```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
                                 2
                            -mu*z
                            -------
    mu      -mu  2*mu - 1  omega
2*mu  *omega    *z        *e
----------------------------------
            Gamma(mu)
```

```
>>> simplify(E(X))
sqrt(mu)*sqrt(omega)*gamma(mu + 1/2)/gamma(mu + 1)
```

```
>>> V = simplify(variance(X))
>>> pprint(V, use_unicode=False)
                     2
         omega*Gamma (mu + 1/2)
omega - -----------------------
        Gamma(mu)*Gamma(mu + 1)
```

```
>>> cdf(X)(z)
Piecewise((lowergamma(mu, mu*z**2/omega)/gamma(mu), z > 0),
        (0, True))
```

References

[R878]

sympy.stats.**Normal**(*name*, *mean*, *std*)

Create a continuous random variable with a Normal distribution.

Parameters

mu : Real number or a list representing the mean or the mean vector

sigma : Real number or a positive definite square matrix,

$\sigma^2 > 0$, the variance

Returns

RandomSymbol

Explanation

The density of the Normal distribution is given by

$$f(x) := \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Examples

```
>>> from sympy.stats import Normal, density, E, std, cdf, skewness,␣
↪quantile, marginal_distribution
>>> from sympy import Symbol, simplify, pprint
```

```
>>> mu = Symbol("mu")
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")
>>> y = Symbol("y")
>>> p = Symbol("p")
>>> X = Normal("x", mu, sigma)
```

```
>>> density(X)(z)
sqrt(2)*exp(-(-mu + z)**2/(2*sigma**2))/(2*sqrt(pi)*sigma)
```

```
>>> C = simplify(cdf(X))(z) # it needs a little more help...
>>> pprint(C, use_unicode=False)
   /  ___          \
   |\/ 2 *(-mu + z)|
erf|---------------|
   \    2*sigma    /   1
-------------------- + -
         2             2
```

```
>>> quantile(X)(p)
mu + sqrt(2)*sigma*erfinv(2*p - 1)
```

```
>>> simplify(skewness(X))
0
```

```
>>> X = Normal("x", 0, 1) # Mean 0, standard deviation 1
>>> density(X)(z)
sqrt(2)*exp(-z**2/2)/(2*sqrt(pi))
```

```
>>> E(2*X + 1)
1
```

```
>>> simplify(std(2*X + 1))
2
```

```
>>> m = Normal('X', [1, 2], [[2, 1], [1, 2]])
>>> pprint(density(m)(y, z), use_unicode=False)
          2          2
         y    y*z   z
       - -- + --- - -- + z - 1
         3     3    3
  ___
\/ 3 *e
------------------------------
             6*pi
```

```
>>> marginal_distribution(m, m[0])(1)
 1/(2*sqrt(pi))
```

References

[R879], [R880]

sympy.stats.**Pareto**(*name*, *xm*, *alpha*)

Create a continuous random variable with the Pareto distribution.

Parameters

xm : Real number, $x_m > 0$, a scale

alpha : Real number, $\alpha > 0$, a shape

Returns

RandomSymbol

Explanation

The density of the Pareto distribution is given by

$$f(x) := \frac{\alpha\, x_m^\alpha}{x^{\alpha+1}}$$

with $x \in [x_m, \infty]$.

Examples

```
>>> from sympy.stats import Pareto, density
>>> from sympy import Symbol
```

```
>>> xm = Symbol("xm", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Pareto("x", xm, beta)
```

```
>>> density(X)(z)
beta*xm**beta*z**(-beta - 1)
```

References

[R881], [R882]

sympy.stats.**PowerFunction**(*name*, *alpha*, *a*, *b*)

Creates a continuous random variable with a Power Function Distribution.

Parameters

alpha : Positive number, $0 < \alpha$, the shape paramater

a : Real number, $-\infty < a$, the left boundary

b : Real number, $a < b < \infty$, the right boundary

Returns

RandomSymbol

Explanation

The density of PowerFunction distribution is given by

$$f(x) := \frac{\alpha (x - a)^{\alpha - 1}}{(b - a)^{\alpha}}$$

with $x \in [a, b]$.

Examples

```
>>> from sympy.stats import PowerFunction, density, cdf, E, variance
>>> from sympy import Symbol
>>> alpha = Symbol("alpha", positive=True)
>>> a = Symbol("a", real=True)
>>> b = Symbol("b", real=True)
>>> z = Symbol("z")
```

```
>>> X = PowerFunction("X", 2, a, b)
```

```
>>> density(X)(z)
(-2*a + 2*z)/(-a + b)**2
```

```
>>> cdf(X)(z)
Piecewise((a**2/(a**2 - 2*a*b + b**2) - 2*a*z/(a**2 - 2*a*b + b**2) +
z**2/(a**2 - 2*a*b + b**2), a <= z), (0, True))
```

```
>>> alpha = 2
>>> a = 0
>>> b = 1
>>> Y = PowerFunction("Y", alpha, a, b)
```

```
>>> E(Y)
2/3
```

```
>>> variance(Y)
1/18
```

References

[R883]

sympy.stats.**QuadraticU**(*name*, *a*, *b*)

Create a Continuous Random Variable with a U-quadratic distribution.

Parameters

a : Real number

b : Real number, $a < b$

Returns

RandomSymbol

Explanation

The density of the U-quadratic distribution is given by

$$f(x) := \alpha(x - \beta)^2$$

with $x \in [a, b]$.

Examples

```
>>> from sympy.stats import QuadraticU, density
>>> from sympy import Symbol, pprint
```

```
>>> a = Symbol("a", real=True)
>>> b = Symbol("b", real=True)
>>> z = Symbol("z")
```

```
>>> X = QuadraticU("x", a, b)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
/                2
|   /  a   b    \
|12*|- - - - + z|
|   \  2   2    /
<-----------------  for And(b >= z, a <= z)
|            3
|    (-a + b)
|
\        0                 otherwise
```

References

[R884]

sympy.stats.**RaisedCosine**(*name*, *mu*, *s*)

Create a Continuous Random Variable with a raised cosine distribution.

Parameters

mu : Real number

s : Real number, $s > 0$

Returns

RandomSymbol

Explanation

The density of the raised cosine distribution is given by

$$f(x) := \frac{1}{2s}\left(1 + \cos\left(\frac{x - \mu}{s}\pi\right)\right)$$

with $x \in [\mu - s, \mu + s]$.

Examples

```
>>> from sympy.stats import RaisedCosine, density
>>> from sympy import Symbol, pprint
```

```
>>> mu = Symbol("mu", real=True)
>>> s = Symbol("s", positive=True)
>>> z = Symbol("z")
```

```
>>> X = RaisedCosine("x", mu, s)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
/   /pi*(-mu + z)\
|cos|------------| + 1
|   \     s      /
<---------------------  for And(z >= mu - s, z <= mu + s)
|         2*s
|
\          0                        otherwise
```

References

[R885]

sympy.stats.**Rayleigh**(*name*, *sigma*)

Create a continuous random variable with a Rayleigh distribution.

Parameters

sigma : Real number, $\sigma > 0$

Returns

RandomSymbol

Explanation

The density of the Rayleigh distribution is given by

$$f(x) := \frac{x}{\sigma^2} e^{-x^2/2\sigma^2}$$

with $x > 0$.

Examples

```
>>> from sympy.stats import Rayleigh, density, E, variance
>>> from sympy import Symbol
```

```
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Rayleigh("x", sigma)
```

```
>>> density(X)(z)
z*exp(-z**2/(2*sigma**2))/sigma**2
```

```
>>> E(X)
sqrt(2)*sqrt(pi)*sigma/2
```

```
>>> variance(X)
-pi*sigma**2/2 + 2*sigma**2
```

References

[R886], [R887]

sympy.stats.**Reciprocal**(*name*, *a*, *b*)

Creates a continuous random variable with a reciprocal distribution.

Parameters

a : Real number, $0 < a$

b : Real number, $a < b$

Returns
RandomSymbol

Examples

```
>>> from sympy.stats import Reciprocal, density, cdf
>>> from sympy import symbols
>>> a, b, x = symbols('a, b, x', positive=True)
>>> R = Reciprocal('R', a, b)
```

```
>>> density(R)(x)
1/(x*(-log(a) + log(b)))
>>> cdf(R)(x)
Piecewise((log(a)/(log(a) - log(b)) - log(x)/(log(a) - log(b)), a <= x),␣
↪(0, True))
```

Reference

sympy.stats.**StudentT**(*name*, *nu*)

Create a continuous random variable with a student's t distribution.

Parameters
nu : Real number, $\nu > 0$, the degrees of freedom

Returns
RandomSymbol

Explanation

The density of the student's t distribution is given by

$$f(x) := \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)}\left(1+\frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}}$$

Examples

```
>>> from sympy.stats import StudentT, density, cdf
>>> from sympy import Symbol, pprint
```

```
>>> nu = Symbol("nu", positive=True)
>>> z = Symbol("z")
```

```
>>> X = StudentT("x", nu)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
           nu   1
```

(continues on next page)

(continued from previous page)

```
         - -- - -
           2    2
 /     2\
 |    z |
 |1 + --|
 \    nu/
-----------------
  ____  /     nu\
\/ nu *B|1/2, --|
        \     2 /
```

```
>>> cdf(X)(z)
1/2 + z*gamma(nu/2 + 1/2)*hyper((1/2, nu/2 + 1/2), (3/2,),
                            -z**2/nu)/(sqrt(pi)*sqrt(nu)*gamma(nu/2))
```

References

[R889], [R890]

sympy.stats.**ShiftedGompertz**(*name*, *b*, *eta*)

Create a continuous random variable with a Shifted Gompertz distribution.

Parameters

b : Real number, $b > 0$, a scale

eta : Real number, $\eta > 0$, a shape

Returns

RandomSymbol

Explanation

The density of the Shifted Gompertz distribution is given by

$$f(x) := be^{-bx}e^{-\eta \exp(-bx)}\left[1 + \eta(1 - e^(-bx))\right]$$

with $x \in [0, \infty)$.

Examples

```
>>> from sympy.stats import ShiftedGompertz, density
>>> from sympy import Symbol
```

```
>>> b = Symbol("b", positive=True)
>>> eta = Symbol("eta", positive=True)
>>> x = Symbol("x")
```

```
>>> X = ShiftedGompertz("x", b, eta)
>>> density(X)(x)
b*(eta*(1 - exp(-b*x)) + 1)*exp(-b*x)*exp(-eta*exp(-b*x))
```

References

[R891]

sympy.stats.**Trapezoidal**(*name, a, b, c, d*)

Create a continuous random variable with a trapezoidal distribution.

Parameters

a : Real number, $a < d$

b : Real number, $a \le b < c$

c : Real number, $b < c \le d$

d : Real number

Returns

RandomSymbol

Explanation

The density of the trapezoidal distribution is given by

$$f(x) := \begin{cases} 0 & \text{for } x < a, \\ \frac{2(x-a)}{(b-a)(d+c-a-b)} & \text{for } a \le x < b, \\ \frac{2}{d+c-a-b} & \text{for } b \le x < c, \\ \frac{2(d-x)}{(d-c)(d+c-a-b)} & \text{for } c \le x < d, \\ 0 & \text{for } d < x. \end{cases}$$

Examples

```
>>> from sympy.stats import Trapezoidal, density
>>> from sympy import Symbol, pprint
```

```
>>> a = Symbol("a")
>>> b = Symbol("b")
>>> c = Symbol("c")
>>> d = Symbol("d")
>>> z = Symbol("z")
```

```
>>> X = Trapezoidal("x", a,b,c,d)
```

```
>>> pprint(density(X)(z), use_unicode=False)
/        -2*a + 2*z
|-------------------------  for And(a <= z, b > z)
|(-a + b)*(-a - b + c + d)
```

(continues on next page)

(continued from previous page)

```
|
|           2
|     --------------         for And(b <= z, c > z)
<     -a - b + c + d
|
|        2*d - 2*z
|-------------------------   for And(d >= z, c <= z)
|(-c + d)*(-a - b + c + d)
|
\            0                      otherwise
```

References

[R892]

sympy.stats.**Triangular**(*name*, *a*, *b*, *c*)

Create a continuous random variable with a triangular distribution.

Parameters

a : Real number, $a \in (-\infty, \infty)$

b : Real number, $a < b$

c : Real number, $a \leq c \leq b$

Returns

RandomSymbol

Explanation

The density of the triangular distribution is given by

$$f(x) := \begin{cases} 0 & \text{for } x < a, \\ \frac{2(x-a)}{(b-a)(c-a)} & \text{for } a \leq x < c, \\ \frac{2}{b-a} & \text{for } x = c, \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{for } c < x \leq b, \\ 0 & \text{for } b < x. \end{cases}$$

Examples

```
>>> from sympy.stats import Triangular, density
>>> from sympy import Symbol, pprint
```

```
>>> a = Symbol("a")
>>> b = Symbol("b")
>>> c = Symbol("c")
>>> z = Symbol("z")
>>> X = Triangular("x", a,b,c)
```

```
>>> pprint(density(X)(z), use_unicode=False)
/    -2*a + 2*z
|-----------------  for And(a <= z, c > z)
|(-a + b)*(-a + c)
|
|       2
|     ------              for c = z
<     -a + b
|
|   2*b - 2*z
|----------------   for And(b >= z, c < z)
|(-a + b)*(b - c)
|
\        0                otherwise
```

References

[R893], [R894]

sympy.stats.**Uniform**(*name*, *left*, *right*)

Create a continuous random variable with a uniform distribution.

Parameters

a : Real number, $-\infty < a$, the left boundary

b : Real number, $a < b < \infty$, the right boundary

Returns

RandomSymbol

Explanation

The density of the uniform distribution is given by

$$f(x) := \begin{cases} \frac{1}{b-a} & \text{for } x \in [a,b] \\ 0 & \text{otherwise} \end{cases}$$

with $x \in [a,b]$.

Examples

```
>>> from sympy.stats import Uniform, density, cdf, E, variance
>>> from sympy import Symbol, simplify
```

```
>>> a = Symbol("a", negative=True)
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")
>>> X = Uniform("x", a, b)
```

```
>>> density(X)(z)
Piecewise((1/(-a + b), (b >= z) & (a <= z)), (0, True))
```

```
>>> cdf(X)(z)
Piecewise((0, a > z), ((-a + z)/(-a + b), b >= z), (1, True))
```

```
>>> E(X)
a/2 + b/2
```

```
>>> simplify(variance(X))
a**2/12 - a*b/6 + b**2/12
```

References

[R895], [R896]

sympy.stats.**UniformSum**(*name*, *n*)

Create a continuous random variable with an Irwin-Hall distribution.

Parameters

n : A positive integer, $n > 0$

Returns

RandomSymbol

Explanation

The probability distribution function depends on a single parameter n which is an integer.

The density of the Irwin-Hall distribution is given by

$$f(x) := \frac{1}{(n-1)!}\sum_{k=0}^{\lfloor x\rfloor}(-1)^k\binom{n}{k}(x-k)^{n-1}$$

Examples

```
>>> from sympy.stats import UniformSum, density, cdf
>>> from sympy import Symbol, pprint
```

```
>>> n = Symbol("n", integer=True)
>>> z = Symbol("z")
```

```
>>> X = UniformSum("x", n)
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
floor(z)
  ___
  \  `
   \         k         n - 1 /n\
    )    (-1) *(-k + z)     *| |
   /                         \k/
  /__,
 k = 0
--------------------------------
            (n - 1)!
```

```
>>> cdf(X)(z)
Piecewise((0, z < 0), (Sum((-1)**_k*(-_k + z)**n*binomial(n, _k),
                (_k, 0, floor(z)))/factorial(n), n >= z), (1, True))
```

Compute cdf with specific 'x' and 'n' values as follows : >>> cdf(UniformSum("x", 5), evaluate=False)(2).doit() 9/40

The argument evaluate=False prevents an attempt at evaluation of the sum for general n, before the argument 2 is passed.

References

[R897], [R898]

sympy.stats.**VonMises**(*name*, *mu*, *k*)

Create a Continuous Random Variable with a von Mises distribution.

Parameters

mu : Real number

Measure of location.

k : Real number

Measure of concentration.

Returns

RandomSymbol

Explanation

The density of the von Mises distribution is given by

$$f(x) := \frac{e^{\kappa\cos(x-\mu)}}{2\pi I_0(\kappa)}$$

with $x \in [0, 2\pi]$.

Examples

```
>>> from sympy.stats import VonMises, density
>>> from sympy import Symbol, pprint
```

```
>>> mu = Symbol("mu")
>>> k = Symbol("k", positive=True)
>>> z = Symbol("z")
```

```
>>> X = VonMises("x", mu, k)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
     k*cos(mu - z)
    e
------------------
2*pi*besseli(0, k)
```

References

[R899], [R900]

sympy.stats.**Wald**(*name*, *mean*, *shape*)

Create a continuous random variable with an Inverse Gaussian distribution. Inverse Gaussian distribution is also known as Wald distribution.

Parameters

mu :

Positive number representing the mean.

lambda :

Positive number representing the shape parameter.

Returns

RandomSymbol

Explanation

The density of the Inverse Gaussian distribution is given by

$$f(x) := \sqrt{\frac{\lambda}{2\pi x^3}} e^{-\frac{\lambda(x-\mu)^2}{2x\mu^2}}$$

Examples

```
>>> from sympy.stats import GaussianInverse, density, E, std, skewness
>>> from sympy import Symbol, pprint
```

```
>>> mu = Symbol("mu", positive=True)
>>> lamda = Symbol("lambda", positive=True)
>>> z = Symbol("z", positive=True)
>>> X = GaussianInverse("x", mu, lamda)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
                                      2
                  -lambda*(-mu + z)
                  -------------------
                               2
                        2*mu *z
  ___   ________
\/ 2 *\/ lambda *e
-------------------------------------
                  ____  3/2
            2*\/ pi *z
```

```
>>> E(X)
mu
```

```
>>> std(X).expand()
mu**(3/2)/sqrt(lambda)
```

```
>>> skewness(X).expand()
3*sqrt(mu)/sqrt(lambda)
```

References

[R901], [R902]

sympy.stats.**Weibull**(*name*, *alpha*, *beta*)

Create a continuous random variable with a Weibull distribution.

Parameters

lambda : Real number, $\lambda > 0$, a scale

k : Real number, $k > 0$, a shape

Returns

RandomSymbol

Explanation

The density of the Weibull distribution is given by

$$f(x) := \begin{cases} \frac{k}{\lambda}\left(\frac{x}{\lambda}\right)^{k-1} e^{-(x/\lambda)^{k}} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Examples

```
>>> from sympy.stats import Weibull, density, E, variance
>>> from sympy import Symbol, simplify
```

```
>>> l = Symbol("lambda", positive=True)
>>> k = Symbol("k", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Weibull("x", l, k)
```

```
>>> density(X)(z)
k*(z/lambda)**(k - 1)*exp(-(z/lambda)**k)/lambda
```

```
>>> simplify(E(X))
lambda*gamma(1 + 1/k)
```

```
>>> simplify(variance(X))
lambda**2*(-gamma(1 + 1/k)**2 + gamma(1 + 2/k))
```

References

[R903], [R904]

sympy.stats.**WignerSemicircle**(*name*, *R*)

Create a continuous random variable with a Wigner semicircle distribution.

Parameters

R : Real number, $R > 0$, the radius

Returns

A RandomSymbol.

Explanation

The density of the Wigner semicircle distribution is given by

$$f(x) := \frac{2}{\pi R^2} \sqrt{R^2 - x^2}$$

with $x \in [-R, R]$.

Examples

```
>>> from sympy.stats import WignerSemicircle, density, E
>>> from sympy import Symbol
```

```
>>> R = Symbol("R", positive=True)
>>> z = Symbol("z")
```

```
>>> X = WignerSemicircle("x", R)
```

```
>>> density(X)(z)
2*sqrt(R**2 - z**2)/(pi*R**2)
```

```
>>> E(X)
0
```

References

[R905], [R906]

sympy.stats.**ContinuousRV**(*symbol, density, set=Interval(-oo, oo), **kwargs*)

Create a Continuous Random Variable given the following:

Parameters

symbol : Symbol

Represents name of the random variable.

density : Expression containing symbol

Represents probability density function.

set : set/Interval

Represents the region where the pdf is valid, by default is real line.

check : bool

If True, it will check whether the given density integrates to 1 over the given set. If False, it will not perform this check. Default is False.

Returns

RandomSymbol

Many common continuous random variable types are already implemented.

This function should be necessary only very rarely.

Examples

```
>>> from sympy import Symbol, sqrt, exp, pi
>>> from sympy.stats import ContinuousRV, P, E
```

```
>>> x = Symbol("x")
```

```
>>> pdf = sqrt(2)*exp(-x**2/2)/(2*sqrt(pi)) # Normal distribution
>>> X = ContinuousRV(x, pdf)
```

```
>>> E(X)
0
>>> P(X>0)
1/2
```

Joint Types

`sympy.stats.`**`JointRV`**`(`*symbol, pdf, _set=None*`)`

Create a Joint Random Variable where each of its component is continuous, given the following:

Parameters

symbol : Symbol

Represents name of the random variable.

pdf : A PDF in terms of indexed symbols of the symbol given

as the first argument

Returns

RandomSymbol

Note

As of now, the set for each component for a `JointRV` is equal to the set of all integers, which cannot be changed.

Examples

```
>>> from sympy import exp, pi, Indexed, S
>>> from sympy.stats import density, JointRV
>>> x1, x2 = (Indexed('x', i) for i in (1, 2))
>>> pdf = exp(-x1**2/2 + x1 - x2**2/2 - S(1)/2)/(2*pi)
>>> N1 = JointRV('x', pdf) #Multivariate Normal distribution
>>> density(N1)(1, 2)
exp(-2)/(2*pi)
```

sympy.stats.**marginal_distribution**(*rv*, **indices*)

Marginal distribution function of a joint random variable.

Parameters

rv : A random variable with a joint probability distribution.

indices : Component indices or the indexed random symbol

for which the joint distribution is to be calculated

Returns

A Lambda expression in sym.

Examples

```
>>> from sympy.stats import MultivariateNormal, marginal_distribution
>>> m = MultivariateNormal('X', [1, 2], [[2, 1], [1, 2]])
>>> marginal_distribution(m, m[0])(1)
1/(2*sqrt(pi))
```

sympy.stats.**MultivariateNormal**(*name*, *mu*, *sigma*)

Creates a continuous random variable with Multivariate Normal Distribution.

The density of the multivariate normal distribution can be found at [1].

Parameters

mu : List representing the mean or the mean vector

sigma : Positive semidefinite square matrix

Represents covariance Matrix. If σ is noninvertible then only sampling is supported currently

Returns

RandomSymbol

Examples

```
>>> from sympy.stats import MultivariateNormal, density, marginal_
↪distribution
>>> from sympy import symbols, MatrixSymbol
>>> X = MultivariateNormal('X', [3, 4], [[2, 1], [1, 2]])
>>> y, z = symbols('y z')
>>> density(X)(y, z)
sqrt(3)*exp(-y**2/3 + y*z/3 + 2*y/3 - z**2/3 + 5*z/3 - 13/3)/(6*pi)
>>> density(X)(1, 2)
sqrt(3)*exp(-4/3)/(6*pi)
>>> marginal_distribution(X, X[1])(y)
exp(-(y - 4)**2/4)/(2*sqrt(pi))
>>> marginal_distribution(X, X[0])(y)
exp(-(y - 3)**2/4)/(2*sqrt(pi))
```

The example below shows that it is also possible to use symbolic parameters to define the MultivariateNormal class.

```
>>> n = symbols('n', integer=True, positive=True)
>>> Sg = MatrixSymbol('Sg', n, n)
>>> mu = MatrixSymbol('mu', n, 1)
>>> obs = MatrixSymbol('obs', n, 1)
>>> X = MultivariateNormal('X', mu, Sg)
```

The density of a multivariate normal can be calculated using a matrix argument, as shown below.

```
>>> density(X)(obs)
(exp(((1/2)*mu.T - (1/2)*obs.T)*Sg**(-1)*(-mu + obs))/
↪sqrt((2*pi)**n*Determinant(Sg)))[0, 0]
```

References

[R907]

sympy.stats.**MultivariateLaplace**(*name, mu, sigma*)

Creates a continuous random variable with Multivariate Laplace Distribution.

The density of the multivariate Laplace distribution can be found at [1].

Parameters

mu : List representing the mean or the mean vector

sigma : Positive definite square matrix

Represents covariance Matrix

Returns

RandomSymbol

Examples

```
>>> from sympy.stats import MultivariateLaplace, density
>>> from sympy import symbols
>>> y, z = symbols('y z')
>>> X = MultivariateLaplace('X', [2, 4], [[3, 1], [1, 3]])
>>> density(X)(y, z)
sqrt(2)*exp(y/4 + 5*z/4)*besselk(0, sqrt(15*y*(3*y/8 - z/8)/2 + 15*z*(-y/
↪8 + 3*z/8)/2))/(4*pi)
>>> density(X)(1, 2)
sqrt(2)*exp(11/4)*besselk(0, sqrt(165)/4)/(4*pi)
```

References

[R908]

sympy.stats.**GeneralizedMultivariateLogGamma**(*syms*, *delta*, *v*, *lamda*, *mu*)

Creates a joint random variable with generalized multivariate log gamma distribution.

The joint pdf can be found at [1].

Parameters

syms : list/tuple/set of symbols for identifying each component

delta : A constant in range $[0, 1]$

v : Positive real number

lamda : List of positive real numbers

mu : List of positive real numbers

Returns

RandomSymbol

Examples

```
>>> from sympy.stats import density
>>> from sympy.stats.joint_rv_types import␣
↪GeneralizedMultivariateLogGamma
>>> from sympy import symbols, S
>>> v = 1
>>> l, mu = [1, 1, 1], [1, 1, 1]
>>> d = S.Half
>>> y = symbols('y_1:4', positive=True)
>>> Gd = GeneralizedMultivariateLogGamma('G', d, v, l, mu)
>>> density(Gd)(y[0], y[1], y[2])
Sum(exp((n + 1)*(y_1 + y_2 + y_3) - exp(y_1) - exp(y_2) -
exp(y_3))/(2**n*gamma(n + 1)**3), (n, 0, oo))/2
```

Note

If the GeneralizedMultivariateLogGamma is too long to type use,

```
>>> from sympy.stats.joint_rv_types import␣
↪GeneralizedMultivariateLogGamma as GMVLG
>>> Gd = GMVLG('G', d, v, l, mu)
```

If you want to pass the matrix omega instead of the constant delta, then use `GeneralizedMultivariateLogGammaOmega`.

References

[R909], [R910]

`sympy.stats.GeneralizedMultivariateLogGammaOmega`(*syms*, *omega*, *v*, *lamda*, *mu*)

Extends GeneralizedMultivariateLogGamma.

Parameters

syms : list/tuple/set of symbols

For identifying each component

omega : A square matrix

Every element of square matrix must be absolute value of square root of correlation coefficient

v : Positive real number

lamda : List of positive real numbers

mu : List of positive real numbers

Returns

RandomSymbol

Examples

```
>>> from sympy.stats import density
>>> from sympy.stats.joint_rv_types import␣
↪GeneralizedMultivariateLogGammaOmega
>>> from sympy import Matrix, symbols, S
>>> omega = Matrix([[1, S.Half, S.Half], [S.Half, 1, S.Half], [S.Half, S.
↪Half, 1]])
>>> v = 1
>>> l, mu = [1, 1, 1], [1, 1, 1]
>>> G = GeneralizedMultivariateLogGammaOmega('G', omega, v, l, mu)
>>> y = symbols('y_1:4', positive=True)
>>> density(G)(y[0], y[1], y[2])
sqrt(2)*Sum((1 - sqrt(2)/2)**n*exp((n + 1)*(y_1 + y_2 + y_3) - exp(y_1) -
exp(y_2) - exp(y_3))/gamma(n + 1)**3, (n, 0, oo))/2
```

Notes

If the GeneralizedMultivariateLogGammaOmega is too long to type use,

```
>>> from sympy.stats.joint_rv_types import␣
↪GeneralizedMultivariateLogGammaOmega as GMVLGO
>>> G = GMVLGO('G', omega, v, l, mu)
```

References

[R911], [R912]

sympy.stats.**Multinomial**(*syms*, *n*, **p*)

Creates a discrete random variable with Multinomial Distribution.

The density of the said distribution can be found at [1].

Parameters

n : Positive integer

Represents number of trials

p : List of event probabilites

Must be in the range of $[0, 1]$.

Returns

RandomSymbol

Examples

```
>>> from sympy.stats import density, Multinomial, marginal_distribution
>>> from sympy import symbols
>>> x1, x2, x3 = symbols('x1, x2, x3', nonnegative=True, integer=True)
>>> p1, p2, p3 = symbols('p1, p2, p3', positive=True)
>>> M = Multinomial('M', 3, p1, p2, p3)
>>> density(M)(x1, x2, x3)
Piecewise((6*p1**x1*p2**x2*p3**x3/
↪(factorial(x1)*factorial(x2)*factorial(x3)),
Eq(x1 + x2 + x3, 3)), (0, True))
>>> marginal_distribution(M, M[0])(x1).subs(x1, 1)
3*p1*p2**2 + 6*p1*p2*p3 + 3*p1*p3**2
```

References

[R913], [R914]

sympy.stats.**MultivariateBeta**(*syms*, **alpha*)

Creates a continuous random variable with Dirichlet/Multivariate Beta Distribution.

The density of the Dirichlet distribution can be found at [1].

Parameters

alpha : Positive real numbers

Signifies concentration numbers.

Returns

RandomSymbol

Examples

```
>>> from sympy.stats import density, MultivariateBeta, marginal_
↪distribution
>>> from sympy import Symbol
>>> a1 = Symbol('a1', positive=True)
>>> a2 = Symbol('a2', positive=True)
>>> B = MultivariateBeta('B', [a1, a2])
>>> C = MultivariateBeta('C', a1, a2)
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> density(B)(x, y)
x**(a1 - 1)*y**(a2 - 1)*gamma(a1 + a2)/(gamma(a1)*gamma(a2))
>>> marginal_distribution(C, C[0])(x)
x**(a1 - 1)*gamma(a1 + a2)/(a2*gamma(a1)*gamma(a2))
```

References

[R915], [R916]

sympy.stats.**MultivariateEwens**(*syms, n, theta*)

Creates a discrete random variable with Multivariate Ewens Distribution.

The density of the said distribution can be found at [1].

Parameters

n : Positive integer

Size of the sample or the integer whose partitions are considered

theta : Positive real number

Denotes Mutation rate

Returns

RandomSymbol

Examples

```
>>> from sympy.stats import density, marginal_distribution,␣
↪MultivariateEwens
>>> from sympy import Symbol
>>> a1 = Symbol('a1', positive=True)
>>> a2 = Symbol('a2', positive=True)
>>> ed = MultivariateEwens('E', 2, 1)
>>> density(ed)(a1, a2)
Piecewise((1/(2**a2*factorial(a1)*factorial(a2)), Eq(a1 + 2*a2, 2)), (0,␣
↪True))
>>> marginal_distribution(ed, ed[0])(a1)
Piecewise((1/factorial(a1), Eq(a1, 2)), (0, True))
```

References

[R917], [R918]

`sympy.stats.MultivariateT(syms, mu, sigma, v)`

Creates a joint random variable with multivariate T-distribution.

Parameters

syms : A symbol/str

For identifying the random variable.

mu : A list/matrix

Representing the location vector

sigma : The shape matrix for the distribution

Returns

RandomSymbol

Examples

```
>>> from sympy.stats import density, MultivariateT
>>> from sympy import Symbol
```

```
>>> x = Symbol("x")
>>> X = MultivariateT("x", [1, 1], [[1, 0], [0, 1]], 2)
```

```
>>> density(X)(1, 2)
2/(9*pi)
```

`sympy.stats.NegativeMultinomial(syms, k0, *p)`

Creates a discrete random variable with Negative Multinomial Distribution.

The density of the said distribution can be found at [1].

Parameters

k0 : positive integer

Represents number of failures before the experiment is stopped

p : List of event probabilites

Must be in the range of $[0, 1]$

Returns

RandomSymbol

Examples

```
>>> from sympy.stats import density, NegativeMultinomial, marginal_
↪distribution
>>> from sympy import symbols
>>> x1, x2, x3 = symbols('x1, x2, x3', nonnegative=True, integer=True)
>>> p1, p2, p3 = symbols('p1, p2, p3', positive=True)
>>> N = NegativeMultinomial('M', 3, p1, p2, p3)
>>> N_c = NegativeMultinomial('M', 3, 0.1, 0.1, 0.1)
>>> density(N)(x1, x2, x3)
p1**x1*p2**x2*p3**x3*(-p1 - p2 - p3 + 1)**3*gamma(x1 + x2 +
x3 + 3)/(2*factorial(x1)*factorial(x2)*factorial(x3))
>>> marginal_distribution(N_c, N_c[0])(1).evalf().round(2)
0.25
```

References

[R919], [R920]

`sympy.stats.`**`NormalGamma`**(*sym, mu, lamda, alpha, beta*)

Creates a bivariate joint random variable with multivariate Normal gamma distribution.

Parameters

sym : A symbol/str

For identifying the random variable.

mu : A real number

The mean of the normal distribution

lamda : A positive integer

Parameter of joint distribution

alpha : A positive integer

Parameter of joint distribution

beta : A positive integer

Parameter of joint distribution

Returns

RandomSymbol

Examples

```
>>> from sympy.stats import density, NormalGamma
>>> from sympy import symbols
```

```
>>> X = NormalGamma('x', 0, 1, 2, 3)
>>> y, z = symbols('y z')
```

```
>>> density(X)(y, z)
9*sqrt(2)*z**(3/2)*exp(-3*z)*exp(-y**2*z/2)/(2*sqrt(pi))
```

References

[R921]

Stochastic Processes

class sympy.stats.**DiscreteMarkovChain**(*sym*, *state_space=None*, *trans_probs=None*)

Represents a finite discrete time-homogeneous Markov chain.

This type of Markov Chain can be uniquely characterised by its (ordered) state space and its one-step transition probability matrix.

Parameters

sym:

The name given to the Markov Chain

state_space:

Optional, by default, Range(n)

trans_probs:

Optional, by default, MatrixSymbol('_T', n, n)

Examples

```
>>> from sympy.stats import DiscreteMarkovChain, TransitionMatrixOf, P, E
>>> from sympy import Matrix, MatrixSymbol, Eq, symbols
>>> T = Matrix([[0.5, 0.2, 0.3],[0.2, 0.5, 0.3],[0.2, 0.3, 0.5]])
>>> Y = DiscreteMarkovChain("Y", [0, 1, 2], T)
>>> YS = DiscreteMarkovChain("Y")
```

```
>>> Y.state_space
{0, 1, 2}
>>> Y.transition_probabilities
Matrix([
[0.5, 0.2, 0.3],
[0.2, 0.5, 0.3],
[0.2, 0.3, 0.5]])
>>> TS = MatrixSymbol('T', 3, 3)
>>> P(Eq(YS[3], 2), Eq(YS[1], 1) & TransitionMatrixOf(YS, TS))
T[0, 2]*T[1, 0] + T[1, 1]*T[1, 2] + T[1, 2]*T[2, 2]
>>> P(Eq(Y[3], 2), Eq(Y[1], 1)).round(2)
0.36
```

Probabilities will be calculated based on indexes rather than state names. For example, with the Sunny-Cloudy-Rainy model with string state names:

```
>>> from sympy.core.symbol import Str
>>> Y = DiscreteMarkovChain("Y", [Str('Sunny'), Str('Cloudy'), Str('Rainy
↪')], T)
>>> P(Eq(Y[3], 2), Eq(Y[1], 1)).round(2)
0.36
```

This gives the same answer as the `[0, 1, 2]` state space. Currently, there is no support for state names within probability and expectation statements. Here is a work-around using `Str`:

```
>>> P(Eq(Str('Rainy'), Y[3]), Eq(Y[1], Str('Cloudy'))).round(2)
0.36
```

Symbol state names can also be used:

```
>>> sunny, cloudy, rainy = symbols('Sunny, Cloudy, Rainy')
>>> Y = DiscreteMarkovChain("Y", [sunny, cloudy, rainy], T)
>>> P(Eq(Y[3], rainy), Eq(Y[1], cloudy)).round(2)
0.36
```

Expectations will be calculated as follows:

```
>>> E(Y[3], Eq(Y[1], cloudy))
0.38*Cloudy + 0.36*Rainy + 0.26*Sunny
```

Probability of expressions with multiple RandomIndexedSymbols can also be calculated provided there is only 1 RandomIndexedSymbol in the given condition. It is always better to use Rational instead of floating point numbers for the probabilities in the transition matrix to avoid errors.

```
>>> from sympy import Gt, Le, Rational
>>> T = Matrix([[Rational(5, 10), Rational(3, 10), Rational(2, 10)],␣
↪[Rational(2, 10), Rational(7, 10), Rational(1, 10)], [Rational(3, 10),␣
↪Rational(3, 10), Rational(4, 10)]])
>>> Y = DiscreteMarkovChain("Y", [0, 1, 2], T)
>>> P(Eq(Y[3], Y[1]), Eq(Y[0], 0)).round(3)
0.409
>>> P(Gt(Y[3], Y[1]), Eq(Y[0], 0)).round(2)
0.36
>>> P(Le(Y[15], Y[10]), Eq(Y[8], 2)).round(7)
0.6963328
```

Symbolic probability queries are also supported

```
>>> a, b, c, d = symbols('a b c d')
>>> T = Matrix([[Rational(1, 10), Rational(4, 10), Rational(5, 10)],␣
↪[Rational(3, 10), Rational(4, 10), Rational(3, 10)], [Rational(7, 10),␣
↪Rational(2, 10), Rational(1, 10)]])
>>> Y = DiscreteMarkovChain("Y", [0, 1, 2], T)
>>> query = P(Eq(Y[a], b), Eq(Y[c], d))
>>> query.subs({a:10, b:2, c:5, d:1}).round(4)
0.3096
>>> P(Eq(Y[10], 2), Eq(Y[5], 1)).evalf().round(4)
0.3096
>>> query_gt = P(Gt(Y[a], b), Eq(Y[c], d))
>>> query_gt.subs({a:21, b:0, c:5, d:0}).evalf().round(5)
0.64705
>>> P(Gt(Y[21], 0), Eq(Y[5], 0)).round(5)
0.64705
```

There is limited support for arbitrarily sized states:

```
>>> n = symbols('n', nonnegative=True, integer=True)
>>> T = MatrixSymbol('T', n, n)
>>> Y = DiscreteMarkovChain("Y", trans_probs=T)
>>> Y.state_space
Range(0, n, 1)
>>> query = P(Eq(Y[a], b), Eq(Y[c], d))
>>> query.subs({a:10, b:2, c:5, d:1})
(T**5)[1, 2]
```

References

[R922], [R923]

absorbing_probabilities()

Computes the absorbing probabilities, i.e. the ij-th entry of the matrix denotes the probability of Markov chain being absorbed in state j starting from state i.

canonical_form() → Tuple[List[*Basic* (page 927)], *ImmutableDenseMatrix* (page 1369)]

Reorders the one-step transition matrix so that recurrent states appear first and transient states appear last. Other representations include inserting transient states first and recurrent states last.

Returns

states, P_new

states is the list that describes the order of the new states in the matrix so that the ith element in states is the state of the ith row of A. P_new is the new transition matrix in canonical form.

Examples

```
>>> from sympy.stats import DiscreteMarkovChain
>>> from sympy import Matrix, S
```

You can convert your chain into canonical form:

```
>>> T = Matrix([[S(1)/2, S(1)/2, 0,      0,      0],
...             [S(2)/5, S(1)/5, S(2)/5, 0,      0],
...             [0,      0,      1,      0,      0],
...             [0,      0,      S(1)/2, S(1)/2, 0],
...             [S(1)/2, 0,      0,      0, S(1)/2]])
>>> X = DiscreteMarkovChain('X', list(range(1, 6)), trans_probs=T)
>>> states, new_matrix = X.canonical_form()
>>> states
[3, 1, 2, 4, 5]
```

```
>>> new_matrix
Matrix([
[  1,   0,   0,   0,   0],
[  0, 1/2, 1/2,   0,   0],
```

(continues on next page)

(continued from previous page)

```
[2/5, 2/5, 1/5,   0,   0],
[1/2,   0,   0, 1/2,   0],
[  0, 1/2,   0,   0, 1/2]])
```

The new states are [3, 1, 2, 4, 5] and you can create a new chain with this and its canonical form will remain the same (since it is already in canonical form).

```
>>> X = DiscreteMarkovChain('X', states, new_matrix)
>>> states, new_matrix = X.canonical_form()
>>> states
[3, 1, 2, 4, 5]
```

```
>>> new_matrix
Matrix([
[  1,   0,   0,   0,   0],
[  0, 1/2, 1/2,   0,   0],
[2/5, 2/5, 1/5,   0,   0],
[1/2,   0,   0, 1/2,   0],
[  0, 1/2,   0,   0, 1/2]])
```

This is not limited to absorbing chains:

```
>>> T = Matrix([[0, 5,  5, 0,  0],
...             [0, 0,  0, 10, 0],
...             [5, 0,  5, 0,  0],
...             [0, 10, 0, 0,  0],
...             [0, 3,  0, 3,  4]])/10
>>> X = DiscreteMarkovChain('X', trans_probs=T)
>>> states, new_matrix = X.canonical_form()
>>> states
[1, 3, 0, 2, 4]
```

```
>>> new_matrix
Matrix([
[   0,    1,   0,   0,   0],
[   1,    0,   0,   0,   0],
[ 1/2,    0,   0, 1/2,   0],
[   0,    0, 1/2, 1/2,   0],
[3/10, 3/10,   0,   0, 2/5]])
```

See also:

sympy.stats.DiscreteMarkovChain.communication_classes (page 2948), *sympy.stats.DiscreteMarkovChain.decompose* (page 2949)

References

[R924], [R925]

communication_classes() → List[Tuple[List[*Basic* (page 927)], *Boolean* (page 1163), *Integer* (page 987)]]

Returns the list of communication classes that partition the states of the markov chain.

A communication class is defined to be a set of states such that every state in that set is reachable from every other state in that set. Due to its properties this forms a class in the mathematical sense. Communication classes are also known as recurrence classes.

Returns

classes

The `classes` are a list of tuples. Each tuple represents a single communication class with its properties. The first element in the tuple is the list of states in the class, the second element is whether the class is recurrent and the third element is the period of the communication class.

Examples

```
>>> from sympy.stats import DiscreteMarkovChain
>>> from sympy import Matrix
>>> T = Matrix([[0, 1, 0],
...             [1, 0, 0],
...             [1, 0, 0]])
>>> X = DiscreteMarkovChain('X', [1, 2, 3], T)
>>> classes = X.communication_classes()
>>> for states, is_recurrent, period in classes:
...     states, is_recurrent, period
([1, 2], True, 2)
([3], False, 1)
```

From this we can see that states 1 and 2 communicate, are recurrent and have a period of 2. We can also see state 3 is transient with a period of 1.

Notes

The algorithm used is of order `O(n**2)` where `n` is the number of states in the markov chain. It uses Tarjan's algorithm to find the classes themselves and then it uses a breadth-first search algorithm to find each class's periodicity. Most of the algorithm's components approach `O(n)` as the matrix becomes more and more sparse.

References

[R926], [R927], [R928], [R929]

decompose() → Tuple[List[*Basic* (page 927)], *ImmutableDenseMatrix* (page 1369), *ImmutableDenseMatrix* (page 1369), *ImmutableDenseMatrix* (page 1369)]

Decomposes the transition matrix into submatrices with special properties.

The transition matrix can be decomposed into 4 submatrices: - A - the submatrix from recurrent states to recurrent states. - B - the submatrix from transient to recurrent states. - C - the submatrix from transient to transient states. - O - the submatrix of zeros for recurrent to transient states.

Returns

states, A, B, C

`states` - a list of state names with the first being the recurrent states and the last being the transient states in the order of the row names of A and then the row names of C. `A` - the submatrix from recurrent states to recurrent states. `B` - the submatrix from transient to recurrent states. `C` - the submatrix from transient to transient states.

Examples

```
>>> from sympy.stats import DiscreteMarkovChain
>>> from sympy import Matrix, S
```

One can decompose this chain for example:

```
>>> T = Matrix([[S(1)/2, S(1)/2, 0,      0,      0],
...             [S(2)/5, S(1)/5, S(2)/5, 0,      0],
...             [0,      0,      1,      0,      0],
...             [0,      0,      S(1)/2, S(1)/2, 0],
...             [S(1)/2, 0,      0,      0, S(1)/2]])
>>> X = DiscreteMarkovChain('X', trans_probs=T)
>>> states, A, B, C = X.decompose()
>>> states
[2, 0, 1, 3, 4]
```

```
>>> A   # recurrent to recurrent
Matrix([[1]])
```

```
>>> B  # transient to recurrent
Matrix([
[  0],
[2/5],
[1/2],
[  0]])
```

```
>>> C  # transient to transient
Matrix([
[1/2, 1/2,   0,   0],
[2/5, 1/5,   0,   0],
```

(continues on next page)

(continued from previous page)

```
[  0,   0, 1/2,   0],
[1/2,   0,   0, 1/2]])
```

This means that state 2 is the only absorbing state (since A is a 1x1 matrix). B is a 4x1 matrix since the 4 remaining transient states all merge into reccurent state 2. And C is the 4x4 matrix that shows how the transient states 0, 1, 3, 4 all interact.

See also:

sympy.stats.DiscreteMarkovChain.communication_classes (page 2948), *sympy.stats.DiscreteMarkovChain.canonical_form* (page 2946)

References

[R930], [R931]

fixed_row_vector()

A wrapper for `stationary_distribution()`.

fundamental_matrix()

Each entry fundamental matrix can be interpreted as the expected number of times the chains is in state j if it started in state i.

References

[R932]

property limiting_distribution

The fixed row vector is the limiting distribution of a discrete Markov chain.

sample()

Returns

sample: iterator object

iterator object containing the sample

stationary_distribution(*condition_set=False*) → Union[*ImmutableDenseMatrix* (page 1369), *ConditionSet* (page 1215), *Lambda* (page 1039)]

The stationary distribution is any row vector, p, that solves p = pP, is row stochastic and each element in p must be nonnegative. That means in matrix form: $(P-I)^T p^T = 0$ and $(1,\ldots,1)p = 1$ where P is the one-step transition matrix.

All time-homogeneous Markov Chains with a finite state space have at least one stationary distribution. In addition, if a finite time-homogeneous Markov Chain is irreducible, the stationary distribution is unique.

Parameters

condition_set : bool

If the chain has a symbolic size or transition matrix, it will return a `Lambda` if `False` and return a `ConditionSet` if `True`.

Examples

```
>>> from sympy.stats import DiscreteMarkovChain
>>> from sympy import Matrix, S
```

An irreducible Markov Chain

```
>>> T = Matrix([[S(1)/2, S(1)/2, 0],
...             [S(4)/5, S(1)/5, 0],
...             [1, 0, 0]])
>>> X = DiscreteMarkovChain('X', trans_probs=T)
>>> X.stationary_distribution()
Matrix([[8/13, 5/13, 0]])
```

A reducible Markov Chain

```
>>> T = Matrix([[S(1)/2, S(1)/2, 0],
...             [S(4)/5, S(1)/5, 0],
...             [0, 0, 1]])
>>> X = DiscreteMarkovChain('X', trans_probs=T)
>>> X.stationary_distribution()
Matrix([[8/13 - 8*tau0/13, 5/13 - 5*tau0/13, tau0]])
```

```
>>> Y = DiscreteMarkovChain('Y')
>>> Y.stationary_distribution()
Lambda((wm, _T), Eq(wm*_T, wm))
```

```
>>> Y.stationary_distribution(condition_set=True)
ConditionSet(wm, Eq(wm*_T, wm))
```

See also:

sympy.stats.DiscreteMarkovChain.limiting_distribution (page 2950)

References

[R933], [R934]

property transition_probabilities

Transition probabilities of discrete Markov chain, either an instance of Matrix or MatrixSymbol.

class sympy.stats.**ContinuousMarkovChain**(*sym, state_space=None, gen_mat=None*)

Represents continuous time Markov chain.

Parameters

sym : Symbol/str

state_space : Set

Optional, by default, S.Reals

gen_mat : Matrix/ImmutableMatrix/MatrixSymbol

Optional, by default, None

Examples

```
>>> from sympy.stats import ContinuousMarkovChain, P
>>> from sympy import Matrix, S, Eq, Gt
>>> G = Matrix([[-S(1), S(1)], [S(1), -S(1)]])
>>> C = ContinuousMarkovChain('C', state_space=[0, 1], gen_mat=G)
>>> C.limiting_distribution()
Matrix([[1/2, 1/2]])
>>> C.state_space
{0, 1}
>>> C.generator_matrix
Matrix([
[-1,  1],
[ 1, -1]])
```

Probability queries are supported

```
>>> P(Eq(C(1.96), 0), Eq(C(0.78), 1)).round(5)
0.45279
>>> P(Gt(C(1.7), 0), Eq(C(0.82), 1)).round(5)
0.58602
```

Probability of expressions with multiple RandomIndexedSymbols can also be calculated provided there is only 1 RandomIndexedSymbol in the given condition. It is always better to use Rational instead of floating point numbers for the probabilities in the generator matrix to avoid errors.

```
>>> from sympy import Gt, Le, Rational
>>> G = Matrix([[-S(1), Rational(1, 10), Rational(9, 10)], [Rational(2,␣
↪5), -S(1), Rational(3, 5)], [Rational(1, 2), Rational(1, 2), -S(1)]])
>>> C = ContinuousMarkovChain('C', state_space=[0, 1, 2], gen_mat=G)
>>> P(Eq(C(3.92), C(1.75)), Eq(C(0.46), 0)).round(5)
0.37933
>>> P(Gt(C(3.92), C(1.75)), Eq(C(0.46), 0)).round(5)
0.34211
>>> P(Le(C(1.57), C(3.14)), Eq(C(1.22), 1)).round(4)
0.7143
```

Symbolic probability queries are also supported

```
>>> from sympy import symbols
>>> a,b,c,d = symbols('a b c d')
>>> G = Matrix([[-S(1), Rational(1, 10), Rational(9, 10)], [Rational(2,␣
↪5), -S(1), Rational(3, 5)], [Rational(1, 2), Rational(1, 2), -S(1)]])
>>> C = ContinuousMarkovChain('C', state_space=[0, 1, 2], gen_mat=G)
>>> query = P(Eq(C(a), b), Eq(C(c), d))
>>> query.subs({a:3.65, b:2, c:1.78, d:1}).evalf().round(10)
0.4002723175
>>> P(Eq(C(3.65), 2), Eq(C(1.78), 1)).round(10)
0.4002723175
>>> query_gt = P(Gt(C(a), b), Eq(C(c), d))
>>> query_gt.subs({a:43.2, b:0, c:3.29, d:2}).evalf().round(10)
0.6832579186
```

(continues on next page)

(continued from previous page)

```
>>> P(Gt(C(43.2), 0), Eq(C(3.29), 2)).round(10)
0.6832579186
```

References

[R935], [R936]

class sympy.stats.**BernoulliProcess**(*sym, p, success=1, failure=0*)

The Bernoulli process consists of repeated independent Bernoulli process trials with the same parameter p. It's assumed that the probability p applies to every trial and that the outcomes of each trial are independent of all the rest. Therefore Bernoulli Processs is Discrete State and Discrete Time Stochastic Process.

Parameters

sym : Symbol/str

success : Integer/str

The event which is considered to be success. Default: 1.

failure: Integer/str

The event which is considered to be failure. Default: 0.

p : Real Number between 0 and 1

Represents the probability of getting success.

Examples

```
>>> from sympy.stats import BernoulliProcess, P, E
>>> from sympy import Eq, Gt
>>> B = BernoulliProcess("B", p=0.7, success=1, failure=0)
>>> B.state_space
{0, 1}
>>> (B.p).round(2)
0.70
>>> B.success
1
>>> B.failure
0
>>> X = B[1] + B[2] + B[3]
>>> P(Eq(X, 0)).round(2)
0.03
>>> P(Eq(X, 2)).round(2)
0.44
>>> P(Eq(X, 4)).round(2)
0
>>> P(Gt(X, 1)).round(2)
0.78
>>> P(Eq(B[1], 0) & Eq(B[2], 1) & Eq(B[3], 0) & Eq(B[4], 1)).round(2)
0.04
>>> B.joint_distribution(B[1], B[2])
```

(continues on next page)

(continued from previous page)

```
JointDistributionHandmade(Lambda((B[1], B[2]), Piecewise((0.7, Eq(B[1],␣
↪1)),
(0.3, Eq(B[1], 0)), (0, True))*Piecewise((0.7, Eq(B[2], 1)), (0.3,␣
↪Eq(B[2], 0)),
(0, True))))
>>> E(2*B[1] + B[2]).round(2)
2.10
>>> P(B[1] < 1).round(2)
0.30
```

References

[R937], [R938]

expectation(*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.

probability(*condition, given_condition=None, evaluate=True, **kwargs*)

Computes probability.

Parameters

condition : Relational

Condition for which probability has to be computed. Must contain a RandomIndexedSymbol of the process.

given_condition : Relational, Logic

The given conditions under which computations should be done.

Returns

Probability of the condition.

class sympy.stats.**PoissonProcess**(*sym, lamda*)

The Poisson process is a counting process. It is usually used in scenarios where we are counting the occurrences of certain events that appear to happen at a certain rate, but completely at random.

Parameters

sym : Symbol/str

lamda : Positive number

Rate of the process, `lambda > 0`

Examples

```
>>> from sympy.stats import PoissonProcess, P, E
>>> from sympy import symbols, Eq, Ne, Contains, Interval
>>> X = PoissonProcess("X", lamda=3)
>>> X.state_space
Naturals0
>>> X.lamda
3
>>> t1, t2 = symbols('t1 t2', positive=True)
>>> P(X(t1) < 4)
(9*t1**3/2 + 9*t1**2/2 + 3*t1 + 1)*exp(-3*t1)
>>> P(Eq(X(t1), 2) | Ne(X(t1), 4), Contains(t1, Interval.Ropen(2, 4)))
1 - 36*exp(-6)
>>> P(Eq(X(t1), 2) & Eq(X(t2), 3), Contains(t1, Interval.Lopen(0, 2))
... & Contains(t2, Interval.Lopen(2, 4)))
648*exp(-12)
>>> E(X(t1))
3*t1
>>> E(X(t1)**2 + 2*X(t2),  Contains(t1, Interval.Lopen(0, 1))
... & Contains(t2, Interval.Lopen(1, 2)))
18
>>> P(X(3) < 1, Eq(X(1), 0))
exp(-6)
>>> P(Eq(X(4), 3), Eq(X(2), 3))
exp(-6)
>>> P(X(2) <= 3, X(1) > 1)
5*exp(-3)
```

Merging two Poisson Processes

```
>>> Y = PoissonProcess("Y", lamda=4)
>>> Z = X + Y
>>> Z.lamda
7
```

Splitting a Poisson Process into two independent Poisson Processes

```
>>> N, M = Z.split(l1=2, l2=5)
>>> N.lamda, M.lamda
(2, 5)
```

References

[R939], [R940]

class sympy.stats.**WienerProcess**(*sym*)

The Wiener process is a real valued continuous-time stochastic process. In physics it is used to study Brownian motion and it is often also called Brownian motion due to its historical connection with physical process of the same name originally observed by Scottish botanist Robert Brown.

Parameters

sym : Symbol/str

Examples

```
>>> from sympy.stats import WienerProcess, P, E
>>> from sympy import symbols, Contains, Interval
>>> X = WienerProcess("X")
>>> X.state_space
Reals
>>> t1, t2 = symbols('t1 t2', positive=True)
>>> P(X(t1) < 7).simplify()
erf(7*sqrt(2)/(2*sqrt(t1)))/2 + 1/2
>>> P((X(t1) > 2) | (X(t1) < 4), Contains(t1, Interval.Ropen(2, 4))).
↪simplify()
-erf(1)/2 + erf(2)/2 + 1
>>> E(X(t1))
0
>>> E(X(t1) + 2*X(t2),  Contains(t1, Interval.Lopen(0, 1))
... & Contains(t2, Interval.Lopen(1, 2)))
0
```

References

[R941], [R942]

class sympy.stats.**GammaProcess**(*sym, lamda, gamma*)

A Gamma process is a random process with independent gamma distributed increments. It is a pure-jump increasing Levy process.

Parameters

sym : Symbol/str

lamda : Positive number

Jump size of the process, `lamda > 0`

gamma : Positive number

Rate of jump arrivals, $\gamma > 0$

Examples

```
>>> from sympy.stats import GammaProcess, E, P, variance
>>> from sympy import symbols, Contains, Interval, Not
>>> t, d, x, l, g = symbols('t d x l g', positive=True)
>>> X = GammaProcess("X", l, g)
>>> E(X(t))
g*t/l
>>> variance(X(t)).simplify()
g*t/l**2
>>> X = GammaProcess('X', 1, 2)
```

(continues on next page)