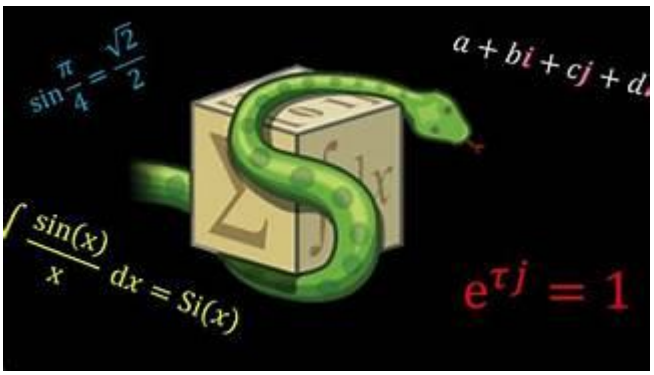
# SymPy




## Google Summer Of Code 2023 Proposal

## Improving and Expanding Functionalities of Control Module

**Anurag Bhat**
**4th April 2023**

# Personal Details

- **Name :** Anurag Bhat
- **Github Profile :** **faze-geek**
- **Email :** bhat.1@iitj.ac.in
- **University :** Indian Institute of Technology Jodhpur
- **Degree :** Bachelor of Technology
- **Major :** Computer Science and Engineering
- **Country of Residence :** India
- **Timezone :** Indian Standard Time (UTC + 5:30)
- **Primary Language :** English

# Introduction

I am **Anurag Bhat**, a third-year student of the Indian Institute Of Technology Jodhpur pursuing a B-Tech in computer science and engineering. IIT-Jodhpur has one of the most rigorous computer science course curriculums in India. My hobbies are playing chess, football and competitive coding. I have got a keen interest in web development over the past year and I enjoy designing web pages even more than creating them and their functionality.

Through the courses I have undertaken in these 3 years I have dealt with a lot of software and hardware. I have taken part in several clubs like the programming and robotics club of my college. I am a part of the student

well-being committee of my college and organize regular events for students, both related/unrelated to technology. I have good communication skills and I'm a fluent English speaker. I gel well with the tech community at IIT-J.

**Some of the relevant courses I have taken are -**

1. Introduction to Computer Science
2. Engineering Mechanics
3. Signals and Systems
4. Mathematics 1 - Calculus
5. Mathematics 2 - Linear algebra and Differential Equation
6. Mathematics For Computing
7. Data Structures and Algorithms
8. Pattern Recognition in Machine Learning
9. Software Engineering

# Programming Experience

My earliest experience of programming was in my 9th grade when I was introduced to JAVA and wrote codes to solve logical problems. In 10th grade, I had won a national level robotics event 'Gizmo-Geeks' in the journey in which I had learned how an Arduino is coded and used.

I have been programming for the past 3 years. My real interaction with programming started when I began the 'Introduction to Computer Science course in college which was a course based purely on python. After that, I started competitive coding to get a good grip on data structures and algorithms. I am now comfortable with coding in
Python, c, c++, HTML, CSS, javascript( for web development ) and Verilog( for hardware breadboard coding ).I have also been interested in physics and physical materials. I have undertaken a project using Data Analysis to predict the fatigue life of Aerospace ball bearings.

I use a Windows 10 operating system and use Visual Studio Code as my primary editor. It is open-source and easy to use. It also has an IntelliSense extension which makes it very user-friendly and allows me to write code very fluently.
I like python as a language because of its extensive pre-defined functions and various libraries which makes problem solving easier. It is also simple to understand because of its similarity to English itself.

I have been using git and GitHub regularly for various things for the past 1 year now. Now I am quite familiar with GitHub workflow and git commands.

Now coming to SymPy the feature which impressed me the most when I first came across it was **pprint()** or pretty print. It is something I never imagined would be present in SymPy and randomly came across it when I approached one of my initial issues. Look how easy it is to visualize things with and without pprint().

```
>>> M=Symbol('M',integer = true)
>>> e = -M/(sqrt(3)/2 - S(1)/2) + M + sqrt(3)*M
>>> e

-M/(-1/2 + sqrt(3)/2) + M + sqrt(3)*M

>>> pprint(e)

     M
- ──────── + M + √3⋅M
    1   √3
  - ─ + ──
    2   2
```

# Contributions To SymPy

I started using SymPy in October 2021 and made my first contribution to the main repository in December. 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.

### Pull Requests

**(Merged) #22640 -** Added Tests in Limits.py for issue which has been fixed in Master
**(Merged) #22647 -** Solves issue of ignoring evaluate=False condition
**(Merged) #22670 -** Modified the as_content_primitive method in core/power.py to deal with expression involving 0 as base
**(Merged) #22681 -** Appropriate Tests Added For residue() Function Which Has Been Fixed On Master
**(Merged) #22927 -** integrals - Added test for integrate() function which has been fixed on master
**(Merged) #23033 -** vector : Scalar condition of vectors made False
**(Merged) #22770 -** Corrected prior form leading to dead code in sympy.polys.polyroots.roots() function
**(Merged) #23074 -** physics : Improved Bode's phase and magnitude plots
**(Merged) #23199 -** calculus : Added condition to improve is-convex results
**(Merged) #22999 -** Fixes results of evalf() function based on precision
**(Merged) #22827 -** physics : completed list of non-SI units -accepted by SI system but missing in SymPy
**(Merged) #22969 -** physics : Added refractive index in Gaussian Beam Parameters and refactored code
**(Merged) #23141 -** integrals : Fixed polytope_integrate for max_degree inputs
**(Merged) #23296 -** physics units : Fixes wrong dimension calculation for functions

**(Unmerged)** **#22688** - Added Essential Information In .evaluate() Doc-string Regarding Use Of Declaring Real Symbols
**(Unmerged)** **#22712** - Fixed limit(x**n, x, oo) Earlier Giving -oo As Answer Irrespective Of Odd/Even n
**(Unmerged)** **#22746** - Moved core/mod.py routines to functions/elementary/integers.py
**(Unmerged)** **#22790** - Added a class method 'nearest_points()' to calculate distance between two lines in 3D.
**(Unmerged)** **#22809** - Made changes for better user-interpretation of series which are demonstrably divergent
**(Unmerged)** **#22869** - combinatorial: Added code for Padovan number
**(Unmerged)** **#22889** - functions : Improved floor and ceiling results
**(Unmerged)** **#23068** - physics : Added solve_vector function in physics vector

### Issues

**(Closed)** **#22819** - Non-SI units mentioned in the SI missing/incomplete in SymPy
**(Open)** **#22935** - Periodicity of trigonometric expression incorrect .
**(Open)** **#22986** - limit(acosh(1 + 1/x)*sqrt(x), x, oo) is evaluated incorrectly
**(Open)** **#23247** - physics : Non-polynomials accepted as num/den inputs for TransferFunction

*Other than these I have reviewed and brainstormed ideas on a few issues. I have tried helping newcomers on the Gitter channel so that they can contribute to SymPy.*

*Note: I believe that I will continue to contribute even during the proposal period so here is the **link** to my updated work.*

# The Project

I would like to describe the project from this point. I hope that I will be able to explain it appropriately and expect improvements in content as well as structure once SymPy mentors go through it. This is intended to be a 175 hours project.

# Overview

The **control** module was added in Gsoc projects to SymPy by ***Naman Gera*** in 2020 and improved by ***Akshansh Bhatt*** in 2021. Their work is truly commendable but SymPy is far from the powerful CST modules of **Wolfram-Alpha** or **MATLAB** .Till now a lot of basic functionality has been added which is common for every control system toolkit. An important guide regarding the value a control package could add to SymPy is this **discussion**.

The advantage this module has is that it can utilize SymPy's symbolic methods instead of numerical methods which can tend to derive slower results when large number systems are involved.

Currently this package has a good scope of improvement. Some Pull requests were made earlier without discussion, to add control systems with nice functionalities ( **#9916** and **#17866** ) whose ideas are still valuable. Some great entry points for work in this module are **Naman's comment on #19761** and **Akshansh's GSoC Report**. While going through the work proposed/done by Naman and Akshansh I found some chunks left whose completion could be the starting points of my work. It is also tempting to look for ideas into older documented open source repositories who develop control packages ,namely **python-control** and **harold**.

# Motivation

I feel that I am a good candidate for this project because I have the theoretical knowledge to complete this project .In my 1st year I have done the basics of

Control Systems, namely ' *Electrical Engineering Circuit Lab* ' and ' *Signals And Systems* '. I have done an intermediate level python project in the above course where the problem statement was to design two systems from scratch - ***One that first performs deblurring and then denoising VS. One that first performs denoising and then deblurring***. We had to report and compare the performance of both systems for the same signals.
Link-
https://drive.google.com/file/d/1iZDy8O7V5Hi703P7BPFScx0bAFgrM6na/view?usp=sharing
I have spent a decent amount of time contributing to SymPy, going through online documentation and will be thoroughly understanding the code base relevant to this project .

# Progress In Past Years

Here are some major goals accomplished by Naman Gera and Akshansh Bhatt in the last two years.

1. Adding **TransferFunction**, **Series**, **Parallel**, and **Feedback** classes, with basic functionality, unit tests, and proper documentation.
2. Adding **TransferFunctionMatrix** class, with basic functionality, unit tests, and proper documentation.
3. Adding **pole_zero** & **bode_plot** using SymPy's native plotting module along with proper documentation.

A few things which were proposed but not implemented in the last two years are -

1. Adding a **StateSpace** class, with basic functionality, unit tests, and proper documentation.
2. Adding **root_locus** & **nyquist_plot** using SymPy's plotting module.
3. Adding extensive **examples**/**illustrations** in documentation from MATLAB or certain college course textbooks.

I believe that the work done in the last 2 years is great and deserves applause, but it is incomplete and has some obvious holes. I would like to complete all the

tasks I mention below so that SymPy has a well-furnished control module by the end of this summer.

## Major Goals -

These are the tasks I have planned throughout the GSoC period. Their implementations will further be equally divided into 4 phases. The priority and scope of these goals can be flexible and changed by discussion with mentors in the future.

1. Completion of work remaining from the past Gsoc period is what I consider to be the goal of highest priority. I would not like to start new work, while there is already pending work to be done .So PR **#22124** will be completed with more examples from documentation of *MATLAB* and *python-control*. I also have a course textbook for my control-systems course and can always refer to that for illustrations. Another pending thing I could observe which Akshansh had attempted but could not get merged due to some performance issues is the **root_locus_plot**. The comments of his PR **#21763** also suggests the addition of **nyquist_plot** and **nichols_plot** along with it .

2. The next goal I have in mind is to fix bugs , make improvements in '**TransferFunction**' / '**TransferFunctionMatrix**' API and add more functionality to the 'TransferFunction' class. These are the issues and potential improvements I discovered while going through the code and online documentation. There are a bunch of features I plan to add ,which I got to know of while comparing SymPy with the CST package of MATLAB. All these topics will be discussed in detail with the mentors before I work on them.

3. Introducing the '**StateSpace**' model for effectively representing a State Space system symbolically. It will be important to add all the relevant attributes and methods. Polishing documentation and code of **#17866** would help the progress of this particular task. Implementing a

**Discrete-time** ‘TransferFunction’ model. Discussing the API and making things compatible with the current implementation is a challenging task as suggested in Akshansh’s Gsoc 2021 report. It has already been a component of the MATLAB CST package from the beginning. As a control module, we have to realize that all signals in practical real life use are always discrete in nature. This is my motivation for adding the Discrete-time ‘TransferFunction’ model to SymPy so that users can have extensive use of SymPy’s CST package in their projects.

# Phases

A brief look at tasks I would aim to complete in each of the 4 phases -

**Phase 1 :**
In this phase, I will focus on my first goal. Completion of previous work, regarding the addition of more illustrations and plots.

**Phase 2 :**
Improvements in the current API, along with some important functionalities being added to ‘TransferFunction’.I expect that my second goal gets completed in this phase.

**Phase 3 :**
A chunk of my third goal should be covered in this phase. I will be starting the new implementations I intend to add to SymPy’s control package, namely the StateSpace system. Along with this concrete discussions for adding the Discrete-time TransferFunction model will be done with mentors.

**Phase 4 :**
This phase is for the completion of the third goal. A basic discrete-time model should be implemented.

# Phases In Detail

Ideas and work of each phase will be explained in detail below.
These are the ideas that I have accumulated by observing work done in previous years and checking out other CST frameworks. Implementation of all ideas would take extensive discussions with mentors along with a bunch of new unit tests to check their functioning.

## Phase 1 -

Phase 1 will be primarily focused on my 1st goal .I will start it with completion of **#22124** (Add examples in control module docs) . A few examples are already present and I will add more of them . For the online docs , I'll refer to examples of MATLAB and Python-control . The course textbook of my university has tons of solved examples which can be added after discussion.

**Plotting** - Visualization is a very important aspect in control systems. SymPy has a plotting module which can be used for the backend.I will be adding some popular plots to SymPy's control plots . These are namely the **root_locus_plot, nyquist plot** and **the nichols plot** . SymPy's plotting module allows dynamic positioning of coordinates ( move the cursor to get values of points on the plot) which will be an added advantage .

### Root Locus Plot -

This is a plot which gives information about the stability of a transfer function. It is basically an extension of the pole zero plot .The difference is that , in the root locus plot the poles and zeros are connected by various curves ( even asymptotes meeting particular lines at infinity ) depending on

the relative stability. Examples of construction of the plot are present in **this document** ( taken from Akshansh Bhatt's proposal ).
This plot depends on -

1. The poles and zeros
2. The number of poles and zeros
3. Centroid of poles and zeros
4. The angle of asymptotes
5. Breaking points

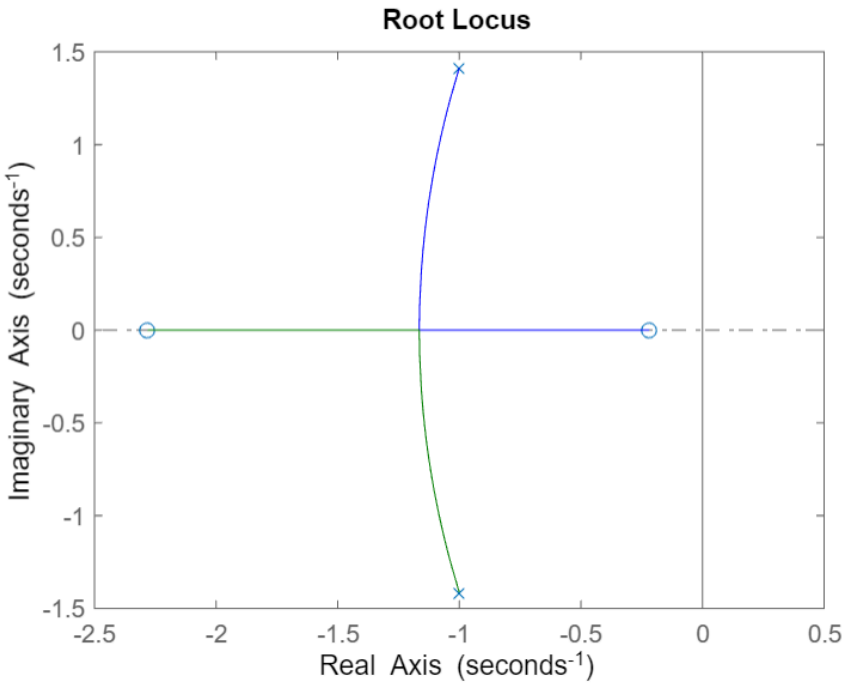
An example of the root locus plot

MATLAB -

```
sys = tf([2 5 1],[1 2 3]);
rlocus(sys)
```

SymPy -

```
>>> tf = TransferFunction(2*s**2 + 5*s + 1,s**2 + 2*s + 3,s)
>>> root_locus_plot(tf)
```



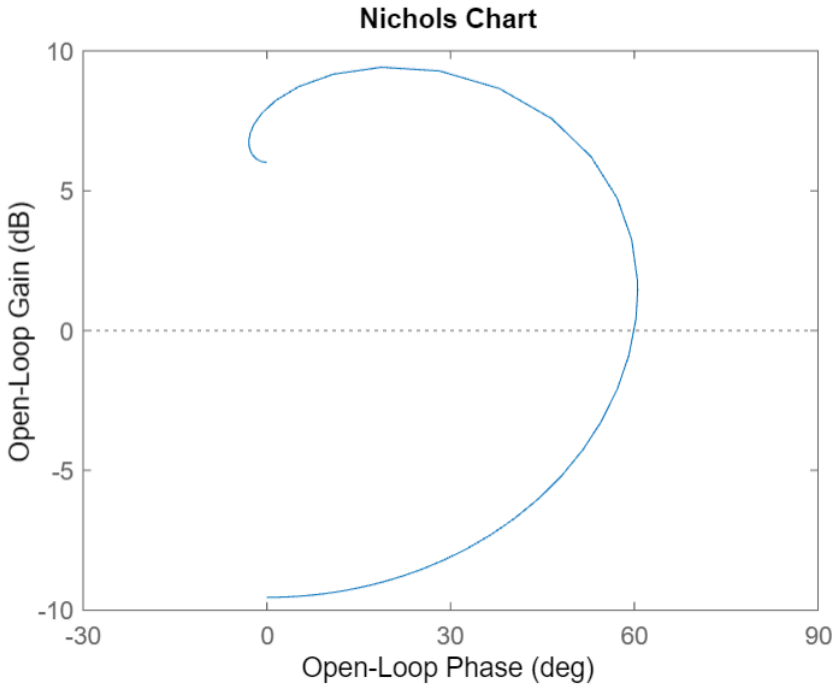

SymPy uses matplotlib as a backend which would need some modification to construct these plots .
There has been some work done on this plot by Akshansh but it needs effective sampling .This **comment** on PR **#21763** explains the current state of sampling and this **suggestion** by **@oscarbenjamin** needs to be addressed .

## Nichols Chart -

Nichols chart is a plot of the frequency response of a dynamic system model. The plot displays the magnitude (in dB) and phase (in degrees) of the system response as a function of frequency.
Here the frequencies in consideration will be decided by system dynamics itself. There can be input arguments for *min_freq* and *max_freq* when the user wants to focus on a range of frequencies( also an option in **MATLAB** and **python control** ).

An example of the nichols chart -

```
>>> tf = TransferFunction(2*s**2 + 5*s + 1,s**2 + 2*s + 3,s)
>>> nichols_plot(tf)
```




## Nyquist Plot -

The Nyquist plot is a graph of the frequency response of a dynamic system model. The plot displays real and imaginary parts of the system response as a function of frequency. It has a contour composed of both positive and negative frequencies. The plot also shows arrows to indicate the direction of increasing frequency for each branch.

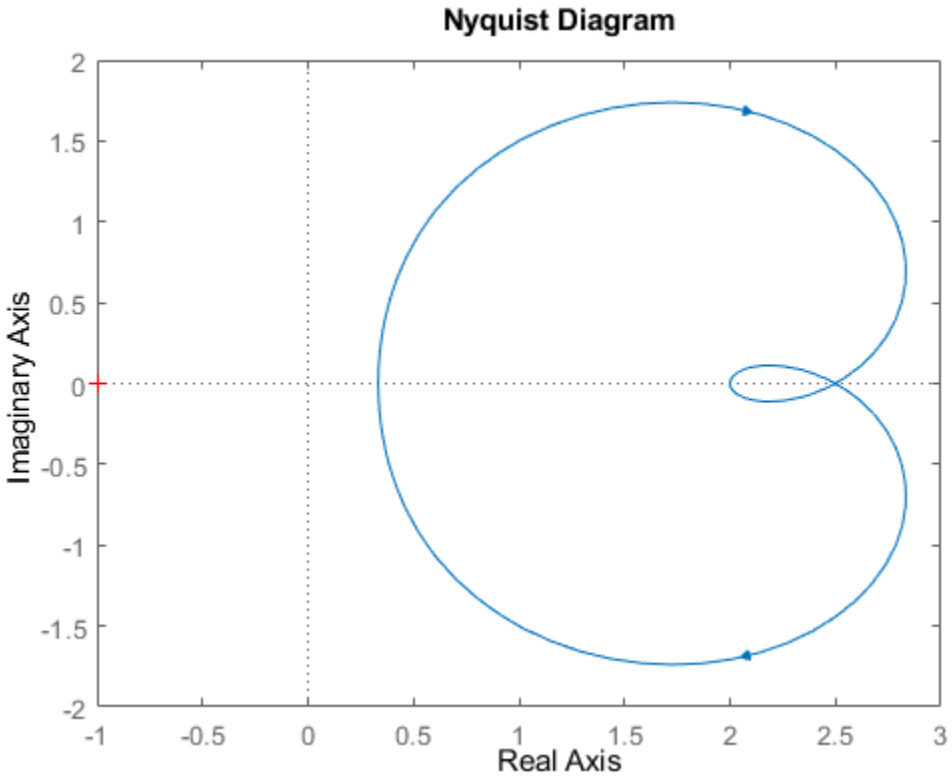
Here the frequencies in consideration will be decided by system dynamics itself .There can be input arguments for *min_freq* and *max_freq* when the user wants to focus on a range of frequencies( also an option in **MATLAB** and **python control** ). A helper function will also be needed to draw arrows in particular directions. A full video tutorial of rules for plotting and deriving

the **Nyquist Stability Criterion** from this **playlist** . **Nyquist stability criterion** is a set of rules to determine stability of a dynamical system which go hand in hand with the plot and can be analyzed with ease.

For implementation I will refer to **python-control** 's nyquist plot. I will be

An example of the nyquist plot -

```
>>> tf = TransferFunction(2*s**2 + 5*s + 1,s**2 + 2*s + 3,s)
>>> nyquist_plot(tf)
```




## Phase 2 -

This phase is relevant to the work I have suggested in my second goal. Some bugs need to be fixed and along with that minor changes can be made to the transfer function and transfer function matrix API. Some of these were pointed out in the previous year's proposal but I believe are not fixed yet. Besides these improvements, I will be adding some handy functionalities to the transfer function model which are a part of MATLAB. I have already started comparing SymPy documentation with MATLAB documentation for the control module and I have observed a bunch of useful functions that will benefit SymPy.

1. I have spotted a bug and raised an issue **#23247** .Transfer function should not be able to take non-polynomial expressions as input (eg - sin(x) , 2^(x)). The fix is simple and this deserves an error message as it does in MATLAB -

```
>>> from sympy.physics.control import TransferFunction
>>> x = Symbol('x')
>>> t = TransferFunction(x**2,sin(x),x)

#Clearly not a polynomial in denom
```

I have already come up with a fix in a branch locally -

```
+        if (num.is_polynomial(var) is not True or
den.is_polynomial(var) is not True):

+     raise TypeError("Numerator and Denominator of
TransferFunction must be a polynomial")
```

But this fix cannot be made instantly .I have observed that previous contributors to **test_lti.py** have not considered this as

a mistake and added examples with non polynomial terms (about 5-6) for eg -

```
tf14 = TransferFunction(a0*s**0.5 + a2*s**0.6 - a1, a1*p**(-8.7), s)
```

where s is raised to .5,hence the numerator is not a polynomial.

Therefore I would have to change those examples which will not be a tough task.

Another direct bug I have spotted is that transfer functions are not treated like basic rational functions in terms of division unlike in MATLAB and python-control.

**MATLAB**

```
>>tf1 = tf([2],[1,1])
>>tf2 = tf([1],[1,1])
>>tf2/tf1

ans =

  s + 1
 -------
 2 s + 2
```

**SymPy**

```
>>> tf1 = TransferFunction(2,s + 1, s)
>>> tf2 = TransferFunction(1,s + 1, s)
>>> tf1/tf2
ValueError: TransferFunction cannot be divided by <class 'sympy.physics.control.lti.TransferFunction'>.
```

This change can be made easily with modifications in the `__truediv__( )` private function.

2. To improve the API of 'TransferFunctionMatrix' Akshansh had modified **lti.py** , he intended to make **ImmutableDenseMatrix** from **sympy.matrices** module as the base class . ImmutableDenseMatrix supports **.subs()** which is a nice functionality for transfer function matrix to have -

```
>>> M = ImmutableDenseMatrix([(s**2 + 2*s + 1)/(s +
1),(s**2 - 9)/(s + 3)])
>>> pprint(M.subs({s:2}))

⎡3 ⎤
⎢  ⎥
⎣-1⎦
```

But this is not possible with 'TransferFunction' / 'TransferFunctionMatrix' as of now .

SymPy being a symbolic library ,**subs( )** and **rewrite( )** are useful properties. 'TransferFunction' is basically a rational function, so it will be a good addition if users can access these methods. 'TransferFunction' inherits 'Basic', so this can be done. 'TransferFunctionMatrix' also as mentioned above lacks these properties.

Current -

```
>>> tf = TransferFunction(s**2 + 2*s + 1,s + 1,s)
>>> pprint(tf.subs({s:2}))
 2
s  + 2·s + 1
─────────────
   s + 1
```

Expected Improvement -

```
>>> pprint(tf.subs({s:2}))
3
```

3. Restructuring some parts of the code , I feel some functions are unutilized .Minor functionalities for 'TransferFunction' that are a part of other CST frameworks like MATLAB can be made a part of SymPy too.As mentioned above I will be comparing MATLAB documentation with SymPy documentation and note if they would be useful for SymPy users.

   Examples -

   - **bandwidth(system)** - First frequency where gain drops by 3dB of DC value. It provides relevant information in linear analysis of the function.
     **bandwidth(system, drop)** - To allow variable drop below 3db.

     Expected -

     ```
     >> sys = tf(s, s + 1, s);
     >> bandwidth(sys)

     0.9976
     #This result shows that the gain of sys drops to 3 dB below its DC value at around 1 rad/s.
     ```

   - **margin(system)** - Gives gain-margin and phase-margin which are used to visualize marginal stability that SymPy's '**is_stable()**' gives no information about.

     **Gain margin** : The greater the **Gain Margin** (GM), the greater the stability of the system. The gain margin refers to the amount of gain, which can be increased or decreased without

making the system unstable. It is usually expressed as a magnitude in dB.

**Formula** -

```
GM = 0  - gain (dB)
```

**Phase margin** : The greater the **Phase Margin** (PM), the greater will be the stability of the system. The phase margin refers to the amount of phase, which can be increased or decreased without making the system unstable. It is usually expressed as a phase in degrees.

**Formula** -

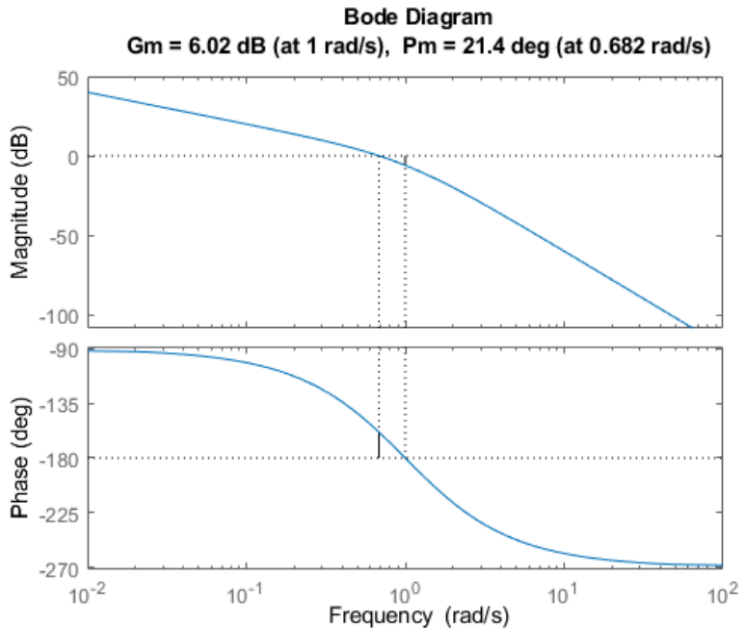
```
PM = phase_lag -( -180 deg)
# Negative PM means an unstable system.
```

Expected -

```
>> sys = tf(1,[1 2 1 0])
>> margin(sys)
{Gain margin : (6.02 dB), Phase margin : (21.4 deg)}
```

A **margin_plot(sys)** can also be added which is present in MATLAB too .It is basically the bode plot but with dotted lines to visualize the phase and gain margins .This can be easily handled since **bode_plot()** is already implemented with all possibilities in consideration.

A sample is shown below -




- **Use Of Minimal Realization( Not utilized in SymPy )**

  'minreal' is a MATLAB function that performs pole-zero cancellation of a transfer function. Although SymPy has 'simplify' / 'cancel' methods which the user can apply manually. But in some instances, it is expected that the result is in a minimal format.
  Consider a feedback system (MATLAB example)-

  **MATLAB**

```
K = 2;
G = tf([1 2],[1 .5 3])
H = feedback(G,K)
H

     s + 2
  ---------------
  s^2 + 2.5 s + 7
```

**SymPy**

```
>>> tf1 = TransferFunction(2, 1, s)
>>> tf2 = TransferFunction(s + 2, s**2 + S(1/2)*s +
3, s)
>>> c = Feedback(tf2, tf1)
>>> pprint(c.doit())
```

```
                    2
      (s + 2) ⋅ ⎛s  + 0.5⋅s + 3⎞
──────────────────────────────────────
   2                      2
⎛s  + 0.5⋅s + 3⎞ ⋅ ⎛s  + 2.5⋅s + 7⎞
```

As you can see here, the results are different . Here actually poles of numerator have been added both in the numerator and in the denominator which is not a good practice / unnecessary for user visualization, as explained by this **link**. This happens in many functions where MATLAB gives the minimal result but SymPy does not.

4. Possible improvements in ‘TransferFunction’ or ‘TransferFunctionMatrix’ API -

    - Making input ‘var’ in ‘TransferFunction’ optional instead of compulsory. Users will only have to specify the variable when there is a conflict. This may be a total change in SymPy convention but is already followed in **Wolfram Mathematica** and **MATLAB**.

Current -

```
>>> tf = TransferFunction(s**2 + 2*s + 1,s + 1)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() missing 1 required
positional argument: 'var'
```

Expected -

```
>>> tf = TransferFunction(s**2 + 2*s + 1,s + 1)
>>> pprint(tf)
 2
s  + 2·s + 1
─────────────
    s + 1
```

When there are multiple variables in num/den expressions an error can be raised

```
>>> tf = TransferFunction(s**2 + 2*s + 1, p + 1)
ValueError: There is a conflict detected for the
value of 'var', please specify it manually.
```

Making the ‘var’ optional would also enable us to input an expression directly . For example -

```
>>> tf = TransferFunction((2*s + 3)/(9*s**3 +
3*s**2 + s))
```

- Allowing a MATLAB-like input scheme where users just have to input coefficients of particular degrees of the Transfer

Function in order.I personally feel it is an efficient method of input and will itself eliminate many input issues, one of them pointed above as use of non-polynomials.

Expected -

```
>>> numerator = 1;
>>> denominator = [2,3,4];

>>> tf = TransferFunction(numerator,denominator)
>>> pprint(tf)

       1
──────────────
   2
2·s  + 3·s + 4
```

- Allowing a declared rational expression to be taken as input in ‘TransferFunctionMatrix’ to reduce size -

  Current -

```
>>> tf1 = TransferFunction(1,1+s,s)
>>> tf2 = TransferFunction(2,2+s,s)
>>> t = TransferFunctionMatrix([[tf1,
tf2],[-tf1,-tf2]])
```

  Expected -

```
>>> exp1 = 1/(1+s)
>>> exp2 = 2/(2+s)
>>> t = TransferFunctionMatrix([[exp1,
exp2],[-exp1,-exp2]])
```

- Other possible improvements I can come up with, are related to how polynomials are returned in the ‘TransferFunction’ model. In many instances, I have observed that polynomials are not returned in their canonical format. Polynomials in the numerator and denominator must always be in their expanded format as in MATLAB. Even if the user inputs a factorized format, there must be a flag, or else the polynomials should always be expanded.

- Adding support for ‘TransferFunctionMatrix’ objects to be instantiated by passing a list of numerators and a common denominator. This can be an alternative way of object creation.( *Referenced directly from Akshansh’s proposal* )

  This makes the API easier for the user . This feature is already present in MATLAB and Wolfram Mathemica .

5. Another questionable issue that is faced are the printing issues with **LaTeX** and **pprint** related to Series / Parallel objects ( depends on the example and the terminal too ) . I will be trying out many examples in Series / Parallel object creation and follow by printing them in both formats to notice potential errors.

These are the changes I would like to make. I believe that it will improve the control module in various ways. Firstly it will become more accessible and easier for users. Secondly, these changes will improve the time and space complexities of many evaluations, speed has always been a priority in SymPy!

Required unit tests will be added in the last week of this phase.

## Phase 3 -

Phase 3 will be about introducing the 'StateSpace' model for effectively representing a State Space system symbolically.
We will use the StateSpace class to represent a linear, time-invariant (LTI) control system.

```
x'(t) = A * x(t) + B * u(t); x in R^n , u in R^k
y(t) = C * x(t) + D * u(t); y in R^m
```

Here in this equation, u(t) is the input signal, and y(t) is the output signal. x(t) is the state of the system. The actual equation has A(t), B(t), C(t) and D(t), but in linear, time-invariant systems, all those four matrices are constant. Here A, B, C and D will be sympy matrices.

Implementation will take place in the following steps. They have been derived from **@sylee957** 's **comment** on **#18460** .

1. Define the constructor with '`__new__`' rather than '`__init__`', such that it 'sympifies' the arguments .

   StateSpaceModel uses '`Basic.__new__(cls, A, B, C, D)`' signature where A, B, C, D are matrices .An optional change can be anchoring the symbol '`s`' for case of symbolic coefficients as done in TransferFunctionModel .

2. '`__eq__`' doesn't have to be defined if it inherits Basic.

3. Define behavior of '`__neg__`' for StateSpace.

```
def __neg__(self):# Negates a state space system.

      return StateSpaceModel(self.args[0], self.args[1],
-self.args[2], -self.args[3])
```

4. Adding Series and Parallel support for StateSpace systems.

```
def series(self, other):# Returns the series interconnection of
the system and another system (other).

      # The implementation in #17866 can be improved further by
using `.args` instead of `.represent` and much more.
```

5. Interconversion between '`StateSpaceModel`' and '`TransferFunctionModel`' .For this I'll refer to this **pdf**, **this wiki**, and this **lecture slide** (resources shared by Akshansh Bhatt ).

   These 2 will be implemented earlier then it will simply use '`.rewrite()`' .

   - '`_eval_rewrite_as_StateSpaceModel( TransferFunctionModel )`'
   - '`_eval_rewrite_as_TransferFunctionModel( StateSpaceModel )`'

```
def _eval_rewrite_as_TransferFunctionModel(self):

      # After this, model interconversion would simply be
`.rewrite(TransferFunctionModel)`.
```

6. **Observability** is a property that indicates that each state of the system is observable from the output, meaning that the value of each state may be deduced .

   These methods will be added in the StateSpaceModel for checking various aspects of observability.

- `is_observable()`
- `Observability_matrix()`
- `observable_subspace()`

```
def observability_matrix(self):

# Description can be found here:
https://in.mathworks.com/help/control/ref/obsv.html#f3-209625
The matrix would be of np rows and n columns.
```

```
def observable_subspace(self):

# The observable subspace only depends on A and C. The
observable subspace of lti system is equal to the image of its
observability matrix.

      return self.observability_matrix().columnspace()
```

```
def is_observable(self):

# Returns a Boolean whether or not the system would be
observable. If the observability matrix is non-singular, then the
system would be observable.

# According to theory, we can just find the determinant.If the
det is 0, then the system is not observable else it is
observable..
```

7. `is_controlable()` can also be added to check for **controllability** of the system.

8. Move '`__str__`', '`__repr__`', '`_repr_latex_`' to '`_print_TransferFunctionModel`' and '`_print_StateSpaceModel`' in StrPrinter, LatexPrinter .

9. Add basic unit tests along the way to check functioning. High level documentation (illustrations / examples) from MATLAB could be adjusted in the next phase.

## Phase 4 -

The start of this phase will be dedicated to some examples for the 'StateSpaceModel' from some reference books, even preferably the course textbook of my university.
Then my focus will be on implementing a Discrete-Time 'TransferFunction' model.
An optional input parameter 'ts' - **sample time** will have to be introduced so that other parts of the code base like already written tests are not affected.

```
ts = 0 (by default) #indicating continuous time model
ts = True #discrete time model but with unknown / unassigned sample time
ts = finite rational #discrete time model with known sample time
```

I plan to make this optional because systems must have compatible timebases in order to be combined. A discrete time system with unspecified sampling time (ts = True) can be combined with a system having a specified sampling time, the result will be a discrete time system with the sample time of the latter system. The `__add__()` , `__sub__()` functions will be modified to conduct this.

Accompanied by parameters and related attributes -

```
>>> tf = TransferFunction(s**2 + 2*s + 1, p + 1, 2)
>>> tf.ts
```

```
2
>>> tf.sfreq() #Sampling frequency which is 2*pi/(ts)
(pi)
```

Followed by 2 state check functions -

- `is_ctime() #Continuous time model`
- `is_dtime() #Discrete time model`

I also intend to add functionalities after discussions with mentors, some basic ones since I aim this to be just an introduction to the discrete-time model framework.
There are a lot of potential functions which are useful and I would add, so pardon me if I forget to mention some of them here .

- `is_stable()`

  ```
  # Addition to function, already present for continuous time .
  # True when poles lie within an open unit disk or absolute
  magnitude of poles of a system is < 1.

  Example-

  >>> tf = TransferFunction(2*s,4*s**3 + 3*s -1,0.1,s)
  >>> tf.is_stable()
  True
  ```

**Sampling** will also be introduced which is a very helpful operation in practical scenarios. Through sampling continuous time models can be converted into discrete time models .It is an important addition, since all the CST packages which I explored while writing this proposal had this method.

Sampling is performed with some assumptions which the user will have to mention .These assumptions are explained in detail in this **MATLAB link**.
The expected api would look like this-

```
>>> from sympy.physics.control import TransferFunction
>>> s = Symbol('s')
>>> tf = TransferFunction(1,1+s,s)
>>> tfd = c2d(tf, ts = 0.1, assumption = 'zero')

  0.09516
 ----------
 z - 0.9048

Sample time: 0.1 seconds
Discrete-time transfer function.


# There are many possible assumptions but the basic ones will be
added.
# zero-order hold - You want an exact discretization in the time
domain for staircase inputs.
# first-order hold - You want an exact discretization in the time
domain for piecewise linear inputs.
```

New set of documentation containing unit tests will be added for the discrete time model . SymPy already has a great set of unit tests in the continuous time model, so refactoring and creating new tests out of them should not be a tough task.

## Timeline -

For an intended 175 hours project.

### Community Bonding Period

- Discuss proposal and relevant points with mentors .
- Learn and test stuff like ‘sympy.matrices’, ‘sympy.plotting’ which I need to be familiar with for the project.
- Finish a couple of unmerged pull requests opened by me.
- Set up a blog for weekly updates .Probably on wordpress.

**Week 1,2,3**

- Start with **Phase 1** coding work.
- Complete remaining pull requests from previous GSoC years.This includes adding documentation and plots.
- Raising issues regularly which my pull requests intend to solve so that the community is aware of the improvements.

**Week 4,5,6**

- Start with **Phase 2** coding work.
- Revamping ‘TransferFunction’ / ‘TransferFunctionMatrix’ in **sympy.physics.control.lti** .Fixing the issues raised in earlier weeks along with new implementations.
- Relevant documentation will be added.

**Week 7,8,9**

- Start with **Phase 3** coding work.
- Adding ‘StateSpaceModel’ and all it’s functionalities mentioned above.
- Since adding this class is a major task ,I’ll adjust documentation in the upcoming phase.

**Week 10,11,12**

- Start with **Phase 4** coding work.
- Make sure that everything added / changed in the previous weeks is tested and documentation is tip top.
- Adding a basic discrete time ‘TransferFunction’ model ,incorporating it with the continuous time ‘TransferFunction’ model.

**Week 13**

- Make sure all pull requests I have opened are successful.
- Submit final evaluations.

## Post Google Summer Of Code

1. After completion of the project, my initial plan is to complete implementations where I will get stuck in the coding period.

2. I will look forward to being a regular contributor and code reviewer for new contributors. I consider GSOC as my work for SymPy in the short term but what I consider more important is my contribution to SymPy in the long term which I believe is to get new contributors and guide them so that the community always grows.

3. One particular step I had in mind was to document a video of SymPy setup (specially for Windows OS) which many new contributors face and failing there instantly demotivates them. I myself faced many issues while setting it up locally. Making some nice SymPy tutorials on platforms like youtube will also be a part of my stretch work.

## Time Commitments ( During Gsoc )

1. I have no major commitments this summer and majority of my time will be spent in GSoC. I can positively say that daily 5-6 hours of my time will be dedicated to SymPy.

2. If I decide to have any change in my plans, it will be communicated to the mentors on a prior basis.

## References

1. Akshansh Bhatt’s proposal

2. Naman Gera's proposal
3. Good previous year proposals at **SymPy Wiki**
4. MATLAB control **documentation**
5. SymPy **Github** page for issues/pull requests/comments

## Acknowledgements & Appreciation

I started my journey in SymPy during Oct 2021 when I had no idea about what open source development means, no knowledge of any version control tools and their use. I have learned a lot in these past months going from communication to technical writing to developing top quality code content to sharpening my math/physics skills. I owe the whole SymPy community for that.

I want to acknowledge some members of the organization who have helped me immensely.

**@oscarbenjamin ( Oscar Benjamin )**
**@smichr ( Christopher Smith )**
**@moorepants ( Jason Moore )**
**@oscargus ( Oscar Gustafsson )**

**Thank-you for going through this proposal.**

**Regards,**
**Anurag Bhat.**