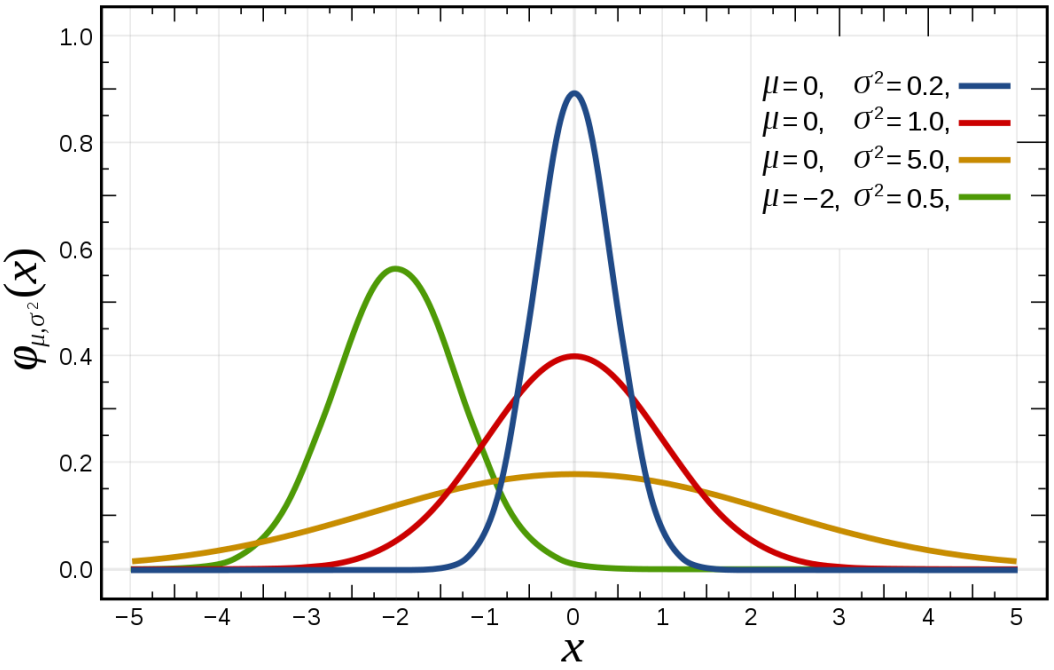
Probability Distributions and Their Use in Simulations

What is a Probability Distribution?

A probability distribution describes how likely certain outcomes of an event are. Probability distributions come up in a lot of different settings, but generally in simulations they are used to randomly generate numbers. Functions in R that look like **rbinom** or **rnorm** are the tools of choice for this task. The "**r"** in these function names stands for "random" and the rest of the name is an abbreviation for the probability distribution that the function uses to generate random numbers.

Probability Distributions are often depicted with graphs that plot particular values on the x-axis (corresponding to events, or rather numbers that describe particular events) and the relative probability of that value on the y-axis:



One way to think about probability distributions is to think about what happens when you generate random numbers using one of the R functions mentioned above. Any individual number generated won't tell you much about the distribution you used to generate the number, but if you generate a crap ton of random numbers from a distribution and plot a histogram of the random numbers, it will take on a shape that looks identical to the plot of the probability distribution (or rather of its probability density/mass function, which is what is plotted above).

The shape and location of a probability distribution is determined by two things:

1. The properties of the particular probability distribution
    a. Example: Normal distributions are symmetric about a mean and can take on any value along the number line whereas exponential distributions have a skew to them and can only take on positive real number values.
2. Parameters that set the finer details of the probability distribution
    a. Example: The figure above depicts several different normal distributions, differing by the choice of parameter values. For a normal distribution, there are two parameters, the mean (μ) and variance (σ^2), which describe the center of the distribution and its width, respectively

When generating random numbers for simulations, you have to consider both of these things. The properties of different probability distributions should be suited to describe whatever you are simulating, and the parameters are chosen so that your simulated data looks reasonable, or at least how you want it to look.

The Two Types of Probability Distributions:

1. Continuous Distributions: These distributions spit out (spit out = what R gives you if you simulate from the respective random number generating function) any number from some portion of the number line (i.e., real numbers). Some only spit out positive numbers (e.g., the exponential distribution), others only give you numbers between certain bounds (e.g., the beta distribution), and some have no bounds at all (e.g., the normal distribution).
    a. If the thing you are simulating doesn't have to be an integer, these are great distributions to look to. For example, simulating fraction news or degradation and synthesis rate constants in a TimeLapse experiment will involve continuous distributions.
2. Discrete Distribution: These distributions only spit out integers. Some only have a few possible outputs (e.g., the Bernoulli distribution only outputs 0 or 1 and the Binomial distribution will only output numbers less than or equal to a chosen upper bound), and some can have a practically infinite set of possible outputs (e.g., the Poisson distribution, which has no hard upper bound).
    a. When simulating things like number of sequencing reads or number of T to C mutations in a given sequencing read, look to these distributions. More specifically, a negative binomial distribution works well for the former and a binomial distribution for the latter.

Math Jargon to Know

1. Random variable
    a. The term random variable has a very particular mathematical definition, but for our sake, we can think of it is as a random number that has some probability of occurring.
    b. For example, the time until a Polymerase initiates at an open promoter has some value that will differ each time that promoter opens up, and will differ across all of the identical promoters in all the cells in your sample. These times until initiation are a random variable that follow some probability distribution (an exponential distribution is the usual model in this case).
    c. Random variables are what functions like **rnorm** and **rbeta** generate in R
2. "X is distributed according to a ___ distribution" or X ~ F(a, b)
    a. This means that X is a random variable, and if you collected a bunch of samples of X (e.g., times until initiation occurs, as in the above example) and plotted a histogram of all your samples, it would look like the density/mass function of the distribution listed. F(a, b) in this case is merely anecdotal, and represents the general form that these statements about probability distributions often take, with a and b being parameters of the distribution F.
3. Moments of a Distribution
    a. Moments are single values that describe broad properties of a distribution. The most common moments to talk about are the mean and variance, which give a rough sense of the center and range of random variables you'd expect to get if you simulated from that particular distribution.
    b. More specifically the nth moment of a probability distribution is the average value of random variables X to the power n (X^n) generated from that distribution. So, the first moment is just the average value of X, or the mean of the distribution, the second moment is the average value of X^2, and so on.
    c. These can be thought of in terms of simulations. Generate 1000000 random variables from a distribution of your choice using R (e.g., **rnorm** or **rpois**). Next raise each random number to the power of n. Finally, average the exponentiated numbers. You have approximated the nth moment of the distribution you simulated. Note, every moment will be a function of the parameters of the distribution, and thus different parameters for the same distribution may yield different values for each of the moments.
    d. Technically, I have only talked about 0-centered moments, that is the average of $(X - 0)^n$. You can also consider moments with a different center, like the mean. Nth moments centered about the mean are then the average value of $(X - \text{mean}(X))^n$. The first moment about the mean is 0 (since it is just mean(X) – mean(X)), but the second moment about the mean is a very commonly discussed moment called the variance. Its square root is the even more well-known standard deviation.

e. When working with simulations from distributions, it is best practice to think about what the mean and variance (or standard deviation) of the distributions you use are. This will help you make informed choices about the parameter values you set for the distributions you use.

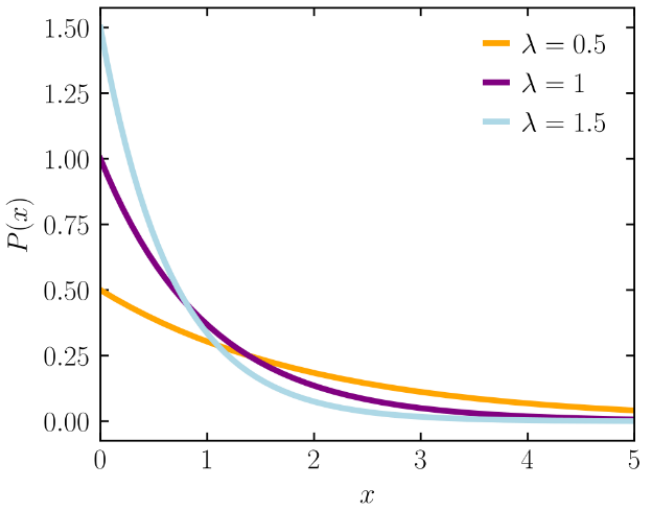
4. Probability Density Function
    a. Probability density functions describe probabilities of various outcomes for CONTINOUS random variables. They are tough to interpret though, because when the random variable is continuous, the probability of any single exact value is 0 (I'm stating this as a fact, but I admit it is unintuitive. This fact can be understood more easily with integral calculus, which is not used here). Thus, we can only talk about the probability that a continuous random variable takes on a range of values. If the width of that range is some small positive number ϵ, then the probability that the random variable falls in that range is ϵ*(average probability density in this range).
    b. Note, it is called a probability DENSITY function because if you multiply it by a "volume" (e.g. the length of an interval on the real line, which is a 1-D volume), you get a probability MASS, which is an actual probability like the ones you are familiar with.
5. Probability Mass Function
    a. Probability mass functions describe probabilities of various outcomes for DISCRETE random variables. They are much easier to interpret because the y-axis value of plots of these are actual probabilities for a particular event, a.k.a. the probability mass for that event.

Important Continuous Distributions to Know

1. The Normal Distribution (Math notation: $X \sim Normal(\mu, \sigma)$ means X is a random variable distributed according to a Normal distribution with mean μ and standard deviation σ) See top of document for picture of probability density functions
    a. Use in simulations
        i. Anytime you want to simulate a continuous random variable that is symmetric about an average value, the normal is a great choice. Even if the thing you are simulating is bounded above and/or below, you can always simulate a transformed version that isn't bounded using a normal distribution, and then transform the simulated values back to the original scale
        ii. Experimental noise and replicate variability are best simulated with normal distributions
        iii. Example: I use normal distributions to simulate effect sizes, like how much the fraction new changes when a particular experimental condition is applied. Fraction news are bounded between 0 and 1, but if I generate a bunch of random normally distributed numbers and then perform the inverse logit transformation on them, they will go from being unbounded to being bounded between 0 and 1. Putting a distribution on the effect size also allows me to model experimental noise in all the parameters of interest (see case study at end of document for details).
    b. R function = **rnorm**
    c. Lower bound = - Infinity; Upper Bound = Infinity
    d. Parameters:
        i. Mean (μ): Tells you where the peak of the distribution is. The Normal distribution is symmetric about its mean
            1. μ can be any real number
        ii. Standard Deviation (σ): Tells you how spread out the distribution is. When generating a random normally distributed number, there is a 95% probability the random number will be within two standard deviations of the mean.
            1. $\sigma > 0$
            2. Sometimes you will see σ^2 listed as a parameter, which is called the variance; literally just the square of the standard deviation
    e. Neat properties
        i. It is symmetric

ii. It pops up all over the place because of the incredibly important Central Limit Theorem. This theorem can be thought of in terms of a simulation:
    1. Choose a random continuous distribution, it doesn't have to be a normal distribution
    2. Generate a crap ton of random numbers from that distribution (e.g., 100000)
    3. Take the average of those random numbers (or you can even just sum them)
    4. Repeat a crap ton of times
    5. Make a histogram of the averages (or sums) and voila, your histogram will look like a normal distribution, no matter what random distribution you chose
        a. The mean and standard deviation will depend on the mean and variance of the distribution you used to generate the random variable

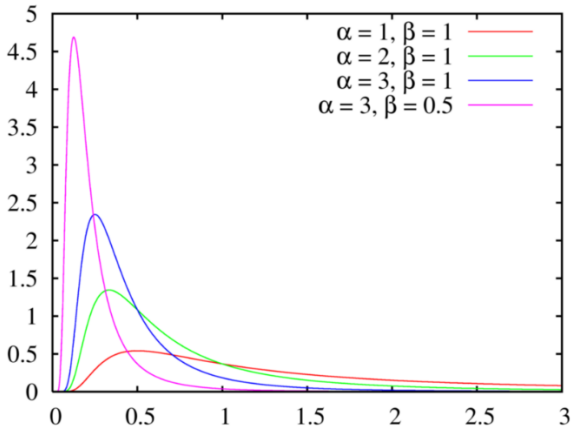
2. The Exponential Distribution (Math notation: X ~ Exponential(λ) means X is distributed according to an Exponential with rate λ)



Exponential(λ) density function

    a. Use in simulations
        i. Exponential distributions arise naturally whenever you are simulating a process that occurs with a constant rate. In that case, the time until the process you are simulating occurs follows an exponential distribution
        ii. Example: I've used Exponential distributions in some STL-seq simulations that explicitly model single Pol II pausing events. The times until initiation and release/termination are modeled as following exponential distributions
    b. R function = **rexp**
    c. Lower bound = 0; Upper bound = Infinity
    d. Parameters
        i. Rate (λ), though sometimes mean (μ = 1/ λ) is used instead
            1. Either way, must be > 0
    e. Moments of Interest
        i. Mean = 1/ λ
        ii. Variance = 1/ λ^2
    f. Neat properties
        i. The memoryless property: Imagine something you are simulating follows an exponential distribution, for example, the time until a reaction occurs. Also assume that you know a reaction hasn't occurred for 10 minutes. The probability of the time until the reaction does occur follows the exact same exponential distribution it did 10 minutes ago. No matter how long you have already waited, the distribution describing the time until the next reaction remains the same.

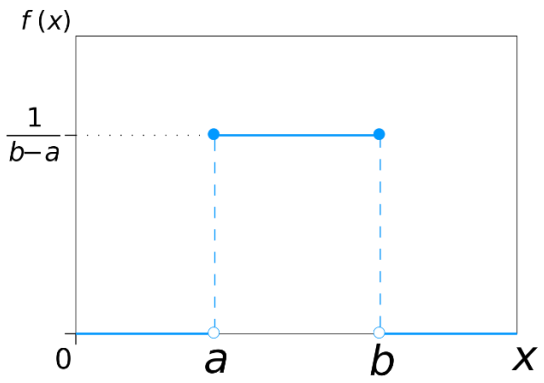
3. The Gamma Distribution (Math notation: X ~ Gamma(α, β) means X is distributed according to a Gamma with shape factor α and scale factor β)



Gamma(α, β) density function

a. Use in simulations
    i. Gamma distributions have typically shown up in simulations in the lab as flexible models for positive random variables. For example, if you want to randomly generate synthesis and degradation rate constants for 10,000 transcripts, a Gamma distribution might be a suitable choice for this.
b. R function = **rgamma**
c. Lower bound = 0; Upper bound = Infinity
d. Parameters
    i. α = shape factor
        1. α > 0
    ii. β (also commonly called λ) = scale factor, or often called the rate
        1. β > 0
e. It's harder to understand how these parameters shape the distribution, but you can get a sense by looking at how they determine the important moments:
        1. Mean = α / β
        2. (Standard deviation)^2 = α / β^2
f. Neat Properties
    i. If you let Z = the sum of n random variables that are independent (their values aren't correlated, though there is a more formal definition) and exponentially distributed with rate = λ, then the distribution of the sum is Gamma(n, λ)
        1. In Math notation: If X_i ~ Exponential(λ) and Z = $\sum_{i=1}^{n} X_i$, then Z ~ Gamma((n, λ)
        2. This makes Gammas a natural model for chemical reactions or other physical processes with multiple rate limiting steps; Always keep in mind that every distribution has multiple personalities. Just because the Gamma distribution has an interesting connection to Exponential distributions doesn't mean that Gamma distributions should only be thought of in terms of this relationship.

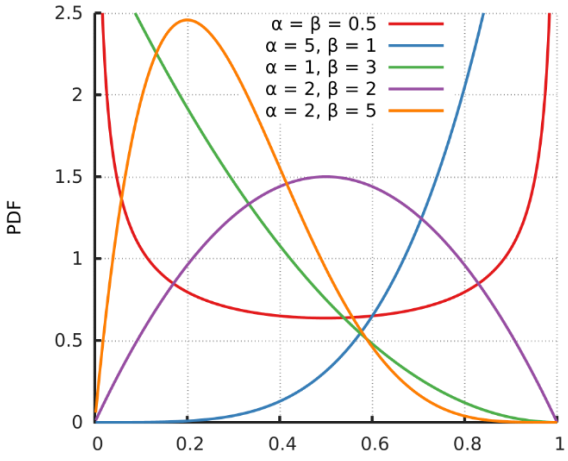
4. The Uniform Distribution (Math notation: X ~ Uniform(a, b) means X is distributed according to a Uniform with lower bound a and upper bound b)



Uniform(a, b) density function

a. Use in simulations
    i. One place Uniform distributions come up is in simulating chemical reactions, like in the Gillespie algorithm simulations of promoter-proximal pausing I have written
        1. In this case, the Uniform distribution is an efficient way to choose between n events that have probabilities p1, p2, …, pn of occurring
        2. Just generate a Uniform(0,1) random variable, call it U. If U < p1, then event 1 occurred; if it is between p1 and p1 + p2, then event 2 occurred, etc.
    ii. Another simpler case of the Uniform in lab simulations is if you just want to randomly generate parameters on some bounded interval. For example, in some of the TimeLapse simulations, we might want to randomly generate L2FC(kdegs) between 1 and 4 (for example), so I use a Uniform(1, 4) to assign a random L2FC(kdeg) to each transcript
b. R function = **runif**
c. Lower bound = a (one of the parameters); Upper bound = b (the other parameter)
d. Parameters
    i. a = lower bound
    ii. b = upper bound
    iii. a and b can be any real number, with a < b
e. Moments of Interest
    i. Mean = (a + b)/2
    ii. Variance = $(b - a)^2/12$
f. Important Properties
    i. Uniform distributions are pretty straightforward, but they do have some badass and interesting properties, though these may not be super important in your simulations
    ii. Assume that a chemical reaction occurs with some rate λ. This means that the time (T) until a single reaction can be well modeled as T ~ Exponential(λ). Now assume that you know a reaction occurred between times t1 and t2. The distribution that describes the probability of the exact time of occurrence of that reaction is Uniform(t1, t2)
    iii. Any random variable can be generated by performing mathematical operations on a Uniform(0,1) random variable. This is a fact that every single random number generating function in R relies on.
        1. The reason for this is that if you define U = F(X), where F(X) is the CDF of a random variable X, then U ~ Uniform(0,1). So by applying the inverse CDF of a random variable to a Uniform(0,1) random variable, you can sample from the distribution that describes X

5. The Beta Distribution (Math notation: X ~ Beta(α, β) means X is distributed according to a Beta distribution with shape factor α and second shape factor β)



Beta(α, β) density function

a. Use in simulations

i. While they are not currently employed in existing simulations, it isn't hard to imagine how a beta distribution could be used. For example, if you want to let the 4sU TimeLapse conversion rate vary from transcript to transcript or experiment to experiment, you could use a beta distribution to randomly generate a 4sU conversion rate for each unique case. Michelle also makes the point that you could use a beta distribution to vary oxidation rates across different s4Us.

b. R function = **rbeta**

c. Lower bound = 0; Upper bound = 1
   i. While this is true, if you add a constant c to a beta distribution, it is still kind of a beta distribution, just with a lower bound of c and an upper bound of 1 + c. Also, if you multiply a beta distribution by a constant d, it is still kind of a beta distribution, just with a lower bound of 0 and an upper bound of d. You can even do both at the same time (multiply by d and add c)! This is called a scaled and translated beta distribution, and can be an interesting way to simulate bounded parameters without using a boring uniform distribution.
      1. Scaling and translating is a neat way to take a distribution and give it a little different flavor by changing some of its properties, so this is a good tool to keep in mind for any distribution.

d. Parameters:
   i. $\alpha > 0$
   ii. $\beta > 0$
   iii. Both parameters are called shape factors, and in R they are referred to as shape1 and shape2

e. Moments of Interest
   i. Mean = $\alpha/(\alpha + \beta)$
   ii. Variance = $\frac{\alpha * \beta}{(\alpha + \beta)^2(\alpha + \beta + 1)}$ (oof)

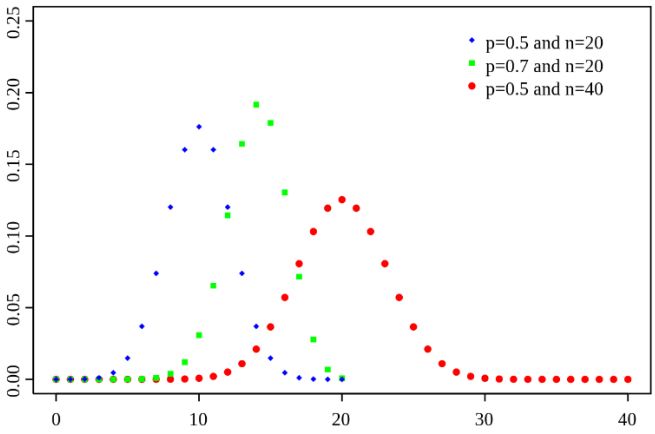
f. Neat Properties
   i. Beta distributions should be thought of as flexible probability distributions to describe probabilities. For example, if you let p ~ Beta(α, β), then p could be used as the probability of success parameter for a Bernoulli, Geometric, Binomial, or other discrete random variable (all described below). This can lead to interesting "mixture distributions" like the beta-binomial distribution

<u>Important Discrete Distributions to Know</u>

1. The Bernoulli Distribution (no picture because that would be a pretty bland picture; we say X ~ Bernoulli(p) to mean X is a Bernoulli trial with probability of success p)
   a. Use in simulations
      i. Since other distributions (e.g., the binomial) are generalizations of multiple Bernoulli trials, Bernoulli random variables are rarely used in simulations currently. That being said, they might see use in the near future
      ii. One possible use of a Bernoulli random variable in our simulations is if you want to determine which particular Us in a read are mutated in a TimeLapse experiment; a vector of Bernoulli random variables could be generated and might look something like: [0 0 0 1 1 0 1], where a 0 at a given position means that U was not mutated, and a 1 means it was mutated
   b. R function = **rbernoulli**
   c. Bounds: Either 0 or 1
   d. Parameters
      i. p = probability that a given sample from the distribution is a 1 rather than a 0
   e. Moments of Interest
      i. Mean = p
      ii. Variance = p(1 – p)

f. Neat Properties
    i. Bernoulli random variables are often called “Bernoulli trials”. These Bernoulli trials form the basis of understanding several of the discrete distributions described below. If a Bernoulli random variable takes the value 1, that is often arbitrarily defined as a success, which is language commonly employed when discussing these discrete distributions

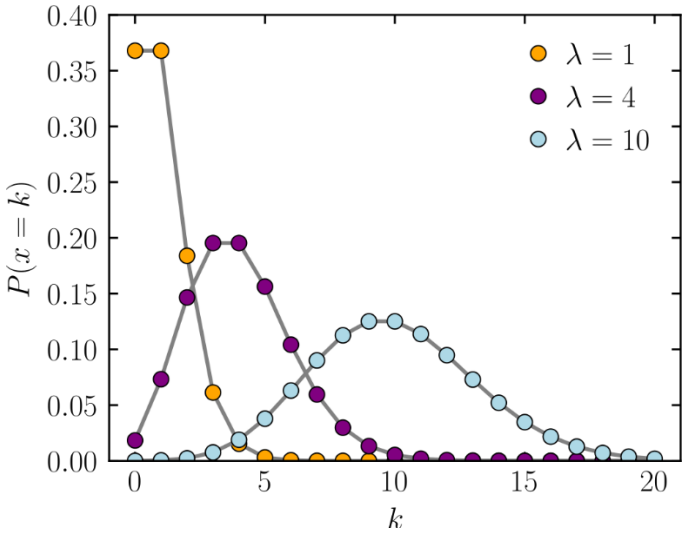
2. The Binomial Distribution (Math notation: X ~ Binomial(n, p) means X is distributed according to a Binomial distribution with n trials and probability of success p on each trial)



Binomial(n, p) mass function

a. Use in simulations
    i. Binomial distributions are our go-to method by which to simulate T to C mutations in RNA-seq reads due to either TimeLapse conversion of 4sU or random sequencing errors.
b. R function = **rbinom**
c. Lower bound = 0; Upper bound = n, where n is often called the “number of trials”
d. Parameters
    i. p = probability of success on each trial
        1. $0 < p < 1$
    ii. n = number of trials
        1. $n > 0$
e. Moments of Interest
    i. Mean = n*p
    ii. Variance = n*p*(1-p)
f. Generative Model (for discrete random variables, often there is a cute picture that can be used to explain where its affiliated probability distribution comes from; this cute picture is what I mean by Generative Model)
    i. Consider n “Bernoulli trials” (random events that result in one of two possible outcomes, denoted 0 and 1). If the probability of success in each individual trial is p, then the number of successes in n trials follows a Binomial distribution
g. Neat Properties
    i. All probability distributions have an associated “normalization factor”. I have avoided writing down the mathematical form of the functions that describe the probability distributions (called probability density/mass functions) I have discussed, but these normalization factors ensure that they are true probability distributions.
        1. I bring this up because the binomial density function normalization factor is famous, and is called the binomial coefficient. It describes the number of ways that you can choose k things from a selection of n, with the order in which you choose the things not mattering:

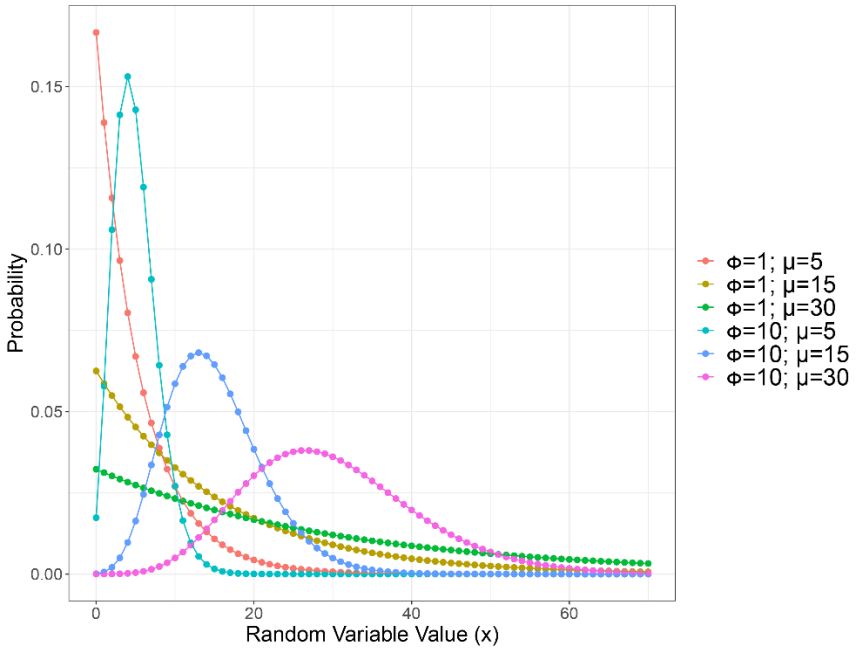
3. The Poisson Distribution (Math notation: X ~ Poisson(λ) means X is distributed according to a Poisson with rate λ)



Poisson(λ) mass function

a. Use in simulations
    i. Interestingly, while Poisson random variables do not show up often in simulations in the lab, they do show up in the statistical models we often analyze simulated data with. While the simulations use a binomial distribution to describe T to C mutations in a TimeLapse experiment, our models approximate this Binomial distribution with a Poisson, since the mutation probability of a given U is pretty low.
b. R function = **rpois**
c. Lower bound = 0; Upper bound = infinity
d. Parameters
    i. λ = rate parameter
        1. λ > 0
e. Moments of Interest
    i. Mean = λ
    ii. Variance = λ (neat!)
f. Neat Properties
    i. Let X ~ Binomial(n, p). Define λ = n*p. If λ << n, then X is approximately a Poisson(λ) random variable.
        1. This means we can understand the Poisson distribution as an approximation of a binomial random variable where the number of successes never gets close to the theoretical maximum n, the number of trials
    ii. It's kind of ridiculous how well Poisson distributions model positive counts
        1. Everything from the number of earthquakes in the U.S., to the number of home runs an MLB player will hit in their career, to the number of 19th century Prussian cavalry killed by their horses kicking them have been successfully modeled with Poisson distributions.
    iii. Poisson distributions show up when modeling chemical reactions using Exponential distributions. If the time until a reaction occurs (such as degradation of an RNA) follows an Exponential distribution with rate = λ, then the number RNAs degraded in a span of t minutes is distributed according to a Poisson(λ*t) distribution

4. The Negative Binomial Distribution (Math notation: X ~ Negative Binomial(μ, ϕ) (sometimes abbreviated NB(μ, ϕ)) means X is distributed according to a Negative Binomial distribution with mean μ and dispersion ϕ)

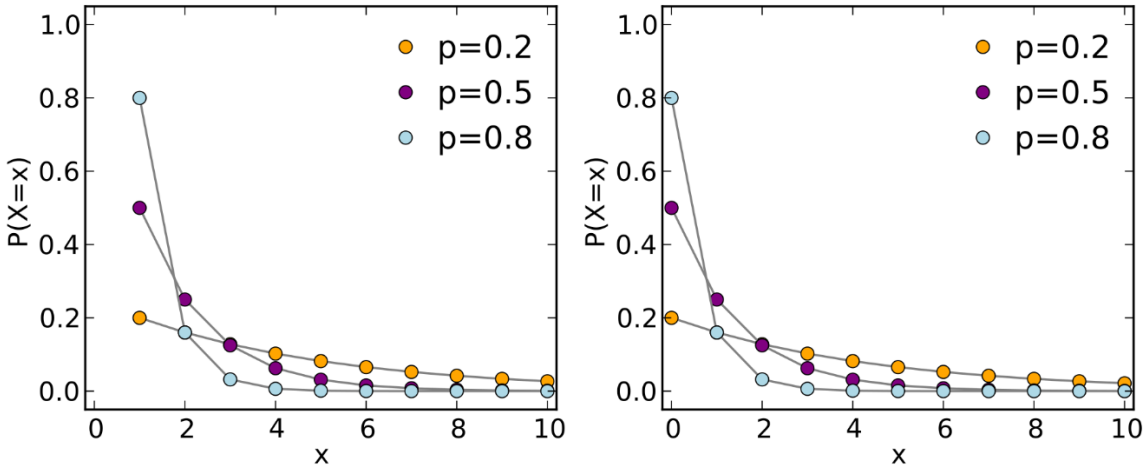


NB(μ, ϕ) mass function

a. Use in simulations
    i. Negative binomial distributions are a beautifully simple model for the number of RNA-seq reads mapping to a transcript present at a particular concentration. It captures the phenomenon of over-dispersion that has been observed in RNA-seq, which is when the variance (i.e., standard deviation squared) is greater than the mean.
b. R function = **rnbinom**
    i. The dispersion parameter is called "size" in the R function
c. Lower bound = 0; Upper bound = infinity
d. Parameters (also called the gamma-poisson distribution parameters)
    i. μ = mean of distribution
        1. $\mu > 0$
    ii. ϕ = dispersion parameter
        1. $\phi > 0$
    iii. This is the parameterization that simulations in the lab almost exclusively rely on; there are other parameterizations and applications of the negative binomial that are actually more common in statistics, but you probably won't have to worry about those
e. Moments of interest
    i. Mean = μ
    ii. Variance = $\mu + (\mu^2/\phi)$
        1. Note, same variance as Poisson if ϕ is really really large
f. Neat properties
    i. The reason the parameterization described above is called a gamma-poisson is that it can be thought of as a mixture of a Gamma and Poisson distribution:
        1. Let L ~ Gamma(α, β) and let X ~ Poisson(L)
            a. So in R, you could imagine simulating L, and using that L as the rate parameter for a simulation from a Poisson distribution
        2. Then X ~ Negative-Binomial($\mu = \alpha/\beta$, $\phi = 1/\beta + 1$)

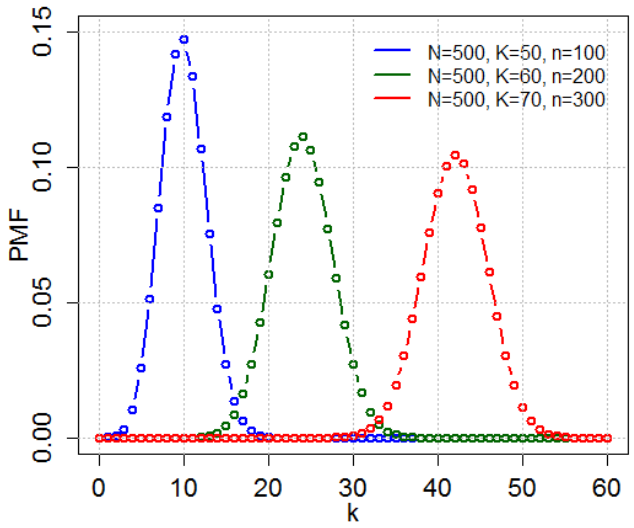
5. The Geometric Distribution (Math notation: X ~ Geometric(p) means X is distributed according to a Geometric distribution with probability of success p)

Geometric(p) mass functions. Left is when defining the random variable as the number of trials until the first success. Right is when defining it as the number of failures until the first success



a. Use in simulations
    i. The Geometric distribution has so far gone unused in simulations in the lab, but it is a very common distribution and could very well show up in a Simon lab simulation one day.
    ii. You might imagine modeling the tethering process in Michelle and Leah's experiments as following a geometric distribution. In this case the "trials" are Us in a particular RNA fragment and a success is MTS-biotin tethering to the fragment. If each U has the same probability of being tethered, then a geometric distribution describes the number of Us until the first successful tether. You could thus generate a random geometric variable and if it is greater than the number of Us in the fragment, tethering does not occur for this fragment
        1. Note, I am not endorsing this use in tethering simulations, lol, it's just a case study. Why might a geometric random variable be a bad model in this case?
b. R function = **rgeom**
c. Lower bound = 0; Upper bound = infinity
d. Parameters
    i. p = probability of success
e. Moments of Interest
    i. Mean = 1/p
    ii. Variance = $(1-p)/p^2$
f. Generative model:
    i. Consider "Bernoulli trials" (random events that result in one of two possible outcomes, denoted 0 and 1). If the probability of success for an individual Bernoulli trial is p, then the number of trials until the first success follows a Geometric distribution
        1. Note, there are two pictures above. That is because some people define a Geometric random variable as the number of **failures** until the first success, in which case 0 is a possibility (if the first trial is a success)
g. Neat properties
    i. The Geometric distribution is memoryless just like the Exponential distribution. In fact, the Geometric distribution is the only memoryless discrete distribution, and the exponential distribution is the only memoryless continuous distribution, so the memoryless property in effect defines these distributions

6. The Hypergeometric Distribution (Math notation: X ~ Hypergeometric(N, K, n) means X is distributed according to a Hypergeometric Distribution with N total objects, K having the feature of interest, and n draws from the objects without replacement)



Hypergeometric(N, K, n) mass function

a. Use in simulations
    i. I have actually used a multi-variate extension of the hypergeometric distribution in a simulation. I used it to model PCR amplification. Each round of PCR, a certain selection of fragments gets amplified, and we can assume for simplicity that no fragment is amplified twice, so that we are essentially randomly selecting RNA fragments WITHOUT REPLACEMENT. Thus, a hypergeometric distribution is a good model! Why multi-variate though? Because there aren't just two classes (e.g., red and blue marbles) but thousands of classes (transcripts from gene 1, transcripts from gene 2, transcripts from gene 3, …)
b. R function = **rhyper**
c. Lower bound = 0; Upper bound = n (the number of samples)
d. Generative model (here it is best to start with a generative model to better understand what all the parameters mean)
    i. Assume you have a jar filled with N marbles. Also, assume you pull out marbles one at a time and observe their color, and importantly, you DON'T replace the marble after pulling it out
    ii. Furthermore, assume that there are two types of marbles in the jar, say red and blue marbles, and that there are K red marbles and N-K blue marbles.
    iii. FURTHERMORE, assume that you pull n marbles out from this jar, and you call a particular pull a success if you pull out a red marble
    iv. Then, the number of "successes" (i.e., the number of red marbles drawn from the jar) follows a hypergeometric distribution
e. Parameters
    i. N = the total number of objects
    ii. K = total number of objects with feature of interest
    iii. n = number of samples taken from the N objects
    iv. All parameters > 0; N ≥ K and N ≥ n
f. Moments of Interest
    i. Mean = n*(K/N)
    ii. Variance = $n * \frac{K}{N} * \frac{N-K}{N} * \frac{N-n}{N-1}$ (wow)
g. Neat properties
    i. There is some neat combinatorics that goes into understanding the hypergeometric probability mass function (the discrete version of a probability density function), but it isn't really worth describing here.
    ii. My favorite thing about the hypergeometric distribution is that there is a built-in correlation structure between draws. When modeling processes with probability distributions, we typically assume that the resulting random variables are independent and identically

distributed (iid), meaning that each draw from the distribution does not depend on the last and that each draw has the same probability density/mass function. Hypergeometric distributions on the other hand describe draws that are NOT independent and NOT identically distributed, because each draw affects the probability you get a success on the next draw, since you aren't replacing the objects you are sampling.

Case Study: Replicate Noise TimeLapse Simulation

I want to walk through the distribution and parameter choices I made in my most recent TimeLapse simulation that models TimeLapse data while taking into account replicate variability in the relevant kinetic parameters. Be warned, this is a fairly extensive example, but I wanted it to be that way so that I could illustrate the practical use of many of the distributions described above.

First, it is important to state the goals of the simulation:

1. Simulate TimeLapse data, which includes simulating:
    a. Read counts
    b. Fraction news
    c. T to C mutations in each read
2. Simulate replicate variability in kinetic parameters:
    a. kdeg
    b. ksyn
    c. fraction new (fn)

Second, we can start to think about the structure of the simulation:

User Input → Simulation of kinetic parameters (and thus fraction news and RNA concentrations) →

Simulation of replicate variability in the average kinetic parameter values →

Simulation of read counts using RNA concentration from last step →

Simulation of T to C mutations in each read using fraction news and read counts simulated in last two steps

Even though User Input is the first part of the simulation, I find myself developing this part as I go. Usually, I don't realize all of the important parameter values I will have to decide on until I start developing the simulation.

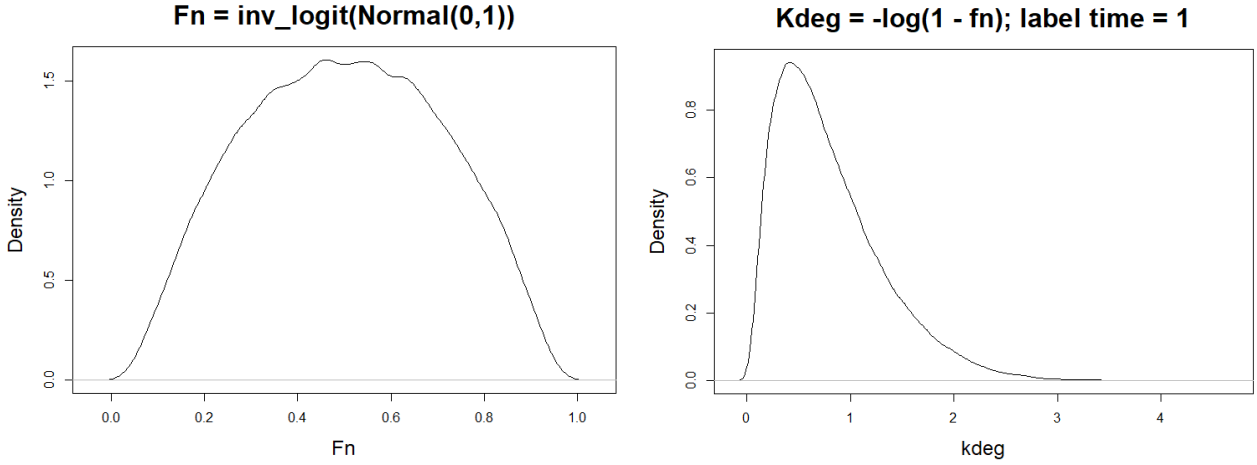
I am kind of glossing over it here, but determining the structure of the simulation is an incredibly important step that can require a lot of thought. Fortunately, a lot of the thought relies on your expertise and familiarity with whatever you are simulating, and not so much with the probability distributions discussed above. That being said, there are several things you have to consider that can be influenced by what is easy and difficult to model with the distributions you have at your disposal:

- What sort of complexity do you want to include? What sort of complexity do you want to ignore?
- What do you want the output of the simulation to be?
- When considering different potential structures, is one going to be easier to simulate than another?

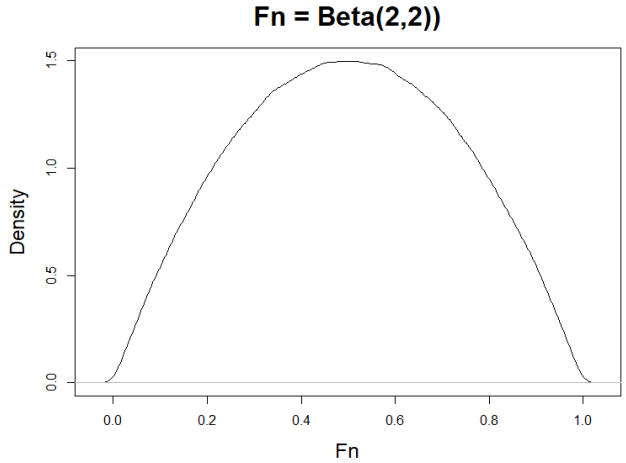
Thus, the third thing we'll do is think about the simulation:

1. Simulating mean kinetic parameters
    a. For every transcript to be simulated, we want to assign a fraction new (fn), synthesis rate (ksyn), and a degradation rate constant (kdeg) to that transcript, so that we can simulate its read counts (which depends on ksyn and kdeg) and the T to C mutations in its sequencing reads (which depends on the fn)
    b. There are a couple ways we could go about doing this, and there are always multiple valid solutions to these simulations

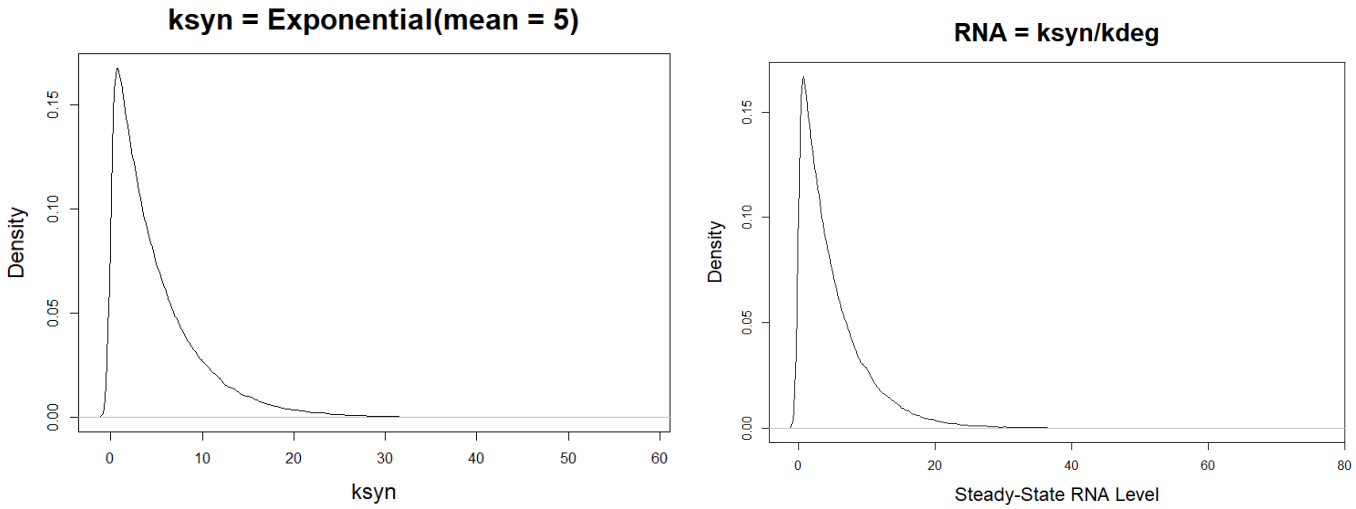
c. One solution (which is what I do), is to randomly draw fns from a transformed normal distribution:
    i. fn = inv_logit(**rnorm**(n = ngene, mean = 0, sd = 1))
        1. ngene is the number of genes to be simulated and inv_logit is the inverse logit function, which takes inputs from -infinity to infinity and spits out a number between 0 and 1
        2. The mean and sd I use were optimized by looking at histograms or density plots of simulations, like that depicted below. I wanted something that would cover a large range of fns, but I also wanted most of the fraction news to be possible to estimate. Really extreme values are hard to estimate because they lead to sequencing reads that are either completely new or completely old, so I chose a distribution symmetric about ½ with a lot of mass around ½
    ii. Then, I can get kdeg = $-\log(1 - \mathrm{fn})/t_{\mathrm{label}}$
        1. Where t_{label} is the label time, another user input to set



d. Similarly, you could use a beta distribution to model the raw, untransformed fraction news.
    i. To get a similar distribution as above, you could play around with the **rbeta** function in R (which simulates random numbers from a beta distribution) and plot densities/histograms to find good choices for the α and β parameters of the Beta distribution
    ii. When going through the distributions above, I mentioned the mean and variance of each distribution, because this can guide your choice
        1. The mean of a Beta is α/(α + β), so if you want the average fn to be ½, you should set α = β
        2. If α > 1 (with α = β), it turns out you will get something that looks like the above distribution, with less variability the larger you make α; one example is below:



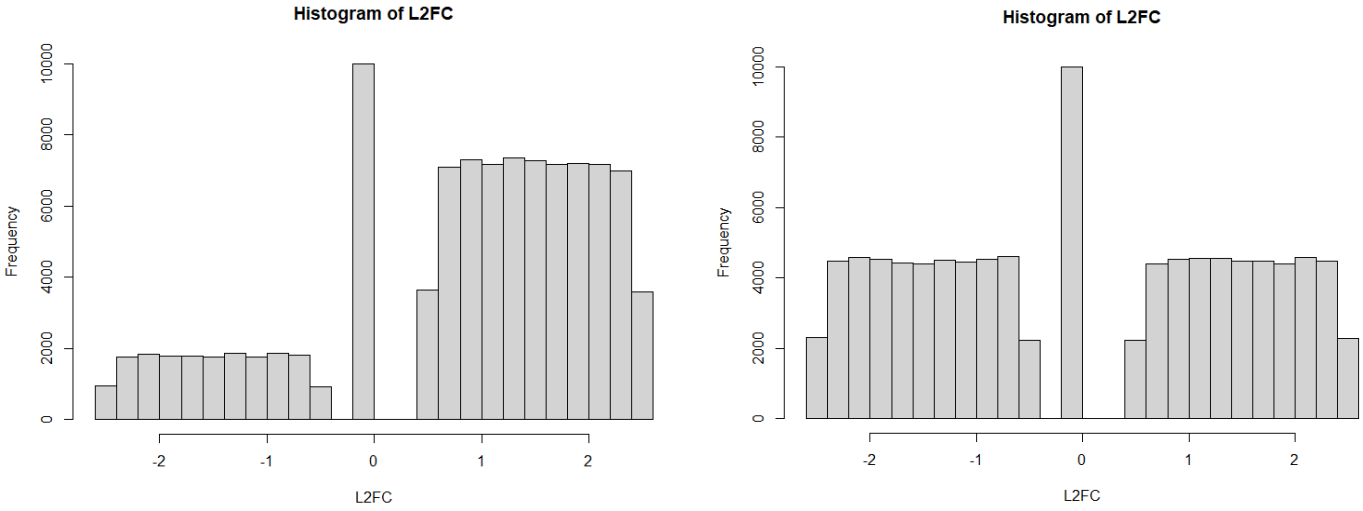
e. You could also simulate kdeg instead of fn, using something like an exponentiated Normal distribution [i.e., kdeg = exp(**rnorm**(…))], or a Gamma distribution. Because we will be using the fn to simulate T to C mutations though, I find it far more natural to simulate the fns directly
f. Finally, you also have to simulate the ksyn's
    i. This is another case where there are MULTIPLE possibilities. Keep in mind though, that the RNA concentration = ksyn/kdeg
        1. Therefore, the relative size of ksyn vs. kdeg will determine how many sequencing reads you expect to get for a particular transcript
    ii. The way I simulate ksyn is as such:
        1. ksyn ~ Exponential(mean = 5*kdeg)
            a. Or in R code: ksyn = **rexp**(rate = 1/(5*kdeg))
        2. The way to understand what this says is that for each kdeg, I call the R function **rexp**, and set its mean (or rate = 1/mean) to 5*kdeg. For example, if kdeg = 1, then the distribution of possible ksyns I might get looks like:



    iii. I went with this particular choice because it means that most of the transcripts I simulate will have non-negligible concentrations, and thus will have at least a couple sequencing reads in each replicate. The exact shape of this distribution is nice too because it disfavors really extreme values of ksyn and thus disfavors extremely abundant RNAs, but it doesn't prevent them. There is no hard cutoff, which I find more natural than say simulating ksyns according to a uniform distribution with some hard lower bound and upper bound.
g. A final important point is one of interpretation. What I am simulating in these examples should be thought of as the average kinetic parameters observed in an experiment. These are the kinetic parameters you would estimate if you conducted 100000000 TimeLapse experiment replicates
    i. The replicate-to-replicate variability in these parameters will be simulated later

2. Simulating the average L2FC(kdeg) and L2FC(ksyn) across replicates
    a. I chose to simulate the average L2FC(kdeg) and L2FC(ksyn) as being either 0 (for non-significant genes) or a non-zero value ranging in magnitude from a hard-set lower bound to a hard-set upper bound
    b. Note, significance and non-significance is defined for both L2FC(kdeg) and L2FC(ksyn) independently. So each transcript can either be significant in both, significant in 1 but not the other, or non-significant in both
    c. Lower and upper bounds will be included in user input
    d. The sign of the change (negative or positive) is randomly selected with some probability
        i. The probability is included in the User input and the choice can be made with a call to **rbernoulli**(p), where **rbernoulli** generates a random 0 or 1 with the probability of 1 being p; If a 1 is generated, the L2FC (ksyn or kdeg) is positive, else it is negative

e. The magnitude of the change is simulated according to a Uniform distribution with the user defined lower and upper bounds
f. The end result looks like this:
    i. Min = 0.5
    ii. Max = 2.5
    iii. Left p = 0.8; Right p = 0.5



g. The parameter selection here is "easy" in that the results are obvious due to the uniform distribution being used. You just have to choose upper and lower bounds that make sense
h. Also, note that I didn't talk about L2FC(RNA) yet. That is because L2FC(RNA) = L2FC(kdeg) + L2FC(ksyn), so we can just simulate L2FC(ksyn) and L2FC(kdeg) and get L2FC(RNA) for free
i. As is always the case, there are many many MANY possible ways to go about simulating L2FC's, and there are plenty of reasons to critique any one choice and to consider different options

3. Simulating technical variability
    a. The L2FC(…)'s allow us to simulate biological variability, i.e., significant differences in a transcripts metabolic kinetics between experimental conditions. Now we want to capture what you might call "technical variability" (though one of its sources is the messiness/stochasticity of biological systems). This variability will be simulated as differences in the kinetic parameters from replicate to replicate, but should be interpreted as capturing variability in the data that gives rise to apparent variability in the kinetic parameters.
    b. To do this, I take the degradation rate constant (kdeg) values simulated in step 1, and add a bit of random noise to them:
    c. kdeg[gene **a,** condition **b,** replicate **c**] =

$$\exp(\ \textbf{rnorm}\ (\ \text{mean} = \log(\ \text{kdeg\_mean}[\textbf{a}]*(2^{\text{L2FC\_kdeg}[\text{gene } \textbf{a}, \text{ condition } \textbf{b}]}),\ \text{sd} = \text{kdeg\_noise}\ )\ )$$

        i. **a, b,** and **c** are indexes for the gene, experimental condition, and replicate IDs
        ii. kdeg_mean is what is simulated in part 1
        iii. L2FC_kdeg is what is simulated in part 2
        iv. So this means that the actual kdeg in a given replicate is pulled from a distribution centered on the mean simulated for that transcript in step 1, but with a little bit of variability captured by the standard deviation term
        v. $2^{\text{L2FC(kdeg)}}$ is the multiplicative term necessary to account for the L2FC(kdeg) simulated in the previous step
    d. kdeg_noise is a parameter to be set at the top of the simulation by the user. While you could again resort to simulations to see what the effect of changing the fn_noise, normal distributions have a simple rule that makes this unnecessary

i. The probability that **rnorm**(mean = μ, sd = σ) gives you a value more than 1*σ away from the mean is 32% The probability that it gives you a value more than 2*σ away from the mean is about 5%. The probability that it gives you a value more than 3*σ away from the mean is about 0.3%.
ii. These general rules can help you determine what standard deviation to set; it is very unlikely that you will get something from **rnorm** more than 3 or 4 standard deviations from the mean, so you can treat this as a practical lower and upper bound.

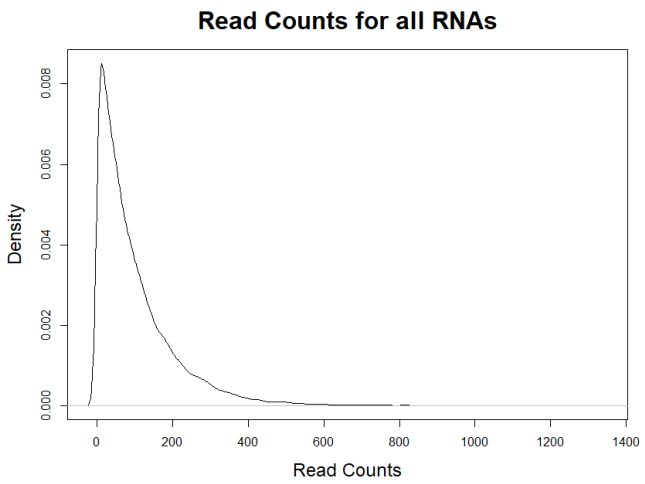
e. Note, I have taken the approach of simulating a L2FC(kdeg) and then a kdeg for each replicate and experimental condition. Another strategy would be to simulate a "fraction new effect size" (eff), which would be between negative infinity and infinity (just like L2FC(kdeg)) that would get added to fn_mean on the inv_logit:

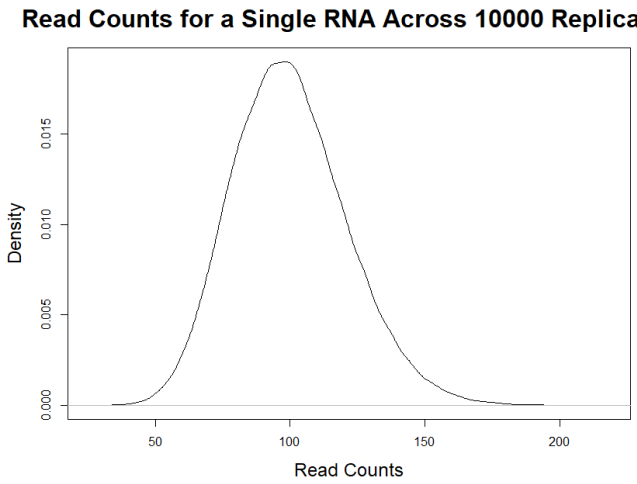
inv_logit(**rnorm** (mean = logit(fn_mean[**a**] + fn_delta[**a, b**] , sd = delta_noise))

f. Fun fact, this is what I actually do now, because the assumption that each transcript has a noise term independent of its individual kinetic properties seems most true for variability in logit-fns rather than in L2FC's. Of course, you could also make the sd term in the replicate variability **rnorms** a function of some other property of the transcript (like its kdeg and/or ksyn)

a. We have explicitly discussed simulating replicate variability in the kdeg and fn for each replicate and each experimental condition; the same exact procedure can be used to simulate replicate variability in ksyn for each replicate and each experimental condition

4. Simulating read counts
   a. At this point, we have simulated values for ksyn, kdeg, and the fn
   b. The steady-state level of RNA is ksyn/kdeg, so we have thus indirectly simulated the RNA concentration in each sample. We can use this fact to simulate RNA-seq read counts
   c. Much literature has discussed the appropriateness of various models for RNA-seq read counts; I prefer to use a negative binomial distribution, one of the most common modeling choices
   d. Two parameters must be specified for a negative binomial distribution: it's mean (μ, which should be proportional to the amount of RNA) and it's dispersion parameter (ϕ).
      i. μ = (steady-state RNA level) * (scale factor)
         1. The scale factor can be interpreted as the average amount of reads that a steady-state RNA level of 1 (units depend on units of degradation rate constant) will yield. A slightly different scale factor can be used for each sample to simulate differences in sequencing depth or RNA isolation
      ii. ϕ can be set equal to a constant that can be informed by simulation; I take a different strategy, following the model of DESeq2. The developers of DESeq2 noted that ϕ tends to depend on the concentration of RNA, and used the following model for ϕ:
         1. $\frac{1}{\phi} = \frac{a1}{\mu} + a2$
         2. a1 and a2 can again be set by testing different values with simulation, or you can use values that the DESeq2 paper found to be appropriate (a1 = 0.5; a2 = 0.03)
   e. When determining what the scale factor (or even the dispersion model, if playing with that) should be, it makes sense to look at two cases:
      i. You can simulate reads for thousands of different RNAs, to get a sense of the distribution of read counts in a single sample
      ii. You can also simulate reads for a single RNA but for thousands of theoretical replicates, to get a sense of the replicate variability in read counts for a single RNA:
         1. Left panel:
            a. kdeg, ksyn, and RNA concentration determined as described above
            b. Scale factor = 20
            c. 100000 RNAs simulated
         2. Right panel:
            a. 100000 replicates for a single RNA with the RNA concentration (ksyn/kdeg) set equal to 5, and a scale factor of 20 used





5. Simulating T to C mutations
    a. For each read, I randomly simulate the number of Us in that read using the read length and a 25% probability of each nucleotide being a U. A binomial distribution is thus appropriate here
        i. Number of Us = rbinom(read_length, p = 0.25)
    b. Each read is designated as new or old using an Bernoulli distribution with probability of new being the fraction new in the sample from which the read comes
        i. Is the read new? = rbernoulli(p = fn[**a, b, c**]
            1. 0 = old, 1 = new
    c. Then, T to C mutations are modeled using a binomial distribution with rate of “success” (mutation) depending on whether the read is old or new
        i. Number of mutations = rbinom(n= Number of Us, p = relevant mutation rate)
    d. Thus, the parameters that the user must specify are:
        i. Read length
        ii. Old read mutation rate
        iii. New read mutation rate
    e. The mutation rates are often informed by previous results that have pinned the mutation rates to be around 0.001 for old reads and between 0.05 and 0.1 for new reads. If investigating the impact of mutation rate on your analysis, these can of course be played with

And with that, we have developed a relatively complex TimeLapse simulation!

Example code for each section of the case study (Full code available on lab Github):

0. User input:

```
#######BEGIN USER INPUT#######
#Label time (in minutes)
tl <- 60

#TL parameters
p_new <- 0.05        #New mutation rate
p_old <- 0.001       #Old mutation rate
read_lengths <- 200  #Read length
p_do <- 0            #Dropout probability

#Number of genes to simulate
ngene <- 500

#Number of experimental Conditions to simulate
num_conds <- 3

#Number of replicates of each experimental condition to simulate
nreps <- 2

#Standard deviation for L2FC of kdeg (L2FC_kd)
noise_deg <- 0.1
#Standard deviation for L2FC of ksyn (L2FC_ks)
noise_synth <- 0.1

#Minimum L2FC_kd for significantly changing genes
low_eff <- 0.3
#Max L2FC_kd for significantly changing genes
high_eff<- 4.0

#Minimum L2FC_ks for significantly changing genes
low_L2FC_ks <- 0.3
#Max L2FC_ks for significantly changing genes
high_L2FC_ks <- 4.0

#Number of transcripts that have a significant L2FC_kd in each experimental condition
#L2FC is always relative to the first condition (the WT condition)
num_kd_DE <- c(0, 450, 0)

#Number of transcripts that have a significant L2FC_ks in each experimental condition
num_ks_DE <- c(0, 0, 450)


#If TRUE, simulates read counts from Negative binomial using ks/kd and scale_factor
sim_read_counts <- TRUE
#Scale factor for Read Count simulation; Only relevant if sim_read_counts is TRUE
scale_factor <- 20 #(one RNA molecule per cell corresponds to <scale_factor> sequencing read per replicate, on

#Set number of sequencing reads to be simulated; Only relevant if sim_read_counts is FALSE
nreads <- 500
```

1. Average kinetic parameters:

```
# Define helper functions:
logit <- function(x) log(x/(1-x))
inv_logit <- function(x) exp(x)/(1+exp(x))

#Initialize matrices
fn <- rep(0, times=ngene*nreps*num_conds)
dim(fn) <- c(ngene, num_conds, nreps)
Counts <- fn
kd <- fn
ks <- fn


#Initialize vectors of mean values for each gene and condition
fn_mean_WT <- rep(0, times=ngene)
fn_mean <- inv_logit(rnorm(n=ngene, mean=0, sd=1.5))
kd_mean <- -log(1-fn_mean)/tl
ks_mean <- rexp(n=ngene, rate=1/(kd_mean*5))

effect_mean <- rep(0, times = ngene*num_conds)
dim(effect_mean) <- c(ngene, num_conds)
L2FC_ks_mean <- effect_mean
L2FC_kd_mean <- effect_mean
```

2. L2FCs:

```
for(i in 1:ngene){

  #Make sure the user didn't input the wrong
  #number of significant genes
  if (i == 1 ){
    if (length(num_kd_DE) > num_conds){
      print("num_kd_DE has too many elements")
      break
    } else if (length(num_kd_DE) < num_conds){
      print("num_kd_DE has too few elements")
      break
    }

    if (length(num_ks_DE) > num_conds){
      print("num_ks_DE has too many elements")
      break
    } else if (length(num_ks_DE) < num_conds){
      print("num_ks_DE has too few elements")
      break
    }
  }

  for(j in 1:num_conds){
    if(j == 1){
      effect_mean[i,1] <- 0
      L2FC_ks_mean[i,1] <- 0
    }else{
      if(i < (ngene-num_kd_DE[j] + 1)){
        effect_mean[i,j] <- 0
      }else{
        if (runif(1) < 0.5){
          effect_mean[i,j] <- runif(n=1, min=low_eff, max=high_eff)
        }else{
          effect_mean[i,j] <- runif(n=1, min=-high_eff, max=-low_eff)
        }

      }
      if(i < (ngene-num_ks_DE[j] + 1)){
        L2FC_ks_mean[i,j] <- 0
      }else{
        if (runif(1) < 0.5){
          L2FC_ks_mean[i,j] <- runif(n=1, min=low_L2FC_ks, max=high_L2FC_ks)
        }else{
          L2FC_ks_mean[i,j] <- runif(n=1, min=-high_L2FC_ks, max=-low_L2FC_ks)
        }

      }
    }
  }
}
```

3. Replicate Variability

```
#SIMULATE L2FC OF DEG AND SYNTH RATE CONSTANTS; REPLICATE VARIABILITY SIMULATED
for(i in 1:ngene){
  for(j in 1:num_conds){
    for(k in 1:nreps){
      fn[i, j, k] <- inv_logit(rnorm(1, mean=(fn_mean[i] + effect_mean[i,j]), sd=noise_deg))
      ks[i,j,k] <- exp(rnorm(1, mean=log((2^L2FC_ks_mean[i,j])*ks_mean[i]), sd=noise_synth))
    }
  }
}

kd <- -log(1 - fn)/tl
```

4. Read Counts

```
#Simulate read counts
if (sim_read_counts == TRUE){
  L2FC_tot_mean <- L2FC_ks_mean - L2FC_kd_mean
  RNA_conc <- ks/kd
  a1 <- 5
  a0 <- 0.03

  for(i in 1:ngene){
    for(j in 1:num_conds){
      for(k in 1:nreps){
        Counts[i, j, k] <- rnbinom(n=1, size=1/((a1/(scale_factor*RNA_conc[i,j,k])) + a0), mu = scale_factor*RNA_conc[i,j,k])
      }
    }
  }
} else{
  Counts <- rep(nreads, times= ngene*num_conds*nreps)
  dim(Counts) <- c(ngene, num_conds, nreps)
}

```

5. TimeLapse simulation

```
 for (s in 1:nsamp){
   mir_data <- vector('list', length = nmir)
   for (mir in 1:nmir){ #mir is feature number index, should change to gene or something

     r <- replicate[s] #Replicate number index
     MT <- mt[s]       #Experimental sample index
     readsize = read_length[mt[s]]

     mir_pold_tc <- p_old[mt[s]]
     mir_pnew_tc <- p_new[mt[s]]

     #Simulate which reads are labeled
     newreads_tc <- rbernoulli(nreads[mir, MT, r], p = fn_s4U[mir, MT, r])# vector of reads, T/F is s4U labeled

     #Simulate the nubmer of Us in each read
     nu <- rbinom(n = nreads[mir,MT,r], size = readsize, prob = 0.25)

     #Number of reads that are new
     newreads_tc <- sum(newreads_tc)

     #Generate number of mutations for new and old reads
     if (!ctl[s]){ #If no s4U added, only old
       nmut_tc <- rbinom(n = nreads[mir,MT,r], size=nu, prob = mir_pold_tc)
     }else {
       nmut_tc_new <- rbinom(n=newreads_tc, size=nu[1:newreads_tc], prob=mir_pnew_tc)
       nmut_tc_old <- rbinom(n=(nreads[mir, MT, r]-newreads_tc), size = nu[(newreads_tc+1):nreads[mir, MT, r]], prob=mir_pold_tc)
       nmut_tc <- c(nmut_tc_new, nmut_tc_old)
     }

```