

Binomial Distribution Computational Analysis in R

Alan Nawoj

August 12, 2026

Background and Motivation

This analysis is the result of my incorrect assumption of how R calculates a **Binomial distribution** for large values of n and my desire for a clear understanding of the logic that is used as well as its assumptions and limitations. Since the discrete Binomial distribution's density formula involves the binomial coefficient $\binom{n}{x}$, which in turn requires the calculation of factorials, my initial assumption was that the logic used by R to compute the Binomial distribution had some sort of "fallback" logic for large values of n where it would instead utilize the Normal approximation to the Binomial distribution to compute the end result and avoid the likely overflow issues that would arise when large factorial computations would exceed a computer's arithmetic limitations.

However, after reviewing the current open-source C code for the `dbinom()` R function at <https://svn.r-project.org/R/trunk/src/nmath/dbinom.c> (R Core Team and Loader 2000b), it became clear that this was **not** how the logic actually works. In fact, the logic for the `dbinom_raw()` function within this source file reveals that the same logic is used for **any** value of n and is largely based on an algorithm presented in (Loader 2002). What follows is a step-by-step analysis of how this algorithm works for any values of n , both large and small.

Binomial Distribution Overview

The Binomial distribution is a discrete probability distribution defined by two parameters, n and p , where n represents the number of Bernoulli trials performed and p represents the probability of success on any single trial. Mathematically, this is defined by the following expression, where X represents a Binomial random variable and x represents one of the possible outcomes between 0 and n inclusive.

$$P(X = x) = \binom{n}{x} p^x (1 - p)^{n-x}$$
$$p(x; n, p) = \frac{n!}{x!(n-x)!} p^x (1 - p)^{n-x}$$

The presence of factorial terms in this formula is what lies at the heart of the challenge of computing an end result probability value in $[0, 1]$ since factorials get very large very quickly.

Key Challenges

The current implementation of the binomial distribution calculations in R stems from addressing two key challenges in computing the formula listed earlier:

1) Overflow with Factorials

For most modern computers, the maximum value that can be stored in a 64-bit double precision number (IEEE 754 Standard) is $\approx 1.79 \times 10^{308}$ (represented by the `DBL_MAX` constant in C). This `DBL_MAX` value gets exceeded as soon as you go from computing the factorial result for 170! to 171! which is a classic *overflow* scenario meaning that the computer is not able to store a value that large and returns `Inf` (infinity) instead. This is problematic since a sample size of $n = 171$ is not even that large for most experiments.

The original solution to this particular factorial overflow problem was simply to convert the computation from linear space to **log space** to greatly reduce the magnitude of the factorial results. By taking the natural log of both sides of the equation above, the following formula was derived to mitigate the overflow challenge:

$$\ln(P(X = x)) = \ln\left(\binom{n}{x} p^x (1-p)^{n-x}\right) \rightarrow \ln\left(\binom{n}{x}\right) + x \ln(p) + (n-x) \ln(1-p)$$

If we then convert the binomial coefficient to its factorial representation (i.e. $n!/(x!(n-x)!)$), the natural logarithm of this quotient turns into the following:

$$\ln(P(X = x)) = \ln(n!) - \ln(x!) - \ln((n-x)!) + x \ln(p) + (n-x) \ln(1-p)$$

The end result of this calculation is then exponentiated back into linear space to get the probability value, i.e. $e^{\ln(P(X=x))} = P(X = x)$. As a side note, implementing this calculation was originally done by calling the **log-gamma function**. Recall that the gamma function Γ is the smooth continuous analogue to the discrete factorial function with the following equivalence, $n! = \Gamma(n+1)$. Therefore, taking the log of both sides of the factorial-gamma formula gives us the equation, $\log(n!) = \log(\Gamma(n+1))$, where the right hand side is implemented by the `lgamma()` function in many math libraries or packages.

2) Catastrophic Cancellation (Precision Loss) in Log Space

While converting the problem calculations to log space and then converting the end result back to linear space effectively solves the overflow challenge, it inadvertently introduces a new issue due to the precision limitations of 64-bit double precision numbers. In a 64-bit double precision number (IEEE 754 Standard), the computer sets aside 1 bit to represent the sign (+ or -) of the number, 11 bits to represent the exponent, and the remaining 52 bits (binary digits) to represent the **significant digits** of the number (also called the mantissa or significand) which equates to about 15 to 17 significant digits in base 10 (decimal). It is key to note that these 15 to 17 significant digits (in base 10) include all significant digits to the left and right of the decimal point. For instance, a really small number like 0.000001234567890123456 uses 16 significant digits all to the right of the decimal point, whereas a number like 12815518.38465817 uses 8 significant digits to the left of the decimal point and 8 to the right of the decimal point (16 total), and a really large number like 123456789012345600000000.0 uses all 16 significant digits to the left of the decimal point.

The issue is that even though the log values of these factorials are much, much smaller than the regular factorial values, they are still quite large. In fact, they are typically **many orders of magnitude** larger than the end result of the log term additions and subtractions. Remember that we must eventually exponentiate the end result of these log term additions and subtractions (i.e. $e^{\ln(P(X=x))} = P(X = x)$) to get a final probability in the range of (0,1) assuming $p \in (0,1)$, so we would expect the end result of the log term additions and subtractions to be a negative number most likely of a magnitude just in the (negative) **tens**. For reference, $\ln(0.0000000000000001) \approx -37$ and of course $\ln(1) = 0$. This means we are likely adding and subtracting numbers that are on the order of magnitude in the *millions* to get an end result that is on the order of magnitude of (negative) *tens* (or ones) in order to then exponentiate that number to get our final probability. Using the reference example just mentioned, if the end result of the log term additions and subtractions came out to -37 , then the final probability would be $e^{-37} \approx e^{\ln(0.0000000000000001)} = 0.0000000000000001$ which is a probability that is virtually zero.

If you have a large log factorial number that it is in the millions, then perhaps *half* (i.e. 8) of the roughly 16 available significant digits get used up by the integer portion and the computer must truncate the fractional

portion of the number down to 8 significant digits. Losing fractional precision at this stage only gets amplified when we raise e to the power of the end result of these log term additions and subtractions. As a result, the final probability $P(X = x)$ loses a lot of accuracy which puts the entire calculation in question, and this sets the stage for the algorithm presented in (Loader 2002).

To emphasize this “orders of magnitude precision challenge”, let us consider a scenario where a coin is flipped 1,000,000 times with a probability of success of 0.5 and we want to know the probability of getting 500,000 heads. The parameters of this example and the resulting calculations using the log space operations *with only 16 significant digits of precision* are as follows:

$$\begin{aligned} n &= 1000000, p = 0.5, x = 500000 \\ \ln(P(X = 500000)) &= \ln(1000000!) - \ln(500000!) - \ln(500000!) + 500000 \ln(0.5) + 500000 \ln(0.5) \\ \ln(P(X = 500000)) &\approx 12815518.38465817 - 12122378.34764511 - 693147.18055995 \approx -7.14354689 \\ e^{\ln(P(X=500000))} &\approx e^{-7.14354689} \\ P(X = 500000) &\approx e^{-7.14354689} \approx \mathbf{0.000789945272673} \end{aligned}$$

If we were able to keep 20 digits of decimal precision on the log factorial results, for instance, as seen in the next two lines, then we would have had the following result which differs from the previous calculation at just the second significant digit (i.e. 0.00078 vs 0.00079) which is a notable difference!

$$\begin{aligned} \ln(P(X = 500000)) &\approx -7.13354688162686448438 \\ P(X = 500000) &\approx e^{-7.13354688162686448438} \approx \mathbf{0.000797884361332} \end{aligned}$$

Notice how we are adding numbers with similar magnitudes in the *millions*, and therefore implicitly having to truncate decimal precision to account for the large integer portion of these numbers, since you only get roughly 15 to 17 significant base 10 digits in a 64-bit double precision number. Subtracting two roughly similar large numbers in the millions clears out most of the stable higher-order digits, leaving mostly the truncated decimal portion. In the end, we get a final result whose most significant digit is in the **ones**. This intermediate result of ≈ -7.14 , unfortunately, incorporates a great deal of precision loss which then gets amplified when it is exponentiated (i.e. $e^{-7.14\dots}$) to get our final probability.

Current R Implementation of Binomial Distribution Calculations

The current version of the binomial density calculation in R based on (Loader 2002) addresses **both** of the challenges discussed in the previous section by still doing calculations in log space but aiming to keep all of the calculations on the **same order of magnitude as the expected log result** before it is then converted back to linear space.

This is cleverly accomplished by replacing those large log factorial terms (e.g. $\ln(n!)$, $\ln(x!)$, etc.) using **Stirling’s Exact Formula** which is defined as $\ln(n!) = \ln(\sqrt{2\pi n}) + n \ln(n) - n + \text{stirlerr}(n)$. This formula can be divided into two parts: the log-factorial approximation part (i.e. $\ln(\sqrt{2\pi n}) + n \ln(n) - n$) and the error function part (i.e. $\text{stirlerr}(n)$). Together, these two parts result in a highly accurate calculation for the natural logarithm of $n!$. Using Stirling’s Exact Formula and performing various algebraic and logarithmic manipulations and simplifications, the end result is a formula with functions that produce values on the *same order of magnitude* as the final result we expect to see. Specifically, these component functions are the **Stirling’s approximation error function** (`stirlerr()` in R) and the **binomial deviance function** (`bd0()` in R).

Let’s first take a high-level look at the source code for R (written in C) that gives the core formula for this binomial distribution calculation before we review the mathematics for how it was derived.

Source Code

In R, the `dbinom()` function calculates the probability mass function (PMF) for the binomial distribution as follows:

```
dbinom(x, size, prob, log = FALSE)
```

Where:

- `x` = number of successes you want to find the probability for
- `size (n)` = total number of independent trials
- `prob (p)` = probability of success on any individual trial
- `log (optional)` = if set to `TRUE`, returns the log-probability; default is `FALSE`

Under the hood, this logic is implemented using the C programming language and the current R source code for this `dbinom()` function written in C can be found here:

R Source Code Repository File Location: <https://svn.r-project.org/R/trunk/src/nmath/dbinom.c>

The following code snippet from this file removes all of the logic checks, comments, etc. that are not part of the core algorithm for computing the probability of $X = x$ where X follows a binomial distribution in order to isolate the key components of how this algorithm works. As you can see, when you call `dbinom()` in R, the user-facing function `dbinom()` hands over the calculations to an internal function called `dbinom_raw()` which, in turn, computes two key intermediate double precision values, `lc` (line 13) and `lf` (line 15) before returning the final result to the user as either `lc - 0.5*lf` (log-probability value) or `exp(lc - 0.5*lf)` (true probability value) in line 17.

```
1 double dbinom(double x, double n, double p, int give_log)
2 {
3     // OMITTING LOGIC / COMMENTS NOT RELEVANT TO CORE BINOMIAL DISTRIBUTION CALCULATIONS
4
5     return dbinom_raw(x, n, p, 1-p, give_log);
6 }
7
8
9 double dbinom_raw(double x, double n, double p, double q, int give_log)
10 {
11     // OMITTING LOGIC / COMMENTS NOT RELEVANT TO CORE BINOMIAL DISTRIBUTION CALCULATIONS
12
13     double lc = stirlerr(n) - stirlerr(x) - stirlerr(n-x) - bd0(x,n*p) - bd0(n-x,n*q);
14
15     double lf = M_LN_2PI + log(x) + log1p(- x/n);
16
17     return R_D_exp(lc - 0.5*lf);
18 }
```

Mathematical Derivation

As mentioned earlier, the large log factorial terms that result in a notable loss of precision when added or subtracted to produce a relatively tiny end result are replaced by Stirling's Exact Formula. The simplified log scale formula then consists of two key functions, the Stirling's approximation error function (`stirlerr()`) and the binomial deviance function (`bd0()`), which produce values that are of the same order of magnitude as the final result.

- *Source Code for `stirlerr()`*: <https://svn.r-project.org/R/trunk/src/nmath/stirlerr.c>
- *Source Code for `bd0()`*: <https://svn.r-project.org/R/trunk/src/nmath/bd0.c>

Deriving this new formula starts with the original binomial distribution function which we saw earlier:

$$P(X = x) = \binom{n}{x} p^x (1-p)^{n-x} = \frac{n!}{x!(n-x)!} p^x (1-p)^{n-x}$$

We then convert this formula to log space as seen previously to avoid 64-bit overflow:

$$\ln(P(X = x)) = \ln(n!) - \ln(x!) - \ln((n-x)!) + x \ln(p) + (n-x) \ln(1-p)$$

The next step involves replacing the log factorial terms with **Stirling's Exact Formula** for a log factorial, which has the expression $\ln(n!) = \ln(\sqrt{2\pi n}) + n \ln(n) - n + \text{stirlerr}(n)$. Note that the **stirlerr**(n) part of the formula is an error function that represents the gap between the Stirling's log factorial approximation (i.e. $\ln(\sqrt{2\pi n}) + n \ln(n) - n$) and the true log factorial value as a function of n . In general, the value of **stirlerr**(n) approaches zero as n gets larger, so the magnitude of **stirlerr**(n) is quite small in most circumstances. Therefore, the three different log factorial terms for the $\ln(P(X = x))$ expression are as follows:

$$\ln(\mathbf{n}!) = \ln(\sqrt{2\pi n}) + n \ln(n) - n + \text{stirlerr}(n)$$

$$\ln(\mathbf{x}!) = \ln(\sqrt{2\pi x}) + x \ln(x) - x + \text{stirlerr}(x)$$

$$\ln((\mathbf{n} - \mathbf{x})!) = \ln(\sqrt{2\pi(n-x)}) + (n-x) \ln((n-x)) - (n-x) + \text{stirlerr}((n-x))$$

After substituting these multi-term expressions back into the log space equation for $\ln(n!)$, $\ln(x!)$, and $\ln((n-x)!)$ respectively, some of the terms end up cancelling out, other terms can be algebraically combined, and we then end up with the intermediate log space equation which is broken up over multiple lines for easier readability:

$$\begin{aligned} \ln(P(X = x)) = & \ln\left(\sqrt{\frac{n}{2\pi x(n-x)}}\right) \\ & + \text{stirlerr}(n) + \text{stirlerr}(x) + \text{stirlerr}(n-x) \\ & + n \ln(n) - x \ln(x) - (n-x) \ln(n-x) \\ & + x \ln(p) + (n-x) \ln(1-p) \end{aligned}$$

To further simplify the last two lines that contain the various logarithmic terms, we can perform an algebraic trick by substituting $x + (n-x)$ in place of n for the $n \ln(n)$ term, combine the resulting like terms, and then apply basic logarithmic rules to end up with the following simplification:

$$\begin{aligned} \ln(P(X = x)) = & \ln\left(\sqrt{\frac{n}{2\pi x(n-x)}}\right) \\ & + \text{stirlerr}(n) + \text{stirlerr}(x) + \text{stirlerr}(n-x) \\ & + x \ln\left(\frac{np}{x}\right) + (n-x) \ln\left(\frac{n(1-p)}{n-x}\right) \end{aligned}$$

If we then invert the fractions inside of the $\ln$ terms on the last line, we can pull out a negative sign and rewrite the log space equation as follows:

$$\begin{aligned} \ln(P(X = x)) = & \ln\left(\sqrt{\frac{n}{2\pi x(n-x)}}\right) \\ & + \text{stirlerr}(n) + \text{stirlerr}(x) + \text{stirlerr}(n-x) \\ & - \left[x \ln\left(\frac{x}{np}\right) + (n-x) \ln\left(\frac{n-x}{n(1-p)}\right) \right] \end{aligned}$$

The bracketed term on the last line is the exact formula (minus a coefficient of 2 in front of the bracket) for a concept known as **binomial deviance** which is a goodness of fit measure between an observed value x and an expected value np . The binomial deviance function is typically written as $D(x; n, p) = x \ln(\frac{x}{np}) + (n-x) \ln(\frac{n-x}{n(1-p)})$ and is implemented in the R source code as `bd0(x,n*p) + bd0(n-x,n*q)` to represent the two terms, where $bd0(x, np) = x \ln(x/np)$ and $bd0(n-x, nq) = (n-x) \ln((n-x)/(nq))$ and where $q = 1 - p$. The binomial deviance formula is essentially the log-likelihood ratio statistic (i.e. log of observed value over expected value) comparing a fitted model with a perfect theoretical model. Crucially, notice that none of the terms left in this altered version of the log space binomial probability evaluation involve any factorials since they were effectively removed when Stirling's Exact Formula was substituted into the original log space equation! All of these terms are on a similar order of magnitude as the expected end result which solves the second challenge (high loss of precision) mentioned earlier.

If we substitute the binomial deviance functions, `bd0(x,n*p) + bd0(n-x,n*q)`, inside of the final bracketed part of this log space expression, we end up with this final version in log space:

$$\begin{aligned} \ln(P(X = x)) = & \ln\left(\sqrt{\frac{n}{2\pi x(n-x)}}\right) \\ & + \text{stirlerr}(n) + \text{stirlerr}(x) + \text{stirlerr}(n-x) \\ & - [bd0(x, np) + bd0(n-x, n(1-p))] \end{aligned}$$

To get the final probability, which is the entire goal of this exercise, we simply raise e to the power of both sides of this equation which gets us back to $P(X = x)$ on the left side, cancels out the $\ln$ of the square root term, and raises the remaining $\text{stirlerr}()$ and $bd0()$ function terms as a power of e . This leaves us with the following formula which works for any values of n in the R implementation of `dbinom()`:

$$P(X = x) = \sqrt{\frac{n}{2\pi x(n-x)}} e^{(\text{stirlerr}(n) - \text{stirlerr}(x) - \text{stirlerr}(n-x) - [bd0(x, np) + bd0(n-x, n(1-p))])}$$

This is the exact formula that is implemented in R for `dbinom()` as listed earlier in the source code highlights.

For instance, you can clearly see how the following variable represents the exponent of this formula:

```
double lc = stirlerr(n) - stirlerr(x) - stirlerr(n-x) - bd0(x,n*p) - bd0(n-x,n*q);
```

The large square root coefficient, however, isn't written into the code in exactly the way it appears above (i.e. $\sqrt{n/[2\pi x(n-x)]}$) mainly for performance and underflow protection reasons. Instead, we perform some algebraic and logarithmic manipulations to the square root coefficient after taking its natural log and raising that as a power of e . This effectively converts the large square root coefficient to the following:

$$e^{\ln(\sqrt{\frac{n}{2\pi x(n-x)}})} = e^{\frac{1}{2} \ln(\frac{n}{2\pi x(n-x)})} = e^{-\frac{1}{2} \ln(\frac{2\pi x(n-x)}{n})} = e^{-\frac{1}{2} \ln(2\pi x(1-\frac{x}{n}))} = e^{-0.5[\ln(2\pi) + \ln(x) + \ln(1-(x/n))]}$$

It is now clear how the bracketed portion of the exponent in this reworked expression of the large square root coefficient maps to the variable `lf` in the source code where the `log1p()` function computes the log of 1 plus its input.

```
double lf = M_LN_2PI + log(x) + log1p(- x/n);
```

The return statement of the binomial distribution probability calculation function is where these two exponent pieces, `lc` and `-0.5*lf` are combined and raised as an exponent of e to give us the final probability value as seen here in this remaining line of code:

```
return R_D_exp(lc - 0.5*lf);
```

As a final note, we have already seen the inner workings of the `bd0(x,np)` function, which is just a variable (e.g. x) times the natural log of a ratio (e.g. x/np), so it does not generate any massive, factorial-scale numbers (see source code at <https://svn.r-project.org/R/trunk/src/nmath/bd0.c> (R Core Team and Loader 2000a)). As for the `stirlerr(n)` function inner workings, the R source code (listed at <https://svn.r-project.org/R/trunk/src/nmath/stirlerr.c> (R Core Team and Loader 2000c)) shows how this function value is either retrieved directly from a precomputed table if $n \leq 15$ or its value is computed from an asymptotic series if $n > 15$. The asymptotic Stirling error function series is an infinite series using Bernoulli numbers and it looks like $\frac{1}{12n} - \frac{1}{360n^3} + \frac{1}{1260n^5} - \frac{1}{1680n^7} + \dots$, but an extremely high precision result can be obtained in just the first dozen or so terms of this series. In fact, in the R source code, when the `stirlerr()` function uses the asymptotic series to compute the error value, the maximum it ever uses is the first 17 terms of the series.

References

- Loader, Catherine. 2002. *Fast and Accurate Computation of Binomial Probabilities*. Lucent Technologies. <https://www.r-project.org/doc/reports/CLoader-dbinom-2002.pdf>.
- R Core Team, and Catherine Loader. 2000a. *Bd0.c: Binomial Deviance Implementation*. R Project Subversion Repository. <https://svn.r-project.org/R/trunk/src/nmath/bd0.c>.
- R Core Team, and Catherine Loader. 2000b. *Dbinom.c: Binomial Probability Distribution Implementation*. R Project Subversion Repository. <https://svn.r-project.org/R/trunk/src/nmath/dbinom.c>.
- R Core Team, and Catherine Loader. 2000c. *Stirlerr.c: Stirling Error Correction Implementation*. R Project Subversion Repository. <https://svn.r-project.org/R/trunk/src/nmath/stirlerr.c>.