Improve CSE and Code Generation in SymPy

About Me :

Contact Information :

- **Name** : Arif Ahmed
- **University** : Birla Institute of Technology and Science, Pilani, Goa Campus
- **Short Bio** : Undergraduate Student of **MSc(Hons.)Mathematics** and **B.E(Hons.)Computer Science**
- **Email-Id** : arif.ahmed.5.10.1995@gmail.com
- **GitHub username** : ArifAhmed1995
- **Blog : https://arif7blog.wordpress.com/**
- **Time-Zone** : IST (UTC + 5:30)

Personal Information :

My name is Arif Ahmed, a second year undergraduate student at BITS Pilani Goa Campus,India. Currently I am pursuing a degree in Mathematics and Computer Science.

I use Linux Mint 18 on my workstation and PyCharm IDE for any Python related work.I have been coding in Python (more than a year), Java(> 2 years), C/C++(more than a year) and R(about a year).I like Python the best of all the languages mentioned because it's easier to work with as compared to the others.

Along with Mathematics courses such as Complex Analysis, Discrete Mathematics, Optimization, Introduction to Algebra, Real Analysis, etc I have also completed various online courses related to Computer Science and Programming.

The online courses successfully completed are :

- Algorithms I - Princeton University
- Introduction to Interactive Programming in Python(Part I) - Rice University
- Introduction to Interactive Programming in Python(Part II) - Rice University
- Principles of Computing(Part I) - Rice University
- Principles of Computing(Part II) - Rice University
- First Five Courses of Data Science Specialization - Johns Hopkins University
- Machine Learning - Stanford University

As part of the above courses I had to code various weekly assignments in Java, Python and R.The most exciting assignments were the ones in Rice University's courses. Unfortunately I cannot upload the assignments on GitHub as the instructors forbade us to do so. In my freshman year, I made a simple simulator in Java which simulates Diffusion of a Gas.

Ever since I was introduced to Python, I liked it far better than any other language. The reason is that one can think more about converting ideas to working code rather than wrestling around with the nuances of the language.One of the advanced features about Python that I liked was the `lambda` operator/function.It's a nifty way to create a nameless function.

One the best features I like about SymPy is it's capability to solve difficult integration problems, especially Indefinite Integrals :

```
>>> Integral(x**3*sin(x*2), x).doit()
-x**3*cos(2*x)/2 + 3*x**2*sin(2*x)/4 + 3*x*cos(2*x)/4 - 3*sin(2*x)/8
```

I am quite familiar with Git/GitHub and have learnt a lot about it whilst opening Pull Requests, creating branches, squashing commits, etc

Contributions :

I stumbled upon SymPy while searching for Mathematics related projects on GitHub, sometime around 20th September 2016.I have attempted to fix issues and understand SymPy's codebase since then.Here is a list of my pull requests :

Merged :

- Geometric sum evaluates correctly for float base**(Easy to Fix)**
- ring series : rs_series extended for logarithm **(Easy to Fix)**
- matrices : re and im works for matrices
- core.evaluate : evaluate(False) works for numeric operators
- evalf : Increase depth of symbolic evaluation for Abs()
- solvers/solveset: Rectify solution for some special equations
- ntheory : Add functionality to return list of all factors
- integrals : Add singularity test for Zero denominator
- core : Float constructor allows to set binary or decimal precision
- sympify : Added support for translating basic numpy datatypes

Unmerged (Open) :

- matrices:Matrix**exponent stays unevaluated for non-integer exponent **(Needs Decision)**
- matrices : Add an optional key 'evaluate' for MatPow **(Needs Decision)**
- Matrix : transpose of non-complex symbol returns the same symbol
- solve : Solutions with spurious real/imaginary terms are discarded
- Add contour plot function
- vector : Add Laplacian function

Introduction :

In the field of Compiler Theory, common subexpression elimination(CSE) is an optimization technique that searches for identical expressions(i.e. evaluates to the same value) within a larger one and analyzes if total evaluation time is decreased when those identical expressions are replaced by a single variable.

From the perspective of a Computer Algebra System(CAS) , CSE serves as a backend to pre-process the input expression and pass on a new optimized one to the solver layer.This in turn , saves computation time [3]

Code generation in SymPy is the process by which a sympy expression or expressions are transformed into executable code in another language like C/C++, Fortran, Julia, Python, etc.

The Project Idea :

My proposal is to improve the Code Generation module in SymPy of which an important part would be to implement an efficient CSE algorithm. The data structures and workflow to implement a robust CSE is detailed here : [1].

The complete details of the algorithm and well as comparison with some other existing techniques are written in this thesis : [4].

It seems that some organizations have a dedicated bug-fixing project offered during the summer. According to me, it's better if such a project is focused on a particular functionality or module. In Phase II of my project, I plan to take up and fix various important issues related to codegen and associated utilities.

Phase I will end with the incorporation of CSE into the code generation module, which has not been done with the existing CSE. This should be an option which the user will be allowed to set, as it is debatable whether this makes generated code better or not. Sometimes it does , sometimes most compilers are good enough and CSE is not really required.

Importance of the Project :

As mentioned earlier, a robust CSE saves computation time by pre-processing expressions thereby making it easier for the solver layer. For a CAS, expressions can be interpreted as polynomials.For example, $(\sqrt{sin\ a}\ -\ 3)(sin\ a + 4)$ can be expressed as a polynomial : $(t\ -\ 3)(t^2\ +\ 4)$ with $t\ =\ \sqrt{sin(a)}$.

Since continuous functions can be approximated by polynomials to a desired degree of accuracy , such polynomials are used as a replacement in many applications.Look at the **Introduction** in : [1].

The current CSE implementation in SymPy is lacking.Consider the following examples :

```
>>> z, u, a, v, b, w, q = symbols('z u a v b w q')
>>> P = z*u**4 + 4*a*v*u**3 + 6*b*u**2*v**2 + 4*u*v**3*w + q*v**4
>>> cse(P)
```

```
([(x0, u**2), (x1, 4*u*v), (x2, v**2)], [a*x0*x1 + 6*b*x0*x2 + q*v**4 + u**4*z +
w*x1*x2])


Multiplications : 19
Additions/Subtractions : 4
```

Normally a implementation of CSE should yield :

```
>>> cse(P)
([(x0, u**2), (x1, u*v), (x2, v**2)], [4*a*x0*x1 + 6*b*x0*x2 + q*x2**2 + x0**2*z +
4*w*x1*x2])


Multiplications : 16
Additions/Subtractions : 4
```

The algorithm mentioned in the paper would yield a more optimized result :

```
>>> cse(P)
([(x0, u**2), (x1, 4*u)], [u*(u*z + a*x1)*x0 + v**2*(u*(x1*w + v*q) + 6*b*x0)])


Multiplications : 13
Additions/Subtractions : 4
```

In some cases SymPy's existing CSE doesn't work at all :

```
>>> x = Symbol('x')
>>> p = x - x**3 + x**5 - x**7
>>> cse(p)
([], [-x**7 + x**5 - x**3 + x])


Multiplications -> 12
Additions/Subtractions -> 3
```

As per the algorithm we get a far more optimized result :

```
>>> x = Symbol('x')
```

```
>>> p = x - x**3 + x**5 - x**7
>>> cse(p)
([(x0, x**2)], [x*(1 - x0*(1 - x0*(1-x0)))])
```

This result is also achieved by current CSE if we tell it to use the horner algorithm :

```
>>> cse(horner(p))
([(x0, x**2)], [x*(1 - x0*(1 - x0*(1-x0)))])
```

However, it would better if the user didn't have to worry about telling CSE to use horner or not in order to get the best result.

As mentioned here [2] , SymPy's current CSE method is used in many open source projects and as such would greatly benefit if the algorithm is improved.

There are various issues with codegen and related utilities in SymPy such as autowrap, ufuncify and lambdify. Such modules are important as they are useful in the scientific community as can be understood from the following issues : #11456 , #11991 , #10522 .

Definitions and Keywords :

1. **Literal** - A variable or constant(e.g : 2, 92, x, a, y).
2. **Cube** - A product of literals each raised to some non-negative integer power along with an associated positive or negative sign (e.g : $3a^2b$, $-2a^3b^2$).
3. **SOP representation** - SOP stands for Sum of Products.In the context of polynomials, it is simply a sum of cubes.
4. **Cube-Free** - A SOP expression is cube-free when there is no cube(except for the cube 1) that divides all cubes of the expression.
5. **Kernel** - Given a polynomial P and a cube c , the expression $\frac{P}{c}$ is a kernel if it is cube-free and is constituted of at least two terms.(e.g : If $P = 3x^4y^2 - 4x^5y^3$ and $c = x^4y^2$ then $\frac{P}{c} = 3 - 4xy$ is a kernel.
6. **Co-Kernel** - The cube utilized to obtain a kernel is a co-kernel. In the above example $c = x^4y^2$ is the co-kernel.

Design :

The important data structures to be used in this project are :

Kernel Cube Matrix(KCM)

Used in detection of multiple cube common sub-expressions. The input expression is acted upon by the Kernelling Algorithm(details mentioned later).A set of kernels and corresponding co-kernels are generated.

A matrix is constructed whose rows denote the co-kernels generated and columns denote distinct cubes which constitute a particular kernel.

$K(i, j) = 1$, if the product of co-kernel in row i and the kernel cube in column j yields a product term in the original set of expressions. We define a “rectangle” as a set of row and columns of the KCM such that all the elements are 1. The “value” of the rectangle is the weighted sum of the number of operations saved by selecting the common sub-expression or factor corresponding to that rectangle.

Therefore , selecting the optimal set of common subexpressions and factors is equivalent to finding a maximum valued rectangle of the KCM.

A prime rectangle is one which is not covered by any other rectangle and thus yields more value than any rectangle it covers.

A rectangle has the following parameters :

- **R** : Number of rows
- $M(R_i)$: Number of multiplications in row i corresponding to a certain kernel/co-kernel pair.
- C : Number of Columns
- $M(C_j)$: Number of multiplications in Column j .
  This actually corresponds to no. of multiplications in a kernel cube denoted by column j .

Each element (i, j) represents a product term equal to the product of co-kernel i and kernel cube j .This element requires a total number of

$M(R_i) + M(C_j) + 1$ multiplications to be computed.

Therefore the total number of multiplications represented by the whole rectangle is equal to :

$$R * \sum M(C_j) + C * \sum M(C_j) + R * C$$

Each kernel corresponds to $C - 1$ addition operations. Therefore , total number of addition operations for a given rectangle : R * (C - 1)

When this rectangle is selected, a common factor corresponding to $\sum M(C_j)$ multiplications and $C - 1$ additions is extracted. This common factor is multiplied by each row, which leads to a further $\sum M(R_i) + R$ multiplications.

Therefore , the value of the rectangle is calculated to be :

$$m * \{ (C-1) * (R + \sum M(R_i)) + (R - 1) * \sum M(C_j) \} + (R - 1) * (C - 1)$$

best described as the weighted sum of the savings in the number of multiplications and additions.

Cube Literal Incidence Matrix(CIM)

Used in detection of single-cube common sub-expressions.

Once again , we find the rectangle that yields a maximum-valued covering of the CIM. A similar algorithm that was used for KCM will be used again. The common cube corresponding to the prime rectangle is obtained by finding the minimum value in each column of the rectangle.

Calculating the value of the rectangle :

- Let $\sum C[i]$ be sum of integer powers in the extracted cube C.
- Now, this cube saves $\sum C[i] - 1$ multiplications in each row of the rectangle.

- Therefore , the value of the rectangle : $(R - 1) * (\sum C[i] - 1)$

Implementation Details :

The pseudocode for the following algorithms can be found in the mentioned paper above[1].

1. **Kernelling Algorithm :** Generate set of kernels and corresponding co-kernels.
   This is performed to construct the KCM , as mentioned earlier.

2. **Distill Algorithm :** Multiple-Cube decomposition and factorization.
   This is a greedy iterative algorithm in which the best prime rectangle is extracted in each iteration :

   Outer loop : Kernels and co-kernels are extracted from set of expressions {p_i}, and KCM is formed from that using the Kernelling Algorithm.This outer loop exits if there Is no favourable rectangle in the KCM.

   Inner loop : Each iteration in the inner loop selects the most valuable rectangle, if present based on the previously defined value function. This rectangle is now added to the set of expressions, and we denote this new expression by a new literal(symbol in SymPy).

3. **Condense Algorithm :** Used for Single-Cube decomposition.
   This is done after the Distill algorithm. In a CIM, single-term common subexpressions appear as rectangles where the rectangle entries are non-zero integers. The CIM is updated by subtracting the extracted cube from all rows in the CIM in which it is contained.This basically extracts the best prime rectangle as decided by the value function described earlier. When no more favourable rectangles exist the algorithm is done.

After this, a new literal(symbol) is added with the extracted value(this value corresponds to the prime rectangle) to the existing literal set.

Non-commutative expressions

A drawback of this algorithm is that it doesn't work for expressions containing non-commutative variables.The issue is in the CIM which doesn't account for non-commutativity. A good start towards such an algorithm could be constructing a non-commutative form of the CIM. It can be constructed column by column (i.e. as a new variable is encountered while scanning current literal add a column for it).

Here is a simple example :

Consider the non-commutative variables a, b and the expression which uses them $a^2b + ba^2$.
We can create the CIM as mentioned in the paper, by simply scanning the individual cubes of an SOP expression and storing them in a matrix with rows denoting individual cubes and columns denoting symbols/variables which constitute the cubes. For the expression $a^2b + ba^2$ this would be the CIM :

	a	b
ba^2	2	1
a^2b	2	1

Commutative form of CIM

Instead , the CIM can be constructed while scanning through the tuple of symbols/variables which constitute each cube in the expression and adding a column if not already encountered. This would correspond to the following CIM :

	a	b	a
a^2b	2	1	0
ba^2	0	1	2

Non-commutative form of CIM

Phase I : Implementing a robust CSE

1. Start off by implementing classes for KCM and CIM data structures :

Prototype for KCM :

```
class KernelCubeMatrix(MutableSparseMatrix, MatrixBase):
        rlabels, clabels = None, None
        @classmethod
        def _new(cls, *args, **kwargs):
                self = object.__new__(cls)
                self._smat = {}
                return self

        def set_rlabels(self, rlabels):
                self.rlabels = rlabels

        def set_clabels(self, rlabels):
                self.clabels = clabels

        def get_rlabels(self):
                return self.rlabels

        def get_clabels(self):
                return self.clabels

        def make(self, expr, pairs):
                #pairs refers to kernel / co-kernel pairs
                self.rlabels, self.clabels = pairs.keys(), pairs.values()
                terms = list(expr.args)
                for i in self.rlabels:
                        for j in self.clabels:
                                if i*j in terms:
                                        self._smat[(terms.index(i), terms.index(j))] = 1
                ...
                ...
```

The above code serves to give an idea of the implementation and is not optimized.

The functions inherited from `MutableSparseMatrix` should be overidden as well. For example, consider the function `row_del` which is inherited. When a row is deleted in the KCM, it means a

co-kernel has been removed , therefore the rlabels have to be updated accordingly.

2. Next are the helper functions :

- Divide, Merge - Required for Kernel and co-kernel extraction.
- Collapse - Required for finding cube intersections.

Implement and test them with some example polynomials.

3. Complete implementing code for :

- Kernel/co-Kernel extraction.
- Kernel intersections.
- Cube intersections.

4. Write unit tests for CSE.

5. Integrate CSE into codegen**.**

Phase II : Improve codegen

1. Make codegen work for matrices (Matrices, SparseMatrices, MatrixSymbol, etc)

2. Support for complex numbers in ufuncify.

3. Reduce runtime complexity of loops using matrix multiplication.

4. Implement other improvements if time permits.

Note : I will write extensive documentation as I code, especially for the CSE algorithm. A lot of modules in SymPy lack proper comments which makes it difficult to understand that code.

Timeline :

Phase I :

Pull requests will be made at the end of **Week 2(after helper functions), 4, 6, 7**

Community Bonding Period(May 4 - May 30):

May 3 - May 18 :

Final Term exams. Not available during this period for any intensive discussion.To make up for this lost time, I'll start fixing the codegen related bugs after mid-term exams(i.e. from April 1st). That will free up time in **Phase II** and maybe I can extend **Phase I** a bit more (6 - 7 weeks).

May 18 - May 30 :

Discuss implementation details with mentor, and begin coding.

Week 1 (May 31 - June 5) :

Implement classes for KCM and CIM.

Week 2 (June 6 - June 12) :

Implement and test helper functions : `Divide`, `Merge`, `Collapse`. Start code for kernel/co-kernel extraction.

Week 3, 4 (June 13 - June 26) :

Finish implementation of Kernel/co-Kernel extraction and Kernel intersections. Two weeks is allocated because this process is the bottleneck in the whole algorithm and hence should be optimized as much as possible.

Week 5, 6 (June 27 - July 10) :

Finish code for cube intersections and unit tests. Make sure CSE implemented is robust and all unit tests pass.

Week 7 (July 11 - July 17) :

Integrate CSE into codegen. Begin **Phase II.**

Phase II :

Pull requests will be made at the end of **Week 9, 10, 11, 12, 13**.

Week 9 (July 25 - July 31) :

Include support for matrices in codegen #11456 . Submit **Phase 2 evaluation report.**

Week 10 (August 1 - August 7) :

Pre-compute constant expressions for code generation.

Week 11 (August 15 - August 21) :

Use techniques mentioned here to reduce runtime complexity of loops.

Week 12 (August 15 - August 21) :

Make lambdify handle derivatives #10844

Week 13 (August 22 - August 29) :

Buffer period. Complete any missing documentation and extra tests (if necessary).

How do I fit in ?

I have been studying SymPy code and contributing for about 8 months now. I have gone through a large part of the code in `codegen.py` and related printing files. Although I wanted to address the codegen related issues right away, it was difficult due to the continuous evaluation pattern followed in my university. I wish to take this summer as an opportunity to get them addressed. After my midterm exams get over by April 1st, I'll start to work on `codegen` issues.

I do have the algorithmic background to complete this project because of online courses where writing concise modular code was a priority. I have no such other plans for the summer and can easily devote

50+ hours per week in order to stick to the proposed timeline and hopefully implement other new features to the codegen module.

Notes

In the community bonding phase, I will discuss with my mentor/s how to tackle expressions containing non-commutative variables in the CSE algorithm. Currently, it seems to be the only drawback in the mentioned algorithm.

My third year starts on **July 30** but there are no major exams for the first one and a half months. Therefore I will be able to devote 40 - 50 per week easily for the whole of Phase II.

Post GSoC

One of the first tasks will be to port the robust CSE into symengine. Currently, I have only a basic understanding of C/C++ , therefore did not include this task in my GSoC timeline. I will further my knowledge in C/C++ in the upcoming semester and will implement this at the earliest.

I recently came to know that the Statistics module of SymPy has a lot of limitations . I would definitely like to expand it.

References :

1. Optimizing Polynomial Expressions by Algebraic Factorization and Common Subexpression Elimination - A.Hosangadi, F.Fallah, R.Kastner

2. Code Generation Notes

3. EUROCAL '87

4.Optimization Techniques for Arithmetic Expressions(Thesis of A.Hosangadi