# GSoC 2021 Application - Mechanics - Implement JointsMethod

Sudeep Sidhu

# About Me

## Basic Information

**Name** - Sudeep Sidhu
**University** - Maharaja Surajmal Institute of Technology, New Delhi
**Email** - sudeepmanilsidhu@gmail.com
**Github profile** - https://github.com/sidhu1012
**Time zone** - IST (UTC +5:30)

## Personal Background

I'm Sudeep Sidhu, a pre - final year student at Maharaja Surajmal Institute of Technology, New Delhi, pursuing a Bachelor of Technology (4 years) in Information Technology.

I spend my free time playing cricket with my friends, spend time with family , play online multiplayer games or hangout with cousins. Physics has been my favourite subject apart from programming and I love to solve dynamics problems since I took Science in higher secondary school and willing to learn more during the project.

## Programming Background

I use Ubuntu 20.04 as my operating system on WSL 2, Windows 10 along with VS Code being my primary source code editor. I am familiar with the `git` and `Github` workflow.

I started programming almost 5 years ago when programming was introduced to me in my higher secondary school. I really enjoyed coding and was amazed by concepts taught with enthusiasm by the teacher.I continued to expand my learning and right now, I feel comfortable writing code in C, C++ and Python.

My curiosity to further learn computer science has increased by taking MOOCs online, with Coursera being my favorite platform. From the past two years, I have continuously worked upon improving my knowledge and skills by proactively learning through several courses online.

Few of the specialization I had the chance to take are IBM AI Engineer; DeepLearning.AI Tensorflow Developer; and few courses like Data for Machine Learning; Python 3: Project-based Python, Algorithms,Data Structures ; Learning Python for Data Analysis and Visualization; Introduction to Git and Github.

I like python because of its English like syntax and extensive inbuilt functions which makes things easier and coding fast.

One of my favorite features of sympy is `pprint.`

```
>>> from sympy.abc import x, y
>>> from sympy import pprint, sqrt
>>> expr = (x**3 - 8*x*y + 2*y**2)/sqrt(27)


>>> pprint(expr)
```

```
   ⎛3                 2⎞
√3⋅⎝x  - 8⋅x⋅y + 2⋅y ⎠
──────────────────────
          9
```

## Contributions

I started using SymPy in August 2020 and made my first contribution to the main repository the same month. I have been continuously contributing to the software since then. I'm a long-term-contributor and will continue to improve this software even after this program is finished.

**Merged PRs**

- #20864 - Add test cases for subs in vector and dyadic. **Fixes** #20437.
- #20741 - Make Matrix expressions simplification possible. **Fixes** #19544.
- #20676 - Make array differentiation work with non sympy expressions. **Fixes** #20655.
- #20666 - Make FiniteSet not contain equal elements. Fixes #20432.
- #20451 - Increase precision of FiniteSet.evalf() . **Fixes** #20379
- #20446 - Added .xreplace() to Vector and Dyadic. **Fixes** #20445.
- #20438 - Make `is-subset` work for ProductSet & FiniteSet and Eq(sets).simplify() work. **Fixes** #19378.
- #20396 - Added .evalf() to Vector and Dyadic. **Fixes** #18064.
- #20353 - Added test case for Gram Schimdt. **Fixes** #9488.
- #20318 - Break ReferenceFrame.orient() into separate methods. **Fixes** #17758.
- #20228 - Fix performance. **Fixes** #20225.
- #20223 - Return NotImplemented instead of TypeError.**Fixes** #20222.
- #20212 - zero raised to power -infinity gives zoo. **Fixes** #19572.

- #20208 - Fix set is equal to set only. **Fixes** #20089.
- #20197 -Drop python2 from entire sympy. **Fixes** #18816.
- #20184 - Drop python2 from vector.
- #20183 - drop py2 from physics.
- #20182 - Drop py2 from polys.
- #20181 - Drop py2 from stats.
- #20175 - Comparing Matrix with object.**Fixes** #19361.
- #20159 - Not implemented for matrix and tensor addition.**Fixes** #18956.
- #20131 - Add warning for all possible paths and cyclic path. **Fixes** #20129.
- #20049 - point.vel() calculates velocity. **Fixes** #17761.
- #20023 - remove python2 from plotting using pyup_dirs.
- #20010 - Dropping unnecessary python2 imports. **Fixes** #18816.
- #20009 - Added missing print parenthesis. **Fixes** #19434.

**Unmerged PRs**

- #21271 - Improve frame caching. **Fixes** #20955 and #21036
- #21137 - Make dynamicsymbols real by default.
- #21027 - Make `dynamicsymbols._t` public and `dynamicsymbol` independent of `.t` . **Fixes** #19434.
- #20830 - Introduce nprime() to generate kth to nth primes. Fixes #19118.
- #20691 - Make inversion of Matrix with MatrixSymbol as element possible. **Fixes** #19162.
- #20493 - [WIP] Add a new helper method for evalf. **Fixes** #20479.
- #20481 - Make NonElementary Integrals work with evalf(). **Fixes** #20133.
- #20200 - Type check for _sympify. **Fixes** #20126.
- #20169 - lens makers formula can calculate focal length for thick lens and plano lens. **Fixes** #20168.
- #20157 - RTM of thick lens.
- #20054 - Gravitational force function for particle.

**Issues raised**

- #20224 -[Questions] Shouldn't Sympy array support broadcasting as numpy array does.
- #20168 - Lens maker formula.
- #20149 - Gaussian optics.
- #20028 - Gravity.

Apart from these, I was also involved in discussions or reviewing some of the PRs and issues namely:

- ***Code Review***
  #21192 - Added documentation for chop function.
  #21199 - Fix Todo.
  #21183 - .evalf() doesn't works properly with evaluate=False.
  #20611 - Fix for limit bug.
  #20785 - Triangle().area for zero area triangle results in attribute error.
  #20813 - Adding inflection point.
  #21092 - Optics: Added one file for implementing functions related to interference.
- ***Discussions*** - #20493, #17714, #20460

# The Project

## Overview and Motivation

The main aim of this project is to implement Joints Method in Sympy under mechanics in physics module as it would open up the use of the library to a much wider set of users because they will be able to construct dynamic models with less knowledge of underlying mathematics.

So exactly how is Joints Method going to help?  Currently to solve kinematical differential equations, users need to define those equations and solve them using Kane's or Lagrange's method. With the help of JointsMethod, users would just need to define the system with joints and body and JointsMethod would form and solve kinematic differential equations all by itself , without the user doing any mathematics.

Some contributors made efforts to add JointsMethod to `sympy.physics.mechanics` in #9809 and #9835. But those were not complete and polished enough to get into the sympy codebase. I'm familiar with the required theory behind this and willing to learn more and improve throughout the project. Throughout this project, two main classes would be implemented, namely: `Joints` and `JointsMethod`.

`Joints` will use physics.mechanics.body, so it would be able to support most of the joints(Pin Joint, Sliding Joint etc). SymPy already has a very rich class KanesMethod that would back our JointsMethod. Joints would be an abstract class which would be inherited by other joints classes and could also be used to define custom joints. JointsMethod would take ground and joints as input and would give solved differential kinematical equations.
I would also love to maintain a blog during this program so that it is easier for others to further expand this project.

## Implementation Plan

I have planned to divide my work into three phases so that the method can be added swiftly and systematically.

- **Phase 1** := Adding `Joints` and `various types of joints` classes. Methods that are already implemented in #9809 will be polished and tests and docstrings would be added for merging to ensure everything is implemented correctly.
- **Phase 2** := In this phase some working example would be added as unit tests for older proposed API to find loopholes in that design and update the API accordingly**.**
- **Phase 3** := Update JointsMethod class. Methods that are already implemented in #9835 will be polished , tests and docstrings would be added for merging to ensure everything is implemented correctly.

## Phase 1

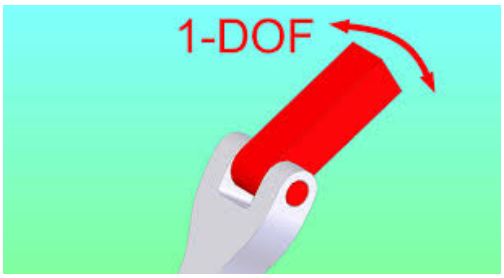
Currently to form a kinematical differential equation for multibody dynamics, all mathematics has to be done by hand using Sympy which makes generation of these equations tough or complicated.

A **mechanical joint** is a section of a machine which is used to connect one or more mechanical part to another. Most mechanical joints are designed to allow relative movement of these mechanical parts of the machine in one degree of freedom, and restrict movement in one or more others.

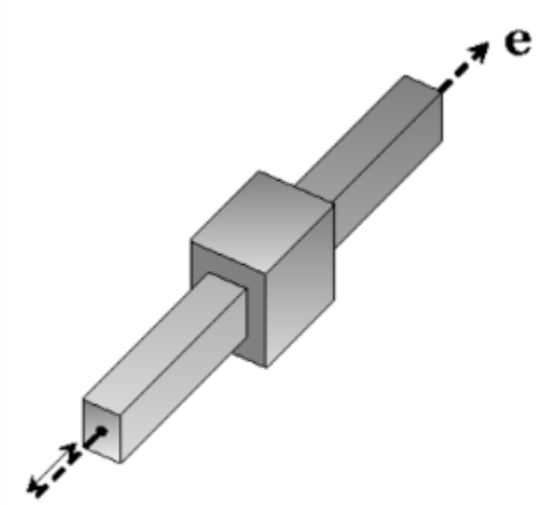
We have class Body which is a common representation of either RigidBody or a Particle in Sympy and class Vector which would serve as building blocks of class Joints and all subclasses of Joints.

The following types of Joints will be enhanced and finished , which would inherit Joints class.

1. **Pin Joint :** A pin joint, also called a revolute joint, is a one-degree-of-freedom kinematic pair. It constrains the motion of two bodies to pure rotation along a common axis. The joint doesn't allow translation, or sliding linear motion. It enforces a cylindrical contact area, which makes it a lower kinematic pair, also called a full joint.




2. **Prismatic Joint :** A prismatic joint provides a linear sliding movement between two bodies, and is often called a slider. The relative position of two bodies connected by a prismatic joint is defined by the amount of linear slide of one relative to the other one. This one parameter movement identifies this joint as a one degree of freedom kinematic pair.



The translational motion of the source and destination attachments at the center of the joint can be computed as:

$$\mathbf{u}_{c,\,src} = (\mathbf{R}_{src} - \mathbf{I}) \cdot (\mathbf{X}_c - \mathbf{X}_{c,\,src}) + \mathbf{u}_{src}$$
$$\mathbf{u}_{c,\,dst} = (\mathbf{R}_{dst} - \mathbf{I}) \cdot (\mathbf{X}_c - \mathbf{X}_{c,\,dst}) + \mathbf{u}_{dst}$$

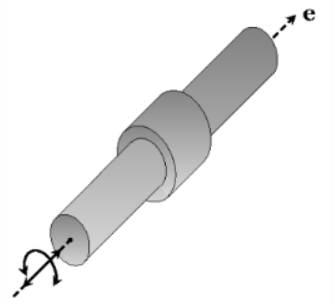
where, for the source and destination attachments, respectively:

- $\mathbf{u}_{c,src}$ and $\mathbf{u}_{c,dst}$ are the displacement vectors for the attachments at the center of joint.
- $\mathbf{R}_{src}$ and $\mathbf{R}_{dst}$ are the rotation matrices describing the rotation of each attachment.
- $\mathbf{X}_{c,src}$ and $\mathbf{X}_{c,dst}$ are the positions of the centroids of the attachments.
- $\mathbf{u}_{src}$ and $\mathbf{u}_{dst}$ are the displacements at the centroids of the attachments.
- $\mathbf{X}_c$ is the joint center.

To formulate this kind of connection for a rigid joint, the motion of the destination attachment is prescribed in terms of the motion of the source attachment:

$$\mathbf{u}_{c,\,dst} = \mathbf{u}_{c,\,src} + \mathbf{u}_r$$
$$\mathbf{b}_{dst} = \mathbf{b}_{src}$$

3. **Cylindrical Joint:** The Cylindrical Joint has one translational and one rotational degree of freedom between the two connected components. The components are free to slide and rotate relative to each other along the axis of joint. A cylindrical joint can be thought of as a combination of a prismatic joint and a hinge joint.

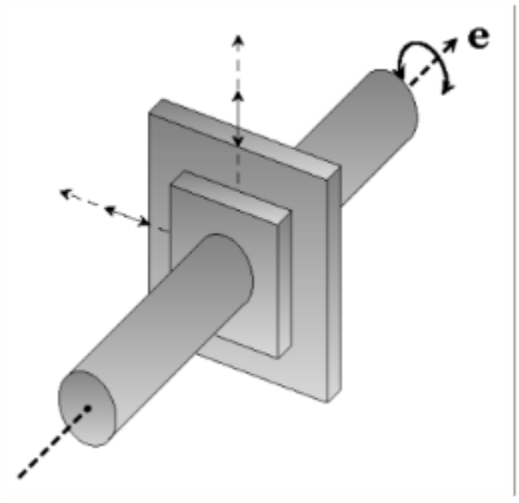



To formulate this kind of connection, the motion of the destination attachment is prescribed in terms of the motion of the source attachment as:

$$\mathbf{b}_{\mathrm{dst}} = \tilde{\mathbf{b}}$$

$$\mathbf{u}_{\mathrm{c,\,dst}} = \mathbf{u}_{\mathrm{c,\,src}} + \mathbf{u}_{\mathrm{r}}$$

4. **Planar Joint:** A joint that allows only translation over a plane and rotation about an axis normal to this plane. This type of joint is produced by a stable object resting on a flat surface. It has three degrees of freedom.



The degrees of freedom are two relative displacements along the second and third axes of the joint ($u_2$, $u_3$) and one relative rotation about the joint axis (θ).
The local coordinate system of a joint, which can be seen as rigidly connected to the source attachment, consists of three axes: joint axis ($\mathbf{e}_1$), second axis ($\mathbf{e}_2$), and third axis ($\mathbf{e}_3$)
The joint formulation is similar to the cylindrical joint; the only difference is in the definition of the relative displacement vector ($\mathbf{u}_r$):

$$\mathbf{u}_r = u_2\mathbf{e}_2 + u_3\mathbf{e}_3$$

Degree of Freedom: The number of independent parameters that define its configuration or state.

As proposed in in PR #9809 will be polished and enhanced:

**Class Joints** : Will work as an abstract class and all types of joints classes will inherit from this class. This class could also be inherited to form custom joints.

- __init__()  would be enhanced and finished and proper docstring will be added.
- _locate_joint_point() which will be renamed to a better and descriptive name(like  _locate_joint_pointofcontact or _locate_joint_pos) . This function would be enhanced and finished with proper docstring.
- _align_axis() would be enhanced and finished with proper docstring.
- apply_joint() would be enhanced and finished with proper docstring.This function would give NotImplementedError and shall be defined by overriding in sub-classes.

**Proposed API**:

```
Class Joint:

    def __init__(self, name, parent, child, parent,
parent_point_pos=None, child_point_pos=None):

        pass


    def _locate_joint_pos(self, parent_point_pos,
child_point_pos):
        pass

    def _align_axis(self, parent_axis, child_axis):
        pass

    def apply_joint():
        return NotImplementedError
```

**Class Pin Joint :** The class Pin Joint would inherit class Joints.

- __init__() would be enhanced and finished and proper docstring will be added.
- apply_joint() definition of kinematics would be enhanced and finished.

**Proposed API**:

```
Class PinJoint:

    def __init__(self, name, parent, child, parent,
parent_point_pos=None, child_point_pos=None):
```

```
        pass

    def apply_joint(coordinate=None, speed=None):

        pass
```

**Example**:

```
>>> theta, omega = dynamicsymbols('theta omega')
>>> child = Body('child')
>>> parent = Body('parent')
>>> P = PinJoint('P', parent, child)
>>> P.apply_joint(coordinates=theta, speeds=omega)

>>> P.kde
Derivative(theta(t), t)- omega(t)
```

**Class Prismatic Joint :** The class Prismatic Joint would inherit class Joints.

- __init__() would be enhanced and finished and proper docstring will be added.
- apply_joint() definition of kinematics would be enhanced and finished.

**Proposed API**:

```
Class PrismaticJoint:

    def __init__(self, name, parent, child, parent,
parent_point_pos=None, child_point_pos=None):

        pass

    def apply_joint(coordinate=None, speed=None):

        pass
```

**Example**:

```
>>> x, v = dynamicsymbols('x v')
>>> child = Body('child')
>>> parent = Body('parent')
>>> P = PrismaticJoint('P', parent, child)
>>> P.apply_joint(coordinates=x, speeds=v)

>>> P.kde
Derivative(x(t), t) - v(t)
```

**Class Cylindrical Joint :** The class cylindrical Joint would inherit class Joints.

- __init__() would be enhanced and finished and proper docstring will be added.
- apply_joint() definition of kinematics would be enhanced and finished.

**Proposed API**:

```
Class CylindricalJoint:

    def __init__(self, name, parent, child, parent,
parent_point_pos=None, child_point_pos=None):

        pass

    def apply_joint(coordinate=None, speed=None):

        pass
```

**Example**:

```
>>> x, v, theta, omega = dynamicsymbols('x v theta omega')
>>> child = Body('child')
>>> parent = Body('parent')
>>> C = CylindricalJoint('C', parent, child)
>>> C.apply_joint(coordinates=[x, theta], speeds=[v,
omega])

>>> P.kde
[Derivative(x(t), t) - v(t), Derivative(theta(t), t) -
omega(t)]
```

**Class Planar Joint :** The class Planar Joint would inherit class Joints.

- __init__() would be enhanced and finished and proper docstring will be added.

- apply_joint() definition of kinematics would be enhanced and finished.

**Proposed API**:

```
Class PlanarJoint:

    def __init__(self, name, parent, child, parent,
parent_point_pos=None, child_point_pos=None):

        pass

    def apply_joint(coordinate=None, speed=None):

        pass
```

**Example**:

```
>>> theta, omega, x_x, v_x, x_y, v_y =
dynamicsymbols('theta omega x_x v_x x_y v_y')
>>> child = Body('child')
>>> parent = Body('parent')
>>> P = PlanarJoint('P', parent, child)
>>> P.apply_joint(coordinates=[theta, x_x, x_y],
speeds=[omega, v_x, v_y])

>>> P.kde
[Derivative(theta(t), t)-omega(t), Derivative(x_x(t), t)-
v_x(t), Derivative(x_y(t), t) - v_y(t)]
```

## Phase 2

In this phase some working example would be added as unit tests for older proposed API to find loopholes in that design and update the API accordingly**.**

Some possible examples are:-

- single pendulum test.
- double pendulum (use two different joints).
- both a tree structure and a serial chain.
- any of the kanesmethod and langrangesmethod tests that do not have constraints can be turned into tests for this class.
- any of the pydy examples that do not have constraints can be used for testing .

Once some decent unit tests are written, final API design for JointsMethod would be formed and the previous one would be updated accordingly.

## Phase 3

Once the Joints class and other Joints have been implemented, we can solve dynamics problem by passing joints to **JointsMethod**

JointsMethod would use Kane's method to solve those equations.

As proposed in in PR #9835 will be polished and enhanced:

- __init__() would be enhanced and finished and proper docstring will be added.
- _generate_bodylist() would be enhanced and finished and proper docstring will be added.
- _generate_forcelist() would be enhanced and finished and proper docstring will be added.

- _generate_q() would be enhanced and finished and proper docstring will be added.
- _generate_u() would be enhanced and finished and proper docstring will be added.
- _generate_kde() would be enhanced and finished and proper docstring will be added.
- _set_kanes() would be enhanced and finished and proper docstring will be added.

**Proposed API**:

```
>>> J = JointsMethod(ground, *joints)
```

**Example:**

```
#Simple double pendulum
>>> ground = Body('ground')
>>> P = Body('P')
>>> R = Body('R')
>>> q1, q2, u1, u2 = dynamicsymbols('q1 q2 u1 u2')
>>> P1 = PinJoint('P1', ground, P)
>>> P1.apply_joint(coordinates=q1, speed=u1)
>>> P2 = PinJoint('P2', P, R)
>>> P2.apply_joint(coordinates=q2, speed=u2)
>>> J = JointsMethod(ground, P1, P2)
```

# Timeline

I have prepared a tentative timeline for the above mentioned tasks. I assure that I'll give all my time to this project and try to finish what I have proposed.

*Community Bonding Period (17th May - 6th June)*

- In this period, I will discuss the project with my mentor so that we come up with an efficient way of implementation. Communicate with my mentors and other selected students about their projects and how we can be of help to each other. **I will start coding in the last week of the community bonding period** and would try to implement Joints abstract class so that I get a good head-start for Phase 1, which is the most important of all. I think one extra week will be beneficial for me in the long run and I will also have fun coding in this period.
- Also I'll be revising multibody dynamics, Kane's method and study joints dynamics.

*Week 1 (7th June - 13th June)* - Phase 1

- Learn the implementation of Pin(revolute) joints.
- Polish and finish Pin Joint class.
- Write docstring and add test cases.

*Week 2 (14th June - 20th June)* - Phase 1

- Learn the implementation of Sliding Joint.
- Polish and finish Sliding Joint class.
- Write docstring and add test cases.

*Week 3 (21st June - 27th June)* - Phase 1

- Learn the implementation of Cylindrical Joint.
- Polish and finish Cylindrical joint class.
- Write docstring and add test cases.

*Week 4 (28th June - 4th July)* - Phase 1

- Learn the implementation of Planar Joint.
- Polish and finish Planar Joint class.
- Write docstring and add test cases.

*Week 5 (5th July - 11th July)* - Phase 2

- Learn the implementation of JointsMethod.
- Write unit tests for previously proposed API

*Week 6(12th July - 18th July)* - Phase 2

- Conclude unit tests
- Finalize API design.

*Week 7(19th July - 25th July)* - Phase 3

- These methods will be polished and finished in this week -- _generate_bodylist, _generate_forcelist, _generate_q
- Write and update docstring.

*Week 8(26th July - 1st August)* - Phase 3

- These methods will be polished and finished in this week -- _generate_u, _generate_kde, _set_kanes.
- Write and update docstring.

*Week 9(2nd August - 8th August)* - Phase 3

- Add a working example for JointsMethod.

*Week 10(9th August - 16th August)* - Phase 3

- Fix bugs if any.
- Finish any remaining work

### Post GSoC

After GSoC I'll love to contribute more and be an active member of the Sympy community . I would even like to become a mentor for next year's GSoC.

# Notes

I've got my end semester exams from in July end but I would still be able to devote 20 to 25 hours per week throughout the GSoC period. I would be able to give all my time and energy to finish the project.

# References

[1] Jason Moore
https://github.com/moorepants

[2] Sahil Shekhawat
https://github.com/sahilshekhawat

[3] Oliver Lee
https://github.com/oliverlee

[4] Some good previous year GSoC applications.

[5] Sahil's PR on Joints
https://github.com/sympy/sympy/pull/9809

[6] Sahil's PR on JointsMethod
https://github.com/sympy/sympy/pull/9835

[7] Wikipedia

[8] Joints at Comsol

[9] Pydy