

Compendium of Distributions, I: Beta, Binomial, Chi-Square, F, Gamma, Geometric, Poisson, Student's t, and Uniform.

Maurice HT Ling

School of Chemical and Life Sciences, Singapore Polytechnic, Singapore

Department of Zoology, The University of Melbourne, Australia

mauriceling@acm.org

Abstract

This manuscript illustrates the implementation and testing of nine statistical distributions, namely Beta, Binomial, Chi-Square, F, Gamma, Geometric, Poisson, Student's t and Uniform distribution, where each distribution consists of three common functions – Probability Density Function (PDF), Cumulative Density Function (CDF) and the inverse of CDF (inverseCDF).

1. Description

Statistical distributions play a central role in statistical inferences to provide a probabilistic measure for use in hypothesis testing. As such, the implementation of statistical distributions and functions is fundamental to high-throughput scientific analyses. This manuscript illustrates the implementation of nine statistical distributions (McLaughlin, 2001); namely Beta, Binomial, Chi-Square, F, Gamma, Geometric, Poisson, Student's t and Uniform; where each distribution consists of three common functions – Probability Density Function (PDF), Cumulative Density Function (CDF) and the inverse of CDF (inverseCDF) – as modelled after Ling (2009). Of these nine distributions presented in this manuscript, two are discrete distributions (Binomial and Poisson) whereas the rest are continuous distributions (Beta, Chi-Square, F, Gamma, Geometric, Student's t and Uniform).

Each distribution can be briefly described as follows:

- Beta distribution is a continuous distribution bounded between zero and one. This constraint rendered its use to model the probability of event occurrences (Keefer et al., 1993) or approximating the value of variables (Haskett et al., 1995).
- Binomial distribution is a discrete distribution commonly used to model the outcomes of a series of experiments with only two possible outcomes per experiment (Van der Geest, 2005).
- Chi-Square distribution is a continuous distribution commonly used to measure the association between two categorical variables (Ugoni and Walker, 1995).
- F distribution (Crofts, 1982) is a joint distribution of two independent variables, each having a Chi-Square distribution. Gamma distribution (Jambunathan, 1954) is related to Chi-Square distribution and had been used in likelihood estimation (Rogers, 2001).

- Geometric distribution is related to Binomial distribution and is commonly used to estimate the probability of the first success or failure in a series of experiments (Miller and Carroll, 1989).
- Poisson distribution is binomial distribution optimized for highly unbalanced occurrences between the two possible outcomes.
- Student's t distribution is Normal distribution catered for small sample size.
- Uniform distribution is used to describe situations whereby all possible outcomes have the same probability of occurrence, like tossing an unbiased dice.

Given that the equation of a distribution is $p(x)$ and area under a distribution is standardized to 1: the Cumulative Density Function (CDF) of value x is the area under the distribution bounded by negative infinity to x ; the Probability Density Function (PDF) is the probability of x for discrete distributions and between $x-h$ and $x+h$ for continuous distribution where h is a small float number; the inverse of CDF (inverseCDF) gives the value of x when given a probability.

$$CDF(x) = \int_{-\infty}^x p(x) dx$$

$$PDF(x) = \int_{x-h}^{x+h} p(x) dx$$

These codes are licensed under Lesser General Public Licence version 3.

2. Code Files

The implementation of the distributions and testing codes are presented in 3 files:

- StatisticsDistribution.py file contains the implementation of each distribution.
- NRPy.py file contains a set of numerical functions from Loredo (2000) and *Numerical Recipes in Pascal* (Press et al., 1989) used by StatisticsDistribution.py.
- DistributionTest.py file contains the test codes for each distribution.

File: StatisticsDistribution.py

```
import random
import math
import NRPy

class Distribution:
    """
    Abstract class for all statistical distributions.
    Due to the large variations of parameters for each distribution,
    it is unlikely to be able to standardize a parameter list for each
    method that is meaningful for all distributions. Instead, the
    parameters to construct each distribution is to be given as
    keyword arguments.
    """

    def __init__(self, **parameters):
        """
```

```

        Constructor method. The parameters are used to construct the
        probability distribution.
        """
        raise NotImplementedError

def CDF(self, x):
    """
    Cummulative Distribution Function, which gives the cummulative
    probability (area under the probability curve) from -infinity
    or 0 to a give x-value on the x-axis where y-axis is the
    probability. CDF is also known as density function.
    """
    raise NotImplementedError

def PDF(self, x):
    """
    Partial Distribution Function, which gives the probability for
    The particular value of x, or the area under probability
    distribution from x-h to x+h for continuous distribution.
    """
    raise NotImplementedError

def inverseCDF(self, probability, start=0.0, step=0.01):
    """
    It does the reverse of CDF() method, it takes a probability
    value and returns the corresponding value on the x-axis.
    """
    raise NotImplementedError

def mean(self):
    """
    Gives the arithmetic mean of the sample.
    """
    raise NotImplementedError

def mode(self):
    """
    Gives the mode of the sample, if closed-form is available.
    """
    raise NotImplementedError

def kurtosis(self):
    """
    Gives the kurtosis of the sample.
    """
    raise NotImplementedError

def skew(self):
    """
    Gives the skew of the sample.
    """
    raise NotImplementedError

def variance(self):
    """
    Gives the variance of the sample.
    """
    raise NotImplementedError

class BetaDistribution(Distribution):
    """
    Class for Beta Distribution.
    """

    def __init__(self, location, scale, p, q):
        """
        Constructor method. The parameters are used to construct the

```

```

probability distribution.

Parameters:
1. location
2. scale (upper bound)
3. p (shape parameter. Although no upper bound but seldom
   exceed 10.)
4. q (shape parameter. Although no upper bound but seldom
   exceed 10.)
"""
self.location = float(location)
self.scale = float(scale)
self.p = float(p)
self.q = float(q)

def CDF(self, x):
    """
    Cumulative Distribution Function, which gives the cumulative
    probability (area under the probability curve) from -infinity
    or 0 to a give x-value on the x-axis where y-axis is the
    probability.
    """
    return NRPy.betainc(self.p, self.q, (x - self.location)/
                        (self.scale - self.location))

def PDF(self, x):
    """
    Partial Distribution Function, which gives the probability
    for particular value of x, or the area under probability
    distribution from x-h to x+h for continuous distribution.
    """
    n = (self.scale - self.location) ** (self.p + self.q - 1)
    n = NRPy.gammln(self.p) * NRPy.gammln(self.q) * n
    n = NRPy.gammln(self.p + self.q) / n
    p = (x - self.location) ** (self.p - 1)
    q = (self.scale - x) ** (self.q - 1)
    return n * p * q

def inverseCDF(self, probability, start=0.0, step=0.01):
    """
    It does the reverse of CDF() method, it takes a probability
    value and returns the corresponding value on the x-axis.
    """
    cprob = self.CDF(start)
    if probability < cprob:
        return (start, cprob)
    while probability > cprob:
        start = start + step
        cprob = self.CDF(start)
    return (start, cprob)

def mean(self):
    """
    Gives the arithmetic mean of the sample.
    """
    n = (self.location * self.q) + (self.scale * self.p)
    return n / (self.p + self.q)

def mode(self):
    """
    Gives the mode of the sample.
    """
    n = (self.location * (self.q - 1)) + (self.scale * \
        (self.p - 1))
    return n / (self.p + self.q - 2)

def kurtosis(self):
    """

```

```

    Gives the kurtosis of the sample.
    """
    n = (self.p ** 2) * (self.q + 2) + \
        (2 * (self.q ** 2)) + \
        ((self.p * self.q) * (self.q - 2))
    n = n * (self.p + self.q + 1)
    d = self.p * self.q * (self.p + self.q + 2) * \
        (self.p + self.q + 3)
    return 3 * ((n / d) - 1)

def skew(self):
    """
    Gives the skew of the sample.
    """
    d = (self.p + self.q) ** 3
    d = d * (self.p + self.q + 1) * (self.p + self.q + 2)
    e = ((self.p + self.q) ** 2) * (self.p + self.q + 1)
    e = (self.p * self.q) / e
    e = e ** 1.5
    return ((2 * self.p * self.q) * (self.q - self.p)) / (d * e)

def variance(self):
    """
    Gives the variance of the sample.
    """
    n = self.p * self.q * ((self.scale - self.location) ** 2)
    d = (self.p + self.q + 1) * ((self.p + self.q) ** 2)
    return n / d

def moment(self, r):
    """
    Gives the r-th moment of the sample.
    """
    return NRPy.beta(self.p + r,
                     self.q)/NRPy.beta(self.p, self.q)

def random(self):
    """
    Gives a random number based on the distribution.
    """
    return random.betavariate(self.p, self.q)

class BinomialDistribution(Distribution):
    """
    Class for Binomial Distribution.
    """

    def __init__(self, success=0.5, trial=1000):
        """
        Constructor method. The parameters are used to construct
        the probability distribution.

        Parameters:
        1. success (probability of success; 0 <= success <= 1)
        2. trial (number of Bernoulli trials)
        """
        self.success = float(success)
        self.trial = int(trial)

    def CDF(self, x):
        """
        Cummulative Distribution Function, which gives the cummulative
        probability (area under the probability curve) from -infinity
        or 0 to a give x-value on the x-axis where y-axis is the
        probability.
        """
        return NRPy.cdf_binomial(x, self.trial, self.success)

```

```

def PDF(self, x):
    """
    Partial Distribution Function, which gives the probability for
    the particular value of x, or the area under probability
    distribution from x-h to x+h for continuous distribution.
    """
    x = int(x)
    return NRPy.bico(self.trial, x) * \
        (self.success ** x) * \
        ((1 - self.success) ** (self.trial - x))

def inverseCDF(self, probability, start=0, step=0.01):
    """
    It does the reverse of CDF() method, it takes a probability
    value and returns the corresponding value on the x-axis.
    """
    cprob = self.CDF(start)
    if probability < cprob:
        return (start, cprob)
    while probability > cprob:
        start = start + step
        cprob = self.CDF(start)
    return (start, cprob)

def mean(self):
    """
    Gives the arithmetic mean of the sample.
    """
    return self.success * self.trial

def mode(self):
    """
    Gives the mode of the sample.
    """
    return int(self.success * (self.trial + 1))

def kurtosis(self):
    """
    Gives the kurtosis of the sample.
    """
    return (1 - ((6 * self.success * (1 - self.success))) /
            (self.trial * self.success * (1 - self.success)))

def skew(self):
    """
    Gives the skew of the sample.
    """
    return (1 - self.success - self.success) / \
        ((self.trial * self.success * (1 - self.success)) ** 0.5)

def variance(self):
    """
    Gives the variance of the sample.
    """
    return self.mean() * (1 - self.success)

class FDistribution(Distribution):
    """
    Class for F Distribution.
    """

    def __init__(self, df1=1, df2=1):
        """
        Constructor method. The parameters are used to construct the
        probability distribution.

```

```

Parameters:
1. df1 (degrees of freedom for numerator)
2. df2 (degrees of freedom for denominator)
"""
self.df1 = float(df1)
self.df2 = float(df2)

def CDF(self, x):
    """
    Cummulative Distribution Function, which gives the cummulative
    probability (area under the probability curve) from -infinity
    or 0 to a give x-value on the x-axis where y-axis is the
    probability.
    """
    sub_x = (self.df1 * x) / (self.df1 * x + self.df2)
    return NRPy.betai(self.df1 / 2.0, self.df2 / 2.0, sub_x)

def PDF(self, x):
    """
    Partial Distribution Function, which gives the probability
    for particular value of x, or the area under probability
    distribution from x-h to x+h for continuous distribution.
    """
    x = float(x)
    n1 = ((x * self.df1) ** self.df1) * (self.df2 ** self.df2)
    n2 = (x * self.df1 + self.df2) ** (self.df1 + self.df2)
    d = x * NRPy.beta(self.df1 / 2.0, self.df2 / 2.0)
    return math.sqrt(n1 / n2) / d

def inverseCDF(self, probability, start=0.0, step=0.01):
    """
    It does the reverse of CDF() method, it takes a probability
    value and the corresponding value on the x-axis.
    """
    cprob = self.CDF(start)
    if probability < cprob:
        return (start, cprob)
    while probability > cprob:
        start = start + step
        cprob = self.CDF(start)
    return (start, cprob)

def mean(self):
    """
    Gives the arithmetic mean of the sample.
    """
    return float(self.df2 / (self.df2 - 2))

class GammaDistribution(Distribution):
    """
    Class for Gamma Distribution
    """
    def __init__(self, location, scale, shape):
        """
        Constructor method. The parameters are used to construct the
        probability distribution.

        Parameters:
        1. location
        2. scale
        3. shape"""
        self.location = float(location)
        self.scale = float(scale)
        self.shape = float(shape)

    def CDF(self, x):

```

```

    """
    Cummulative Distribution Function, which gives the cummulative
    probability (area under the probability curve) from -infinity
    or 0 to a give x-value on the x-axis where y-axis is the
    probability.
    """
    return NRPy.gammp(self.shape,
                      (x - self.location) / self.scale)

def inverseCDF(self, probability, start=0.0, step=0.01):
    """
    It does the reverse of CDF() method, it takes a probability
    value and the corresponding value on the x-axis.
    """
    cprob = self.CDF(start)
    if probability < cprob:
        return (start, cprob)
    while probability > cprob:
        start = start + step
        cprob = self.CDF(start)
    return (start, cprob)

def mean(self):
    """
    Gives the arithmetic mean of the sample.
    """
    return self.location + (self.scale * self.shape)

def mode(self):
    """
    Gives the mode of the sample.
    """
    return self.location + (self.scale * (self.shape - 1))

def kurtosis(self):
    """
    Gives the kurtosis of the sample.
    """
    return 6 / self.shape

def skew(self):
    """
    Gives the skew of the sample.
    """
    return 2 / math.sqrt(self.shape)

def variance(self):
    """
    Gives the variance of the sample.
    """
    return self.scale * self.scale * self.shape

def qmean(self):
    """
    Gives the quantile of the arithmetic mean of the sample.
    """
    return NRPy.gammp(self.shape, self.shape)

def qmode(self):
    """
    Gives the quantile of the mode of the sample.
    """
    return NRPy.gammp(self.shape, self.shape - 1)

class ChiSquareDistribution(GammaDistribution):
    """
    Chi-square distribution is a special case of Gamma distribution

```

where location = 0, scale = 2 and shape is twice that of the degrees of freedom.

```

"""
def __init__(self, df=2):
    """
    Constructor method. The parameters are used to construct
    the probability distribution.

    Parameters:
    1. df = degrees of freedom"""
    GammaDistribution.__init__(self, location=0, scale=2,
                              shape=float(df) / 2.0)

```

```

class GeometricDistribution(Distribution):

```

```

    """
    Geometric distribution is the discrete version of Exponential
    distribution.
    """

```

```

def __init__(self, success=0.5):
    """
    Constructor method. The parameters are used to construct the
    probability distribution.

    Parameters:
    1. success (probability of success; 0 <= success <= 1;
       default = 0.5)
    """
    self.prob = float(success)

```

```

def CDF(self, x):
    """
    Cummulative Distribution Function, which gives the cummulative
    probability (area under the probability curve) from -infinity
    or 0 to a give x-value on the x-axis where y-axis is the
    probability.
    """
    total = self.PDF(1)
    for i in range(2, int(x) + 1):
        total += self.PDF(i)
    return total

```

```

def PDF(self, x):
    """
    Partial Distribution Function, which gives the probability
    for particular value of x, or the area under probability
    distribution from x-h to x+h for continuous distribution.
    """
    return self.prob * ((1 - self.prob) ** (x - 1))

```

```

def inverseCDF(self, probability, start=1, step=1):
    """
    It does the reverse of CDF() method, it takes a probability
    value and the corresponding value on the x-axis.
    """
    cprob = self.CDF(start)
    if probability < cprob:
        return (start, cprob)
    while probability > cprob:
        start = start + step
        cprob = self.CDF(start)
    return (start, cprob)

```

```

def mean(self):
    """
    Gives the arithmetic mean of the sample.

```

```

        """
        return 1/self.prob

def mode(self):
    """
    Gives the mode of the sample.
    """
    return 1.0

def variance(self):
    """
    Gives the variance of the sample.
    """
    return (1 - self.prob) / (self.prob ** 2)

class PoissonDistribution(Distribution):
    """
    Class for Poisson Distribution. Poisson distribution is binomial
    distribution with very low success - that is, for rare events.
    """

    def __init__(self, expectation=0.001):
        """
        Constructor method. The parameters are used to construct the
        probability distribution.

        Parameters:
        1. expectation (mean success probability; lambda)
        """
        self._mean = float(expectation)

    def CDF(self, x):
        """
        Cummulative Distribution Function, which gives the cummulative
        probability (area under the probability curve) from -infinity
        or 0 to a give x-value on the x-axis where y-axis is the
        probability.

        """
        return NRPy.cdf_poisson(x + 1, self._mean)

    def PDF(self, x):
        """
        Partial Distribution Function, which gives the probability
        for particular value of x, or the area under probability
        distribution from x-h to x+h for continuous distribution.
        """
        return (math.exp(-1 ** self._mean) *
                (self._mean ** x) / NRPy.factorial(x))

    def inverseCDF(self, probability, start=0.001, step=1):
        """
        It does the reverse of CDF() method, it takes a probability
        value and the corresponding value on the x-axis.
        """
        cprob = self.CDF(start)
        if probability < cprob:
            return (start, cprob)
        while probability > cprob:
            start = start + step
            cprob = self.CDF(start)
        return (start, cprob)

    def mean(self):
        """
        Gives the arithmetic mean of the sample.
        """

```

```

        return self._mean

def mode(self):
    """
    Gives the mode of the sample.
    """
    return int(self._mean)

def variance(self):
    """
    Gives the variance of the sample.
    """
    return self._mean

class TDistribution(Distribution):
    """
    Class for Student's t-distribution.
    """

    def __init__(self, location=0.0, scale=1.0, shape=2):
        """Constructor method. The parameters are used to construct
        the probability distribution.

        Parameter:
        1. location (default = 0.0)
        2. scale (default = 1.0)
        3. shape (degrees of freedom; default = 2)"""
        self._mean = float(location)
        self.stdev = float(scale)
        self.df = float(shape)

    def CDF(self, x):
        """
        Cummulative Distribution Function, which gives the cummulative
        probability (area under the probability curve) from -infinity
        or 0 to a give x-value on the x-axis where y-axis is the
        probability.
        """
        t = (x - self._mean) / self.stdev
        a = NRPy.betainc(self.df / 2.0, 0.5, self.df /
            (self.df + (t * t)))
        if t > 0:
            return 1 - 0.5 * a
        else:
            return 0.5 * a

    def PDF(self, x):
        """
        Calculates the density (probability) at x with n-th degrees of
        freedom as:

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

        for all real x. It has mean 0 (for n > 1) and variance
        n/(n-2) (for n > 2)."""
        a = NRPy.gammln((self.df + 1) / 2)
        b = math.sqrt(math.pi * self.df) *
            NRPy.gammln(self.df / 2) * self.stdev
        c = 1 + (((x - self._mean) / self.stdev) ** 2) / self.df
        return (a / b) * (c ** ((-1 - self.df) / 2))

    def inverseCDF(self, probability, start=0.0, step=0.01):
        """
        It does the reverse of CDF() method, it takes a probability
        value and the corresponding value on the x-axis.
        """
        cprob = self.CDF(start)
        if probability < cprob:

```

```

        return (start, cprob)
    while probability > cprob:
        start = start + step
        cprob = self.CDF(start)
    return (start, cprob)

def mean(self):
    """
    Gives the arithmetic mean of the sample.
    """
    return self._mean

def mode(self):
    """
    Gives the mode of the sample.
    """
    return self._mean

def kurtosis(self):
    """
    Gives the kurtosis of the sample.
    """
    a = ((self.df - 2) ** 2) * NRPy.gammln((self.df / 2) - 2)
    return 3 * ((a / (4 * NRPy.gammln(self.df / 2))) - 1)

def skew(self):
    """
    Gives the skew of the sample.
    """
    return 0.0

def variance(self):
    """
    Gives the variance of the sample.
    """
    return (self.df / (self.df - 2)) * self.stdev * self.stdev

class UniformDistribution(Distribution):
    """
    Class for Uniform distribution.
    """

    def __init__(self, location, scale):
        """
        Constructor method. The parameters are used to construct the
        probability distribution.

        Parameter:
        1. location
        2. scale
        """
        self.location = float(location)
        self.scale = float(scale)

    def CDF(self, x):
        """
        Cummulative Distribution Function, which gives the
        cumulative probability (area under the probability curve)
        from -infinity or 0 to a give x-value on the x-axis where y-
        axis is the probability.
        """
        return (x - self.location) / (self.scale - self.location)

    def PDF(self):
        """
        Partial Distribution Function, which gives the probability
        for particular value of x, or the area under probability

```

```

distribution from x-h to x+h for continuous distribution.
"""
return 1.0/(self.scale - self.location)

def inverseCDF(self, probability, start=0.0, step=0.01):
    """
    It does the reverse of CDF() method, it takes a probability
    value and the corresponding value on the x-axis.
    """
    cprob = self.CDF(start)
    if probability < cprob:
        return (start, cprob)
    while probability > cprob:
        start = start + step
        cprob = self.CDF(start)
    return (start, cprob)

def mean(self):
    """
    Gives the arithmetic mean of the sample.
    """
    return (self.location + self.scale) / 2.0

def median(self):
    """
    Gives the median of the sample.
    """
    return (self.location + self.scale) / 2

def kurtosis(self):
    """
    Gives the kurtosis of the sample.
    """
    return -1.2

def skew(self):
    """
    Gives the skew of the sample.
    """
    return 0.0

def variance(self):
    """
    Gives the variance of the sample.
    """
    return ((self.scale - self.location) ** 2) / 12

def quantile1(self):
    """
    Gives the 1st quantile of the sample.
    """
    return ((3 * self.location) + self.scale) / 4

def quantile3(self):
    """
    Gives the 3rd quantile of the sample.
    """
    return (self.location + (3 * self.scale)) / 4

def qmean(self):
    """
    Gives the quantile of the arithmetic mean of the sample.
    """
    return 0.5

def random(self, lower, upper):
    """
    Gives a random number based on the distribution.

```

```

    """
    return random.uniform(lower, upper)

```

File: NRP.py

```

import math

class MaxIterationsException(Exception):
    """
    Exception to catch maximum looping.
    """
    pass

def beta(z, w):
    """Beta function.
    Depend: gammln
    @see: NRP 6.1

    @param z: float number
    @param w: float number
    @return: float number"""
    return math.exp(gammln(z) + gammln(w) - gammln(z+w))

def betacf(a, b, x):
    """
    Continued fraction for incomplete beta function.
    Adapted from salstat_stats.py of SalStat
    (www.sf.net/projects/salstat)
    Ref; NRP 6.3
    """
    iter_max = 200
    eps = 3.0e-7

    bm = az = am = 1.0
    qab = a + b
    qap = a + 1.0
    qam = a - 1.0
    bz = 1.0 - qab * x / qap
    for i in range(iter_max + 1):
        em = float(i + 1)
        tem = em + em
        d = em * (b - em) * x / ((qam + tem) * (a + tem))
        ap = az + d * am
        bp = bz + d * bm
        d = -(a + em) * (qab + em) * x / ((qap + tem) * (a + tem))
        app = ap + d * az
        bpp = bp + d * bz
        aold = az
        am = ap / bpp
        bm = bp / bpp
        az = app / bpp
        bz = 1.0
        if (abs(az - aold) < (eps * abs(az))):
            return az

def betai(a, b, x):
    """
    Incomplete beta function


$$I_{-sub-x}(a,b) = 1/B(a,b) * \int_0^x t^{(a-1)}(1-t)^{(b-1)} dt$$


    where  $a, b > 0$  and  $B(a,b) = G(a)G(b)/(G(a+b))$  where  $G(a)$  is the
    gamma function of  $a$ .

    Adapted from salstat_stats.py of SalStat
    (www.sf.net/projects/salstat)
    Depend: betacf, gammln

```

```

@see: NRP 6.3
"""
if (x < 0.0 or x > 1.0):
    raise ValueError('Bad value for x: %s' % x)
if (x == 0.0 or x == 1.0):
    bt = 0.0
else:
    bt = math.exp(gammln(a+b) - gammln(a) - gammln(b) + a *
                  math.log(x) + b * math.log(1.0-x))
if (x < (a + 1.0) / (a + b + 2.0)):
    return bt * betacf(a, b, x) / float(a)
else:
    return 1.0 - bt * betacf(b, a, 1.0 - x) / float(b)

def bico(n, k):
    """
    Binomial coefficient. Returns n!/(k!(n-k)!)
    Depend: factln, gammln
    @see: NRP 6.1

    @param n: total number of items
    @param k: required number of items
    @return: floating point number
    """
    return math.floor(math.exp(factln(n) - factln(k) - factln(n-k)))

def factln(n):
    """
    Natural logarithm of factorial: ln(n!)
    @see: NRP 6.1

    @param n: positive integer
    @return: natural logarithm of factorial of n
    """
    return gammln(n + 1.0)

def gammln(n):
    """
    Complete Gamma function.
    @see: NRP 6.1 and
    http://mail.python.org/pipermail/python-list/2000-
    June/039873.html

    @param n: float number
    @return: float number
    """
    gammln_cof = [76.18009173, -86.50532033, 24.01409822,
                  -1.231739516e0, 0.120858003e-2, -0.536382e-5]
    x = n - 1.0
    tmp = x + 5.5
    tmp = (x + 0.5) * math.log(tmp) - tmp
    ser = 1.0
    for j in range(6):
        x = x + 1.
        ser = ser + gammln_cof[j] / x
    return tmp + math.log(2.50662827465 * ser)

def gammf(a, x):
    """
    Gamma incomplete function, P(a,x).
    P(a,x) = (1/gammln(a)) * integral(0, x, (e^-t)*(t^(a-1)), dt)
    Depend: gser, gcf, gammln
    @see: NRP 6.2

    @param a: float number
    @param x: float number
    @return: float number
    """

```

```

    if (x < 0.0 or a <= 0.0):
        raise ValueError('Bad value for a or x: %s, %s' % (a, x))
    if (x < a + 1.0):
        return gser(a, x)[0]
    else:
        return 1.0 - gcf(a, x)[0]

def gammq(a, x):
    """
    Incomplete gamma function:  $Q(a, x) = 1 - P(a, x) = 1 - \text{gammp}(a, x)$ 
    Also commonly known as Q-equation.
    @see: http://mail.python.org/pipermail/python-list/2000-June/039873.html
    """
    if (x < 0.0 or a <= 0.0):
        raise ValueError('Bad value for a or x: %s, %s' % (a, x))
    if (x < a + 1.0):
        a = gser(a, x)[0]
        return 1.0 - a
    else:
        return gcf(a, x)[0]

def gcf(a, x, itmax=200, eps=3.e-7):
    """
    Continued fraction approx'n of the incomplete gamma function.
    @see: http://mail.python.org/pipermail/python-list/2000-June/039873.html
    """
    gln = gammaln(a)
    gold = 0.0
    a0 = 1.0
    a1 = x
    b0 = 0.0
    b1 = 1.0
    fac = 1.0
    n = 1
    while n <= itmax:
        an = n
        ana = an - a
        a0 = (a1 + a0 * ana) * fac
        b0 = (b1 + b0 * ana) * fac
        anf = an * fac
        a1 = x * a0 + anf * a1
        b1 = x * b0 + anf * b1
        if (a1 != 0.0):
            fac = 1.0 / a1
            g = b1 * fac
            if (abs((g - gold) / g) < eps):
                return (g * math.exp(-x + a * math.log(x) - gln), gln)
            gold = g
        n = n + 1
    raise MaxIterationsException('Maximum iterations reached: %s'
                                  % abs((g - gold) / g))

def gser(a, x, itmax=700, eps=3.e-7):
    """
    Series approximation to the incomplete gamma function.
    @see: http://mail.python.org/pipermail/python-list/2000-June/039873.html
    """
    gln = gammaln(a)
    if (x < 0.0):
        raise ValueError('Bad value for x: %s' % a)

    if (x == 0.0):
        return(0.0, 0.0)
    ap = a
    total = 1.0 / a

```

```

delta = total
n = 1
while n <= itmax:
    ap = ap + 1.0
    delta = delta * x / ap
    total = total + delta
    if (abs(delta) < abs(total) * eps):
        return (total * math.exp(-x + a * math.log(x) - gln), gln)
    n = n + 1
raise MaxIterationsException('Maximum iterations reached: %s, %s'
                             % (abs(delta), abs(total) * eps))

def cdf_binomial(k, n, p):
    """
    Cummulative density function of Binomial distribution. No
    Reference implementation.
    Depend: betai, betacf, gammln
    @see: NRP 6.3

    @param k: number of times of event occurrence in n trials
    @param n: total number of trials
    @param p: probability of event occurrence per trial
    @return: float number - Binomial probability
    """
    return betai(k, n - k + 1, p)

def cdf_poisson(k, x):
    """
    Cummulative density function of Poisson distribution from 0 to
    k - 1 inclusive. No reference implementation.
    Depend: gammq, gser, gcf, gammln
    @see: NRP 6.2

    @param k: number of times of event occurrence
    @param x: mean of Poisson distribution
    @return: float number - Poisson probability of k - 1 times of
    Occurrence with the mean of x
    """
    return gammq(k, x)

```

File: DistributionTest.py

```

import unittest
import StatisticsDistribution as N

class testBeta(unittest.TestCase):
    def testCDF1(self):
        p = N.BetaDistribution(location=0, scale=1,
                               p=1, q=2).CDF(1.0)
        self.assertAlmostEqual(p, 1.0000, places=4)
    def testCDF2(self):
        p = N.BetaDistribution(location=0, scale=1,
                               p=6, q=2).CDF(1.0)
        self.assertAlmostEqual(p, 1.0000, places=4)

class testBinomial(unittest.TestCase):
    def testCDF1(self):
        p = N.BinomialDistribution(trial=1000,
                                   success=0.5).CDF(500)
        self.assertAlmostEqual(p, 0.5126125, places=4)
    def testPDF1(self):
        p = N.BinomialDistribution(trial=20,
                                   success=0.5).PDF(10)
        self.assertAlmostEqual(p, 0.1762, places=4)
    def testPDF2(self):
        p = N.BinomialDistribution(trial=20, success=0.1).PDF(1)

```

```

        self.assertAlmostEqual(p, 0.27017, places=4)
def testPDF3(self):
    p = N.BinomialDistribution(trial=20, success=0.1).PDF(3)
    self.assertAlmostEqual(p, 0.19012, places=4)
def testPDF4(self):
    p = N.BinomialDistribution(trial=20, success=0.1).PDF(4)
    self.assertAlmostEqual(p, 0.08978, places=4)

class testChiSquare(unittest.TestCase):
    def testCDF0_1(self):
        p = N.ChiSquareDistribution(df=1).CDF(2.706)
        self.assertAlmostEqual(p, 0.9000, places=4)
    def testCDF0_2(self):
        p = N.ChiSquareDistribution(df=1).CDF(3.841)
        self.assertAlmostEqual(p, 0.9500, places=4)
    def testCDF0_3(self):
        p = N.ChiSquareDistribution(df=1).CDF(10.828)
        self.assertAlmostEqual(p, 0.9990, places=4)
    def testCDF10_1(self):
        p = N.ChiSquareDistribution(df=10).CDF(15.987)
        self.assertAlmostEqual(p, 0.9000, places=4)
    def testCDF10_2(self):
        p = N.ChiSquareDistribution(df=10).CDF(18.307)
        self.assertAlmostEqual(p, 0.9500, places=4)
    def testCDF10_3(self):
        p = N.ChiSquareDistribution(df=10).CDF(29.588)
        self.assertAlmostEqual(p, 0.9990, places=4)
    def testinverseCDF1(self):
        p = N.ChiSquareDistribution(df=10).inverseCDF(0.9)[0]
        self.assertAlmostEqual(p, 15.9870, places=2)
    def testinverseCDF2(self):
        p = N.ChiSquareDistribution(df=10).inverseCDF(0.95)[0]
        self.assertAlmostEqual(p, 18.307, places=1)
    def testinverseCDF3(self):
        p = N.ChiSquareDistribution(df=10).inverseCDF(0.999)[0]
        self.assertAlmostEqual(p, 29.588, places=2)

class testF(unittest.TestCase):
    def testCDF1(self):
        p = N.FDistribution(df1=3, df2=5).CDF(1.0)
        self.assertAlmostEqual(p, 0.535145, places=4)
    def testCDF2(self):
        p = N.FDistribution(df1=3, df2=5).CDF(2.0)
        self.assertAlmostEqual(p, 0.767376, places=4)
    def testCDF3(self):
        p = N.FDistribution(df1=3, df2=5).CDF(3.0)
        self.assertAlmostEqual(p, 0.866145, places=4)
    def testPDF1(self):
        p = N.FDistribution(df1=3, df2=5).PDF(1.0)
        self.assertAlmostEqual(p, 0.361174, places=4)
    def testPDF2(self):
        p = N.FDistribution(df1=3, df2=5).PDF(2.0)
        self.assertAlmostEqual(p, 0.1428963, places=4)
    def testPDF3(self):
        p = N.FDistribution(df1=3, df2=5).PDF(3.0)
        self.assertAlmostEqual(p, 0.066699, places=4)
    def testinverseCDF1(self):
        p = N.FDistribution(df1=3,
                           df2=5).inverseCDF(0.535145)[0]
        self.assertAlmostEqual(p, 1.0000, places=2)
    def testinverseCDF2(self):
        p = N.FDistribution(df1=3,
                           df2=5).inverseCDF(0.767376)[0]
        self.assertAlmostEqual(p, 2.00, places=2)
    def testinverseCDF3(self):
        p = N.FDistribution(df1=3,
                           df2=5).inverseCDF(0.866145)[0]
        self.assertAlmostEqual(p, 3.00, places=2)

```

```

class testGamma(unittest.TestCase):
    def testCDF1(self):
        p = N.GammaDistribution(location=0, scale=4,
                                shape=4).CDF(7.0)
        self.assertAlmostEqual(p, 0.1008103, places=4)
    def testCDF2(self):
        p = N.GammaDistribution(location=0, scale=4,
                                shape=4).CDF(7.5)
        self.assertAlmostEqual(p, 0.1210543, places=4)
    def testCDF3(self):
        p = N.GammaDistribution(location=0, scale=4,
                                shape=4).CDF(8.0)
        self.assertAlmostEqual(p, 0.1428765, places=4)
    def testinverseCDF1(self):
        p = N.GammaDistribution(location=0, scale=4,
                                shape=4).inverseCDF(0.1008103)[0]
        self.assertAlmostEqual(p, 7.0000, places=4)
    def testinverseCDF2(self):
        p = N.GammaDistribution(location=0, scale=4,
                                shape=4).inverseCDF(0.1210543)[0]
        self.assertAlmostEqual(p, 7.5000, places=4)
    def testinverseCDF3(self):
        p = N.GammaDistribution(location=0, scale=4,
                                shape=4).inverseCDF(0.1428765)[0]
        self.assertAlmostEqual(p, 8.00, places=2)
    def test_kurtosis(self):
        p = N.GammaDistribution(location=0, scale=4,
                                shape=4).kurtosis()
        self.assertAlmostEqual(p, 1.50, places=2)

class testGeometric(unittest.TestCase):
    def testCDF1(self):
        p = N.GeometricDistribution(success=0.4).CDF(1)
        self.assertAlmostEqual(p, 0.4000, places=4)
    def testCDF2(self):
        p = N.GeometricDistribution(success=0.4).CDF(4)
        self.assertAlmostEqual(p, 0.8704, places=4)
    def testCDF3(self):
        p = N.GeometricDistribution(success=0.4).CDF(6)
        self.assertAlmostEqual(p, 0.953344, places=4)
    def testPDF1(self):
        p = N.GeometricDistribution(success=0.4).PDF(1)
        self.assertAlmostEqual(p, 0.4000, places=4)
    def testPDF2(self):
        p = N.GeometricDistribution(success=0.4).PDF(4)
        self.assertAlmostEqual(p, 0.0864, places=4)
    def testPDF3(self):
        p = N.GeometricDistribution(success=0.4).PDF(6)
        self.assertAlmostEqual(p, 0.031104, places=4)
    def testinverseCDF1(self):
        p = N.GeometricDistribution(
            success=0.4).inverseCDF(0.4)[0]
        self.assertAlmostEqual(p, 1.0000, places=4)
    def testinverseCDF2(self):
        p = N.GeometricDistribution(
            success=0.4).inverseCDF(0.8704)[0]
        self.assertAlmostEqual(p, 4.0000, places=4)
    def testinverseCDF3(self):
        p = N.GeometricDistribution(
            success=0.4).inverseCDF(0.953344)[0]
        self.assertAlmostEqual(p, 6.0000, places=4)

class testPoisson(unittest.TestCase):
    def testCDF5_1(self):
        p = N.PoissonDistribution(expectation=5).CDF(1)
        self.assertAlmostEqual(p, 0.04, places=2)
    def testCDF5_2(self):

```

```

        p = N.PoissonDistribution(expectation=5).CDF(10)
        self.assertAlmostEqual(p, 0.986, places=3)
    def testCDF7_1(self):
        p = N.PoissonDistribution(expectation=7).CDF(10)
        self.assertAlmostEqual(p, 0.901, places=3)
    def testCDF7_2(self):
        p = N.PoissonDistribution(expectation=7).CDF(13)
        self.assertAlmostEqual(p, 0.987, places=3)
    def testinverseCDF5_1(self):
        x = N.PoissonDistribution(
            expectation=5).inverseCDF(0.04)[0]
        self.assertAlmostEqual(x, 1.000, places=2)
    def testinverseCDF5_2(self):
        x = N.PoissonDistribution(
            expectation=5).inverseCDF(0.986)[0]
        self.assertAlmostEqual(x, 10.000, places=2)
    def testinverseCDF7_1(self):
        x = N.PoissonDistribution(
            expectation=7).inverseCDF(0.901)[0]
        self.assertAlmostEqual(x, 10.000, places=2)

class testT(unittest.TestCase):
    def testinverseCDF1_1(self):
        x = N.TDistribution(location=0.0, scale=1.0,
                           shape=2).inverseCDF(0.9)[0]
        self.assertAlmostEqual(x, 1.886, places=1)
    def testinverseCDF1_2(self):
        x = N.TDistribution(location=0.0, scale=1.0,
                           shape=2).inverseCDF(0.95)[0]
        self.assertAlmostEqual(x, 2.920, places=3)
    def testinverseCDF1_3(self):
        x = N.TDistribution(location=0.0, scale=1.0,
                           shape=2).inverseCDF(0.99)[0]
        self.assertAlmostEqual(x, 6.965, places=1)
    def testinverseCDF1_4(self):
        x = N.TDistribution(location=0.0, scale=1.0,
                           shape=2).inverseCDF(0.999)[0]
        self.assertAlmostEqual(x, 22.328, places=1)
    def testinverseCDF5_1(self):
        x = N.TDistribution(location=0.0, scale=1.0,
                           shape=5).inverseCDF(0.9)[0]
        self.assertAlmostEqual(x, 1.480, places=3)
    def testinverseCDF5_2(self):
        x = N.TDistribution(location=0.0, scale=1.0,
                           shape=5).inverseCDF(0.95)[0]
        self.assertAlmostEqual(x, 2.020, places=2)
    def testinverseCDF5_3(self):
        x = N.TDistribution(location=0.0, scale=1.0,
                           shape=5).inverseCDF(0.99)[0]
        self.assertAlmostEqual(x, 3.365, places=2)
    def testinverseCDF5_4(self):
        x = N.TDistribution(location=0.0, scale=1.0,
                           shape=5).inverseCDF(0.999)[0]
        self.assertAlmostEqual(x, 5.899, places=2)

class testUniform(unittest.TestCase):
    def testCDF1(self):
        p = N.UniformDistribution(location=1.0,
                                  scale=3.0).CDF(1.5)
        self.assertTrue(abs(p / 0.25 - 1) < 0.01)
        self.assertAlmostEqual(p, 0.2500, places=4)
    def testCDF2(self):
        p = N.UniformDistribution(location=1.0,
                                  scale=3.0).CDF(2.5)
        self.assertTrue(abs(p / 0.75 - 1) < 0.01)
        self.assertAlmostEqual(p, 0.7500, places=4)
    def testCDF3(self):

```

```

    p = N.UniformDistribution(location=-1.0,
                             scale=1.0).CDF(0)
    self.assertAlmostEqual(p, 0.5)
def testPDF1(self):
    p = N.UniformDistribution(location=1.0,
                             scale=3.0).PDF()
    self.assertTrue(p == 0.5)
def testPDF2(self):
    p = N.UniformDistribution(location=1.0,
                             scale=3.0).PDF()
    self.assertTrue(p == 0.5)
def testinverseCDF1(self):
    p = N.UniformDistribution(location=1.0,
                             scale=3.0).inverseCDF(0.25)[0]
    self.assertAlmostEqual(p, 1.50, places=1)
def testinverseCDF2(self):
    p = N.UniformDistribution(location=1.0,
                             scale=3.0).inverseCDF(0.75)[0]
    self.assertAlmostEqual(p, 2.50, places=1)
def test_mean(self):
    p = N.UniformDistribution(location=1,
                             scale=2).mean()
    self.assertAlmostEqual(p, 1.5)

if __name__ == '__main__':
    unittest.main()

```

3. References

- Crofts, Alberts E. 1982. On a Property of the F Distribution. *Trabajos de Estadística y de Investigación Operativa* 33, 110-111.
- Haskett, Jonathan D., Pachepsky, Yakov A., Acock, Basil. 1995. Use of the beta distribution for parameterizing variability of soil properties at the regional level for crop yield estimation. *Agricultural Systems* 48, 73-86.
- Jambunathan, M. V. 1954. Some Properties of Beta and Gamma Distributions. *Annals of Mathematical Statistics* 25, 401-405.
- Keefer, Donald L. and Verdini, William A. 1993. Better Estimation of PERT Activity Time Parameters. *Management Science* 39(9), 1986-1991
- Loredo, Tom. 2000. Where are the Maths Functions? Python mailing list [online]. Available at <http://mail.python.org/pipermail/python-list/2000-June/039873.html>. Last assessed on 11th June, 2009.
- Ling, Maurice HT. 2009. Ten Z-test Routines from Gopal Kanji's 100 Statistical Tests. Submitted.
- McLaughlin, Michael. 2001. *Regress+ - A Compendium of Common Probability Distributions*.
- Miller, Gary L., Carroll, Bradley W. 1989 Modeling Vertebrate Dispersal Distances: Alternatives to the Geometric Distribution. *Ecology* 70, 977-986.
- Press, William H., Flannery, Brian P., Teukolsky, Saul A., and Vetterling, William T. 1989. *Numerical Recipes in Pascal*. Cambridge University Press, Cambridge.
- Rogers, J. S. 2001. Maximum likelihood estimation of phylogenetic trees is consistent when substitution rates vary according to the invariable sites plus gamma distribution. *System Biology* 50, 713-722.
- Ugoni, Antony, Walker, Bruce F. 1995. The Chi-Square Test: An Introduction. *COMSIG Review* 4, 61-64.
- Van der Geest, P. A. G. 2005. The binomial distribution with dependent Bernoulli trials. *Journal of Statistical Computation and Simulation* 75, 141 – 154.