GSOC 2025 Proposal - Improving Relational Reasoning and Introducing Quantifier Support in SymPy's Assumptions Framework

Table of Contents :

Me the Person

Personal Information:

- **Name :** Krishnav Bajoria
- **Email :** bajoriakrishnav@gmail.com
- **University :** IIT BHU, Varanasi, India
- **GitHub :** https://github.com/krishnavbajoria02
- **TimeZone :** IST (UTC +5:30)
- **Linkedin :** https://in.linkedin.com/in/krishnav-bajoria

Background Overview:

I am Krishnav Bajoria, a third-year student in Mathematics and Computing at IIT (BHU), Varanasi, ranked among the top 5 in my batch academically. My strong foundation in mathematics and computer science includes coursework in data structures, algorithms, discrete mathematics, number theory, optimization techniques, linear algebra, multivariate calculus, and numerical approximation methods. I possess technical expertise in programming languages such as Python, C++, C, and Java, alongside practical experience in web and app development. My proficiency extends to machine learning (ML) and artificial intelligence (AI), making me confident in my ability to contribute effectively to SymPy's assumptions project.
I have significant research experience that aligns with the goals of this project. I was selected for the prestigious Global Research Internship Program at Changwon National University, South Korea, where I developed a novel architecture for object detection in autonomous vehicles. This work led to a research paper co-authored with my mentor, Professor Oh Seol Kwon, which is currently submitted to an SCIE journal. Additionally, I undertook a museum dataset scraping project where I curated a dataset of over 80 million images of artifacts and objects from museums worldwide. I trained models like BLIP and CLIP to create a virtual reality headset-based

guide for museum visits, which has resulted in another research paper under development.
Beyond research, I have prior software development experience from my internship at PepsiCo-Varun Beverages Limited. There, I developed an app for bottle detection and recognition using 3D point-of-view depth analysis. I also created a sentiment polarity tool that dynamically extracts comments related to their products to analyze overall sentiment trends and recent news. My technical contributions demonstrated my ability to solve complex real-world problems using advanced computational techniques.
I have also excelled in competitive environments, being a national finalist in hackathons such as the Goldman Sachs Hackathon and Convolve 2.0 (a Pan-IIT data science competition). Additionally, I with my team registered the best Bitcoin trading strategy at the Inter-IIT Technical Meet at IIT Bombay, having delivered exceptional performance metrics.
With my strong academic background, proven research capabilities, software development experience, and competitive achievements, I am confident in my ability to make meaningful contributions to SymPy’s Assumptions Project during Google Summer of Code 2025.

Me the Programmer

I primarily use **VS Code** as my coding platform due to its ease of use, wide range of extensions, indexing capabilities, and debugging tools, which make it ideal for both development and experimentation. Additionally, I use **Sublime Text** for lightweight editing tasks because of its speed and simplicity, especially when working with smaller scripts or quick edits. For Python-based machine learning projects requiring intensive computations and module integration, I rely on **Jupyter Notebook** and **Google Colab**, which provide an interactive environment for data analysis and model training.I primarily use **Windows** but also am comfortable with **Ubuntu**.
I have over **seven years** of programming experience, starting in high school, with around three to four years focused on Python. My programming journey includes creating diverse projects such as a text-to-voice application, chatbot models (text-to-text), object detection

pipelines for autonomous vehicles, and bottle detection classifiers for PepsiCo's coolers. All of my projects and related work can be found here in my resume.
My experience with Python spans various domains, including symbolic computation, machine learning, and web development. I admire Python's simplicity and readability, which allow concise code compared to other languages requiring verbose syntax. Features like **lambda functions** and **list comprehensions** stand out for their expressiveness and ability to streamline complex operations. The vast ecosystem of libraries supported by Python's strong community further enhances its versatility which is another striking feature of Python.
One of my favorite features of SymPy is its ability to handle **symbolic computations** effectively. For example, SymPy allows defining symbols that retain their exact mathematical properties rather than approximating them numerically. This enables operations like algebraic simplification, differentiation, integration, and solving equations symbolically.
Another interesting feature is the **assumptions** part of SymPy.
The very fact that SymPy uses assumptions to simplify expressions and uses it in their solvers is something very unique I have seen till now which really fascinated me.

Example:

```
from sympy import Symbol, sqrt, solve, Eq

# Define a symbol with an assumption that it is positive
x = Symbol('x', positive=True)

# Simplify an expression using the assumption
expr = sqrt(x**2)
print(f"Simplified Expression: {expr}")  # Output: x

# Use assumptions in solving equations
eq = Eq(x**2 - 4, 0)
```

```
solutions = solve(eq)
print(f"Solutions: {solutions}")  # Output: [2]

# Check assumptions for further queries
print(f"Is x positive? {x.is_positive}")  # Output: True
```

The fact that SymPy simplifies sqrt(x**2) to x when x is declared as positive, and otherwise keeps it as |x|, is truly remarkable. Additionally, when solving equations like x**2 - 4 = 0, where the solutions could be 2 and -2, SymPy chooses 2 because of the positive assumption assigned to x. This unique feature of leveraging assumptions for simplifications and solving equations is one of the most fascinating aspects I have noticed in SymPy.
Regarding version control systems like Git, I have been using Git extensively since my first year . While I am comfortable with basic Git commands such as git init, git commit, git push, and branching workflows, there are still advanced features I am learning. Despite occasional confusion with complex Git operations, I enjoy the challenge of exploring new tools and workflows.

Contributions to SymPy

I started using SymPy in December 2024 and made my first contribution to the main repository in January 2025. I have been consistently contributing to the organization since then. I plan to be a long-term contributor and will continue to improve this software even after this program is finished.I understand that each GSoC applicant must submit at least one code-related patch to SymPy's main repository, demonstrating both familiarity with the workflow and proficiency in Python.Below is a brief summary of my contributions so far:

PRs Merged:

1) **https://github.com/sympy/sympy/pull/27512** **-** Fixes prime handler for integer exponentiation

2) **https://github.com/sympy/sympy/pull/27511** - Fixes bug with algebraic exponential handler

3) **https://github.com/sympy/sympy/pull/27552** **-** Added cmath support for lambdify and implemented CmathPrinter

4) **https://github.com/sympy/sympy/pull/27555** - Fixes handling of integer exponentiation in rational assumptions

5) **https://github.com/sympy/sympy/pull/27666** - Improves Q.infinite(Expr) handling for symbolic conditions and complex infinity

6) **https://github.com/sympy/sympy/pull/27685** - Fixes IntegerPredicate for Abs to handle non-integer inputs correctly

7) **https://github.com/sympy/sympy/pull/27693** - Enhances Pow handler to correctly handle zero base cases in Q.real evaluation

8) **https://github.com/sympy/sympy/pull/27708** - Fixes ask(Q.finite(x**-1), Q.real(x)) to handle potential division by zero

9) **https://github.com/sympy/sympy/pull/27741** - Fixes IntegerPredicate Handling for Pow and Mul Expressions

10) **https://github.com/sympy/sympy/pull/27520** - Added NaN handler for NegativePredicate (Coauthor)

PRs Open (Unmerged):

1) **https://github.com/sympy/sympy/pull/27679** - Enhances PositivePredicate for real base and even exponent in power expressions

2) **https://github.com/sympy/sympy/pull/27686** - Fixes RecursionError in ODE solver when using Float coefficients (Draft)

3) **https://github.com/sympy/sympy/pull/27678** **-** Adds features to infer real variables from Q.gt and Q.lt (Draft)

4) **https://github.com/sympy/sympy/pull/27519** - Fixes RealPredicate for Mul to handle uncertainty in zero-checks correctly (Draft)

5) **https://github.com/sympy/sympy/pull/27549** - Fixes incorrect handling of infinite imaginary parts in _Imaginary_number handler (Coauthor)

Issues Opened:

1) **https://github.com/sympy/sympy/issues/27739** - Issue regarding incorrect Behavior in ask for Integer Division

2) **https://github.com/sympy/sympy/issues/27748** - Issue regarding ask(Q.noninteger(x), Q.integer(x)) returns None Instead of False and to implement a NonIntegerHandler

3) **https://github.com/sympy/sympy/issues/27705** - Issue regarding inconsistency regarding ask in evaluating the finiteness of oo * y when y is zero.

4) **https://github.com/sympy/sympy/issues/27673** - Issue regarding implementing comprehensive testing framework for lambdify backends

5) **https://github.com/sympy/sympy/issues/27707** - Issue with the handling of assumptions for the expression x**-1 when x is real and could potentially be zero

Apart from this I have taken active part in discussions in the SymPy community by answering queries on Google group discussions on mailing list, and reviewing others' PRs from time to time.

I am currently actively reviewing these PRs:

1) **https://github.com/sympy/sympy/issues/27739** - Issue regarding incorrect Behavior in ask for Integer Division

2) **https://github.com/sympy/sympy/pull/27696** - PR regarding adding handlers for Q.transcendental predicate.

About the Project

From this section onward, I will discuss the project, specifically addressing its structure, the planned deliverables, the project's impact and feasibility, and the expected outcomes that it aims to achieve.This is intended to be a 350 hours project.

Overview of Two-Fold Plan for Enhancing SymPy's Assumptions System

This overview outlines the two core pillars of the plan. In subsequent sections, I will expand on each component.

1. Relational Reasoning & Incremental SAT Solvers

- Relational Gaps: Address missing equivalences and enable inferences without explicit Q.real declarations.
- Incremental SAT solver: Integrate incremental SAT solver to enhance ask/satask performance by reusing clauses across queries.

2. Quantifier Support
   - Universal/Existential Quantifiers: Introduce Q.forall/Q.exists predicates enabling advanced logical queries about mathematical properties across domains
   - Implementation approach will combine Z3-inspired pattern-based instantiation with SymPy's declarative assumptions framework.

---

1. Relational Reasoning & Incremental SAT Solvers

I will now discuss the structure of the relational reasoning and SAT integration project, building upon the foundational work done by TiloRC in his 2023 GSOC project. His project focused on enhancing relational assumptions through the implementation of linear real arithmetic (LRA) SAT solvers, which significantly improved the handling of linear inequalities. More information regarding his project can be found here : TiloRC's report 2023.

TiloRC's work marked a pivotal moment, as it was the first time SymPy's assumptions system gained the capability to better understand and utilize linear inequalities. My goal of this ‘pillar’ is to carry this work further by addressing existing gaps and identifying low-hanging fruit that can enhance reliability, integrity, and overall utility in relational reasoning, and by addition of an incremental SAT solver which will improving the efficiency and speed of the assumptions system when it comes to dealing with linear inequalities.

In this endeavor, I would like to first address the low-hanging fruit in enhancing SymPy's relational reasoning, which involves fixing several key issues related to inequalities and equality predicates.

A. Addressing Low-Hanging Fruit:

First, I aim to establish these equivalences and implications:

```
Q.eq(x, 0) ⇐⇒ Q.zero(x)
Q.gt(x,y) =⇒ Q.ge(x,y)
Q.lt(x,y) =⇒ Q.le(x,y)
```

These equivalences and implications are crucial for enhancing the robustness and reliability of SymPy's relational reasoning capabilities. By addressing these foundational issues, we can significantly improve the system's ability to handle complex logical queries and assumptions. Some discussions about these can be found in this issue.

B. Addressing the Limitation of Explicit Real Variable Declarations in LRA SAT

One of the key limitations in TiloRC's 2023 GSOC project was that the Linear Real Arithmetic (LRA) SAT solver worked only on expressions where all variables were explicitly declared as real. This limitation is detailed in this GitHub issue. My proposal aims to resolve this issue, enabling SymPy's assumptions system to infer the real nature of variables automatically in certain cases, making it more powerful and user-friendly.

Current Behavior[1]

As pointed out in the issue , when using SymPy's ask function with inequalities, it requires all variables to be explicitly declared as real for the LRA SAT solver to work correctly. This behavior is illustrated below:

[1] Taken directly from this issue: https://github.com/sympy/sympy/issues/27470

```
from sympy import ask, symbols
# Variables not explicitly declared as real
x, y, z = symbols('x y z')
print(ask(x > z, (x > y) & (y > z)))  # Returns None
# Variables explicitly declared as real
x, y, z = symbols('x y z', real=True)
print(ask(x > z, (x > y) & (y > z)))  # Returns True
```

This limitation forces users to manually declare variables as real=True, which can be cumbersome and unintuitive. Ideally, we want the ask function to infer correctly without requiring explicit real=True declarations.

Proposed Solution

To implement this fix, I propose enhancing the inference mechanism within the lra_satask function to handle single-variable inequalities intelligently while appropriately managing complex expressions.

1. Inference Logic for Single Variables

- For predicates like Q.gt(x, y) or Q.lt(x, y), we can infer that both x and y are real because strictly greater/lesser inequalities inherently require real-valued operands.
- However, for expressions like Q.gt(x + y, 0) or Q.lt(x * y, 0), we cannot conclusively infer that x and y are individually real. These cases require explicit declarations or additional assumptions.

2. Implicit Real Assumptions for Single Variables

- Modify the lra_satask function to automatically add Q.real assumptions for single variables involved in inequalities if they are not explicitly declared as real.
- This enhancement will allow the LRA SAT solver to make deductions without requiring manual declarations of real=True.

Desired Behavior/Expected outcome[2] :

The desired behavior is for the assumptions system to automatically infer the real nature of variables involved in inequalities where possible.

```
x, y, z = symbols('x y z')
assert ask(x > z, (x > y) & (y > z)) is True
assert ask(Q.gt(x, z), Q.gt(x, y) & Q.gt(y, z)) is True
assert ask((x > y) | (x <= y)) is None
assert ask((x > y) | (x <= y), Q.eq(x, u) & Q.eq(y, v)) is
None
assert ask(x > z, ~Q.le(x, y) & ~Q.le(y, z)) is None
```

This behavior ensures that users can work with inequalities seamlessly without needing explicit declarations for every variable.

C. Incremental SAT Solver Component

To conclude the first part of my project, I aim to address another limitation in SymPy's assumptions system: its speed and efficiency when handling relational queries. This issue is particularly evident in the performance of satask, the core bottleneck for many queries involving relational predicates. By implementing an incremental SAT solver, I propose to enhance the speed and efficiency of satask, making it more practical for real-world use cases. More regarding this can be found in this issue.

Current Performance Issues[3]

The current implementation of satask evaluates boolean formulas for satisfiability by solving them from scratch each time. This approach is inefficient, especially when formulas are highly similar and share common subformulas. For example, consider the following query:

```
from sympy import ask, Q
from sympy.abc import x, y
```

[2] As desired in the comment of this issue by TiloRC
[3] Taken directly from this issue

```
ask(Q.negative(x), Q.prime(x) | Q.positive(x))
```

Here's how satask works:

1. $Constructs\ two\ boolean\ formulas$
    - $F1\ =\ Q.negative(x) \wedge Q.prime(x) \vee Q.positive(x) \wedge rules$
    - $F2\ =\ \neg Q.negative(x) \wedge Q.prime(x) \vee Q.positive(x) \wedge rules$
2. $Checks\ satisfiability\ of\ F1\ and\ F2\ independently.$
3. $Based\ on\ the\ results:$
    - $If\ F1\ is\ satisfiable\ but\ F2\ isn't,\ returns\ True.$
    - $If\ F2\ is\ satisfiable\ but\ F1\ isn't,\ returns\ False.$
    - $If\ both\ are\ satisfiable,\ return\ None.$

$While\ this\ approach\ works,\ it\ is\ slow\ because\ the\ SAT\ solver\ does\ not\ share\ any\ work\ checking\ F1\ and\ F2,\ even\ though\ they\ are\ highly\ similar.$

Proposed Solution: Incremental SAT Solver

To address this inefficiency, I propose integrating an incremental SAT solver into SymPy's assumptions system. Incremental SAT solvers allow for reusing intermediate results between related queries by maintaining a "state" that can be updated with additional constraints or rolled back to previous checkpoints.

How it Works

1. $Initial\ Validation:\ Begin\ by\ ensuring\ the\ assumptions\ are\ consistent\ using\ the\ formula.$
    - $F_0\ =\ assumptions \wedge rules$
    - $If\ F_0\ is\ unsatisfiable,\ immediately\ trigger\ an\ exception\ without\ further\ evaluation.$
2. $State\ Retention:\ Store\ the\ SAT\ solver's\ state\ after\ verifying\ F_0\ to\ prevent\ redundant\ computations\ of\ common\ subformulas.$
3. $Progressive\ Validation:$
    - $Introduce\ constraints\ for:$

$F_1 = Q.negative(x) \wedge Q.prime(x) \vee Q.positive(x) \wedge rules,\ then\ assess\ satisfiability\ using\ the\ stored\ state.$

- $Similarly,\ evaluate\ F_2 = \neg Q.negative(x) \wedge Q.prime(x) \vee Q.positive(x) \wedge rules\ by\ extending\ the\ existing\ state..$

Required Functionality

To implement this, these two features need to be added in SAT solver :

- **State Preservation:** Save and reload progress during computation.
- **Dynamic Constraints:** Modify constraints incrementally without restarting.

Implementation Plan

This part of the project is fairly new to me but exciting nonetheless. I have already collected resources and research papers to study during the community bonding period if my proposal gets selected. I enjoy learning new things and am eager to take on this challenge with guidance from mentors and thorough research. Also in terms of implementation, I think there will be quite a bit of refactoring code when implementing the incremental SAT solver.

Research Resources

- CryptoMiniSat repository: A state-of-the-art incremental SAT solver with Python bindings.
- Incremental SAT Solver Paper: A detailed explanation of incremental SAT solving techniques.

2. Adding Support for Quantifiers in Assumptions System

The second part of my project focuses on adding support for quantifiers—both universal (forall) and existential (there_exists)—to SymPy's assumptions system. This enhancement will allow users to reason about logical statements involving quantifiers, significantly increasing the power and versatility of the assumptions framework.

Current State and Motivation

Currently, SymPy's assumptions system lacks native support for quantifiers like forall (universal quantifier) and there_exists (existential quantifier). This limits its ability to handle advanced logical reasoning tasks. For example:

- We cannot query whether a property holds for all elements of a set (e.g., "Are all primes integers?").
- Similarly, we cannot reason about the existence of an element satisfying a property (e.g., "Does there exist an even prime number?").

Adding support for quantifiers will enable queries like:

```
ask(forall(Q.prime(x), Q.integer(x)))  # Are all primes integers?
ask(there_exists(Q.even(x), Q.prime(x)))  # Does there exist an even prime?
```

This enhancement will bring SymPy closer to the reasoning capabilities of formal logic systems and make it more useful for users working with mathematical proofs, symbolic reasoning, and automated theorem proving.

Below discussed are two approaches I have in mind currently for implementing basic functionality of quantifiers in the SymPy assumptions framework. Based on discussions with my mentor I would finalise the implementation plan for adding quantifier support.

Proposed Approach 1

To implement quantifier support, I propose adopting a declarative programming paradigm similar to the allargs approach already used in SymPy. The allargs function allows rules to be written declaratively, making them easier to read, maintain, and extend. Here's an example of how allargs is currently used:

```
[4]@class_fact_registry.multiregister(Add)
def _(expr):
    return [
        allargs(x, Q.positive(x), expr) >>
Q.positive(expr),
        allargs(x, Q.negative(x), expr) >>
Q.negative(expr),
        allargs(x, Q.real(x), expr) >> Q.real(expr),
        allargs(x, Q.rational(x), expr) >>
Q.rational(expr),
        allargs(x, Q.integer(x), expr) >> Q.integer(expr),
        exactlyonearg(x, ~Q.integer(x), expr) >>
~Q.integer(expr),
    ]
```

This declarative approach makes it straightforward to define rules like "If all arguments are positive, then the sum is positive." I propose extending this paradigm to support quantifiers.

Implementation Plan

1. Syntax Design

The first step is to define how users will interact with quantifiers in the assumptions system. While the exact syntax will require further discussion with my mentor to ensure consistency with SymPy's codebase, here are some initial ideas:

Universal Quantifier (forall):

```
ask(forall(Q.prime(x), Q.integer(x)))  # Are all primes
integers?
```

- This query would evaluate whether the predicate Q.integer(x) holds true for all values of x satisfying Q.prime(x).

[4] Current implementation

Existential Quantifier (there_exists):

```
ask(there_exists(Q.even(x), Q.prime(x)))  # Does there
exist an even prime?
```

- This query would evaluate whether there exists at least one value of x satisfying both Q.even(x) and Q.prime(x).

2. Declarative Implementation

The implementation of quantifiers will follow a declarative approach similar to allargs. For instance:

- forall can be implemented as a rule that checks whether a predicate holds true for all elements in a domain.
- there_exists can be implemented as a rule that checks whether at least one element satisfies a predicate.

Example (conceptual):

```
forall(predicate, domain) >> result
there_exists(predicate, domain) >> result
```

3. Integration with Existing Framework

The quantifier rules will be integrated into the existing assumptions framework so that they can interact seamlessly with other predicates and relational reasoning logic.

Proposed Approach 2 (Based on the current implementation of quantifiers in Z3)

Z3's Quantifier Implementation: Core Concepts

Z3 handles quantifiers using pattern-triggered E-matching and model-based quantifier instantiation (MBQI). Below is a detailed breakdown of its workflow:

1. Key Terminology

- **E-Matching:** A technique to instantiate quantifiers by matching patterns against terms in an E-graph (a data structure that tracks equivalence classes of terms modulo equalities).
- **Patterns/Triggers:** Subexpressions in quantified formulas that include all bound variables. These guide Z3 on when to instantiate quantifiers.
  Example: In $\forall x.\ f(x) > 0$, $f(x)$ is a trigger.
- **Ground Terms:** Concrete terms without variables (e.g., $f(5)$).
- **MBQI:** A counter-example refinement loop where Z3 constructs candidate models and checks if they violate quantifiers, adding instantiations to eliminate invalid models.

2. Detailed Workflow in Z3

- **Pattern Annotation:**
  Users annotate quantifiers with triggers to guide instantiation.
- **E-Matching Engine:**
  - Maintains an E-graph of terms and equalities.
  - Matches triggers against ground terms in the E-graph.
  - Instantiates quantifiers with substitutions from matches.
    Example: If $f(5)$ exists, instantiate $x=5$ in $\forall x.\ f(x) > 0$ → check $f(5) > 0$.
- **Model-Based Quantifier Instantiation (MBQI)**:
  - Builds candidate models for quantifier-free formulas.
  - Checks if models satisfy quantifiers. If not, generates instantiations to refine the model.
  - *Example*: For $\exists x.\ f(x) = 0$, MBQI might propose $x=3$ and check $f(3)=0$.
- **Handling Undecidability:**
  Z3 prioritizes heuristics (patterns, MBQI) over completeness, as first-order logic is undecidable. Users must provide meaningful triggers for efficiency.

Adapting Z3's Approach for SymPy

1. Declarative Pattern Triggers

- Mechanism: Use SymPy's Wild symbols to define patterns.

```
from sympy import Wild
from sympy.assumptions import Q

x = Symbol('x')
a = Wild('a')  # Pattern variable
expr = Q.forall(x, x > a, patterns=[x > a])  # Trigger on x
> a
```

- Workflow:
    - Match ground terms (e.g., 3 > 2) against patterns.
    - Instantiate quantifiers when matches occur.

2. Simplified E-Matching

Replace Z3's E-graph with SymPy's algebraic substitution and simplification:

```
def ematch(expr, pattern, assumptions):
    match = expr.match(pattern)  # Uses SymPy's .match()
    if match:
        substituted = conclusion.subs(match)  # Substitute
variables
        return ask(substituted, assumptions)
    return None
```

Example: For Q.forall(x, x > a), if expr = 5 > 3, match a=3 and check 5 > 3.

3. Domain Inference

Leverage SymPy's assumptions to infer quantification domains:

```
ask(Q.forall(x, x**2 >= 0),Q.integer(x))  # Domain: ℤ →
Checks ∀x∈ℤ, x²≥0
```

Explicit Domains: Allow manual specification (e.g., domain=Q.integer(x)).

4. Incremental Instantiation

Track ground terms in the assumption context to trigger quantifier checks:

```
class QuantifierTracker:
    def __init__(self):
        self.ground_terms = set()  # Track terms like
prime(3), prime(5)

    def add_term(self, term):
        for pattern in active_patterns:
            if term.match(pattern):

self.instantiate_quantifier(pattern.quantifier, term)
```

If this approach is followed then a tentative roadmap could be as follows:

Phase	Focus Area	Deliverables
Phase 1	Quantifier AST Nodes	Q.forall/Q.exists API
Phase 2	Pattern Tracking & E-Matching	Integration with satask

Phase 3	Domain Inference & Simplification	Automated ℝ/ℤ handling

Research Resources

Since this area is relatively new to me, I have already started collecting resources and papers that will guide me during implementation. Some of these include:

1. Z3 Docs
2. E-Graph Representation
3. E-Matching Paper
4. Pattern Inference
5. PR #7608 -Implementation of First Order Logic

During the community bonding period, I plan to study these resources thoroughly while discussing implementation details with my mentor.

Execution / Project Timeline and Structure

Bonding Period

During the bonding period, I will focus on understanding SymPy's assumptions system and its integration with relational reasoning and SAT solvers. This includes studying the existing codebase, researching incremental SAT solvers, and discussing the plan of execution with my mentor. I will also connect with other contributors to learn about their projects and how they might intersect with mine.As I'm already a SymPy developer I'll not waste time learning basic things about SymPy. I can use this time to start working on my project. The speed might be slow as I'll have exams during this period.
From my past experiences, I understand that things can change quickly—for example, a bug might arise that needs immediate attention before proceeding further. While I will propose a tentative timeline here, this

period will help clarify expectations, refine the plan, and set the stage for smooth collaboration. This timeline will also serve as a way to track progress and ensure that any delays are identified early.

Phase 1: Weeks 1–4 (Relational Reasoning Enhancements)

Focus: Fixing relational gaps and addressing explicit real variable limitations.

Week 1	Implement equivalences and implications for relational predicates, ensuring robust handling of inequalities.
Weeks 2-3	Resolve the limitation where variables must be explicitly declared as real for LRA SAT to work. Enhance lra_satask to infer real assumptions for single-variable inequalities while managing complex expressions appropriately.
Week 4	Write comprehensive tests for relational reasoning improvements and also serve as a buffer week for the tasks mentioned in weeks 1-3.

Deliverables:

- Fixed equivalences and implications for relational predicates.
- Enhanced inference logic in lra_satask for real variables.
- A suite of tests verifying the reliability of relational reasoning enhancements.

Phase 2: Weeks 5–8 (Incremental SAT Solver Integration)
Focus: Speeding up satask by integrating an incremental SAT solver.

Week 5	Study the existing satask implementation in depth and identify bottlenecks in performance. Begin integrating an incremental SAT solver to reuse intermediate results between related queries.
Weeks 6-7	Implement functionality for setting checkpoints and adding constraints incrementally within the SAT solver. Optimize satask to leverage these features effectively.
Week 8	Write benchmarks comparing the performance of satask before and after incremental SAT solver integration. Address edge cases and finalize tests for speed improvements.Also serves as a buffer week for the tasks mentioned in weeks 5-7.

Deliverables:

- Integrated incremental SAT solver with checkpointing and constraint addition functionality.
- Optimized satask with improved speed and efficiency.

- Performance benchmarks demonstrating significant gains.
- Further if time permits here would also extend this speed up to the SMT solver. Although the SMT solver hasn't yet been integrated, if time permits I would surely like to integrate this completely as this would really enhance the speed of SymPy assumptions.

Phase 3: Weeks 9–11 (Quantifier Support Implementation)

Focus: Adding support for universal (forall) and existential (there_exists) quantifiers.

Week 9	Design syntax for quantifiers in SymPy's assumptions system (e.g., ask(forall(Q.prime(x), Q.integer(x)))). Discuss implementation details with mentors to ensure consistency with existing codebase conventions.
Week 10	Implement basic support for universal (forall) and existential (there_exists) quantifiers using the chosen approach. Ensure seamless integration with existing predicates and reasoning logic.
Week 11	Write tests for quantifier support, including queries like "Are all primes integers?" or "Does there exist an even prime number?"

Deliverables:

- Syntax design for universal and existential quantifiers in SymPy's assumptions system.
- Declarative implementation of quantifiers integrated into the framework.
- Comprehensive tests validating quantifier functionality.

Phase 4: Week 12 (Buffer and Finalization)

Focus: Implementing any remaining work or finalizing deliverables by adding test cases/ implementation for edge cases that might arise.

Week 12	This will serve as a buffer week for finalizing any remaining work and would serve as validating and documenting all work done in the past 11 weeks.

Other Commitments During GSoC

I have no other commitments during the summer, allowing me to dedicate my full attention to this project. My semester exams conclude on May 12, after which I will have a month-long break, ensuring uninterrupted focus. During the community bonding period (May 12–May 29), I plan to first focus on further research, reading documentation, and discussing finer details of the project with the community and my mentor.
Once the coding period officially begins on May 29, I will dedicate 30-35 hours per week to the project. If I fall behind during any given week, I will use weekends to catch up and ensure that all deliverables are met on time. From my past experiences, I know that unforeseen challenges, such as bugs or unexpected issues, can arise and affect progress. If such situations occur, I will promptly inform my mentors and adjust my workload

accordingly. I am fully committed to allocating additional time if needed to ensure the successful completion of this project.

Post-GSoC

Beyond GSoC, I plan to remain an active contributor to SymPy by refining and extending the features implemented during this project based on community feedback. My primary focus will be on improving the assumptions system further, optimizing performance, and addressing any new challenges that arise.
Additionally,I also plan to mentor new contributors, sharing my knowledge of SymPy’s codebase and helping them navigate their initial contributions effectively.
By staying engaged with the SymPy community post-GSoC, I hope to play an active role in shaping the future of symbolic mathematics in SymPy while continuing to grow as a developer and contributor.

Notes

- I have no major commitments during the summer, allowing me to dedicate my full attention and energy to SymPy throughout the 12 weeks of Google Summer of Code.
- I am genuinely excited about the prospect of seeing my work being used and benefiting others. If, for any reason, my contributions are not merged into the master branch by the end of the summer, I am fully committed to continuing my efforts beyond the summer to ensure they are completed and integrated successfully.

References

1) Tilo Reneau-Cardoso’s proposal

2) Sudhanshu Mishra’s proposal

3) Anurag Bhat’s proposal

4) Other good proposals on Sympy wiki page

5) GitHub Discussions

6) Some of the content of this proposal has been taken directly from SymPy GitHub issues page to address the current performance issues I aim to solve through my proposal. :

- Issue 27680
- Issue 27470
- Issue 27477

7) Some of the content of this proposal has been taken directly from the Google group discussion .

Acknowledgements & Appreciation

I began my journey of contributing to SymPy in January 2025, stepping into the world of open source with little knowledge of how to contribute effectively. At first, I faced numerous challenges, made mistakes, and stumbled at various points. However, the incredible support and guidance from the SymPy community have been instrumental in helping me grow over the past three months. Through this experience, I have not only learned how to write high-quality code but also how to think critically, design

solutions thoughtfully, collaborate effectively with others, and use constructive feedback from reviews to continuously improve myself. I am deeply grateful to the SymPy community for fostering such an encouraging and enriching environment for contributors.
I want to acknowledge some members of the organization who have helped me immensely.

@TiloRC (Tilo Reneau-Cardoso)
@asmeurer (Aaron Meurer)
@oscarbenjamin (Oscar Benjamin)
@bjodah (Bjorn Dahlgren)

Thank you for going through this proposal.
- Krishnav Bajoria -
