Calculation of the special functions

Scope

This document provides details on the implementation of the special functions required for calculations of probability density functions (PDF), cummulative distribution functions (CDF) and quantile functions (QF, a.k.a. inverse cummulative distribution function ICDF). Only the real number arguments are considered, in which case the respective functions yield real numbers; although they can be continued onto complex numbers.

The concerned special functions can be implemented as polynomial or rational functions approximations or constructed from other special functions, implemented in this library or in the Standard Python Library.

The proposed implementation is either inspired by or translated into Python from [Ref 1][^Ref1], further on referred to as *Numerical Recipes*.

Background - combinatorics functions

Considering the discrete distributions the probability is often related to *combinatorics*, i.e. number of combinations under specific limitations.

The first function to consider is *factorial*, which calculates the number of possible arrangement of exactly *n* unique objects, if the order is important (i.e. {1, 2, 3} and {2, 1, 3} are considered to be different outcomes of the experiment). Basically, for the first position we have *n* possible choices of an object, for the second position - *n-1* possible choice (since one object is already removed), for the third position - *n-2* possible choice, etc., until only two objects remain with 2 possible choices, then for the last remaining object there is only a single possibility. Thus, the total number of possible arrangements, i.e. *full permutation* is (note different notations often used):

$$ P_n \equiv {}*n P_n \equiv P(n,n) = n * (n-1) * (n-2) * ... * 2 * 1 = n! = \prod*{k=1}^n{k} \equiv \prod_{k=0}^{n-1}{(n-k)} $$

Basically, permutation of an empty set is 1, since there are no elements, and there is only one possible way to select nothing from any set. Similarly, for the set containing only 1 element there is only one possible way to select this element. Thus, 0! = 1 and 1! = 0. Therefore, it is possible to define factorial recursively as:

$$n! = \{ \ 1; \mathtt{if}; n = 0 \ 1; \mathtt{if}; n = 1 \ n * (n-1)!; \forall; n > 1, ; n \in \mathbb{N}$$

The factorial function is available as *math.factorial*() function in Standard Python Library.

The second relevant function is *partial permutation* or *k-permutation*, which calculates the number of possible arrangement of exactly *k* unique objects chosen of a set of *n* unique objects, if the order is important (i.e. {1, 2, 3} and {2, 1, 3} are considered to be different outcomes of the experiment). Similarly, for the first position we have *n* possible choices of an object, for the second position - *n-1* possible choice (since one object is already removed), for the third position - *n-2* possible choice, etc., but we stop at the k-th object, for which there are still *n-k+1* possible choices. Thus (note the different often used notations)

$$ A_n^k \equiv {}*n P_k \equiv P(n,k) = n * (n-1) * (n-2) * ... * (n-k+1) = \frac{n!}{(n-k)!} = \prod*{m=n-k+1}^n{m} \equiv \prod_{m=0}^{k-1}{(n-m)} \newline A_n^k = \frac{n}{n-k} A_{n-1}^k = n A_{n-1}^{k-1} = (n-k+1) A_n^{k-1} $$

This function is implemented as *math.perm*(n, k) in the Standard Python Library (version >= 3.8). In the case of an older Python interpreter is should be impemented as looped multiplication form (n-k+1) towards n instead of ratio of factorials in order to achieve beter accuracy in favour of the $\mathtt{exp}[\mathtt{ln}(\Gamma(n+1)) - \mathtt{ln}(\Gamma(k+1))]$ approach[^1], because Python supports an arbitrary length integer arithmetics.

The last relevant function is *combinations*, which says how many unique sub-sets of k elements can be formed from a set of *n* unique elements, i.e. selection with the order being unimportant (i.e. {1, 2, 3} and {2, 1, 3} are the same sub-set). Basically, there are A_n^k in total ways to select k elements from a set of n elements, whereas within a selected sub-set of k elements there are k! ways to re-arrange the elements. Therefore, the number of combinations is (note the different notations often used):

$$ C_n^k \equiv \binom{n}{k} = \frac{A_n^k}{P_k} = \frac{n!}{k! (n-k)!} \equiv \frac{A_n^{n-k}}{P_{n-k}} = C_n^{n-k} \newline C_n^k = \frac{n}{k} C_{n-1}^{k-1} = \frac{n}{n-k} C_{n-1}^k = \frac{n-k+1}{k} C_n^{k-1} = C_{n-1}^k + C_{n-1}^{k-1} $$

This function is implemented as *math.comb*(n, k) in the Standard Python Library (version >= 3.8). In the case of an older Python interpreter is should be impemented as follows in order to achieve beter accuracy:

- If k > (n-k) - calculate A_n^{n-k} and (n-k)!, then $C_n^k = \frac{A_n^{n-k}}{(n-k)!}$
- Otherwise - calculate A_n^k and k!, then $C_n^k = \frac{A_n^k}{k!}$

This approach is preferable to $\mathtt{exp}[\mathtt{ln}(\Gamma(n+1)) - \mathtt{ln}(\Gamma(k+1)) - \mathtt{ln}(\Gamma(n-k+1))]$ approach[^2], because Python supports an arbitrary length integer arithmetics.

Inverse error function

The *error function* erf() is defined as:

$$\mathtt{erf}(x > 0) = \frac{2}{\sqrt{\pi}} \int_0^x {e^{- t^2} dt} \newline \mathtt{erf}(0) = 0 \newline \mathtt{erf}(x < 0) = - \mathtt{erf}(-x)$$

The *inverse error function* is the solution of equation $\mathtt{erf}(x) = y; \Rightarrow; x = \mathtt{erf}^{-1}(y)$, whith the following (obvious) prorperties:

$$\mathtt{erf}(\mathtt{erf}^{-1}(y)) = y \newline \mathtt{erf}^{-1}(\mathtt{erf}(x)) = x \newline \mathtt{erf}^{-1}(-y) = - \mathtt{erf}^{-1}(y)$$

This function is defined on the range (-1, 1); it is monotonically growing, and it yields values in the range $(-\infin, +\infin)$.

Even though the inverse error function cannot be represented in terms of simple analytical functions, it can be represented as an infinite power series[^3]

$$\mathtt{erf}^{-1}(x) = \sum_{k=0}^{\infin}{\frac{c_k}{2k+1} \left(\frac{\sqrt{\pi}}{2} x \right)^{2k+1}} = \newline = \frac{\sqrt{\pi}}{2} \left(x + \frac{\pi}{12} x^3 + \frac{7 \pi^2}{480} x^5 + \frac{127 \pi^3}{40320} x^7 + ... \right)$$

However, this series converges slowly, especially with the argument approaching ± 1. Instead, a *rational function* approximation is used for the calculation of the inverse error function. A rational function is a ratio of two finite polynomials:

$$R_{n,m}(x) = \frac{P_n(x)}{Q_m(x)} = \frac{p_0 + p_1 * x + p_2 * x^2 + \ldots + p_n * x^n}{q_0 + q_1 * x + q_2 * x^2 + \ldots + q_m * x^m}$$

Specifically concerning the inverse error function algorithm AS241[^Ref2] can be used, wich defines 3 distict rational functions of 7th-7th power for each of the regions: central / core $\mathtt{abs}(x) \le 0.85$, tails $0.85 < \mathtt{abs}(x) \le 1 - 2.77759 \times 10^{-11}$ and far tails $1 - 2.77759 \times 10^{-11} < \mathtt{abs}(x) < 1$. This algorithm was proposed in 1988 for the double precision floating point calculations.

Note, that the actual algorithm defines the quantile function of Z-distribution (see DE002 document)

$$\Phi^{-1}(p) = \sqrt{2}\mathtt{erf}^{-1}(2p - 1)$$

where 0 < p < 1, so the tranformation of the algorithm is trivial

$$\mathtt{erf}^{-1}(x) = \Phi^{-1}(\frac{x}{2} + 0.5)/\sqrt{2}$$

where -1 < x < 1. Furthermore, the polynomials should be calculated using itterative procedure instead of the direct implementation of the formula[^4], i.e.:

$$P_n(x) = p_0 + p_1 * x + p_2 * x^2 + ... + p_n * x^n = \newline = p_0 + x * (p_1 + x * (p_2 + x * (...(p_{n-1} + x * p_n))))$$

Beta function

The *beta function* is defined as:

$$B(x, y) = \int_0^1 t^{x-1}(1-t)^{y-1}dt$$

In the case of *real* arguments *x* and *y*, it is defined for *x* > 0 and *y* > 0. It also can be defined in terms of *gamma function*, i.e.

$$B(x, y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$$

where gamma function is defined

$$\Gamma(x) = \int_0^{\infin} t^{x-1}e^{-t}dt$$

for x > 0. Note, that $\Gamma(n \in \mathbb{N}) = (n-1)!$ on the natural numbers. The gamma function is implemented as *math.gamma*() in the Standard Python Library

Thus, the (complete) beta function can be calculated as:

$$B(x,y) = \mathtt{exp}\left[\mathtt{ln}\left(\frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}\right)\right] = \mathtt{exp}\left[\mathtt{ln}(\Gamma(x)) + \mathtt{ln}(\Gamma(y)) - \mathtt{ln}(\Gamma(x+y))\right]$$

for the better precision, where $\mathtt{ln}(\Gamma(x))$ is implemented as *math.lgamma*() function in the Standard Python Library, and exp() is *math.exp*() function.

Also note the following properties of the beta function:

```
B(x,y) = B(x + 1, y) + B(x, y + 1) \newline
B(x + 1, y) = B(x,y) * \frac{x}{x + y} \newline
B(x, y + 1) = B(x,y) * \frac{y}{x + y} \newline
B(x,y) = B(y,x)
```

Incomplete beta functions

The generalization of the *beta function*, i.e. *incomplete beta function* is defined as:

$$B(z;x,y) = \int_0^z t^{x-1}(1-t)^{y-1}dt$$

for 1 > z > 0; and its *regularized* version is

$$I_z(x,y) = \frac{B(z;x,y)}{B(x,y)}$$

with the edge cases:

```
I_0(x > 0, y>0) = 0 \newline
B(0; x > 0, y >0) = 0 \newline
I_1(x > 0, y>0) = 1 \newline
B(1; x>0, y>0) = B(x,y)
```

and the symmetry relation:

```
I_z(x,y) = 1 - I_{1 -z}(y,x) \newline
B(z; x, y) = B(x,y) - B(1-z; y, x)
```

The regularized incomplete beta function can be calculated using the series expansion[^5]

```
I_z(x,y) = \frac{z^x*(1-z)^y}{x * B(x,y)} \times \left( 1 + \sum_{n=0}^{\infin} {c_n * z^{n + 1}} \right) ; \mathtt{where} \newline
c_n = \frac{B(x+1, n + 1)}{B(x + y, n + 1)} = \frac{B(x+1, n) * n / (x + 1 + n)}{B(x+y, n) * n / (x + y + n)} = \frac{x + y + n}{x + 1 + n} * c_{n - 1} ; \mathtt{and} \newline
c_0 = \frac{B(x + 1, 1)}{B(x + y, 1)}
```

However, this series converges slowly. Instead, a continued fraction representation can be utilized

```
I_z(x,y) = \frac{z^x*(1-z)^y}{x * B(x,y)} \times \left( \cfrac{1}{1+\cfrac{d_1}{1+\cfrac{d_2}{1+\cdots}}} \right) ; \mathtt{where} \newline
d_{2m+1} = - \frac{(x + m) * (x + y + m) * z}{(x + 2m) * (x + 2m  + 1)} \newline
d_{2m} = \frac{m * (y - m)*z}{(x + 2m - 1) * (x + 2m)}
```

which converges rapidly for

$$z < \frac{x+1}{x+y+2} \equiv 1 - z > \frac{y+1}{x+y+2}$$

In the opposite case

$$z > \frac{x+1}{x+y+2} \equiv 1 - z < \frac{y+1}{x+y+2}$$

the symmetry relation is to be used for the computations.

Incomplete gamma functions

The (complete) gamma function is defined as integral from 0 to infinity, which can also be split into two integrals

$$\Gamma(x) = \int_0^{\infin} t^{x-1}e^{-t}dt = \int_0^y t^{x-1}e^{-t}dt + \int_y^{\infin} t^{x-1}e^{-t}dt = \gamma(x,y) + \Gamma(x,y)$$

where $\gamma(x,y) = \int_0^y t^{x-1}e^{-t}dt$ is the *lower incomplete gamma function* and $\Gamma(x,y) = \int_y^{\infin} t^{x-1}e^{-t}dt$ is the *upper incomplete gamma function*. In practice, the *requlirized* versions of incomplete gamma functions are often used, which are:

$$
P(x,y) = \frac{\gamma(x,y)}{\Gamma(x)} \newline
Q(x,y) = \frac{\Gamma(x,y)}{\Gamma(x)} \newline
P(x,y) + Q(x,y) = 1
$$

Note that they are defined for x > 0 and y > 0, with the edge cases:

$$
P(x > 0 , 0) = 0 \newline
\gamma(x > 0, 0) = 0 \newline
Q(x > 0, 0) = 1 \newline
\Gamma (x > 0, 0) = \Gamma(x)
$$

The lower incomplete gamma function can be calculated using the series expansion[^6]

$$
\gamma(x,y) = e^{-y} y^x \sum_{n=0}^{\infin} {\frac{\Gamma(x)}{\Gamma(x + 1 + n)} y^n} = e^{-y} y^x \sum_{n=0}^{\infin} {b_n y^n} ; \mathtt{where} \newline
b_0 = 1 / x\newline
b_n = 1 / \prod_{i = 0}^{n} {(x + i)}
$$

which follows from the property of the gamma function $\Gamma(x+1) = x * \Gamma(x)$. This series converges rapidly for $y < x + 1$. For the opposite case $y > x + 1$ the continued fraction expression for the upper incomplete gamma function converges faster:

$$
\Gamma(x,y) = e^{-y} y^x \left(\cfrac{1}{y + \cfrac{1 - x}{1 + \cfrac{1}{y + \cfrac{2 - x}{1 + \cfrac{2}{y + \cdots}}}}} \right) = \newline
= e^{-y} y^x \left(\cfrac{1}{y + 1 -x - \cfrac{1 - x}{y + 3 - x - \cfrac{2 * (2 - x)}{y + 5 - x - \cdots}}} \right)
$$

Thus, depending on the relation between the arguments either of the expressions should be used in order to calculate the value of the function directly or via its complementary counterpart.

References

[^Ref1]: [Ref 1]: William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery. **Numerical Recipes in C: The Art of Scientific Computing**. 2nd Ed. Cambridge University Press (1992). ISBN: 0-521-43108-5

[^Ref2]: [Ref 2]: Michael J. Wichura. *Algorithm AS241: The Percentage Points of the Normal Distribution.* Journal of Royal Statistical Society. Series C (Applied Statistics), Vol. 37, No. 3 (**1988**), pp. 477-484

[^1]: Numerical Recipes. p 214.

[^2]: Numerical Recipes. p 215.

[^3]: Wikipedia - error function

[^4]: Numerical Recipes. pp. 173 - 175

[^5]: Numerical Recipes. pp. 226 - 228

[^6]: Numerical Recipes. pp. 216 - 219