Improving and Extending `stats` module

Smit Lunagariya

March 28, 2020

Abstract

`SymPy` is currently supporting Stochastic Processes, Random matrices, Joint, Continuous, and Discrete distributions in its `stats` module. Compound Distributions introduced in 2018, does not provide complete support. Therefore, I plan to do the following during this summer.

- Sampling from external libraries.
- Adding and testing currently implemented Stochastic Processes.
- Assumptions of dependence between Random Variables.
- Improving and adding support for Compound Distributions.
- Improving Joint Distributions Framework and adding more distributions.
- Adding support for Mixture Distributions.

Contents

1 About Me

1.1 The Student

- **Name:** Smit Lunagariya
- **Email:** smitlunagariya.mat18@itbhu.ac.in
- **GitHub:** https://github.com/Smit-create
- **Gitter-id:** Smit-create

1.2 The University

- **Name:** Indian Institute of Technology - (BHU), Varanasi
- **Major:** Mathematics and Computing
- **Year:** Sophomore
- **Degree:** Integrated Dual Degree

1.3 Programming Experience & Mathematical Background

I have been programming for the last two years, and for the past year in python. Apart from python, I have experience in C, C++, Java. I work on Ubuntu 18.04 LTS and use the `atom` as my editor because it has an inbuilt version control system and is easy and simple to use. I use `pdb` as a python debugger. I am also familiar with Git and GitHub workflow.

I have background knowledge in Mathematical and Computer Science courses which include Linear Algebra and Abstract Algebra, Probability and Statistics, Information Technology Python Workshop, Data Structure, Mathematical Methods(ongoing), Numerical Techniques(ongoing), Operating Systems(ongoing), and Algorithms(ongoing).

I have also undertaken many online courses including Deep Learning, Advanced Reinforcement Learning, Convolutional Neural Networks, and Stochastic Processes(NPTEL).

I use Python for programming as well as for scientific computing. The syntax of the python programming helps to do complex coding in fewer steps and also helps to enhance the code readability. The latest feature of python which I have used is `walrus operator (:=)` which allows us to assign variables inside an expression and was introduced in Python 3.8.

I began to use `SymPy` in the first week of December, 2019 and was quite impressed with multiple functionalities provided by it. My most favourite feature in `SymPy` is Fourier and Laplace Transforms which I also use for my academic course on Mathematical Methods.

```
>>> from sympy import *
>>> from sympy.abc import x, s
>>> fourier_transform(1/(1+x**2), x, s)
pi*exp(-2*pi*s)
>>> laplace_transform(cos(2*x)*x**3, x, s)
(6*(s**4 - 24*s**2 + 16)/((s**2 + 4)**2*(s**4 + 8*s**2 + 16)), 0, True)
>>> laplace_transform(sin(3*x)/x, x, s)
(-atan(s/3) + pi/2, 0, True)
```

1.4 My Motivation

I had found my interest in Probability and Statistics during my school days and continued exploring the literature related to it. It also forms one of the most important basis of growing Artificial Intelligence. Therefore, I started exploring the `stats` module of `SymPy`. I was really fascinated by its ability to symbolically solve the complicated expressions of `density`, `expectation` and `cdf` with the support of powerful `integrate` and `solvers` module.

Moreover, it also supports Stochastic Processes and Random matrices which are my favourite features in the `stats` module. Stochastic Processes were introduced in the last year's GSoC and can be extended to provide support for more of such processes. I found that my interest matches with `SymPy` and therefore started exploring it more to know its powerful features.

1.5 Contributions

I started using `SymPy` in the beginning of the December, 2019. I made my first contribution in the middle of December, 2019 and continued to explore, learn and contribute to the library consistently.

1.5.1 Merged PR's - Related to `stats`

- `18935` - Added coskewness function
- `18655` - Increases code coverage of crv_types(stats)
- `18639` - Corrected Documentation in finite rv types
- `18614` - Fixed DiscreteUniform to produce correct density and cdf
- `18577` - Fixed compute_expectation to use non-sympify parameters
- `18446` - Added Moyal Distribution
- `18336` - Fixed Wrong Integral from SingleContinuousPSpace
- `18311` - Fixed a bug in probability method in ContinuousPSpace
- `18300` - Added Median
- `18289` - Added sampling methods of Discrete Random Variables
- `18233` - Added Hermite Distribution
- `18173` - Added Bernoulli Process
- `18152` - Added Powerfunction Distribution
- `18096` - Added Levy Distribution

1.5.2 Merged PR's - Related to other modules

- `18698` - Fixes sqf_list to combine same multiplicity factors
- `18677` - Added backend option in plotting
- `18640` - Fixes TypeError in piecewise_simplify
- `18605` - Fixes Idx object to accept non-integer bound
- `18597` - sqrt(x).is_negative return False instead of None
- `18566` - Fixed bug in Heuristic gcd for polynomial gcd
- `18537` - Fixed aspect_ratio parameter with MatplotlibBackend
- `18535` - Fixes polynomial solve with Golden Ratio and Tribonacci Constant
- `18472` - Handing Integers containing I
- `18450` - Fixes recursion error in Limit
- `18434` - Handling Float Integrals
- `18405` - Fixes test failure with cache off
- `18390` - Removed deprecated in assumptions

- 18276 - Using gmpy in integer_nthroot and igcd
- 18159 - Fix Attribute error in matrix and polynomial multiplication
- 18141 - Fixes ValueError from meijerg
- 18061 - Added assumption in DynamicSymbols

1.5.3 Open PR's

- 18754 - Added sampling methods of crv_types
  - Working on one of important the part of the proposal i.e. Sampling from External Libraries
  - Discussing and implementing various API designs that involves changing the `sample` methods across many stats files, such that the implementation is in an organized manner and makes it easy to extended for adding more of such external libraries.

1.5.4 Open Issues

- 18920 - Incorrect plot of exponential expression
- 18901 - Incorrect plot with constants
- 18796 - Integrate: Takes long time and produces unevaluated output
- 18702 - solveset with quartic equations is slow and gives complicated results
- 18423 - Test failure with cache off
- 18359 - solveset produces two different results
- 18323 - Incorrect output while comparing symbolic expression
- 18299 - Quantile function gives NaN

1.5.5 Discussions

- 18718 - Reparametrization and KL Divergence
  - Discussing the API and examples related to KL Divergence Implementation.
- 18730 - Mixture models and Mixture Distributions
  - Discussing the API for implementation of Mixture Distributions.
  - Proposing an idea for the stats plot wrapper for plotting `cdf` and `density` of random variables.

1.5.6 Reviewed PR's

- 18961 - Giving 'digits' a 'bits' argument
- 18960 - [WIP] Add multinomial coefficients
- 18958 - Including Polar Decomposition
- 18952 - Adding Multinomial Coefficients
- 18951 - 90 degree rotation of matrix elements
- 18950 - for 90 degree rotation of matrix elements

- `18947` - Solves bug in limit
- `18945` - Move Matrix test out of test_matrices.py
- `18923` - Test for the stats module
- `18905` - Tangent on the curve at a particular point
- `18903` - Evaluate nested floor/ceiling
- `18902` - tangent to curve
- `18900` - Functions and test cases added
- `18880` - adding latus rectum in parabola file in geometry module
- `18875` - Quartenion: added tests to increase test coverage
- `18858` - Improve code coverage of test_polygon.py
- `18856` - Singularity handling for plots improved
- `18734` - Docstring updated to clarify the statement about load value
- `18681` - Added Refine function for symmetric matrices

2 The Project

In this section I will provide the details and explanation regarding my project during this summer. I will first explain the plan briefly and then provide theory and implementation design in the further sections.

2.1 Brief Overview

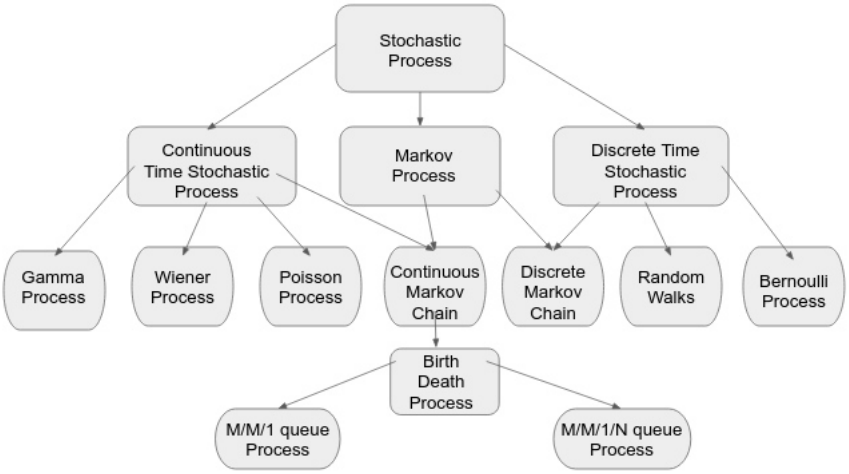
The `stats` module provides a powerful support for finite, continuous, and discrete distributions. Joint and Compound distributions lack a well defined framework as it will require code cleaning and improvement. I will be also focusing on extending the support of Stochastic Processes by adding more of such process and also implement **Mixture Distributions which are currently not supported by the `stats` module**. Therefore, my focus during the summer would be on the following:

- Extend the Stochastic processes
- Improve the implementation of Joint and Compound Distributions and add more examples
- Implement Mixture Distributions
- Adding assumptions of dependence
- Sampling from external libraries

2.2 The Plan

In this section, I will explain the plan and workflow of my project. The execution details regarding the same are presented in the following subsection. I would also discuss in detail regarding the validity of my plan with the mentor and would refine it further under his guidance during the project period.

Stochastic processes were introduced in the `stats` module in last year's GSoC and currently support Markov Processes(both Continuous and Discrete), and Bernoulli Process. I would further extend this to provide support for more of the Stochastic Processes. The below given flowchart shows the hierarchy order in the `stochastic_process_types.py`. Implementation design and code for each of the process and its examples follows in the next subsection.



The compound distributions were introduced in the `stats` module in GSoC 2018, but they do not provide complete support and require much refinement. The following snippet shows some of the failing examples of Compound Distributions:

```
>>> from sympy.stats import *
>>> from sympy.abc import x
>>> G = Gamma('G',2,3)
>>> T = Poisson('T', G)
>>> density(T)
CompoundDistribution(PoissonDistribution(G))
>>> density(T)(x)
Integral(G*G**x*exp(-4*G/3)/(9*factorial(x)), (G, 0, oo))
>>> cdf(T)(x)
Traceback (most recent call last):
...
AttributeError: 'JointPSpace' object has no attribute 'compute_cdf'
>>> E(T)
Traceback (most recent call last):
...
NotImplementedError: Expectations of expression with unindexed
joint random symbols cannot be calculated yet.
```

Currently, it shares a joint probability space with joint distributions. This needs to be changed and **provide a new Compound Probability Space** and more examples will be added related to compound distributions. I would continue the PR #17036 by Ritesh Kumar which require many changes in the Compound Probability Space which I have proposed in the following subsection

along with the example outputs.

One of the interesting part of this project would be to provide support for Mixture Distributions. These are currently not supported.

```
>>> from sympy.stats import *
>>> from sympy.abc import x
>>> N = Normal('N',1, 2)
>>> M = Normal('M', 3, 4)
>>> X = 0.3*N + 0.7*M
>>> density(X)(x).simplify()
0.230245262479199*exp(-0.0609756097560978*x**2 + 0.292682926829268*x - 9/32)
/sqrt(pi)
>>> P(X > 1)
Traceback (most recent call last):
...
sympy.integrals.meijerint._CoeffExpValueError: expr not of form a*x**b:
1.25560538116592
>>> cdf(X)(x)
Traceback (most recent call last):
...
ValueError: CDF not well defined on multivariate expressions
```

Adding Mixture Distributions would include creating a new file named `mixture_rv.py` consisting of temporary `MixturePSpace`(it will map to corresponding Probability Space), class `MixtureDistribution`, and a RV `Mixture` - Random Variable representing `MixtureDistribution` and other helper functions. I have provided the complete implementation code and the examples of using Mixture Distributions in the next subsection.

2.3 Execution and Implementation

In this section, I will discuss the theory and the implementation code of the project. I will divide the whole project into four phases which include Community Period, and three official coding phases and explain each of these separately.

2.3.1 Community Bonding Period

During this phase, I will start discussing the API and the implementation details of the Compound Distributions and the Mixture Distributions with the mentor. Since I have been contributing to the library for the past four months, it would be easier for me to get into the community.

As many distributions are supported in the `stats` module under Discrete and Continuous Random variable, I would like to add more of such distributions as some of them might be useful in further implementation of Joint and Compound Distributions. While adding these distributions, I will make sure to increase the code coverage and add all the necessary tests, upon which I worked in my PR #18655. I will be adding the following distributions during this phase:

- Borel Distribution (Discrete)
- Conway-Maxwell-Poisson Distribution (Discrete)
- Lomax Distribution (Continuous)
- Symmetric Pareto Distribution (Continuous)
- Bounded Pareto Distribution (Continuous)

All the above distributions will follow the same template as:

```
class NameofDistribution(SingleContinuousDistribution/SingleDiscreteDistribution):
    _argnames = (...)
    set = # the set on which the distribution is defined

    @staticmethod
    def check(...):
        # check the validity of the arguments passed

    def pdf(self, ...):
        # the probability density fucntion of the distribtuion

    # Add further ad-hoc methods only if integrate fails to produce
    # the expected results
```

An instance of adding such distributions is shown by the implementation of Bounded Pareto Distribution:

```
class BoundedParetoDistribution(SingleContinuousDistribution):
    _argnames = ('alpha', 'left', 'right')

    @property
    def set(self):
        return Interval(self.left, self.right)

    @staticmethod
    def check(alpha, left , right):
        _value_check(alpha.is_positive, "Shape must be positive.")
        _value_check(left.is_positive, "Left value should be positive.")
        _value_check(right > left, "Right should be greater than left.")

    def pdf(self, x):
        alpha, left, right = self.alpha, self.left, self.right
        num = alpha * (left**alpha) * x**(-alpha-1)
        den = 1 - (left/right)**alpha
        return num/den

def BoundedPareto(name, alpha, left, right):
    """Create a continuous random variable with a Bounded Pareto Distribution"""
    return rv(name, BoundedParetoDistribution, (alpha, left, right))
```

Example:

```
>>> from sympy.stats import *
>>> from sympy import symbols
>>> from sympy.abc import x
>>> L, H = symbols('L, H', positive=True)
>>> B = BoundedPareto('B', 1, L, H)
>>> E(B)
H*L*log(H)/(H - L) - H*L*log(L)/(H - L)
```

```
>>> cdf(B)(x)
Piecewise((-H*L/(x*(H - L)) + H/(H - L), L <= x), (0, True))
>>> density(B)(x)
L/(x**2*(1 - L/H))
```

Presently, `stats` provides symbolic classes of `Probability`, `Expectation`, `Variance` and `Covariance` in `symbolic_probability.py`. **The class `Probability` lacks `doit` method**. Therefore, I will work upon adding the `doit` method to this class during this phase. This can be implemented as:

```
def doit(self, **hints):
        expr = self.args[0]
        condition = self._condition
        if not expr.has(RandomSymbol):
            return expr

        if isinstance(expr, And):
            rv_combs = list(itertools.combinations(expr.args, 2))
            if all([not dependent(*rv_pair) for rv_pair in rv_combs]):
                return Mul(*[Probability(arg, condition).doit() for
                                arg in expr.args])

        if isinstance(expr, Or):
            return Add(*[Probability(arg, condition) for arg in expr.args])-
                                Probability(And(*expr.args), condition).doit()
        if condition is not None:
            return Probability(And(expr, condition)).doit()
                                /Probability(condition).doit()
        return self
```

Examples of using this `doit` method of class `Probability` are:

```
>>> from sympy import And, Or
>>> from sympy.stats import Probability, Bernoulli
>>> A = Bernoulli('A', 0.1)
>>> B = Bernoulli('B', 0.2)
>>> C = Bernoulli('C', 0.3)
>>> Probability(Or(A>0, B>0)).doit()
Probability(A > 0) - Probability(A > 0)*Probability(B > 0) + Probability(B > 0)
>>> Probability(And(A>0, B>0), C>0).doit()
Probability(A > 0)*Probability(B > 0)
```

Apart from these, I will also work upon adding the sampling methods of continuous, finite and discrete random variables from the external libraries. I am currently working upon adding the sampling methods for continuous random variables in my PR #18754. Once the implementation design is approved, I will add these methods for finite and discrete random variables.

2.3.2 Phase 1

During the first phase, I will work completely upon extending the support of Stochastic Processes and testing the existing algorithms.

A stochastic process means that one has a system for which there are observations at certain times, and that the outcome, that is, the observed value at each time is a random variable. It is concerned with concepts and techniques, and is oriented towards a broad spectrum of mathematical, scientific and engineering interests. Stochastic processes are widely used as mathematical models of systems and phenomena that appear to vary in a random manner.

I would follow the hierarchy flowchart as shown in the previous section and generalize some of the methods that are common to many of such processes. The functions or the data structure that are common to some of the process are implemented as:

```
dict_rv_subs = {
'Poisson': lambda rv: Poisson(rv.name, lamda=rv.pspace.process.lamda*rv.key),
'Wiener': lambda rv: Normal(rv.name, mean=0, std=sqrt(rv.key)),
'Bernoulli': lambda rv: Bernoulli(rv.name, p=rv.pspace.process.p,
'RandomWalk': lambda rv: Bernoulli(rv.name,p=rv.pspace.process.p,succ=1,fail=-1),
'Gamma': lambda rv: Gamma(rv.name, theta=1/rv.pspace.process.jump_size,
                                k=rv.pspace.process.jump_rate*rv.key)

        # This dictionary maps indexed RV with simple RV having same name and
        # distributionas of the indexed RV which helps in computing using the
        # functions that are presently implemented for simple RV
               }
```

The above dictionary is common to five processes and helps to create a map between the indexed and the simple random variable. Using this dictionary in the below function, `_rvindexed_subs` we will substitute the indexed RV.

```
def _rvindexed_subs(process, expr, condition=None):
        """
        Substitutes the RandomIndexedSymbol with the RandomSymbol with
        same name, distribution and probability as RandomIndexedSymbol.
        """

        rvs_expr = random_symbols(expr)
        if len(rvs_expr) != 0:
            swapdict_expr = {}
            for rv in rvs_expr:
                if isinstance(rv, RandomIndexedSymbol):
                    newrv = dict_rv_subs[process](rv)
                    swapdict_expr[rv] = newrv
            expr = expr.subs(swapdict_expr)

        rvs_cond = random_symbols(condition)
        if len(rvs_cond) != 0:
            swapdict_cond = {}
            for rv in rvs_cond:
                if isinstance(rv, RandomIndexedSymbol):
                    newrv = dict_rv_subs[process](rv)
                    swapdict_cond[rv] = newrv
            condition = condition.subs(swapdict_cond)
        return expr, condition
```

The following function is used to calculate the expectation of the expression having the indexed

random variable. This will calculate the expectation for the indexed RV of the processes in the `dict_rv_subs` with the aid of `dict_rv_subs` and `_rvindexed_subs`.

```
def _expectation_stoch(process, expr, condition=None, evaluate=True, **kwargs):
        """
        Computes expectation.

        Parameters
        ==========

        expr: RandomIndexedSymbol, Relational, Logic
            Condition for which expectation has to be computed. Must
            contain a RandomIndexedSymbol of the process.
        condition: Relational, Logic
            The given conditions under which computations should be done.

        Returns
        =======

        Expectation of the RandomIndexedSymbol.
        """
        new_expr, new_condition = _rvindexed_subs(process, expr, condition)
        new_pspace = pspace(new_expr)
        if new_condition is not None:
            new_expr = given(new_expr, new_condition)
        if new_expr.is_Add:  # As E is Linear
            return Add(*[new_pspace.compute_expectation(
                         expr=arg, evaluate=evaluate, **kwargs)
                         for arg in new_expr.args])
        return new_pspace.compute_expectation(
                new_expr, evaluate=evaluate, **kwargs)
```

Now, the following function is used to calculate the probability of the expression having the indexed random variable. This will calculate the probability of expression containing the indexed RV of the processes in the `dict_rv_subs` with the aid of `dict_rv_subs` and `_rvindexed_subs`.

```
def _probability_stoch(process,condition,given_condition=None,evaluate=True,**kw):
        """
        Computes probability.

        Parameters
        ==========

        condition: Relational
                Condition for which probability has to be computed. Must
                contain a RandomIndexedSymbol of the process.
        given_condition: Relational/And
                The given conditions under which computations should be done.

        Returns
        =======

        Probability of the condition.
```

```
    """
    new_condition, new_givencondition = _rvindexed_subs(process, condition,
                                                        given_condition)
    if isinstance(new_givencondition, RandomSymbol):
        condrv = random_symbols(new_condition)
        if len(condrv) == 1 and condrv[0] == new_givencondition:
            return BernoulliDistribution(_probability_stoch(process,
                                                            new_condition), 0, 1)
        if any([dependent(rv, new_givencondition) for rv in condrv]):
            return Probability(new_condition, new_givencondition)
        else:
            return _probability_stoch(process, new_condition)
    if new_givencondition is not None and \
            not isinstance(new_givencondition, (Relational, Boolean)):
        raise ValueError("%s is not a relational or combination of\
                relationals" % (new_givencondition))
    if new_givencondition is False:
        return S.Zero
    if new_condition is True:
        return S.One
    if new_condition is False:
        return S.Zero
    if not isinstance(new_condition, (Relational, Boolean)):
        raise ValueError("%s is not a relational or combination of\
                relationals" % (new_condition))
    if new_givencondition is not None: # If there is a condition
    # Recompute on new conditional expr
        return _probability_stoch(process, given(new_condition,
                                  new_givencondition, **kw), **kw)
    result = pspace(new_condition).probability(new_condition, **kw)
    if evaluate and hasattr(result, 'doit'):
        return result.doit()
    else:
        return result
```

Hence, these functions `_rvindexed_subs`, `_expectation_stoch`, `_probability_stoch` along with the `dict_rv_subs` helps in the expectation and probability computation of the expression involving indexed RV. This is done to avoid code repetition and maintain organized way of implementation.

Also, `joint_distribution` method of `class StochasticProcess` needs a minor fix in-order to produce the result in the case when the distribution of the process is time dependent as in case of Poisson, Wiener etc. This below change would make it work for all the cases.

```
if not args[0].pspace.process.distribution:
# checks if there is any distribution available
        return JointDistribution(*args)
```

I will now begin explaining the processes individually.

- **Poisson Process**

A Poisson Process is a model for a series of discrete event where the average time between events is known, but the exact timing of events is random. The arrival of an event is independent of the event before (waiting time between events is memory-less). Therefore, Poisson Process is a Discrete state Continuous Time Stochastic Process.

The probability of the random variable $N(t)$ following the Poisson process being equal to n is given by:

$$P\{N(t) = n\} = \frac{(\lambda t)^n \exp(-\lambda t)}{n!}$$

The implementation of Poisson Process is given in the following code snippet using the functions that are described above.

```
class PoissonProcess(ContinuousTimeStochasticProcess):
    index_set = _set_converter(Interval(0, oo))

    def __new__(cls, sym, lamda):
        sym = _symbol_converter(sym)
        lamda = _sympify(lamda)
        return Basic.__new__(cls, sym, lamda)

    @property
    def state_space(self):
        return S.Naturals0

    @property
    def symbol(self):
        return self.args[0]

    @property
    def lamda(self):
        return self.args[1]

    def distribution(self, rv):
        return PoissonDistribution(self.lamda*rv.key)

    def density(self, x):
        return (self.lamda*x.key)**x / factorial(x) * exp(-(self.lamda*x.key))

    def expectation(self, expr, condition=None, evaluate=True, **kwargs):
        """Computes expectation."""
        return _expectation_stoch('Poisson',expr,condition,evaluate,**kwargs)

    def probability(self, condition,given_condition=None,evaluate=True,**kwargs):
        """Computes probability."""
        return _probability_stoch('Poisson', condition, given_condition,
                                                    evaluate, **kwargs)
```

Following are the examples of using the Poisson Process:

```
>>> from sympy import Eq
>>> from sympy.stats import PoissonProcess, E, P
```

```
>>> X = PoissonProcess('X', 3)
>>> E(X(2))
6
>>> P(Eq(X(3), 2))
81*exp(-9)/2
>>> X.joint_distribution(X(1),  X(2))
JointDistributionHandmade(Lambda((X(1),  X(2)),
3**X(1)*6**X(2)*exp(-9)/(factorial(X(1))*factorial(X(2)))))
>>> N = X(1) + X(2) + X(3)
>>> E(N)
18
```

- **Wiener Process**

The Wiener process can be constructed as the scaling limit of a random walk, or other discrete-time stochastic processes with stationary independent increments. This is known as `Donsker's` theorem. The process has independent increments means that if $0 \leq s_1 < t_1 \leq s_2 < t_2$ then $W_{t_1} - W_{s_1}$ and $W_{t_2} - W_{s_2}$ are independent random variables, and the similar condition holds for n increments. Hence, the Wiener process is Continuous state Continuous time Stochastic Process.

The unconditional probability density function, which follows normal distribution with mean = 0 and variance = t, at a fixed time t:

$$f_{W_t}(x) = \frac{\exp(-x^2/2t)}{\sqrt{2\pi t}}$$

The implementation of the Wiener Process is given in the following code snippet:

```
class WienerProcess(ContinuousTimeStochasticProcess):
    index_set = _set_converter(Interval(0,oo))

    def __new__(cls, sym):
        sym = _symbol_converter(sym)
        return Basic.__new__(cls, sym)

    @property
    def state_space(self):
        return S.Reals

    @property
    def symbol(self):
        return self.args[0]

    def distribution(self, rv):
        return NormalDistribution(mean=0, std=sqrt(rv.key))

    def density(self, x):
        return exp(-(x)**2/(2*x.key)) / (sqrt(2*pi)*sqrt(x.key))

    def expectation(self, expr, condition=None, evaluate=True, **kwargs):
        return _expectation_stoch('Wiener',expr,condition,evaluate,**kwargs)

    def probability(self, condition,given_condition=None,evaluate=True,**kwargs):
```

```
        return _probability_stoch('Wiener', condition, given_condition,
                                                    evaluate, **kwargs)
```

Following are the examples of using the Wiener Process:

```
>>> from sympy.stats import WienerProcess, P , E, variance
>>> W = WienerProcess('W')
>>> E(W(4.67))
0
>>> variance(W(4))
4
>>> P(W(4.5) > 2)
0.122251191528291*sqrt(2)
>>> W.joint_distribution(W(3), W(5))
JointDistributionHandmade(Lambda((X(3), X(5)),
sqrt(15)*exp(-X(3)**2/6)*exp(-X(5)**2/10)/(30*pi)))
```

- **Random Walk**

A Random walk also known as a stochastic or random process, describes a path that consists of a succession of random steps on some mathematical space such as the integers. An elementary example of a random walk is the random walk on the integer number line, Z , which starts at 0 and at each step moves +1 or −1 with some probability. Hence, random walk is the Discrete state and Discrete Time Stochastic Process. The probability distribution of the random walk at some particular time can be interpreted as the Bernoulli Process with $succ = 1$ and $fail = -1$ and p being the probability of getting success.

$$P\{X(t) = 1\} = p \text{ and } P\{X(t) = -1\} = 1 - p$$

The implementation of the Random Walk is given in the following code snippet:

```
class RandomWalk(DiscreteTimeStochasticProcess):
    index_set = S.Naturals0

    def __new__(cls, sym, p):
        _value_check(p >= 0 and p <= 1, 'Value of p must be between 0 and 1.')
        sym = _symbol_converter(sym)
        p = _sympify(p)
        return Basic.__new__(cls, sym, p)

    @property
    def state_space(self):
        return S.Integers

    @property
    def symbol(self):
        return self.args[0]

    @property
    def p(self):
        return self.args[1]
```

```
    def distribution(self):
        return BernoulliDistribution(self.p, succ=1, fail=-1)

    def density(self, x):
        return Piecewise((self.p, Eq(x, 1)),
                         (1 - self.p, Eq(x, -1)),
                         (S.Zero, True))

    def path(self, position=0, steps=5):
        """
        Parameters
        ==========

        position : Integer
          Intial position
        steps: Positive Integer
          Number of steps required in the path.

        Returns
        =======

        path : list
          List of positions at a particular step.
        """
        path = []
        for step in range(steps):
            if random.random() <= self.p:
                position += 1
            else:
                position -=1
            path.append(position)
        return path

    def expectation(self, expr, condition=None, evaluate=True, **kwargs):
        return _expectation_stoch('RandomWalk',expr,condition,evaluate,**kwargs)

    def probability(self,condition,given_condition=None,evaluate=True,**kwargs):
        return _probability_stoch('RandomWalk', condition, given_condition,
                                                        evaluate, **kwargs)
```

The examples of the Random Walk are:

```
>>> from sympy.stats import RandomWalk, E, P
>>> from sympy import Eq
>>> R = RandomWalk('R', 0.6)
>>> E(3*R[4])
0.6
>>> R.path(steps=9)
[1, 0, 1, 0, 1, 2, 3, 4, 5] # Random results
>>> P(Eq(R[4], -1))
0.4
>>> progress = R[1] + R[2] + R[3]
>>> P(Eq(progress, 1))
```

```
0.432
>>> R.joint_distribution(R[8], R[10])
JointDistributionHandmade(Lambda((R[8], R[10]), Piecewise((0.6, Eq(R[10], 1)),
(0.4, Eq(R[10], -1)), (0, True))*Piecewise((0.6, Eq(R[8], 1)),
(0.4, Eq(R[8], -1)), (0, True))))
```

- **Gamma Process**

A Gamma process is a random process with independent gamma distributed increments. The parameter γ controls the rate of jump arrivals and the scaling parameter λ controls the jump size. The marginal distribution of a gamma process at time t is a gamma distribution with mean $\gamma t/\lambda$ and variance $\gamma t/\lambda^2$. Therefore, Gamma Process is the Continuous State Continuous Time Stochastic process. The probability density function f at some time t is given by

$$f(x;t,\lambda,\gamma) = \frac{\lambda^{\gamma t} x^{\gamma t-1} \exp(-\lambda x)}{\Gamma(\gamma t)}$$

The implementation of the Gamma Process is given in the following code snippet:

```
class GammaProcess(ContinuousTimeStochasticProcess):
    index_set = _set_converter(Interval(0,oo))

    def __new__(cls, sym, jump_size, jump_rate):
        sym = _symbol_converter(sym)
        jump_size = _sympify(jump_size)
        jump_rate = _sympify(jump_rate)
        return Basic.__new__(cls, sym, jump_size, jump_rate)

    @property
    def state_space(self):
        return _set_converter(Interval(0,oo))

    @property
    def symbol(self):
        return self.args[0]

    @property
    def jump_size(self):
        return self.args[1]

    @property
    def jump_rate(self):
        return self.args[2]

    def distribution(self, rv):
        return GammaDistribution(k=rv.key*self.jump_rate, theta=1/self.jump_size)

    def density(self, x):
        k = x.key*self.jump_rate
        theta = 1/self.jump_size
        return x**(k - 1) * exp(-x/theta) / (gamma(k)*theta**k)
```

```
    def expectation(self, expr, condition=None, evaluate=True, **kwargs):
        return _expectation_stoch('Gamma',expr,condition,evaluate,**kwargs)

    def probability(self,condition,given_condition=None,evaluate=True,**kwargs):
        return _probability_stoch('Gamma', condition, given_condition,
                                                        evaluate, **kwargs)
```

The examples of using Gamma Process are:

```
>>> from sympy.stats import GammaProcess, P, E, variance
>>> from sympy import symbols
>>> y, t, l = symbols('y, t, l', positive=True)
>>> G = GammaProcess('G', l, y)
>>> E(G(t))
t*y/l
>>> variance(G(t))
t*y/l**2
>>> G = GammaProcess('G', 2, 1)
>>> P(G(3) > 4)
41*exp(-8)
>>> G.joint_distribution(G(2), G(4))
JointDistributionHandmade(Lambda((G(2), G(4)),
32*exp(-2*G(2))*exp(-2*G(4))*G(2)*G(4)**3/3))
```

- **Birth Death Process**

The birth–death process is a special case of Continuous-time Markov process where the state transitions are of only two types: "birth", which increases the state variable by one and "deaths", which decreases the state by one. When a birth occurs, the process goes from state n to $n+1$. When a death occurs, the process goes from state n to state $n-1$. The process is specified by birth rates $\{\lambda_i\}_{i=1,2,\ldots,\infty}$and death rates $\{\mu_i\}_{i=1,2,\ldots,\infty}$. Hence, Birth-Death Process is the Discrete State Continuous Time Stochastic process. The implementation of the Birth Death Process is given in the following code snippet:

```
class BirthDeathProcess(ContinuousMarkovChain):

    def __new__(cls, sym, state_space, birth_rate=None, death_rate=None):
        sym = _symbol_converter(sym)
        state_space = _set_converter(state_space)
        if (birth_rate and death_rate) is not None:
            if state_space.is_finite_set and
        not (len(birth_rate) == len(death_rate) == len(state_space)):
                raise(ValueError, "Birth and death rates should\
                                                be defined for each state.")
            birth_rate = ImmutableMatrix(birth_rate)
            death_rate = ImmutableMatrix(death_rate)
        return Basic.__new__(cls, sym, state_space, birth_rate, death_rate)

    @property
    def birth_rate(self):
        return self.args[2]
```

```
    @property
    def death_rate(self):
        return self.args[3]

    @property
    def generator_matrix(self):
        if not self.state_space.is_finite_set:
            return None
        n = len(self.state_space)
        mat = zeros(n, n)
        mat[0, 1] = self.birth_rate[0]
        mat[0, 0] = -self.birth_rate[0]
        mat[n-1, n-1] = -self.death_rate[n-1]
        mat[n-1, n-2] = self.death_rate[n-1]
        for s in range(1, n-1):
            mat[s, s+1] = self.birth_rate[s]
            mat[s, s] = -(self.birth_rate[s] + self.death_rate[s])
            mat[s, s-1] = self.death_rate[s]
        return mat
```

Examples of using the Birth Death Process are:

```
>>> from sympy.stats import BirthDeathProcess, P, E
>>> from sympy import symbols, Eq
>>> B = BirthDeathProcess('B', [0,1], [1,1], [1,2])
>>> B.limiting_distribution()
Matrix([[2/3, 1/3]])
>>> A, t = B.generator_matrix, symbols('t', positive=True)
>>> B.transition_probabilities(A)(t)
Matrix([[2/3 + exp(-3*t)/3, 1/3 - exp(-3*t)/3],
[2/3 - 2*exp(-3*t)/3, 1/3 + 2*exp(-3*t)/3]])
>>> E(B(2), Eq(B(0), 1))
2*exp(-6)/3 + 1/3
>>> P(Eq(B(1), 1), Eq(B(0), 1))
2*exp(-3)/3 + 1/3
```

- **Queueing Process**

A queueing process is a model of waiting lines, constructed so that queue length and waiting times can be predicted. Networks of connected queues allow similar models for more complex situations where routing between queues plays a role. Queueing Process represents a queue with general arrival and service distributions. The Queueing Processes can be interpreted as the special cases of Birth Death process in which the birth rate represents the arrival rate and the service rate represents the death rate. The number of servers and the length of the queue depends on the type of the Queueing model. I will be working on adding the M/M/1 and M/M/1/N Queueing models and will discuss necessary changes in the API with mentor. Following is the code for both of these Queueing models:

```
class MMOne(BirthDeathProcess):
    def __new__(cls, sym, arrival_rate, service_rate):
        sym = _symbol_converter(sym)
        arrival_rate = _sympify(arrival_rate)
```

```
        service_rate = _sympify(service_rate)
        state_space = S.Naturals0
        return Basic.__new__(cls, sym, state_space, arrival_rate, service_rate)

    @property
    def arrival_rate(self):
        return self.args[2]

    @property
    def service_rate(self):
        return self.args[3]

    def limiting_distribution(self):
        return GeometricDistribution(self.arrival_rate/self.service_rate)

# Will be implementing expectation and probability methods for Queueing Process
```

For M/M/1/N Queueing models:

```
class MMOneN(BirthDeathProcess):
    def __new__(cls, sym, num_servers, arrival_rate, service_rate):
        sym = _symbol_converter(sym)
        num_servers = _sympify(num_servers)
        arrival_rate = _sympify(arrival_rate)
        service_rate = _sympify(service_rate)
        state_space = set(range(num_servers))
        birth_rate = [arrival_rate]*num_servers
        death_rate = [service_rate]*num_servers
        return BirthDeathProcess.__new__(cls, sym, state_space,
                                         birth_rate, death_rate)

    @property
    def num_servers(self):
        return len(self.args[1])

    @property
    def arrival_rate(self):
        return self.args[2][0]

    @property
    def service_rate(self):
        return self.args[3][0]

    def eff_arrival_rate(self):
        return self.arrival_rate*(S.One - self.limiting_distribution()
                                              [self.num_servers-1])

    def throughput(self):
        return self.service_rate*(S.One - self.limiting_distribution()[S.Zero])

# Will be implementing expectation and probability methods for Queueing Process
```

Examples of using both of these Queueing models are:

```
>>> from sympy.stats import MMOne, MMOneN
>>> M = MMOne('M', 1, 2)
>>> M.limiting_distribution()
GeometricDistribution(1/2)
>>> M = MMOneN('M', 3, 1, 2)
>>> M.limiting_distribution()
Matrix([[4/7, 2/7, 1/7]])
>>> M.eff_arrival_rate()
6/7
>>> M.throughput()
6/7
```

I will be adding these processes in `stochastic_process_types.py` and add the tests of each them in order to increase the code coverage and check the validity of the implemented code.

2.3.3 Phase 2

During this phase, I will work upon adding the Compound Distributions, create a well defined frame work and improved code quality for Joint Distributions.

- **Compound Distributions**

A compound distribution is the probability distribution that results from assuming that a random variable is distributed according to some parameterized distribution, with (some of) the parameters of that distribution themselves being random variables.

Presently, `SymPy` does support Compound Distributions, but the implementation is not well structured and not well defined. Compound Distribution is created in `joint_rv.py` and is provided with the same Probability Space as the Joint distributions, this also leads to a mix-up in the `joint_rv.py`.

I went through the PR #17036 by Ritesh Kumar in last year's GSoC who had implemented the similar logic, but can be used for only Normal Distributions(with only mean as RV) . Also the compound probability space created in that patch contains many methods that are not useful for the compound distributions . Therefore, it needs some refinement such as some methods like compute_expectation, marginal_distribution, etc. are not useful when this compound distribution is mapped to some continuous or discrete distribution. I would continue and complete the PR and make it generalize to be able to extend its scope by adding more examples on the top of the existing one. I will make two files, `compound_rv.py` and `compound_rv_types.py` similar to other stats files. The `compound_rv_types.py` will contain a dictionary that will map compound distributions to the corresponding Continuous or Discrete Distribution. After cleaning up Compound Probability space and I will create a new file for adding examples named `compound_rv_types.py` in the following way:

```
def create_compound(sym, dist):
    comp_dist = compound_rv_map[dist.compound_distribution.__class__.__name__]
                                                                (sym, dist)
    return comp_dist if comp_dist is not None else dist

compound_rv_map = {
    'NormalDistribution': lambda sym, dist: compute_normal(sym, dist),
    'PoissonDistribution': lambda sym, dist: compute_poisson(sym, dist),
```

```
    'ExponentialDistribution': lambda sym, dist: compute_exponential(sym, dist),
    'GammaDistribution': lambda sym, dist: compute_gamma(sym, dist)
                                }

def compute_normal(sym ,dist):
    mean, std = dist.compound_distribution.args
    if random_symbols(mean):
        from sympy.stats.crv_types import NormalDistribution
        if isinstance(mean.pspace.distribution, NormalDistribution):
            mu, sigma = mean.pspace.distribution.args
            return SingleContinuousPSpace(sym, NormalDistribution(mu,
                                sqrt(sigma ** 2 + std ** 2)))
    else:
        from sympy.stats.crv_types import (RayleighDistribution,
                                           LaplaceDistribution)
        if isinstance(std.pspace.distribution, RayleighDistribution):
            sigma = std.pspace.distribution.args[0]
            return SingleContinuousPSpace(sym, LaplaceDistribution(mean, sigma))

def compute_poisson(sym, dist):
    from sympy.stats.drv_types import NegativeBinomialDistribution
    from sympy.stats.crv_types import GammaDistribution
    rate = dist.compound_distribution.args[0]
    if isinstance(rate.pspace.distribution, GammaDistribution):
        k, theta = rate.pspace.distribution.args
        return SingleDiscretePSpace(sym, NegativeBinomialDistribution(k,
                                                theta/(1+theta)))

# Similarly will contain the functions for gamma, exponential and other such
# compounded distributions which can be extended easily.
```

This will be accessed by Compound Probability space with the help of function `create_compound` with the following change in the `__new__` method of CompoundPSpace.

```
class CompoundPSpace(ProductPSpace):
    """
    Represents a compound probability space. Represented using a distribution
    and parameter(s) of which one or more are distributed
    according to some distribution.
    """

    def __new__(cls, sym, dist):
        if isinstance(sym, str):
            sym = Symbol(sym)
        if not isinstance(sym, Symbol):
            raise TypeError("s should have been string or Symbol")
        dist = create_compound(sym, dist)
        if not isinstance(dist, CompoundDistribution):
            return dist
        return Basic.__new__(cls, sym, dist)
```

Using the above changes, we can use the compound distributions as shown in the following

examples:

```
>>> from sympy.stats import *
>>> Y = Rayleigh('Y', 4)
>>> X = Normal('X', 1, Y)
>>> X.pspace.distribution
LaplaceDistribution(1, 4)
>>> P(X > 1)
1/2
>>> Y = Gamma('Y', 3, 2)
>>> X = Poisson('X', Y)
>>> X.pspace.distribution
NegativeBinomialDistribution(3, 2/3)
>>> P(X > 4)
416/729
>>> Y = Normal('Y', 1, 2)
>>> X = Normal('X', Y, 2)
>>> P(X > 4)
3*(-4*sqrt(pi)*erf(3/4)/3 + 4*sqrt(pi)/3)/(8*sqrt(pi))
>>> X.pspace.distribution
NormalDistribution(1, 2*sqrt(2))
```

- **Joint Distributions**

Moving the Compound Distributions to a new file, will make `joint_rv.py` solely devoted to joint distributions.

The two classes namely `MultivariateNormalDistribution` and `MultivariateLaplaceDistribution` of joint distribution in `joint_rv_types.py` are **presently not mapped with the multivariate random variable**. They are presently created using continuous random variables of Normal and Laplace and also there are no tests in either of the test files for these distributions. I will map them with a multivariate random variable and use this random variable when someone tries to create a Multivariate RV using continuous Normal and Laplace random variables. Currently we have in `crv_types.py` as:

```
from sympy.stats.joint_rv_types import MultivariateNormalDistribution
        return multivariate_rv(
            MultivariateNormalDistribution, name, mean, std)
```

So, I will create a `MultivariateNormal` and `MultivariateLaplace` RV in `joint_rv_types.py`, as follows:

```
def MultivariateNormal(syms, mean, std):
    """Creates a joint random variable with multivariate normal distribution."""
    return multivariate_rv(MultivariateNormalDistribution, syms, mean, std)

def MultivariateLaplace(syms, mean, std):
    """Creates a joint random variable with multivariate laplace distribution."""
    return multivariate_rv(MultivariateLaplaceDistribution, syms, mean, std)
```

This will help in maintaining the uniformity throughout the `stats` module and also I will add the relevant tests for both of these classes.

The test file of `joint_rv.py`, i.e. `test_joint_rv.py` also needs attention as there are many tests currently which are tested using **string comparison** that needs to be changed to `dummy_eq`, also all

the imports are not moved to the top of the file, etc. Hence, `test_joint_rv.py` also needs code cleaning and improved code quality.

After creating a well defined framework and improved code quality in `joint_rv.py`, I will add more joint distributions in `joint_rv_types.py`. I will be following the same pattern of the current implementation and will add the distributions which will follow the below given template.

```
class NameofDistribution(JointDistribution):
    _argnames = (...)
    is_Continuous = True/False

    @property
    def set(self):
        pass
        # Interval/set on which distribution is defined

    @staticmethod
    def check(arguments):
        pass
        # check if the arguments passed are valid

    def pdf(self, *args):
        pass
        # probability density function of the distribution

    def marginal_distribution(self, indices, *syms):
        pass
        # override the marginal_distribution only if the JointDistribution fails
        # to produce it correctly
```

I will be adding the following joint distributions :

1. Wishart
2. Matrix Gamma
3. Normal Inverse Gamma
4. Inverse Wishart
5. Normal Wishart
6. Inverse Matrix Gamma

An instance of the Matrix Gamma is:

```
class MatrixGammaDistribution(JointDistribution):
    _argnames = ('alpha', 'beta', 'scale_matrix')
    is_Continuous = True

    @staticmethod
    def check(alpha, beta, scale_matrix):
        if not isinstance(scale_matrix, MatrixSymbol):
            _value_check(scale_matrix.is_positive_definite, "The shape\
                                    matrix must be positive definite. ")
```

```
        _value_check(scale_matrix.shape[0] == scale_matrix.shape[1], "Should\
                                be square matrix")
        _value_check(alpha.is_positive, "Shape parameter should be positive.")
        _value_check(beta.is_positive, "Scale parameter should be positive.")

    @property
    def set(self):
         k = self.scale_matrix.shape[0]
         return S.Reals**k

    def pdf(self, *args):
        alpha, beta, scale_matrix = self.alpha, self.beta, self.scale_matrix
        p = scale_matrix.shape[0]
        args = ImmutableMatrix(args)
        sigma_inv_x = -scale_matrix.inv()*args/beta
        return ((det(scale_matrix))**(-alpha))*(det(args)**(alpha -(p+1)/2))*\
            exp(sigma_inv_x.trace())/(beta**(p*alpha))/gamma(alpha)
```

While adding these distributions as well as the examples of compound distributions, I will make sure to add all the relevant tests and examples.

2.3.4 Phase 3

During this phase, I will work on adding the Mixture Distributions, adding Assumptions of dependence and adding densities of Circular ensembles in Random Matrices.

- **Mixture Distributions**

A mixture distribution is a mixture of two or more probability distributions. Random variables are drawn from more than one parent population to create a new distribution. The parent populations can be univariate or multivariate, although the mixed distributions should have the same dimensionality. In addition, they should either be all discrete probability distributions or all continuous probability distributions.

The distributions can be made up of different distributions (e.g. a normal distribution and a t-distribution) or they can be made up of the same distribution with different parameters. For adding the support of mixture distributions, I will create a new file consisting of all the functions and classes needed for Mixture Distributions. I will create a temporary Mixture Probability Space which will ultimately map to the corresponding, Continuous, discrete, finite or Joint probability space. The class `MixtureDistribution` will be containing methods such as set on which distribution is defined, probability density function, etc. The class `MixtureDistribution` will be accessible with the help of `Mixture` random variable. This can be done in the following by creating the new file `mixture_rv.py`:

```
def rv(symbol, cls, args):
    dist = cls(sympify(args))
    pspace = MixturePSpace(symbol, dist)
    return pspace.value

def mixture_map(sym, distribution):
    x = Symbol('x')
```

```
    mixt_map = {
    'is_Continuous': lambda sym, distribution: SingleContinuousPSpace(sym,
        ContinuousDistributionHandmade(distribution.pdf, distribution.set)),
    'is_Joint' : lambda sym, distribution: JointPSpace(sym,
        JointDistributionHandmade(distribution.pdf, distribution.set)),
    'is_Discrete' : lambda sym, distribution: SingleDiscretePSpace(sym,
        DiscreteDistributionHandmade(distribution.pdf, distribution.set)),
    'is_Finite': lambda sym, distribution: SingleFinitePSpace(sym,
        FiniteDistributionHandmade(distribution.pdf, distribution.set))
    }
    return mixt_map[dist_type_check(distribution)](sym, distribution)

def dist_type_check(dist):
    dist = list(dist.wt_dict)[0]
    if isinstance(dist.pspace.distribution, ContinuousDistribution):
            return 'is_Continuous'
    elif isinstance(dist.pspace.distribution, JointDistribution):
            return 'is_Joint'
    elif isinstance(dist.pspace.distribution, DiscreteDistribution):
            return 'is_Discrete'
    else:
            return 'is_Finite'

class MixturePSpace(PSpace):
    """This is just a temporary ProbabilitySpace as it will
    map to the original ProbabilitySpace based on the type of distribution
    passed.
    """
   def __new__(cls, sym, distribution):
        if isinstance(sym, str):
            sym = Symbol(sym)
        if not isinstance(sym, Symbol):
            raise TypeError("sym should have been string or Symbol")
        pspace = mixture_map(sym, distribution)
            return Basic.__new__(cls, sym, distribution, pspace)

    @property
   def value(self):
            return self.args[2].value

class MixtureDistribution(Basic, NamedArgsMixin):
    """Represents the Mixture distribution"""
   _argnames = ('wt_dict')

    def __new__(cls, wt_dict):
            wt_list = [_sympify(wt) for wt in wt_dict.values()]
            set_ = list(wt_dict)[0].pspace.domain.set
           for rv in wt_dict.keys():
                if not isinstance(rv, RandomSymbol):
                    raise TypeError("Each of element should be a random variable")
                if rv.pspace.domain.set != set_:
                    raise ValueError("Each random variable should be defined\
```

```
                        on same set")
            for wt in wt_list:
                if not wt.is_positive:
                    raise ValueError("Weight of each random variable should\
                                                    be positive")
            return Basic.__new__(cls, wt_dict)

    @property
    def set(self):
            return list(self.wt_dict)[0].pspace.distribution.set

    def pdf(self, x):
            y = Symbol('y')
            sum_wt = sum(self.wt_dict.values())
            wt_dict = {}
            for rv, wt in self.wt_dict.items():
                wt_dict[rv] = _sympify(self.wt_dict[rv]/sum_wt)
            pdf_ = S(0)
            if isinstance(list(wt_dict)[0].pspace.distribution,
                                        SingleFiniteDistribution):
              for rv in wt_dict.keys():
                pdf_ = pdf_ + wt_dict[rv]*rv.pspace.distribution.pmf(y)
            else:
              for rv in wt_dict.keys():
                pdf_ = pdf_ + wt_dict[rv]*rv.pspace.distribution.pdf(y)
            return Lambda(y, pdf_)(x)

def Mixture(name, wt_dict):
    """Creates a random variable with mixture distribution"""
    return rv(name, MixtureDistribution, wt_dict)
```

Examples of using the Mixture Distributions is:

```
>>> from sympy.stats import *
>>> M = Normal('M', 3, 4)
>>> N = Normal('N', 1, 2)
>>> X = Mixture('X', {N:3, M:5})
>>> E(X).simplify()
9/4
>>> P(X>0)
-5*erfc(3*sqrt(2)/8)/16 - 3*erfc(sqrt(2)/4)/16 + 1
>>> variance(X).simplify()
-1325*erfc(3*sqrt(2)/32)/256 - 1325*erf(3*sqrt(2)/32)/256
- 267*erfc(5*sqrt(2)/16)/256 - 267*erf(5*sqrt(2)/16)/256 + 597/32
```

- **Assumptions of Dependence**

Currently, `stats` **module does not support dependence of random variables**. The following code snippets shows that the density and expectation of the random variables does not evaluate and takes undefined time for computation.

```
>>> from sympy.stats import *
```

```
>>> from sympy import And
>>> N = Normal('N', 1, 2)
>>> M = Normal('M', 3, 4)
>>> density(N + M, And(N > 0, M > 3))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyboardInterrupt
>>> density(N + M, M > 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyboardInterrupt
>>> E(N + M, N > 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyboardInterrupt
```

An attempt was made in the last year's GSoC to add these dependencies in PR #17387 by Gagandeep Singh, I would continue this PR as the most common assumption of dependency is `Covariance` or correlation coefficient between two random variables. I would discuss more with the mentor regarding other assumptions and complete the PR by adding other required methods in the class `DependentPSpace` which include `compute_expectation`, `compute_cdf`, etc.

- **Mode and Pearson Mode Skewness**

The mode of the data is one of the most important statistical and analytical tool. Presently, `stats` module doesn't have any function to compute the mode of a random variable.

The mode of a set of data values is the value that appears most often. If X is a discrete random variable, the mode is the value $x(i.e, X = x)$ at which the probability mass function takes its maximum value. When the probability density function of a continuous distribution has multiple local maxima it is common to refer to all of the local maxima as modes of the distribution. A mode of a continuous probability distribution is often considered to be any value x at which its probability density function has a locally maximum value, so any peak is a mode.

The implementation of mode for continuous random variable is just by its definition according to which it is the value at which the derivative of the density function goes to zero. This will be done with separate cases, as for finite distributions, it will return the position at which the `pmf` is maximum, and for continuous it will solve using the derivative of the density function. For instance, the mode for the Normal distribution is same as its mean:

```
>>> from sympy.stats import Normal, density, Binomial
>>> from sympy import solveset, S, diff
>>> from sympy.abc import x, m, s
>>> X = Binomial('X', 4, S.Half)
>>> d = density(X).dict
>>> max(d, key=d.get)
2                       # mode of X
>>> N = Normal('N', m, s)
>>> d = density(N)(x)
>>> solveset(diff(d, x), x)
FiniteSet(m)            # mode of N
```

Pearson mode skewness helps to find out if we have positive or negative skewness. If we have a distribution and we know the mean, mode, and standard deviation (σ), then the Pearson mode skewness formula is:

$$\frac{(mean - mode)}{\sigma}$$

Therefore, after adding the function to calculate the mode of the random variable, I will be adding the function to calculate the Pearson mode skewness and find whether the distribution is positively or negatively skewed.

- **Circular Ensembles**

Circular Ensembles were introduced in Random Matrices in the last year's GSoC by Gagandeep Singh. Going through his report, I found that the densities of the Circular Ensembles are yet to be added as it involves Haar measure. Moreover, it can be implemented after reading this paper. I would go through this paper again during the summer and add the densities of the Circular Ensembles under `random_matrices.py`

2.3.5 Stretch Goal

I would like to add methods which create random plots of the Stochastic processes, which makes its easy to visualize the effects of various parameters passed while creating the instance of the process. I will discuss more on this with the mentor regarding the API and implementation details.

3 The Timeline

I will be completing the proposed project and make `stats` more powerful to handle the various functionalities by increasing its scope and making it more robust. I also assure that I will be devoting my summer vacation to this project. The present COVID-19 outbreak is expected to cause disruptions in my college academic schedule for the summer. Nevertheless, I have planned for eventuality and will take utmost care to make sure that it does not affect my project during the summer. My current semester examination is expected to complete around end of May, hence during this time, I will manage to devote 20-25 hours per week. It is expected to have summer vacation just before the official coding period begins and, therefore, I will be easily able to devote 45-50 hours per week solely to this project. I neither have any commitments during this summer, so I will be available to work even more if it is needed to complete the project within the time bounds. My next semester will commence from July 23. However, during beginning of the semester, I don't have much load and work, hence, I will be able to continue with the same pace and devote 40-45 hours per week during the final phase of the project. I will try to be ahead of deadlines so that we don't face pressure, however, if in case, due to some reasons, we lag behind, I will devote extra time to the project to cover it.

3.1 Community Bonding Period

Since I have been contributing to the library for the past four months, it would be easier for me to start working on my project and discuss the implementation details further with the mentor. I would also explore other modules during this phase that are related to `stats` such as `integrate`, `solvers`, etc. and get familiar with the functionalities supported by them.

3.1.1 Week 1 (May 4 - May 11)

- Discuss the API related to Mixture and Compound Distributions with the mentor, and study the codebase of stats and its helper modules- solvers and integrate.

3.1.2 Week 2 (May 11 - May 18)

- Adding Borel Distribution, Conway-Maxwell-Poisson Distribution, Lomax Distribution.
- Adding their test cases.

3.1.3 Week 3 (May 18 - May 25)

- Adding Symmetric Pareto Distribution, Bounded Pareto Distribution.
- Adding `doit` method in symbolic `Probability`.
- Adding tests cases of each of these implemented classes and functions.

3.1.4 Week 4 (May 25 - May 31)

- Finalize the API for adding sampling methods in PR #18754 and add the same for discrete and finite distributions.

3.2 Phase 1

3.2.1 Week 5 (June 1 - June 8)

- Testing the current framework and algorithms.
- Adding functions that are common to some process(generalized framework) to avoid code repetition.
- Adding Poisson Process and its test cases.

3.2.2 Week 6 (June 8 - June 15)

- Adding Gamma Process and its test cases.
- Adding Wiener Process and its test cases.

3.2.3 Week 7 (June 15 - June 22)

- Adding Random Walk and its test cases.
- Adding Birth Death Process and its test cases.

3.2.4 Week 8 (June 22 - June 29)

- Adding Queuing Processes and its types with their test cases.

3.3 Phase 2

3.3.1 Week 9 (June 29 - July 6)

- Creating a new file for Compound Distributions and adding support for it.

3.3.2 Week 10 (July 6 - July 13)

- Continuing the Compound Distributions as adding more examples and tests.
- Cleaning and Improving the Joint Distributions.

3.3.3 Week 11 (July 13 - July 20)

- Adding Wishart Distribution , Matrix Gamma Distribution, Normal Inverse Gamma Distribution and their test cases.

3.3.4 Week 12 (July 20 - July 27)

- Adding Inverse Wishart Distribution, Normal Wishart Distribution, Inverse Matrix Gamma Distribution and their test cases.

3.4 Phase 3

3.4.1 Week 13 (July 27 - August 3)

- Creating a new file and adding support for Mixture Distributions.

3.4.2 Week 14 (August 3 - August 10)

- Adding Assumptions of Dependence between the random variables and its examples and test cases.

3.4.3 Week 15 (August 10 - August 17)

- Adding mode and Pearson mode skewness functions and their test cases.

3.4.4 Week 16 (August 17 - August 24)

- Adding densities of Circular Ensembles.

3.5 Project Deliverables

- Complete proposed project plan within the given time bound.
- Complete Documentation of each functions or classes added following numpy docstyle.
- Test cases for each newly as well as improved functions or classes to increase the code coverage with an aim of above `95%` for the `stats` module.
- Code quality following the PEP8 style guide.

3.6 Post GSoC

I will firstly focus on the work that has remained incomplete during the project and try to complete it as soon as possible and get it merged. I would even like to continue to contribute to sympy as it matches my two major interests which include Mathematics and Computation. I would help the new contributors and also keep contributing to the one of the best Computer Algebra System-SymPy. Being contributing to this library for past few months, makes me proud to be a part of such a helpful and wonderful community. I will also do research regarding more robust techniques and features that can be added to `stats` and make it more powerful.

I would also like to thank **Gagandeep Singh**(@czgdp1807), **Aaron Meurer**(@asmeurer), **Kalevi Suominen**(@jksuom), **Christopher Smith**(@smichr), **Oscar Benjamin**(@oscarbenjamin), **S.Y. Lee**(@sylee957) and all other members of the community who helped me a lot in contributing to this library and provided motivation to learn and explore more in the library.

4 References

4.1 Community Bonding Period

- Borel Distribution
- Conway Maxwell Poisson Distribution
- Lomax Distribution
- Symmetric Pareto Distribution
- Bounded Pareto Distribution
- Sampling Methods
  - SciPy docs
  - PyMc docs
  - NumPy docs

4.2 Phase 1

- Stochastic Process-1
- Stochastic Process-2
- Random Walk
- Wiener Process
- Poisson Process
- Birth Death Process
- Gamma Process
- Queueing Process

4.3 Phase 2

- Compound Distribution-1
- Compound Distribution-2 -sympy PR #17036 by Kumar Ritesh
- Joint Distributions
  - Wishart Distribution
  - Normal-Wishart Distribution
  - Matrix Gamma Distribution
  - Normal-inverse-gamma Distribution
  - Inverse-Wishart Distribution
  - Inverse matrix Gamma Distribution

4.4 Phase 3

- Mixture Distribution-1
- Mixture Distribution-2
- Assumption of Dependence - sympy PR #17387 by Gagandeep Singh
- Mode
- Pearson Mode Skewness
- Circular Ensembles - Gagandeep Singh's report GSoC 2019
- Paper related to Circular Ensembles

4.5 Past Year Proposals

- Gagandeep Singh(@czgdp1807) GSOC-2019
- Jogi Miglani(@jmig5776) GSOC-2019
- Ritesh Kumar(@ritesh99rakesh) GSOC-2019

4.6 Potential Mentor

- **Gagandeep Singh -** @czgdp1807