**CSC 202 Final Project Spring’15**

**Design and Implement the following algorithms, using NetBeans,and or any other compliers you chose..**

**Due Date is : April 30th.. No Late project will be accepted otherwise there is 50% point deduction.**

1. **Statistics Functions**

**Statisticians use various probability distributions and special statistical functions. Statistics texts often present these functions in tables in their appendices. There are also software packages that contain these functions. However, if a program you’re writing needs to use these functions, the tables in MatLab , Maple and Mathematica software packages are not what you want. You want code for those functions in your program!**

**This project’s description gives you algorithms for several popular statistical functions. The following UML class diagram shows how they are related.**

**Gamma**

+IMAX : int = 100

+ERROR : double = 1.0e-10

+logGamma(a : double) : double

+gamma(a : double) : double

+gamma(a : double, x : double) : double

+factorial(n : int) : double

+main(args : String[]) : void

**IncompleteGamma**

+incompleteGamma(a : double, t : double) : double

-series(a : double, t : double) : double

-continuousFraction(a : double, t : double) : double

+cumulativePoisson(m : int, x : double) : double

+cumulativeNormal(x : double) : double

+chiSquare(chiSquared : double, degreesFreedom : int) : double

+main(args : String[]) : void

**IncompleteBeta**

+incompleteBeta(a : double, b : double, x : double) : double

-continuousFraction(a : double, b : double, x : double) : double

+studentT(degFreedom : int, t : double) : double

+main(args : String[]) : void

**Beta**

+beta(a : double, b : double) : double

+beta(a : double, b : double, x : double) : double

+logBeta(a : double, b : double) : double

+main(args : String[]) : void

**Binomial**

+combinations(n : int, k : int) : double

+coefficients(n : int) : double[]

+probability(n : int, k : int, p double) : double

+cumulativeProbability(n : int, k : int, p : double) : double

+main(args : String[]) : void

**As in Java’s Math class, everything should be static. Give each class its own main driver, to demonstrate the functions in that class.**

* 1. **Gamma class**

**Provide IMAX and ERROR constants as common values for two derived classes – the IncompleteGamma class and the IncompleteBeta class.**

**In effect, the gamma function returns the “factorial” of a real number. The logGamma method is used not only for the gamma function itself but also for many other statistical functions. We usually compute the natural logarithm of gamma instead of gamma, because gamma itself can easily overflow, and most applications use ratios of gamma, and the ratios are typically much smaller than the individual gamma values. The algorithm for logGamma is a “magic” recipe invented by C. Lanczos, and we’ll just give it to you.**

**First, define an array of 6 constants:**

**coef[] ← {76.18009173, -86.50532033, 24.01409822,**

**-1.231739516, 0.120858003e-2, -0.536382e-5}**

**Then, provide an if statement that terminates execution if the parameter, a, is negative. Then do this:**

**n ← a – 1**

**temp ← n + 5.5**

**temp ← temp – (n + 0.5) \* loge(temp)**

**sum = 1**

**for all coef**

**n ← n + 1**

**sum ← coef / n**

**return –temp + loge(2.50662827465 \* sum)**

**In the gamma method, just exponentiate logGamma. When the argument, a, is an integer, gamma should return a value that is equal to the factorial of (a - 1). When a is not integer, gamma should return a value that varies continuously between the values returned by the adjacent integers. In other words, gamma(a), represented in mathematical formulas as Γ(a), is a floating point version of factorial(a - 1), or (a - 1)!. How about that!**

**The gamma distribution function, γ(a,x), generated by the two-argument gamma method, should simply display the formula:**

**γ(a,t) = x(a-1) e-t / Γ(a)**

**This gives the probability of taking a time somewhere in the range between t – ½ and t + ½ to complete a task. In general, the task is composed of several random-length subtasks, which must be done sequentially – not in parallel. The parameter, a, approximately represents the number of independent subtasks in the overall task.**

**Here is what the gamma distribution function looks like for various values of a:**

****

a = 4.0

a = 2.0

a = 1.0

a = 0.5

**The abscissa is t, and the ordinate is the value of γ(a, t). The black curve is for a = 0.5. The magenta curve is for a = 1.0. The yellow curve is for a = 2.0. The turquoise curve is for a = 4.0. When a = 1.0, there is exactly one subtask, and the gamma distribution becomes the simple exponential distribution. The area under each of these curves is 1.0.**

**In the factorial method, use two alternative algorithms. If n <= 20, fill the static array from nHi = 0 up to the currently needed factorial by multiplying the previous array element by the next long integer. If n > 20, long overflows, and you must use exp(logGamma(n + 1)).**

**In the main method, ask the user to enter a and print gamma(a) and factorial(a - 1). Then ask the user to enter t and print gamma(a,t). The Γ(a) values should be the same as (a - 1)! when a is an integer, and you can check your gamma(a) method by comparing its results with hand-calculated values of the factorial. You can check your gamma(a,t) method for a = 0.5, 1.0, 2.0, and 4.0, by comparing its outputs at various t with the curves displayed above.**

* 1. **IncompleteGamma class**

**The incomplete gamma function, ІΓ(a,t), gives the probability of completing a sequential independent random processes in some time less than t. This is the area under one of the gamma curves (like those shown above) to the left of some particular time, t. So, for any value of a, the incomplete gamma function rises monotonically from 0 to 1.0 as t increases from 0 to +infinity.**

**The algorithm for computing the incomplete gamma function is another magic recipe.[[1]](#footnote-1) In the incompleteGamma method, print a diagnostic message and terminate execution if a <= 0 or if t < 0. If t < a + 1, call the series helper method and return the value the helper returns. Otherwise, call the continuousFraction helper method, and return 1.0 minus the value the helper returns.**

**In the series method, initialize: aPlus ← a, term ← 1/a, and sum ← term. Then implement this algorithm:**

**for i from 0 to IMAX**

**aPlus++**

**term ← term \* t / aPlus**

**sum ← sum + term**

**if (abs(term / sum) < ERROR) break**

**After the for loop, if i >= IMAX, print a message indicating not enough iterations. Then return the value:**

**sum \* exp(-t + a \* loge(t) - logGamma(a))**

**Notice how you will normalize the answer by subtracting Γ(a) in the exponent. This will utilize a method inherited from the Gamma class.**

**In the continuousFraction method, initialize:**

**fac ← 1.0;**

**a0 ← 1.0;**

**b0 ← 0.0;**

**a1 ← t;**

**b1 ← 1.0;**

**gold ← 0.0;**

**g ← 0.0.**

**Then implement this algorithm:**

**for i from 0 to IMAX**

**ip1 ← i + 1**

**ip1ma ← ip1 - a**

**a0 ← (a1 + a0 \* ip1ma) \* fac**

**b0 ← (b1 + b0 \* ip1ma) \* fac**

**ip1f ← ip1 \* fac**

**a1 ← t \* a0 + ip1f \* a1**

**b1 ← t \* b0 + ip1f \* b1**

**if a1 > 1.0 // renormalize**

**fac ← 1.0 / a1**

**g ← b1 \* fac**

**if abs((g - gold) / g) < ERROR**

**break**

**else**

**gold ← g**

**After the for loop, if i >= IMAX, print a message indicating not enough iterations. Then return the value:**

**g \* Math.exp(-t + a \* Math.log(t) - logGamma(a))**

**Again, notice how you will normalize by subtracting Γ(a) in the exponent.**

**The cumulativePoisson method is supposed to generate the probability of having no more than “m” events in a time interval in which the expected number of events is x. This method should return the value generated by:**

**1.0 - incompleteGamma(m + 1, x)**

**The cumulativeNormal method is supposed to generate the area under a normalized Gaussian from –infinity up to an arbitrary input value, x. For this method, first compute:**

**value ← incompleteGamma(0.5, x \* x / 2.0)**

**Then, if x > 0 return:**

**0.5 + 0.5 \* value**

**Otherwise, return:**

**0.5 - 0.5 \* value**

**Assume some random process can be characterized by a particular distribution curve, like one of the gamma distribution curves shown above. Then, with vertical lines, partition that assumed curve into *bins*, and find the fraction of the total area under the curve in each bin minus the area between the adjacent pair of vertical lines that bound that bin. That area is the fraction of all experimental samples that should fall in that bin, if the assumed theoretical distribution is correct. *Chi Square* is the sum over all bins of the square of the difference between the observed number of observations in a bin and the theoretically expected number of observations for that bin, with each squared difference divided by the theoretically expected number for that bin. A large value of Chi Square says the assumed distribution does not fit the data very well. The question is, how large is too large? For the answer, a user should input the calculated Chi Square value, along with degrees of freedom (number of bins minus number of parameters adjusted to fit the distribution function to the data) into the chiSquare method. The probability returned by the chiSquare method is the probability that a correctly assumed distribution could result in a Chi Square value at least as large as the one input. If the value returned is greater than 0.1, the assumed distribution is believable. If it is less than 0.001, the assumed distribution is suspect.**

**Begin the chiSquare method by printing a diagnostic message and terminating execution if the input Chi Square value is less than zero or the degrees of freedom is less than one. If the input is OK, return the value given by:**

**1.0 - incompleteGamma(0.5 \* degreesFreedom, 0.5 \* chiSquare)**

**In the main method, ask the user to select among the alternatives: incompleteGamma(g), cumulativePoisson(p), normal(n), chiSquared(x). For the ‘g’ case, ask the user to enter values for “a” and “t” and print the value returned by the incompleteGamma method. For the ‘p’ case, ask the user to input the maximum number of events and the expected number of events, and print the value returned by the cumulativePoisson method. For the ‘n’ case, ask the user to enter a value for the normalized Gaussian random variable, x, and print the value returned by the cumulativeNormal method. For the ‘x’ case, ask the user to enter values for Chi Square and degrees of freedom, and print the value returned by the chiSquare method.**

***Sample sessions:***

***Select incompleteGamma(g), cumPoisson(p), cumNormal(n), chiSquared(x): g***

***Enter number of sequential random processes: 4***

***Enter normalized time: 3***

***Completion probability = 0.35276811126498503***

***Select incompleteGamma(g), cumPoisson(p), cumNormal(n), chiSquared(x): p***

***Enter expected number of events: 4***

***Enter maximum number of events: 3***

***probability of no more than 3 events is 0.4334701202981951***

***Select incompleteGamma(g), cumPoisson(p), cumNormal(n), chiSquared(x): n***

***Enter normalized Gaussian argument, x: 2.0***

***Cumulative Normal = 0.9772498680498296***

***Select incompleteGamma(g), cumPoisson(p), cumNormal(n), chiSquared(x): x***

***Enter Chi Square value: 22.307***

***Enter degrees of freedom: 15***

***Chi-Square Probability = 0.10000318893899185***

* 1. **Beta class**

**The beta function is Β(a,b) = Γ(a) Γ(b) / Γ(a + b). The logBeta method should return the value:**

**logGamma(a) + logGamma(b) - logGamma(a + b)**

**The two-parameter beta method should return the value:**

**exp(logBeta(a, b)).**

**The beta distribution function, β(a,b,x), is given by:**

**β(a,b,x) = x(a – 1.0) \* (1.0 – x)(b – 1.0) / Β(a,b)**

**This distribution function provides a wide range of useful shapes in the range 0.0 < x < 1.0: When a = 1 and b = 1, it generates the uniform distribution. When a = 2 and b = 1, it generates a linear up-ramp. When a = 1 and b = 2, it generates a linear down ramp. When a = 2 and b = 2, it generates an inverted parabola. When a < 1 and b < 1, it generates a bowl with infinitely high sides. The areas under all of these curves are exactly equal to 1.0. The three-parameter beta method should return the value generated by the java expression:**

**Math.exp((a - 1) \* Math.log(x) +**

**(b - 1.0) \* Math.log(1.0 - x) - logBeta(a, b))**

**The main method should ask the user to enter values for “a” and “b,” and print the value returned by the two-parameter beta method. Then it should ask the user to enter a value for “x,” and print the value returned by the three-parameter beta method.**

**Sample session:**

**Enter Beta parameters a and b: *5.0 1.5***

**Beta(a,b)= 0.07388167387892877**

**Enter x: *0.9***

**beta(a,b,x)= 2.808234118025584**

* 1. **IncompleteBeta class**

**The incomplete beta function, ІB(a,b,x), gives the area under β(a,b,x) between 0.0 and x. So, the incomplete beta function rises monotonically from 0 to 1.0 as x increases from 0 to 1.0.**

**In the incompleteBeta method, if x < 0 or x > 1.0, print an out-of-range message, and terminate execution. Then if x is equal to either 0.0 or 1.0, make the assignment:**

**multiplier ← 0.0**

**Otherwise, make the assignment:**

**multiplier ←**

**exp(a \* loge(x) + b \* loge(1.0 - x) - logBeta(a, b))**

**Then if x < (a + 1.0) / (a + b + 2.0) and a > 0, return the value:**

**multiplier \* continuousFraction(a, b, x) / a**

**Otherwise return the value:**

**1.0 - multiplier \* continuousFraction(b, a, (1.0 - x)) / b**

**The continuousFraction method does the real work in the evaluation of the incomplete beta function. It’s another “magic” recipe. Start by initializing several variables:**

**ap1 ← (a + 1.0);**

**am1 ← (a - 1.0);**

**apb ← (a + b);**

**az ← 1.0;**

**am ← 1.0;**

**bz ← 1.0 - apb \* x / ap1;**

**bm ← 1.0.**

**Then implement this algorithm:**

**for i from 1 through IMAX**

**twoI ← i + i**

**numerator ← i \* (b - i) \* x / ((am1 + twoI) \* (a + twoI))**

**ap ← az + numerator \* am**

**bp ← bz + numerator \* bm**

**numerator ← -(a + i) \* (apb + i) \* x /**

**((ap1 + twoI) \* (a + twoI))**

**app ← ap + numerator \* az**

**bpp ← bp + numerator \* bz**

**aOld ← az**

**am ← ap / bpp**

**bm ← bp / bpp**

**az ← app / bpp**

**bz ← 1.0**

**if abs(az - aOld) < ERROR \* abs(az)**

**break**

**After the for loop, if i > IMAX, print a message indicating excessive error. And finally, return the final value of az.**

**The studentT method computes the probability that the true mean of a measured phenomenon is within a range that is plus or minus [t \* (measured standard deviation)] from the measured mean.[[2]](#footnote-2) To use this method, you compute t by dividing (the difference between the mean of the measurements and some reference value) by the standard deviation determined from the same measurements. Then you call the studentT method with this computed value of t as the first argument, and degrees of freedom (one less than the number of measurements) as the second argument. If the probability returned by studentT is less than some acceptably small value, like 0.02, the chance that the true mean of the measured phenomenon is on the other side of the reference value is only 1% (half of this returned probability).**

**Implement the studentT method by making the assignment:**

**x ← degFreedom / (degFreedom + t \* t)**

**And then returning the value returned by:**

**incompleteBeta(0.5 \* degFreedom, 0.5, x)**

**In the main method, ask the user to select either incompleteBeta(b) or studentT(t). For case ‘b’, ask the user for Beta arguments “a” and “b”, and print the value returned by the incompleteBeta method. For case ‘t’, ask the user for degrees of freedom and the student t value, and print the value returned by the studentT method.**

**Sample sessions:**

**Select incompleteBeta(b) or studentT(t): *b***

**Enter Beta parameters a and b: *5.0 1.5***

**Enter x: *0.9***

**incomplete beta function= 0.7761721343078992**

**Select incompleteBeta(b) or studentT(t): *t***

**Enter degrees of freedom: *15***

**Enter value of student t: *1.753***

**two-sided overlap probability= 0.10000889662154729**

* 1. **Binomial class**

**This class will use the one-parameter gamma method inherited from the Gamma class and the incompleteBeta method inherited from the IncompleteBeta class.**

**In the combinations method, if k < 0 or k > n, print an outside-of-range message and terminate the execution. Then, implement this:**

**logCoef ←**

**logGamma(n + 1) - logGamma(k + 1) - logGamma(n - k + 1)**

**coef ← exp(logCoef)**

**And return the rounded value of coef.**

**In the coefficients method, create an array of length n + 1, load each element, k, of this array with the value returned by combinations(n, k), and return a reference to the new array. The elements of this array should be equal to the coefficients of a binomial.**

**In the probability method, if k > n, return 0.0. Otherwise, return the value of this expression:**

**combinations(n,k) \* pk (1 - p)(n - k)**

**If the probability of a binary event is p per trial, this returns the probability of that event occurring exactly k times in n trials.**

**In the cumulativeProbability method, if k > n return 0.0. Otherwise, return the value of this expression:**

**incompleteBeta(k, (n - k + 1), p)**

**Assuming the probability of a binary event is p per trial, this returns the probability of that event occurring k or more times in n trials.**

**In the main method, ask the user to select between coefficients(c) or probability(p). For the ‘c’ case, ask the user to enter the order, n. Then print the value of each of the coefficients in the array returned by a call to the coefficients method. For the ‘p’ case, ask the user to enter the total number of trials, the number of successes or minimum successes, and the single-trial probability. Then print the value returned by a call to the probability method, and also print the value returned by a call to the cumulativeProbability method.**

**Sample sessions:**

**Select coefficients(c) or probability(p): *c***

**Enter order: *4***

**1.0**

**4.0**

**6.0**

**4.0**

**1.0**

**Select coefficients(c) or probability(p): *p***

**Enter total number of trials: *6***

**Enter successes or minimum successes: *4***

**Enter single-trial probability: *.5***

**probability of 4 successes = 0.234375**

**probability of at least 4 successes = 0.34375000002342193**

1. Based on algorithms in Section 6.2 in Press, et. al., *Numerical Recipes in C*, Cambridge University Press (1991). [↑](#footnote-ref-1)
2. See, for example, Section 4.5 in Law and Kelton, *Simulation Modeling & Analysis*, McGraw-Hill (1982). [↑](#footnote-ref-2)