Implementing a SymPy module for Integration of Homogeneous functions over Polytopes

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 :

The Project Idea :

The idea is to implement a module in sympy to integrate homogeneous functions over polytopes using the techniques described in [1].

Importance of the Project :

The ability to integrate polynomial functions over polytopes and polyhedra is required in computational methods such as extended finite elements, embedded interface methods, virtual element method and weak Galerkin method (see Introduction in [1]). Other uses are in Computer Graphics (rigid body simulations of solids)[2].

Theory :

Definitions and Keywords :

- **Polytope :** A geometric object with flat sides, and may exist in any general number of dimensions.
- **Cubature Rule :** A definite formula to compute numeric integral for a certain type of polytope.

Generalised Stokes's Theorem

In the context of this project,

$$\int_P (\nabla . X) f(x)\, dx + \int_P < X,\ \nabla f(x) >\ dx\ = \int_{\partial P} < X,\ n > f(x)\, d\sigma$$

$< . , . >$ denotes the inner product of vectors.

To understand the requirement of this theorem, lets us consider : $f(x) = 1$ and $X = x$

Euler's homogeneous function theorem states $qf(x) = < \nabla f(x), x > \forall x \in R^d$. We use this and substitute value of $f(x)$ and X in above formula.

Therefore the above rule translates to : $\int_P dx = \frac{1}{d+q} \sum_{i=1}^{m} \frac{b_i}{\|a_i\|} \int_{F_i} d\sigma$

d : This is the dimension of the polytope.

m : Number of facets.

F_i : One of the m facets of the polytope.

$d\sigma$: Lebesgue measure on the boundary that contains facet F_i.

a_i : Normal of the hyperplane H_i in which facet F_i lies .

b_i : Parameter of the hyperplane H_i: $a_i^T x = b_i$.

The LHS of the equation denotes the volume of the polytype and the RHS denotes the total sum of the volumes of the simplexes that project from the origin to the boundary facets.

This theorem is useful because it translates the problem of integrating a polynomial function over a polytope to integrating it over the boundary of the polytope which is computationally easier to deal with.

Note that this was possible because we assumed that the function $f(x)$ is homogeneous. However, any arbitrary polynomial function can also be decomposed into a sum of such homogeneous polynomials[1].

In [1] most of the mathematical derivations are concerned with extending the cubature rules which was devised by Lasserre for convex polytopes to both convex and non-convex polytopes. Extensions are also provided for polar curves and homogeneous functions in polar form.

Design :

The algorithms described in the paper require the following as input :

Vertices of a Polytope (In Cartesian Coordinates)

There should be an option for the user to manually enter a list of tuples which correspond to the vertices, or to use a polytope from the available library.

```
>> polytope_integrate(vertices = [(0, 0), (0, 1), (1, 1), (1, 0)], ....)
>> polytope_integrate(vertices = ('regular-hexagon', 4), ....)
```

4 refers to the radius. Here, the vertices will refer to an object for regular-hexagon. That object will already have pre-defined boundary equations therefore computation time for calculating the integral will be reduced as the convex hull algorithm will not be required here. Obviously, for a hexagon, the reduction in run-time may be less evident but there are many common polygons used in fracture mechanics, computer graphics, etc for which given only a set of vertices, the boundary equations are expensive to find. Hence, it would be great to have a library of such polytope objects. I will find out what are such common polytopes through discussion with my mentor/s who are connected to people working in such fields.

Connectivity of the vertices

There should be an option for the user to manually enter the boundary equations between vertices, however if the user does not enter them then the polytope will be interpreted as a convex polytope and the boundary equations can be calculated by finding the convex hull of the vertices (this is required to be done for finding a_i and b_i corresponding to a facet F_i later anyway).

```
>> x, y = symbols('x y')
>> polytope_integrate(boundaries=[x+y=2, x=y, x=0], ...)
>> polytope_integrate(vertices=[(0,0),(1,1),(0,2)], ...)
```

In the first case boundaries were supplied and there should be a check to see if the region is bounded or not.

In the second case, the vertices are interpreted to be part of a convex hull and the required boundary equations are calculated using the convex hull algorithm. In my opinion, it will be better if an error is thrown when both boundaries and vertices are given. This will save computation time, otherwise we'll need to check if the boundaries and vertices match to

give the same polytope.

For concave polytopes the user would have to specify the connectivity equations between vertices as there is no way to know how the user has defined the boundary connections between the vertices of his/her concave polytope.

Polynomial Function

This is the function which to integrate over the polytope. Note that every polynomial can be interpreted as a sum of homogeneous functions so all the over techniques which hold true for such functions are true for all polynomials as well. A plain SymPy expression (i.e univariate or multivariate polynomial) containing symbols should suffice here.

Implementation Details :

The Polynomial data structure used in this project is already implemented in SymPy. Here we focus on the algorithms described in the paper :

1. **Main Integration Algorithm** : Integration of polynomial over arbitrary polytope.
2. **Integration Reduction Algorithm** : Further reduction of integration of polynomial (Helper method for main algorithm).
3. **Convex Hull Algorithm** : This will be used to compute the hyperplane equations when only vertices are supplied.

Main Integration Algorithm

These implementation details definitely aren't final and will be the main topic of discussion with my mentor/s during the Community Bonding Period.

1. Determine the dimension, d of the polytope :

   If vertices are supplied then we simply take the length of the tuple of any vertex. If vertices are not supplied then we need to fetch the dimension from the facet boundaries of the polytope. The LHS of the relational can be coerced to Poly class.

Then length of `Poly(expr).gens` is equal to $d-1$.

2. Determine m, the number of hyperplanes :

   If the facet boundaries are supplied, then m is simply the length of the facet boundary list.

   If vertices are supplied then the hyperplane equations will have to be first computed by using a convex hull algorithm. This is not only required for finding the value of m but also for calculating the values of a_i and b_i, both of which are required in calculating the result.

3. Iterate over the hyperplanes (H_i s) and calculate a_i and b_i for each H_i :

   The technique is mentioned clearly in Section 4 of the paper. Here, SymPy Matrices will suffice for the task.

4. Decompose the input polynomial into $p+1$ homogeneous polynomials :

   This can be done by creating an empty list of size $p+1$. Then scan the given polynomial term by term, find out the total degree of each term (i.e sum of exponents of variables) and store in the correct cell corresponding to that degree.

5. Compute the integral using the **Integration Reduction Algorithm** :

```
result = 0
for j in range(0, p + 1):
        h = 0
        for i in range(1, m+1):
                g = integration_reduction(F[i], a, b, d - 1, g[j])
                h = h + g * (b[i]/norm(a[i]))
        h = h/(d + j)
        result = result + h
return result
```

Integration Reduction Algorithm

With the use of Generalised Stokes Theorem, integration over the polytope was translated to integrating over the boundary facets. Now, we can extend this further and translate it

to integrating over the edges made by the facets.

```
def integration_reduction(F_k, a, b, vars, d, f):
        #vars → This is the list of variables of which constitute the polytope.
        g = 0
        df = [0 for i in range(0,d)]
        x0 = b[k] * (a[k].transpose()).inverse()
        for i in range(1, d+1):
                df[i - 1] = diff(f, vars[i])
                if df[i - 1] != 0:
                        g = g + integration_reduction(F_k, a, b, vars, d, x0[i - 1] * df[i-1])
        for j in range(1, m+1):
                F_kj = intersection(F_k, F_j)
                if not F_kj:
                        d_kj = boundary_edge_length(F_k, F_j)
                        if is_vertex(F_kj):
                                g = g + d_kj * f.subs({x, v_kj})
                        else:
                                g = g + integration_reduction(F_kj, a, b, vars, d-1,
                                                              d_kj*f.subs({x : v_kj}))
            return g
```

`boundary_edge_length`, `intersection`, `is_vertex` are the helper functions.

Convex Hull Algorithm

There are many efficient algorithms currently for finding out the convex hull for a given set of points. I will need to research and find out the best one suited to this task.

After the convex hull is found, all we need to do is to iterate through the list of points that belong to the convex hull and construct the boundary equations between consecutive points in the list.

For 2D polytopes, we need to consider every two consecutive points. The boundary equations will be lines for 2D polytopes.

For 3D polytopes, a facet must consist of minimum three points. Therefore, we take the

first three points in the list, construct a plane equation and check to see if the next point lies on the plane or not. If it does add it to the current facet, or else save the current facet and put the new point as the beginning of a new facet.

Note : It is crucial that the points be traversed in a counterclockwise fashion for 3D polytopes (per facet of the polytope) and clockwise for 2D polytopes or this mentioned algorithm will fail.

Timeline :

Pre-GSoC period (April 3 - April 25):

Since this idea was posted quite late [2], I did not have the time to properly understand all the mathematical details in [1]. I will take this time to properly study the paper and understand all the background mathematical details and the techniques to implement the cubature rules for integration on polar curves and using polar functions. I understand it's not very much different from the cartesian technique. I will keep updating this Google Doc and add one markdown file as well to the SymPy wiki.

Phase I : Implementing the proposed integration module

Community Bonding Period(May 4 - May 30):

May 3 - May 18 :

Final Term exams. Not available during this period for any intensive discussion. I am well aware of the SymPy community rules as I have contributing for about 8 months. Also, I will have made up for this lost time earlier in the pre-GSoC period by thoroughly studying the paper mentioned and trying out some python code snippets.

May 18 - May 30 :

Discuss implementation details with mentor, and begin coding.

Week 1 (May 31 - June 5) :

Start off by defining a class and it's class methods called `polytope_integrate`. Structure will be very similar to that of the already well-built `integrals` class in SymPy. **(Pull Request #1)**

Week 2 (June 6 - June 12) :

Begin writing and testing the helper functions such as `boundary_edge_length`, `intersection`, `is_vertex`. They should work for both 2D and 3D polytope test data. **(Pull Request #2)**

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

Start and finish the Convex Hull Algorithm. I would need to discuss with my mentor which particular one to use as there are many such algorithms available currently. Finish implementation of convex hull for both 2D and 3D sets of points. **(Pull Request #3)**

Two weeks are allocated because I would need to make sure that the algorithm is fast and robust.

Week 5 (June 27 - July 3) :

Start and finish coding the Integration Reduction Algorithm. As usual , write unit tests for it. **(Pull Request #4)**

Week 6, 7, 8 (July 4 - July 24) :

Begin and finish coding the main integration algorithm. It is common in software development for issues to crop up when integration of many functions are done. Hence three weeks is allotted for this task. At the end Week 8 we will have a fully functional module to integrate a homogeneous function over a polytope with everything being expressed in Cartesian form. **(Pull Request #5)**

Phase II : Extending the functionality of the module

Week 9, 10 (July 25 - August 7) :

Add polar form support to the integration algorithm. **(Pull Request #6)**

Week 11, 12 (August 8 - August 21) :

Add common polytope classes for instant use by the user. Look at **Vertices of a Polytope (In Cartesian Coordinates)** under heading **Design** for clarification. **(Pull Request #7)**

Week 13 (August 22 - August 29) :

Write up any missing documentation and add extra unit tests if necessary. **(Pull Request #8)**

Notes :

I like to follow the paradigm of test driven development and write extensive documentation (docstrings and comments) as I code. Many modules in SymPy are poorly documented which makes it difficult to understand that section of code.
My third year starts on **July 30** but there are no major exams for the first one and a half months. Also , I have designed my timeline such that almost all the major implementations are done in **Phase I.**

How do I fit in ?

I have been studying SymPy code and contributing for about 8 months now, therefore understand how SymPy code is structured and written. Also, being a Mathematics student, I have taken courses such as Measure & Integration, Real Analysis, Complex Analysis, etc and hence will not shy away from the theoretical knowledge that might be required for completing this project.
I do have the programming 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 extend the capabilities of the proposed integration module.

Post GSoC

One of the first tasks will be to port this code into symengine. I am not so conversant in C/C++ right now, therefore did not add this task in my timeline.

Later on, I wish to improve the Statistics module of SymPy as it has a lot of limitations.

References :

1. Numerical integration of homogeneous functions on convex and nonconvex polygons and polyhedra - Eric B.Chen, Jean B.Lasserre, N.Sukumar
2. GSoC project idea : Integration of homogeneous functions over polytopes