GSoC 2020 Application - Implement a control systems package for SymPy

About Me

Basic Information

Name - Naman Gera
University - Indian Institute of Information Technology, Guwahati
Email - namangera15@gmail.com
Github profile - https://github.com/namannimmo10
Stack Overflow profile - namannimmo
Time zone - IST (UTC +5:30)

Personal Background

I'm Naman Gera, a sophomore at Indian Institute of Information Technology, Guwahati, pursuing a Bachelor of Technology (4 years) in Electronics and Communication engineering.

I spend my free time playing soccer with my friends, reading books (poetry and biographies) or playing chess online. A few of the courses I had the chance to take at school are Linear Algebra and calculus; complex analysis; Differential equations; Signals and Systems (covering some part of Control theory); principles of communication; Object-Oriented Design and Analysis; & Data Structures.

Programming Background

I use Ubuntu 19.04 as my operating system, along with VS Code being my primary source code editor because it has a nice debugger. I try to complete some of the mundane tasks using the terminal. I am familiar with the `git` and `Github` workflow.

I started programming almost 2 years ago through the introductory CS course called CS50 by Harvard University on edX.com. I was really amazed by the enthusiasm of the professor to teach some fundamental CS concepts to beginners from scratch. Post the course, I continued to expand my learning and right now, I feel comfortable writing code in C, C++, Python, and Java.

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 like these:

1. Python Data Structures
2. Programming for Everybody in Python
3. Learning How to Learn

I like python because of its English like syntax, and because of its advantages over other languages. I'm amongst the top 3 committers to this ProjectEuler focused repository as can be seen here because I really like problem-solving in general and I frequently take part in coding competitions including ACM ICPC, and on various other platforms.

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

```
>>> from sympy.abc import x, y
```

```
>>> from sympy import pprint, sqrt
>>> expr = (x**2 - 5*x*y + 10*y**2)/sqrt(8)
>>> pprint(expr)

   ⎛ 2             2⎞
√2⋅⎝x  - 5⋅x⋅y + 10⋅y ⎠
──────────────────────
          4
```

Contributions

I started using SymPy in November 2019 and made my first contribution to the main repository the next 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

- #18100 - Change `Basic.NaN` to `S.NaN`, it raised `AttributeError` earlier. Fixes #4841.
- #18107 - Fix bug in `cancel` function. Fixes #17843.
- #18109 - Add `global_evaluate[0]` condition in `add`, `sub`, `mul`, `div` methods in `Infinity` and `NegativeInfinity` class. Fixes #18052.
- #18149 - Add regression test for `subs`. Fixes #11746.
- #18187 - Remove unused imports from test_evaluate.py
- #18192 - Handle `Mul.is_imaginary` for infinite values. Fixes #17556.
- #18222 - Fix bug in intersection of two `Line3D`. Fixes #8615.
- #18250 - [feature] Make Curve Callable with parameter values.
- #18264 - Remove `filterfalse` and Python 2 related things from compatibility file.
- #18275 - Completes one todo task in shor.py

- #18286 - [WIP] Remove python 2 related things from the entire codebase. Helps in fixing #17943.
- #18302 - Cleaning up compatibility.py. Helps in fixing #17943.
- #18335 - [feature] Add `bisectors` method for `Polygon` class.
- #18339 - Sympified output numbers for `factorial` and `binomial` used with `Mod`. Fixes #18338.
- #18362 - Use `__index__` instead of checking for explicit list of types. Fixes #16779.
- #18373 - Removes an unreachable part of code.
- #18409 - Improves code for `gcdex_diophantine`.
- #18463 - Add check for integrating `Piecewise` function. Closes #7370.
- #18498 - Add some of the missing `core` docstrings.
- #18601 - Remove `_print_GeometricEntity` method from `StrPrinter` class.
- #18612 - Dropping python 2 support. Helps in fixing #17943.
- #18633 - Fix random failing test in `test_tensor_operators.py`. Fixes #18620.
- #18828 - Improve `free_symbols` method in class `Point`.

Unmerged PRs

- #18169 - Add `dra`, `drm` functions in `ntheory` module. Further improved `dra` (O(1) time complexity), but couldn't get merged because a fellow contributor was working on adding a similar feature.
- #18240 - `Crypto`: Add XOR cipher.
- #18329 - Add `factmod` function in `ntheory` module.
- #18429 - Implement Chinese remainder theorem over a cartesian product of vectors.
- #18436 - Add `control` package to `sympy.physics`.
- #18517 - Add `Min/Max` support for `decompogen`.

Issues raised

- #18144 - `trigsimp` does not `simplify` expression.
- #18372 - Sympy docs show Sympy 1.5.1 twice.
- #18460 - Discussion regarding the API for `control` package.
- #18514 - `exp(x)*erf(x*I)` and `exp(-x)*erf(x*I)` don't work.

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

- *Code Review* - #18849, #18834, #18502.
- *Discussions* - #18627, #16500.

Questions answered on stack overflow related to SymPy

- [permalink] - Integrate exponential function using sympy.
- [permalink] - How can I specify an integer index in sympy?
- [permalink] - How to use a list of symbols for sympy calculation in python?

The Project

Overview and Motivation

The main aim of this project is to implement in sympy a basic Control System functionality that would benefit a Control System engineer. So what exactly is CST? CST is the mathematical process of designing a machine to control another machine. The machine to be controlled is called the *plant*, and the machine doing the controlling part is called the *controller*. Both these machines are referred to as a system.

Some contributors made efforts to add `control` package to `sympy.physics` in #17866 and #12189. But those were not polished enough to get into the sympy codebase. As I have completed some parts of this section from my course, I'm familiar with the required theory behind this. Throughout this project, two classes would be implemented, namely: `StateSpaceModel` and `TransferFunctionModel`.

`StateSpaceModel` will use sympy matrix, so it would be able to support most of the symbolics. SymPy already has a very rich polynomial module that will back our well defined `TransferFunctionModel` class. TFM will subclass `Mul` and SSM will subclass `Basic`. My proposal aims at adding a package that would be a full symbolic model more than just a simple solver. I would also love to maintain a blog during this program so that it is easier for others to further expand this package.

Implementation Plan

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

- **Phase 1** := Adding `StateSpaceModel` and `TransferFunctionModel` class. Methods that are already implemented in #17866 will be polished for merging. In the last week of this phase, I will also write the required tests to ensure that everything is implemented correctly.
- **Phase 2**:= These methods will be implemented --
    * `observability_matrix` (for `StateSpaceModel`)
    * `observable_subspace` (for `StateSpaceModel`)
    * `is_observable` (for `StateSpaceModel`)
    * `__neg__` (for both)
    * `_eval_rewrite_as_StateSpaceModel` (for `TransferFunctionModel`)
    * `_eval_rewrite_as_TransferFunctionModel` (for

`StateSpaceModel`)
Tests will be added for these methods in the last week of this phase. Also, documentation and examples for phase 1 will be covered.

- **Phase 3**:= `bode_plot` and `root_locus_plot` will be implemented, along with the final documentation. Bugs will be resolved if any.
- **Phase 4**:= Preparing documentation of examples and tutorials for our implemented control systems package (Week 10).
  Adding `IVPSolution` to represent a solution of initial value problems, along with required tests and documentation (Week 11 and 12).

Phase 1

We use our `StateSpaceModel` 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
```

Where `u(t)` is an input signal, `y(t)` is the corresponding output and `x(t)` is the state of the system.
Here, A, B, C, and D are sympy matrices.

We use `TransferFunctionModel` class to also represent a linear, time-invariant control system. This class introduces a transfer function matrix G in Laplace space. The input-output relation for the system in Laplace space is given by:

```
Y(s) = G(s) * U(s);     s in C
```

where `U(s)` is the input to the system in Laplace space and `Y(s)` is the corresponding output.

A **transfer function** basically represents a ratio of output to the input:

$$\frac{Y(s)}{U(s)} = \frac{c_m s^m + c_{n-1} s^{n-1} + \ldots + c_1 s + c_0}{a_n s^n + a_{n-1} s^{n-1} + \ldots + a_1 s + a_0}.$$

An important property that the transfer function may have is that of being *proper*. So in this phase, I also implement a function `is_proper` which checks if the degree of the numerator is less than or equal to the degree of the denominator.
As we know that the transfer function is a ratio of polynomials, we already have a way to represent that using sympy's core class `Mul`. We are now able to use all the features from that class, for example -- `as_numer_denom`, expressing in evaluated and non evaluated form and much more.

```
>>> from sympy import Mul
>>> from sympy.abc import s
>>> expr = Mul(s**2 - 3*s + 4, 1/(s**4 - s*2))
>>> pprint(expr)
 2
s  - 3·s + 4
─────────────────────
   4
  s  - 2·s
>>> expr.as_numer_denom()
(s**2 - 3*s + 4, s**4 - 2*s)
```

Both the classes have already been proposed in #17866, I will continue working in #18436 (WIP) which I had raised before. We can use these guidelines (suggested by @sylee957) to make this a candidate for a full symbolic model. All that can be done while we subclass `Basic`.

1. Define the constructor with `__new__` than `__init__` (as implemented in the PR), such that the sympification of arguments is possible.
    - So, the `StateSpaceModel` uses `Basic.__new__(cls, A, B, C, D)` signature where `A, B, C, D` are matrices.
    - `TransferFunctionModel` uses `Basic.__new__(cls, num, denom)` signature where `num, denom` are polynomials, each representing the numerator and denominator of the transfer function as explained above.
    - `represent`, `block_represent` will be derived from `.args` rather than storing `represent` and `block_represent` by mutating the objects.
2. `__eq__` don't have to be defined because it inherits `Basic`.
3. `__str__` , `__repr__`, `__repr_latex_` will be moved under `_print_TransferFunctionModel` and `_print_StateSpaceModel` in `StrPrinter` and `LatexPrinter`.
4. `_eval_rewrite_as_TransferFunctionModel` and `_eval_rewrite_as_StateSpaceModel` to be implemented in phase two. And then the model interconversion would be `.rewrite(StateSpaceModel)` and `.rewrite(TransferFunctionModel)`.

In the docstrings, `symbols` will be used instead of `var`, and plenty of examples and use cases will be added.

Along with addressing all the comments, I'm gonna define `total_degree` for `TransferFunctionModel`. This gives back the total degree of the rational function. If the degree of the denominator is greater than that of the numerator, we get a negative result.

```
def total_degree(en):
    num, denom = en.as_numer_denom()
    return degree(num) - degree(denom)
```

This function will help in the simplification of `is_proper` from this:

```
def is_proper(m, s, strict=False):
    if strict is False:
        return all(degree(en.as_numer_denom()[0], s) <=
                   degree(en.as_numer_denom()[1], s) for
en in m)
    else:
        return all(degree(en.as_numer_denom()[0], s) <
                   degree(en.as_numer_denom()[1], s) for
en in m)
```

To this:

```
def is_proper(m, strict=False):
    if strict is False:
        return all(total_degree(en) <= 0 for en in m)
    else:
        return all(total_degree(en) < 0 for en in m)
```

Some examples for methods in `StateSpaceModel` are:

```
>>> from sympy import Matrix, Symbol
>>> from sympy.physics.control import StateSpaceModel
>>> A, B, C, D = Matrix([1, 2]), Matrix([2, 3]),
Matrix([2]), Matrix([0])
>>>
>>> # now creating a state space system
```

```
>>> ssm = StateSpaceModel(A, B, C, D)
>>> ssm
StateSpaceModel(
Matrix([
[1],
[2]]),
Matrix([
[2],
[3]]),
Matrix([[2]]),
Matrix([[0]]))
```

We can use less matrices as well, and the rest will be filled with a minimum of zeros.

```
>>> ssm_2 = StateSpaceModel(A, B)
>>> ssm_2
Matrix([
[1],
[2]]),
Matrix([
[2],
[3]]),
Matrix([[0]]),
Matrix([[0]]))
```

We can also use a `TransferFunctionModel` to create a `StateSpaceModel`. Like this:

```
>>> s = Symbol('s')
>>> from sympy.physics.control import TransferFunctionModel
>>> tfm = TransferFunctionModel(Matrix([1/s, s/(1+s**2)]))
```

```
>>> StateSpaceModel(tfm)
StateSpaceModel(
Matrix([
[0, -1, 0],
[1, 0, 0],
[0, 1, 0]]),
Matrix([
[1],
[0],
[0]]),
Matrix([
[1, 0, 1],
[1, 0, 0]]),
Matrix([
[0],
[0]]))
```

Symbolic evaluation of `StateSpaceModel` is possible -

```
>>> ssm = StateSpaceModel(Matrix([[-1, 1], [1, -1]]),
ones(2, 1), eye(2))
>>> u = Matrix([exp(2*t)])  # input
>>> x = Matrix([1, 0])      # initial state
>>>
>>> # symbolic evaluation
>>> ssm.evaluate(u, x, t)   # `t` is a Symbol.
Matrix([
[cosh(2*t)],
[sinh(2*t)]])
```

Finally, series and parallel interconnection of two `StateSpaceModel`

```
>>> A, B, C, D = Matrix([1, 2]), Matrix([2, 3]),
Matrix([2]), Matrix([0]);
>>> ssm_1 = StateSpaceModel(A, B, C, D)
>>> ssm_2 = StateSpaceModel(A, B)
>>> ssm_1.series(ssm_2) #series interconnection.
StateSpaceModel(
Matrix([[1, 0], [2, 0], [4, 1], [6, 2]]),
Matrix([[2], [3], [0], [0]]),
Matrix([[0, 0]]),
Matrix([[0]]))

>>> ssm_1.parallel(ssm_2) #parallel interconnection.
StateSpaceModel(
Matrix([[1, 0], [2, 0], [0, 1], [0, 2]]), #deliberately
written horizontally.
Matrix([[2], [3], [2], [3]]),
Matrix([[2, 0]]),
Matrix([[0]]))
```

API (with some basic examples) for `TransferFunctionModel` --

```
>>> G = Matrix([1/s, 1/(s + 1)])
>>> # creating a TransferFunctionModel from a matrix.
>>> tfm = TransferFunctionModel(G)

>>> tfm.G # this gives back the Matrix G.

>>> # Create TransferFunctionModel from StateSpaceModel
>>> tfm_2 = TransferFunctionModel(StateSpaceModel(A, B, C,
D), s)
```

```
>>> s = Symbol('s')
>>> u = Matrix([1/s])
>>> # example for `evaluate` method.
>>> TransferFunctionModel(Matrix([s/(1 +
s**2),1/s])).evaluate(u, s)
Matrix([
[1/(s**2 + 1)],
[     s**(-2)]])

>>> a, b = symbols('a b')
>>> tfm_1 = TransferFunctionModel(Matrix([1 / (a * s)]))
>>> tfm_2 = TransferFunctionModel(Matrix([(b + s) / (a +
s)]))
>>> tfm_1.series(tfm_2)  #series interconnection of two TFM
TransferFunctionModel(Matrix([[(b + s)/(a*s*(a + s))]]))

>>> tfm_1.parallel(tfm_2) #parallel interconnection.
```

Required tests will be added in the last week of this phase to ensure that everything is implemented correctly. If possible, I'll start coding in the last week of the community bonding period so that I get a good headstart and work in the next phases will proceed swiftly.

Phase 2

Now I'll further add more features to this control systems package by opening up a PR. These are the following methods that I will implement.

- `observability_matrix` (in `StateSpaceModel`)
- `observable_subspace` (in `StateSpaceModel`)
- `is_observable` (in `StateSpaceModel`)

- `__neg__` (in `TransferFunctionModel`)
- `__neg__` (in `StateSpaceModel`)
- `_eval_rewrite_as_TransferFunctionModel` (in SSM)
- `_eval_rewrite_as_StateSpaceModel` (in TFM)

Observability - Given a multiple-state system in StateSpace form, it may be desirable to know which states are observable. Observability is a property that indicates that each state of the system is observable from the output, that is, the value of each state may be deduced.
Observability matrix can be constructed as below for a state space system {A, B, C, D} or order n. This matrix would be of np rows and n columns for n-by-n matrix A and p-by-n matrix C.

$$\begin{bmatrix} C^T & A^T C^T & \dots & (A^T)^n C^T \end{bmatrix}$$

If the observability matrix is *non singular*, the system is observable.

Another thing that can be useful is `observable_subspace` of an LTI system. It is equal to the image of its `observability_matrix`. The implementation for that is simple, like this:

```
def observable_subspace(self):
    return self.observability_matrix().columnspace()
```

`is_observable` := `is_observable()` returns a Boolean indicating whether or not a system is observable. There are many approaches to check if the system is observable or not.

- *Observability matrix test* - The system is observable if and only if rank *O* = n (where *O* is the observability matrix!). This one is easy to implement.
- *Eigenvector test for observability* - An LTI system is said to be observable if and only if no eigenvector of A is in the kernel of C.

Proposed API:

```
>>> ssm.observability_matrix() # depends only on A and C.
>>> ssm.observable_subspace() # returns a list of vectors,
and depends only on A and C
>>> ssm.is_observable() # returns a Boolean
```

Negation - Sometimes, negation of a system is required while solving control problems. So, I think `__neg__` would be a nice addition for `StateSpaceModel` and `TransferFunctionModel`.

`__neg__` for `StateSpaceModel`: This method negates a state space system
`__neg__` for `TransferFunctionModel`: This method negates a transfer function.

```
def __neg__(self):
    return StateSpaceModel(self.args[0], self.args[1],
                           -self.args[2], -self.args[3])
----------------------------------------------------------------
>>> A, B, C, D = Matrix([1, 2]), Matrix([2, 3]),
Matrix([2]), Matrix([0]);

>>> ssm = StateSpaceModel(A, B, C, D)
>>> ssm.__neg__()
StateSpaceModel(
Matrix([
[1],
[2]]),
Matrix([
[2],
[3]]),
Matrix([[-2]]),
```

```
Matrix([[0]]))
```

Similarly, we can have `__neg__` for `TransferFunctionModel`, whose API would be something like this -

```
>>> expr = Mul(s**2 - 3*s + 4, 1/(s**4 - s*2))
>>> expr.__neg__()
(-s**2 + 3*s - 4)/(s**4 - 2*s)
>>> pprint(_)
   2
- s  + 3⋅s - 4
──────────────────────
    4
   s  - 2⋅s
```

As we've seen that `StateSpaceModel` already takes in four arguments, so now there's a need for model interconversion handled as a special case for one argument.

```
def _eval_rewrite_as_TransferFunctionModel(self):
    return self.rewrite(TransferFunctionModel)

def _eval_rewrite_as_StateSpaceModel(self):
    return self.rewrite(StateSpaceModel)
```

After defining both of them, model interconversion would be as simple as `.rewrite(StateSpaceModel)` & `.rewrite(TransferFunctionModel)`

All the required tests will be added along in the same PR. So, with this our basic control systems functionality is implemented!

Phase 3

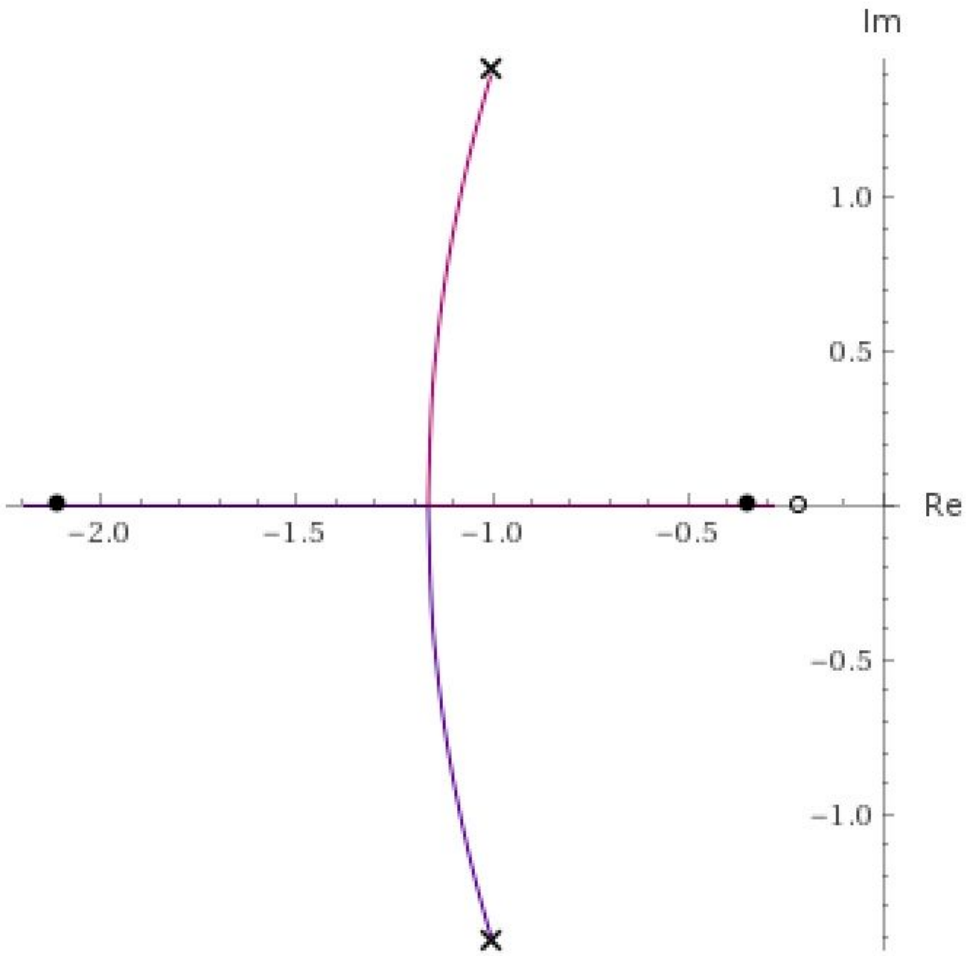
Plotting - Visualization is an important aspect while dealing with control problems. SymPy has a Plotting module which can be used as our backend. In this phase, I'll add functionality for `root_locus_plot` and `bode_plot` because these are two of the most popular control related plots used.

- *Root Locus Plot* - As we know that our transfer function is a ratio of polynomials, we can get the root locus for that, given the provided gains.

```
>>> tf = (2*s**2 + 5*s + 1)/(s**2 + 2*s + 3)
>>> pprint(tf)
   2
2·s  + 5·s + 1
──────────────────────────
  2
 s  + 2·s + 3
>>> tf.root_locus_plot(gains) # `gains` is a range.
```

Following is a rough idea of what the plot would look like for `gains` between 0 and 10. The purple (dark blue) plot is the *locus*, **x** is for *poles* and **o** is for *zeros* for our transfer function.

SymPy's plotting module supports dynamic positioning of the coordinates, so getting value of any quantity by just moving the cursor over it would be an added advantage for us. Some hints can be taken from this blog post.



(shown for gain between 0 and 10)

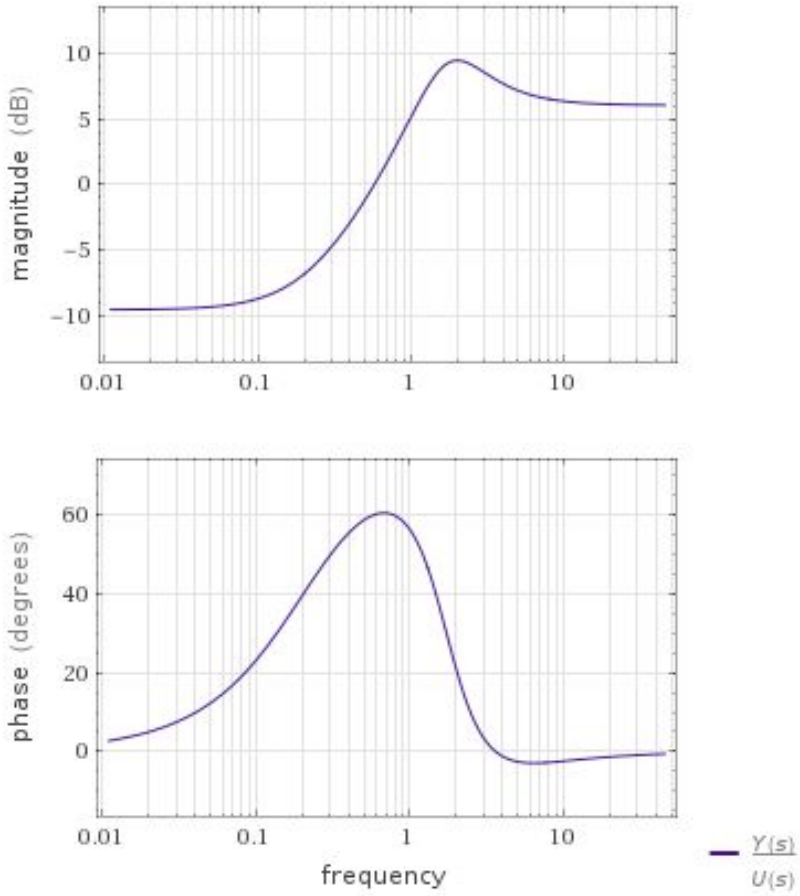
- *Bode Plot* - It is another commonly used control related plot. The bode plot for a linear, time invariant system with transfer function $H(s)$
  consists of a magnitude plot and a phase plot.

  1) Bode Magnitude plot - is the graph of the function `|H(s=jw)|` of frequency `w`. The w-axis of the magnitude plot is logarithmic and the magnitude is given in decibels, i.e., a value for the magnitude `|H|` is plotted on the axis at `20 log |H|` (base 10).
  2) Bode phase plot - is the graph of the phase (commonly expressed in degrees), of the transfer function as a function of

`w(frequency)`. The value for the phase is plotted at the linear vertical axis.

```
>>> # transfer function
>>> tf = (2*s**2 + 5*s + 1)/(s**2 + 2*s + 3)
>>> tf.bode_plot(range) # pass frequency range.
```

Following is a rough idea of what the plots would look like:



Examples will also be added in the same PR. Some of them can be taken from this manuscript.

Phase 4

By the time I would reach this phase, I would have implemented a control package for SymPy along with the tests and docs.

However I think that we should have a nice documentation of a bunch of examples depicting the uses of this package. People tend to use the software more if the docs are top-notch. So in the first week of this phase (Week 10), I'm gonna prepare a well structured docs which will show each and every feature and assists the users in realizing these features along with examples of control problems solved using the implemented API.

There is already a great python package, python-control, that implements basic control systems functionality but I think the docs are not that great, with too few examples. I want SymPy's control package to be a *full symbolic model* with a great documentation, like it already has.

I can easily find a bunch of examples in this manuscript and in this book - *Linear Systems Theory* by *Joao P. Hespanha*, 2009.

In the following weeks (Week 11, 12), I'll add an object, `IVPSolution`, that represents the solution of an ODE initial value problem. Given an ODE of type `Eq(f(x).diff(x), x)` with an initial condition `Eq(f(x), 0)`, we'll be able to define an object `IVPSolution(t, (Eq(f(x).diff(x), x),), (Eq(f(x), 0))` which would mathematically represent `f(t)` (`f` is the solution of the initial value problem).

We can now apply `subs`, `evalf`, or `lambdify` for further work as per our requirement. The idea (suggested by @oscarbenjamin in #18023) would be to get numerical solutions for ODE.

`IVPSolution.evalf` will use the odefun of mpmath to give us numeric solutions to ODE initial value problems.

Like this for example:

```
>>> ft = IVPSolution(t, (Eq(f(x).diff(x), x),), (Eq(f(x),
0)));
>>> # In this example, f(t) = exp(t)
>>> ft.evalf(subs={t:1})
2.71828182845905
```

Required tests and documentation will be added along in the same PR.

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 (April 27 - May 17)

- In this period, I will discuss the project with my mentor so that we come up with an efficient way of implementation. My end sem exams will be over by 2nd of May and if possible, **I will start coding in the last week of the community bonding period** so that I get a good head-start for Phase 1, which is the most important of all.
- Also I'll be studying the Plotting and matrices module in more detail to utilize it to the fullest in the official coding period.

Week 1, 2, 3 (May 18 - June 7) - Phase 1

- Add `StateSpaceModel` and `TransferFunctionModel` class.

- Required tests will be added to ensure that everything is implemented correctly.
- Since the major task is to set up both these classes, I'll write documentation in phase 2.

Week 4, 5, 6 (June 8 - June 28) - Phase 2

- These methods will be implemented in this phase -- `observability_matrix`, `observable_subspace`, `is_observable`, `__neg__` (for `TransferFunctionModel` and `StateSpaceModel`), `_eval_rewrite_as_TransferFunctionModel`, `_eval_rewrite_as_StateSpaceModel`.
- Tests and documentation for this newly implemented functionality will be added in the same PR that I'll open specifically for this phase.
- Also, documentation for phase 1 will be added.

Week 7, 8, 9 (June 29 - July 19) - Phase 3

- Implementation of `bode_plot` and `root_locus_plot` will be done in a new PR along with documentation and examples.
- Will do required changes to make plots look more visually appealing, readable and useful.
- Bugs will be taken care of, if any.

Week 10, 11, 12 (July 20 - Aug 10) - Phase 4

- *Week 10* - I will prepare a proper documentation of examples and tutorials for this package.
- *Week 11, 12* - Add `IVPSolution` to represent solution of an ODE initial value problem along with required tests and documentation.

Week 13 (Aug 10 - Aug 17)

- I will make sure all the PRs are successfully merged into the codebase.
- Submitting final evaluations.

Post GSoC

After working with SymPy for about nine months, I would still be eager to contribute more to this software. If the circumstances are good enough, I will become a mentor for next year's GSoC.

Notes

I've got no other commitments apart from this project, and thus would be able to devote 40 to 50 hours per week. My college will restart on the 2nd of August, and in the first month we have a light coursework, so I would be able to give all my time and energy to finish the project.

References

[1] - Oscar Benjamin
https://github.com/oscarbenjamin

[2] - Oscar's issue on `IVPSolution`.
https://github.com/sympy/sympy/issues/18023

[3] - Feedback Systems (Second Edition)
http://www.cds.caltech.edu/~murray/books/AM08/pdf/fbs-xferfcns_12Aug2019.pdf

[4] - Python Control
https://python-control.readthedocs.io/en/latest/index.html

[5] - Bicycle Control Design
https://plot.ly/python/v3/ipython-notebooks/bicycle-control-design/

[6] - SY Lee
https://github.com/sylee957

[7] - Discussion with mentors in this issue.
https://github.com/sympy/sympy/issues/18460

[8] - Some good previous year GSoC applications.

[9] - Gagandeep's PR on control systems.
https://github.com/sympy/sympy/pull/17866