving factorials .

Following basic big Oh pattern for growth of functions , we have $\text{const} < \log(x) < x^n < 2^x$ or $\exp(x) < \text{factorial}(x) < x^x$.Following results should be improved here , these works on other sister CAS which have some implementation of calculating the order term like Sage / diofant .

```
>>> #ON MASTER
>>> O(2**x + factorial(x), (x, oo))
O(exp(x*log(2)) + factorial(x), (x, oo))
>>>
>>> O(2**x + factorial(x) + x**x, (x, oo))
O(exp(x*log(x)) + factorial(x), (x, oo))
>>>
>>> O(x + factorial(x), (x, oo))
O(x*(1 + factorial(x)/x), (x, oo))
>>>
>>> # Better results could be
>>>
>>> O(2**x + factorial(x), (x, oo))
O(factorial(x), (x, oo))
>>>
>>> O(2**x + factorial(x) + x**x, (x, oo))
O((1/x)**(-x), (x, oo))
>>>
>>> O(x + factorial(x), (x, oo))
O( factorial(x) , (x, oo))
```

The Pr referred in the immediate next section ([#22247](#)) through which the code for order is being refactored is responsible for this and helps us get through most of the results here except a couple results

```

>>> O(1/x**2 + factorial(x) , (x, oo))
O(factorial(x), (x, oo))
>>>
>>> O(2**x + factorial(x) + x**x, (x, oo))
O(exp(x*log(x)), (x, oo))
>>>
>>> O(2**x + x**x, (x, oo))
O(exp(x*log(x)), (x, oo))
>>>
>>> O(x + x**x, (x, oo))
O(exp(x*log(x)), (x, oo))
>>>
>>> O(x + factorial(x), (x, oo))
# hangs infinitely
>>>
>>> O(2**x + factorial(x), (x, oo)) # Only case giving wrong result
O(exp(x*log(2)), (x, oo))

```

The second error is because of line 250 in the following block

```

if expr.is_Add:
    # Here orders is [O(exp(log(2)/_Dummy_22)),
    O(factorial(1/_Dummy_22))]
    new_expr = Order(Add(*orders), *pts)
    # new_expr returns O(exp(log(2)/_Dummy_22))
    # instead of O(factorial(1/_Dummy_22))
    # leading to wrong answer in the case above
    if new_expr.is_Add:
        new_expr = Order(Add(*[a.expr for a in new_expr.args]), *pts)
    -----
>>> O(O(exp(log(2)/x)) + O(factorial(1/x)))
O(exp(log(2)/x)) # expected O(factorial(1/x))

>>> # This happens because of a failing limit on master
>>> gruntz(exp(-log(2)/x)*factorial(1/x), x, 0) # expected oo
0
>>> # The rewritten function after calculating
>>> # mrv term isn't correct

```

And the first error is related to the following block from line 205 to 207 and `factor` method in general. If we comment out the following block, we get the result quite fast.

```
if expr.is_Add:
    expr = expr.factor()
```

Hence these two cases would be addressed along with the issue referred below.

- [#22247](#) - Refactoring Order for Multivariate expressions and improving individual if-else blocks to handle specific cases . This would end up removing the block from line 224 to 227

```
if expr.is_Add:
    lst = expr.extract_leading_order(args)
    expr = Add(*[f.expr for (e, f) in lst])
```

And we could eventually deprecate the `extract_leading_order` method by the end of this improvement ([#22252](#))

Though the main issue reported is being solved by the Pr, some cases pointed out here need careful observation.

1. Block under consideration

```
# Lines 246 - 250
if expr.is_Add:
    new_expr = Order(Add(*orders), *pts)
    if new_expr.is_Add:
        new_expr = Order(Add(*[a.expr for a in new_expr.args]), *pts)
    expr = new_expr.expr
```

As stated correctly by [Kalevi's Comment](#), The order of a sum cannot in general be derived from the orders of its

arguments . Fortunately there is just a single test in `test_order.py`, which comfortably passes this but Obviously if explicitly cases are pointed out which go through this block , this would in most cases lead to bugs.

As the above issue involving `factorials` also shows that this block of code could get buggy , the handling here needs to be changed to something like the `_eval_as_leading_term` method of class `Add`.

2. The above block might also be responsible for errors in Multivariate cases .These comments by **Kalevi**([1st](#), [2nd](#), [3rd](#)) need to be answered and discussed as the concerns raised here are quite logical but have been ignored in sympy as of now. This would involve refactoring all code and tests(adding new tests and improving previous ones) for `multivariate` cases of type `Add`.



jksuom commented on Nov 25, 2021

Member



It is not clear how `Order` should be defined in case of several variables. I would expect that examples like this

```
>>> O(x + 2*y)
O(x + y, x, y)
```



would mean that $|x + 2y| < M|x + y|$ for some constant M , at least when x and y are close to 0, but that is obviously not true: $x + 2y$ need not be 0 when $x + y$ is 0 (except when both variables are 0). This type of definition might be meaningful but probably impossible to implement in general.

I think that the only feasible definition in multivariable case is the one where leading terms are taken in a definite order. First, the leading term of $f(x, y)$ wrt. x would be something like $x^{**n}*g(y)$ and then the leading term wrt. y would be constant times $x^{**n}*y^{**m}$. I don't know how useful that would be but at least it could be implemented once the leading term methods are in place.

The theory pointed out here is that, $f(x)=O(g(x))$ as $x \rightarrow p$ iff there exists M : $|f(x)| \leq M |g(x)|$ in a neighborhood of $x=p$. Most code for multivariate O probably doesn't use this definition. For most expressions, it is doubtful that there are too many possible correct (and simple) transformations for the O constructor. There might be many ways to go about here

- Drop constants for Mul type expressions
- Take leading terms sequentially according to passed arguments
- Take leading term only when one of the terms in the expression is univariate otherwise return self
- Restrict order to univariate cases

I would feel going with the 2nd case won't be tough through the pr(just have to remove a couple lines) , hence if that's the best approach to go about here(will be discussed thoroughly with mentors) , it can be achieved .

Another issue([#6822](#)) is somewhat related to the above , hence this will be addressed too.

- [Substitution for Order](#) - Issues [#19120](#), [#10290](#), [#7872](#)

All these issues are related .The issues are related to `_eval_subs` method of `Order`.

```

>>> # On Master
>>> O(x).subs(x, O(x))
>>> (x + O(x**2)).subs(x, x + O(x**2))
>>> O(x**2).subs(x, x + O(x**2))
>>> O(x**2 + 1, (x, oo)).subs(x, x + O(x**2, (x, oo)))
>>> (x - x**3/6 + x**5/120 - x**7/5040 +
O(x**8)).subs(x, sin(x).series(n=8)).expand()
>>> (cos(x).series(n=8)).subs(x, sin(x).series(n=8)).expand()

File "c:\users\anuto\sympy\sympy\core\basic.py", line 953, in subs
    rv = rv._subs(old, new, **kwargs)
File "c:\users\anuto\sympy\sympy\core\cache.py", line 72, in wrapper
    retval = cfunc(*args, **kwargs)
File "c:\users\anuto\sympy\sympy\core\basic.py", line 1067, in _subs
    rv = fallback(self, old, new)
File "c:\users\anuto\sympy\sympy\core\basic.py", line 1039, in fallback
    arg = arg._subs(old, new, **hints)
File "c:\users\anuto\sympy\sympy\core\cache.py", line 72, in wrapper
    retval = cfunc(*args, **kwargs)
File "c:\users\anuto\sympy\sympy\core\basic.py", line 1065, in _subs
    rv = self._eval_subs(old, new)
File "c:\users\anuto\sympy\sympy\series\order.py", line 479, in
_eval_subs
    res = [dict(zip((d, ), sol))]
TypeError: 'ConditionSet' object is not iterable

```

Couple of examples from the gruntz thesis Chapter 6 regarding calculation of asymptotic series, also fail due to this.

```

>>> # On Master
>>> a, b = symbols('a b', integer=True)
>>> e = exp(1/x + exp(-x**2) * (exp(a*x) - exp(b*x))) - exp(1/x)
>>> e.aseries(x, n=3, hir=True)
Traceback (most recent call last):
  File "C:\Users\anuto\sympy\sympy\core\expr.py", line 3176, in
aseries
  File "C:\Users\anuto\sympy\sympy\core\basic.py", line 1071, in _subs
    rv = self._eval_subs(old, new)
  File "C:\Users\anuto\sympy\sympy\series\order.py", line 474, in
_eval_subs
    res = [dict(zip((d, ), sol))]
TypeError: 'ConditionSet' object is not iterable
>>>
>>> # expected result
>>> e.aseries(x, n=3, hir=True)
E**(-2*x**2)*(E**(2*a*x + 1/x)/2 + E**(2*b*x + 1/x)/2 - E**(a*x + b*x +
1/x)) + E**(-x**2)*(E**(a*x + 1/x) - E**(b*x + 1/x)) + O(E**(-3*x**2), (x,
-oo))

```

Though I already have a Pr where an **implementation has been proposed** ([22339](#)), and solves almost all basic cases it hasn't been reviewed to the fullest yet and the code there could be improved to cater to more cases (the pr can address most of the basic cases like the one shown below but needs some work and reviews to address more examples which also include couple examples from gruntz thesis)

```
>>> (cos(x).series(n=8)).subs(x,sin(x).series(n=8)).expand()
1 - x**2/2 + 5*x**4/24 - 37*x**6/720 + O(x**8)
>>>
>>> (x**2+ O(x**2) + 1/x**2).subs(x, x + O(x**2))
(x + O(x**2))**(-2) + O(x**2)
```

I found that most other CAS having **Order** implementation, including our sister library diofant is able to return correct results here. Though correct results were returned, these take time . I tried a couple examples from gruntz which wouldn't work currently on sympy and though diofant returns correct results,it takes 10 seconds each . Some intuition behind the performance aspect for such expression was given by Oscar Benjamin([Oscar's Comment](#)), hence even that should be taken care of .

- Another somewhat related issue here is [#15915](#)

I found the implementation details and theory given behind this issue by [@Osidharth](#) in his proposal(linked below) last year to be quite appropriate here and should be followed for the implementation .

- Miscellaneous -

This section addresses some already existing unmerged Pr's and some other prompted ideas which would be really great additions to have in sympy

1. Implementing limit for a symbolic point -

Finish work on [#22463](#) which implements function `_limit_for_symolic_point` for evaluating limit when `z0` is passed as Symbol. The function mentioned above is a helper function to the `Limits` class. It has been long known that sympy isn't able to handle limit computations at variable symbolic points. We also raise an error in some circumstances in the limits code for this as follows.

```
if(z0.has(z)):
    raise NotImplementedError("Limits approaching a variable point are"
        " not supported (%s -> %s)" % (z, z0))
```

The corresponding issue where this would come into picture is this one [#21029](#), and it is addressed as follows.

```

# On Master
>>> x, z = symbols('x z')
>>> pprintlimit((x**3 + 2*x + 1)/(x**2 - 1), x, z)
      3
      z  + 2·z + 1
      -----
      2
      z  - 1
>>> limit((x**3 + 2*x + 1)/(x**2 - 1), x, z).subs(z, 1)
zoo
>>> limit((x**3 + 2*x + 1)/(x**2 - 1), x, z).subs(z, oo)
Nan
>>> # On Committed Branch
>>> limit((x**3 + 2*x + 1)/(x**2 - 1), x, z)
{
  -oo          for z = -oo
|
|  oo          for z = -1 ∨ z = 1 ∨ z = oo
|
| 3
| z  + 2·z + 1
| -----
|                   otherwise
|  2
|  z  - 1
}
>>> limit((x**3 + 2*x + 1)/(x**2 - 1), x, z).subs(z, 1)
oo
>>> limit((x**3 + 2*x + 1)/(x**2 - 1), x, z).subs(z, -oo)
-oo

```

The pr is currently under reviewed and code/functionality here can be improved by a good margin. The implementation currently works only for expressions having finite discontinuities and I feel that as a first step this could be extended for expressions having **periodic infinite discontinuities** too, hence that also comes under my plan of implementation . Hence for something like an **imageset** for eg `limit(tan(x)/x, x, z)` or `limit(tan(x), x, z)` where we have a periodic repetition of discontinuous points , we could try to calculate the limit at the first point and generalize it overall !!

```
>>>singularities(tan(x)/x , x)
Union(FiniteSet(0), ImageSet(Lambda(_n, 2*_n*pi + pi/2),
Integers), ImageSet(Lambda(_n, 2*_n*pi + 3*pi/2), Integers))
```

2. Improving coverage for summations.py

A particular genre of issue has been reported more than a couple of times now on the github page under the labels of `concrete` and `series`. Some examples being [#14410](#), [#14579](#) and [#22178](#). These issues mainly point out sympy's inability to compute summations from a finite lower bound to an infinite upper bound .

For computing Sums such as `Sum(1/x - 3, (x, 1, oo)).doit()`, where the expressions involved are of type `Add`. The code in the `doit` method deals with this as follows

1. Splits expressions about the operator
2. Calculates individual sums
3. Adds the sums up and return the answer

Executing this would lead to `oo - oo` being computed which would evaluate to `nan` . Hence if we see carefully, the results returned should be based on the term which would end up being dominant at infinity. This calls for an implementation based on **Order**.

I have a pr ([#22193](#)) already implementing this but the code written there is quite inefficient and was just written just to see whether an Order based implementation would suffice. It needs some good reviews before going in. Some results based on the pr.

```
>>> Sum(3 - sqrt(x) , (x, 1, oo)).doit() == -oo
>>> Sum(1/(sqrt(x)) - 3, (x, 1, oo)).doit() == -oo
```

3. Extending Functionality of limitseq.py to limits.py -

There is some functionality which is possible through the `limit_seq` method but can't be done through the `limit` method . It's the functionality of dealing with **alternating sign limits** or rather calculating **limits of sequences with alternating sign at infinity** .For eg

```
>>> # Implementation through limits
>>> limit((-1)**n/n**2, n, oo)
.....
NotImplementedError: Result depends on the sign of I
>>>
>>> limit((2*n + (-1)**n)/(n + 1), n, oo)
.....
NotImplementedError: Result depends on the sign of I
>>>
>>> limit((-2)**(n+1)/(n + 3**n), n, oo)
.....
NotImplementedError: Result depends on the sign of sign(log(2) +
I*pi)
>>>
>>> Implementation through limit_seq
>>> limit_seq((-1)**n/n**2)
0
>>> limit_seq((-2)**(n+1)/(n + 3**n))
0
>>> limit_seq((2*n + (-1)**n)/(n + 1))
2
```

A concern for this was raised on the github page through [#19823](#). As [Kalevi's](#) comment correctly says , It could be possible to make the `limit` method handle the general case by adding a suitable heuristic argument .

Fortunately a fellow contributor had tried implementing a fix for this and had done a somewhat good job with [#20478](#) , currently the Pr is stalled and has been labeled as `please take over` . The Pr hasn't been reviewed to the fullest and the approach used there may or may not be the best way to solve this (it tries to inculcate `limit_seq` method in `limits.py` under some cases) , hence the Pr needs reviews and through testing of what it claims to solve . The comments/few reviews given on the Pr hasn't been addressed by the original author and he's no longer working on the issue . Hence I would confirm the approach

(some parts are doubtful to me as of now) and finish the incomplete work there .

4. Implementing Shackell's Algorithm

[@ArighnalITG](#)'s proposal from 2019 pointed out a good alternate algorithm from [gruntz-thesis](#) , known as the Shackell's Algorithms .

The **Gruntz's algorithm** which is kind of the main backbone behind limit and series computations currently in sympy especially when the functions involved lack leading term and series implementations . But as he correctly wrote, the gruntz algorithm can get stuck up with heavy rewrites and also it involves heavy recursive calls and it is already too technical for contributors and even solving new issues through gruntz would involve heavy debugging and fixing something might easily break another test.

This made me go through the **Shackell's Algorithm**(based on **nested forms**) in Section 6.1, and as I went through the theory there , it looks to me that this has potential and maybe its compatibility should be tested out with sympy .

Although [@ArighnalITG](#) gave us a convincing breakdown and analysis of where the algorithm could help us along with a general API for its implementation and some tests where the performance gets better I'm not sure why it wasn't tried out during his Gsoc period and I would like to try it out if there's appetite for an alternative algorithm to **gruntz** .

The general API proposed in his proposal looks quite good for a start and I would like to build upon that.

5. Finish Working on other prominent Prs

- [#22806](#) - Fixes missing and mathematically incorrect terms being returned in Multivariate series expansions (+100 - 7)
- [#22804](#) - Extends functionality of `taylor_term` method for multivariate expressions (+307 - 71)
- [#23229](#) - Implements methods `rs_sec` and `rs_csc` for handling univariate ring series expansions (+94 - 1)

NOTE - I have created a separate ["Stretch Goals"](#) doc , which I would be like to follow and complete if I end up finishing ahead of schedule . These are issues/implementations which don't have a confirmed/most appropriate approach yet . Although I've tried out quite some ways to address these, they can only be put to use after having a chat with the mentors on the same and confirming the functionality, usage of the newly written code.

Proposed Timeline and Milestones

This section provides a basic overview regarding how I plan to utilize the Pre-Gsoc, Gsoc contributing and the Post Gsoc period .

- Pre Gsoc /Application Review Period -

1. I would be having my end semester evaluations a week after the Gsoc Application period(**19th April**) ends and the examination period would be running for a week or two. I will be having my last examination on **May 5th** , hence I will be devoting my time here.
2. I would be resuming contributions to sympy after this. I am quite passionate about a Pr of mine ([#22698](#)) which is implementing a `hyperbola.py` file. I have already added a basic structure and functionality for the hyperbola class which is about 60-70 percent of the work . I would be glad to finish the remaining work as this is a long time pending issue in sympy.
3. Go through almost all issues of the series/limits label and give them a read . I would most likely encounter some issue which is close to what I plan to fix/implement and hence make note of such issues.

- May 20 - June 13 (Community Bonding Period) -

1. Using the community bonding period in its true sense , I would be interacting with the mentors , senior members and my fellow selected contributors and know about their projects and preferred working domains in sympy.
2. Coding wise I would be going through the series and limits code from start till end to understand the complete workflow and related function calls. I realize that doing this would eventually help me throughout the contribution period.
3. I also plan to start early, maybe by the last week of the community bonding period , which implies that I would be building on top of the pseudocode I've already provided in this doc and keeping some blocks of working code ready for what I plan to tackle initially.

● Gsoc Period -

I plan to divide the Gsoc contributing period into **4 phases of three weeks each** and will try to give a detailed breakdown of what I intend to execute each week.

Phase 1 (Week 0 - 3)

- I will start by addressing all buggy leading term and series methods for functions involving branch cuts/ branch points . This will involve improving leading term and series methods for the log function to use log based rewrites later . Hence a pr would be made for this first .
- Then I would start reframing and rewriting leading term and series methods for almost all inverse trig/hyperbolic functions as these are the functions to be considered here .
- These methods are very much fragile and special testing and observation needs to be done for the points in question here which are the branch points.
- In the last 1 to 0.5 weeks of phase 1, I would start with the second topic as written above which is fixing and implementing leading term, nseries and aseries methods for special and elementary functions. There are 11 things to tackle under this topic and I plan to tackle at least 2 of them.

Phase 2 (Week 3 - 6) -

- I would continue addressing issues from the second topic . It is quite possible that not all series or leading term methods are required for a function as also indicated in the proposal above as sometimes just the leading term method would suffice , hence all remaining 9 issues would be addressed accordingly as per the needed methods.
- Write extensive tests for all expansions implemented and also for the limits involving these special functions.
- Separate prs based would be made here based on clubbing related functions , which implies for example Issues related to all bessell functions would be tackled together through a pr and issues

/implementations based on the gamma functions would be tackled together through another separate pr. This would make reviewing code and having back and forth discussions with the mentors more smoother.

Phase 3 (Week 6 - 9)

- I would start phase 3 by addressing issues as pointed out under the gruntz section . Many of these need implementations or fixes in the `_eval_rewrite_as_tractable` method hence a single pr would be made first addressing all these issues.
- A separate Pr would then be made for solving the last issue shown in the gruntz section . This would also call for deprecating the `calculate_series` method from the codebase and building an approach based around the `leadterm` method. As this would lead to bugs initially a separate pr would be made for this. The new method would be documented and thoroughly tested.
- I would be shifting my focus to refactoring code in `order.py` next . The first issue shown as mentioned above involves deprecating the `extract_leading_order` method and also fixing a gruntz error. Hence a separate Pr would be made for this .
- Code here would be improved in blocks one at a time . This might involve a good amount of testing hence tests will be added and modified when required.

Phase 4 (Week 9 - 12)

- I would finish my work based on order term by fixing the buggy `_eval_subs` method which would tackle the last two issues mentioned under the order section. I already have pr implementing the first fix (though hasn't been reviewed to a good extent) and the second fix will be made through newer commits on the pr.
- All miscellaneous issues will then be addressed . The best part about this section is that there is proposed code/pseudocode/API proposed already through me and previous gsocers , hence we just need to channel and integrate these newer ideas into already existing code . The topics here are quite different and maybe

unrelated to each other (though related to the series module as a whole) , hence separate prs would be used for each of these.

- I would like to confirm testing and documentation for all newly written code during this period .

Lastly if time permits I would like to address a good chunk of the stretch goals as the issues there are quite intriguing and would need some discussion . I would like to use the last 1 week at least as a buffer week, where I fix anything left from my proposal and also ponder about the content from the stretch goals doc.

- **Post Gsoc Period -** I would be doing the following things once my tenure of contribution is over

1. Create a dedicated Wiki page or blog regarding the state of the series module . Though there are a couple pages on sympy wiki doing this to some extent, hence I would either be updating those or making a new one for future work by new contributors. This would also involve opening up any good first issue I find in the series module for making it easier for fellow contributors to hop in and start contributing.
2. Finish all stretch goals even after project completion. As most of the stretch goals were issues first reported by me on the github page , I would be glad to finish implementing them or jump in anytime to help fellow contributors wanting to tackle those issues.
3. I plan to keep contributing even after the Gsoc period ends . First because I gained a lot of experience in the last few months of contributing and secondly because sympy is quite dear to me as it was the first library I contributed to when I was a complete beginner in open source.

I also plan on mentoring a project in the series/ring_series module in future and also take part next year (**if sympy allows multiple participations**) under some other topics which I prefer like group theory , probability and stats etc.

- **Time Commitment -**


1. I don't have any other significant commitments during the Gsoc contribution period , hence I would be able to dedicate around 30 hours per week quite easily during Gsoc . That's already 10 more than what the Gsoc website quotes for a 175 hour project(which is 20 hours) . If anytime I sense the project has started lagging behind to the proposed timeline, it wouldn't be a big concern as I could invest more time (even 40 hours) whenever it is needed.
2. If I am caught up with any work (personal or miscellaneous) and would probably be missing a deadline or a weekly meeting I would be informing the mentor way beforehand and would also indicate the span of dates where I would put in more time and get the work done .

I plan to finish the Gsoc project with almost 1.5 to 2 weeks of buffer i.e. within a span for 10 weeks rather than the standard 12 week mark . This will give me more time to address anything left or give more importance to any issue that demands more time and debugging. This also gives me more time to document my work and test the newly added features and improve the series module as a whole.

As I finish writing my proposal here I would like to acknowledge the great amount of support I got during the past few months and especially when I was an amateur. I would like to thank Kalevi Suominen (jksuom), Sidharth Mundhra (Osidharth), Oscar Benjamin (oscarbenjamin), Christopher Smith (smichr) who have helped me a lot in contributing by reviewing my PRs and guiding me whenever I was stuck.

References and Links

1. [Arighna Chakrabarty's Proposal](#)
2. [Sidharth Mundra's Proposal](#)
3. [Sartaj Singh's Proposal](#)
4. [Computing Limits in Symbolic Manipulation Systems -Gruntz's Thesis](#)
5. [Dominik Gruntz and Wolfram Koepf, Formal Power Series](#)
6. [Wolfram Koepf, Power Series in Computer Algebra](#)

- 
7. [Manuel Kauers, Computing Limits of Sequences](#)
 8. [Branch cuts/ Branch points Wiki](#)
 9. [Branch Cut Wolfram](#)
 10. [Power series and Asymptotic expansions for Bessel and Modified Bessel functions](#)