Instructions for the Singh, et al assignment

Specific instructions

- Follow the instructions in the general implementation instructions.
- Reproduce figures 2, 3, 5, and 6.
- Submit your write up (as a PDF) and your code (as a single .m file) on Stellar. The write up should follow the general implementation instructions, and your code should be well commented and should run in one go.
- Implement the model with the changes noted in the following section.

Notes about the Model and changes:

All of the information about the model is in the supplemental information. However, we are not following the model exactly.

- In the paper, they use $\Delta t = 10^{-5}$s. We will instead be calculating the Δt as a random exponentially distributed number with rate equal to the sum of all the reaction rates (Gillespie Algorithm).
- In the paper they use P_{syn} as a probability, but it is not bounded by 1. I artificially capped P_{syn} at 1 using an if statement. The probability of walking (not resynthesizing ATP) would therefore be $1-P_{syn}$.
- For figure 3, they say they use 5000 simulations. We will only do 50. Do not do what they say in the paper; use 50. Only do 50 simulations for figure 3.

Because we are not following the model exactly, you may have some differences between the plots you generate and those in the paper. I have included plots that I got from my implementation of this model on the next page of this document.

Also keep in mind that this is a Monte Carlo model. Even if we implemented everything the same way, we won't get identical numbers. When deciding if your figure is correct, look at the magnitude of the data points and the general shape of any curves. You may also find it useful to run your code multiple times and see how variable it is to convince yourself it is correct and reasonable.

Figure specific notes:

Figure 2:

Feel free to plot movement using a line plot. You do not have to recreate the figure with the missing vertical lines. This plot will look the most different between your implementation and the paper. The most important aspects of this figure will be the relative shapes of the curves (which one stalls earlier, big steps or small steps). Don't get stuck making this figure perfect. The accuracy of the other figures is a much better indicator of whether or not your implementation is correct.

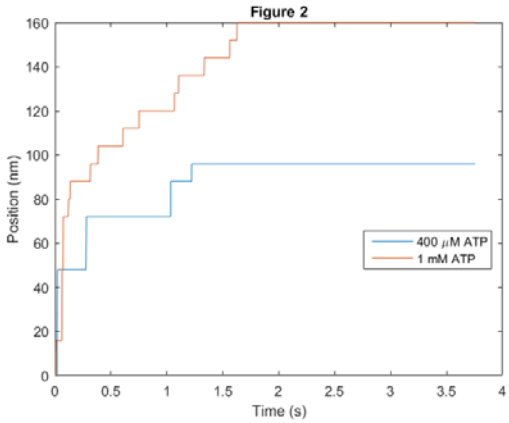


Figure 3:

The y-axis numbers are quite reproducible (the first point should be around 0.4 and it should plateau just under 1.1.). Only run 50 simulations for this figure. 50!

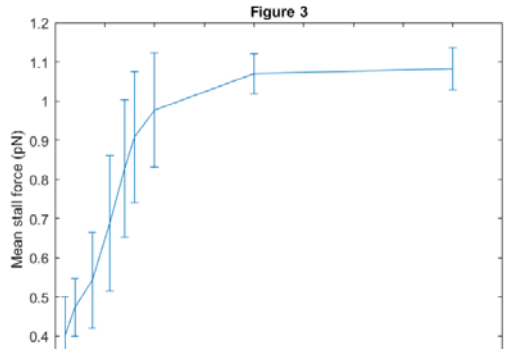


Figure 5:

For figures 5 and 6 the model is changed so that force does not depend on distance. In these simulations, the model is run at constant force and average speed is calculated over the 50 seconds. These numbers (including error bars) will also match well between your implementation and the paper.

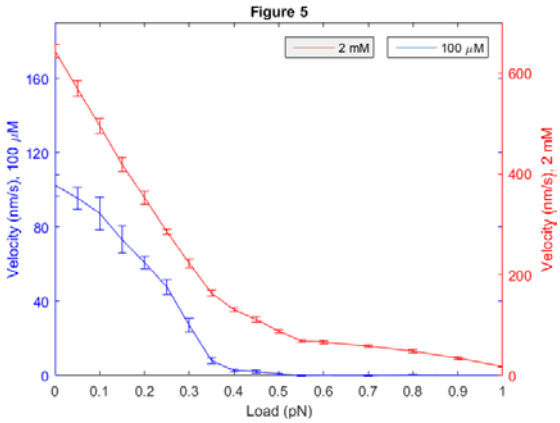


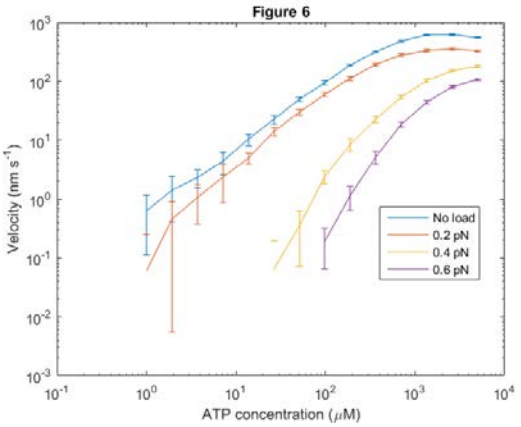
Figure 6:

See the note above for figure 5. Also, make sure to use evenly spaced points in log space, not in normal numbers (1, 10, 100 not 1, 50, 100). One easy way to do this in Matlab is:

X = 10.^linspace(0, 6, 14);

Note the dot operator!

The numbers and error bars may vary at low velocities, but the shape and relative spacing of the curves is consistent.



Tips for implementing the model:

How long should it take my code to run?

There are at least two ways to write your code: using lots of conditional statements or using matrix operations. The former is the brute force method. For most people it will be more straightforward. However, it might be slower. Matlab was built for matrix operations, and if you can think of a way to implement this using matrices, your code may run faster. In my experience, however, the biggest determinant of speed is pre-allocation of matrices.

I implemented this model last year using conditional statements and did not pre-allocate. It took about 7 minutes to run (without modification). This past week I changed my code to pre-allocate matrices and now it only takes 1 minute to run. Isaac implemented the model using matrix operations and pre-allocating matrices. His code took about 3 minutes to run (without modification). Many people last year had code that took "too long" or "Hours when I was doing it wrong. A few minutes when I was doing it right."

What does it mean to pre-allocate, and how does that work with the variables we have?

You have two main variables that you have to update and track with each loop iteration: the time and the position. To pre-allocate them means you initialize these variables as some Nx1 (or 1xN) vector where N does not equal 1. Before you start running the Monte Carlo algorithm, you only have one time value (0) and one position value (0), but you create space to store these values later on.

With these pre-allocated matrices, you have to make sure you put the new information in the correct space, the next available space. One simple way to do this is using a counting variable. Below I've named it count. I initialize count as 1 and use it to index the time and position vectors. This is the important part: every time I update time or position, I increase count by 1. This will allow you to keep track of where the next available space is.

You may end up pre-allocating these matrices much larger than necessary. Maybe you initialize them as 1000x1 vectors but only 500 Monte Carlo loops are necessary. You don't want to output a bunch of zeros, because that will affect how your figures look and how any calculations (velocity, stall force) work. See below for how I used count to only get the parts I wanted.

```
time = zeros(1000, 1);
position = zeros(1000, 1);
count = 1;

time(count+1, 1) = time(count, 1)+dT;
position(count+1, 1) = position(count, 1)+dX;
count = count+1;

time_out = time(1:count, 1);
position_out = position(1:count, 1);
```

How do you recommend writing and debugging?
Only work on one figure at a time. There's no need to run figure 2 every time you check to see if figure 3 is working. You can comment/uncomment to control what runs or use another method. Whatever you do, make sure what you turn in has all of your code in one file and will generate all of the requested figures if run (make sure everything is uncommented before turn-in).

How many random numbers should I generate?
You should generate two random numbers per standard iteration (think about how this changes following hydrolysis). One random number will be used to calculate the time step, and one random number will be used to pick which transition to execute given the probabilities of each occurring.

What should I be careful about?
Infinite loops! We'll be using random time steps, so we don't know how many iterations we need to reach a final time. To accomplish this in code, you'll use a while loop. Given a variable tracking the time (total_time) and a target time to run the simulation (target_time):

```
while total_time<target_time
    % Algorithm
end
```

While loops are really useful for processes like this, but they're also dangerous. If you don't update total_time appropriately, the loop may never end. Be extra careful when setting up these while loops for that reason. Note that the later figures also require while loops but don't track time.

How do you plot figure 5 with two y axes and errorbars?

```
figure
[AX] = plotyy(X1, Y1, X2, Y2);
hold(AX(1), 'on')
errorbar(AX(1), X1, Y1, Error1, 'b');
hold(AX(2), 'on')
errorbar(AX(2), X2, Y2, Error2, 'r');
xlabel('Load (pN)')
ylabel(AX(1), 'Velocity (nm/s), 100 \muM ATP')
ylabel(AX(2), 'Velocity (nm/s), 2 mM ATP')
legend(AX(1), '100 \muM')
legend(AX(2), '2 mM')
ylim(AX(1), [0, 190])
set(AX(1), 'YTIck', [0, 40, 80, 120, 160])
ylim(AX(2), [0, 700])
set(AX(2), 'YTick', [0, 200, 400, 600])
set(AX, {'ycolor'}, {'b'; 'r'})
title('Figure 5')
```

Where do I start?
Write out a Markov diagram for the process. Include the possible states and the mathematical descriptions for transition probabilities. This is a good starting place for a matrix-based approach to the implementation. Such a matrix will be NxN where N is the number of states. The *i, j* value (row *i* column *j*) describes the rate or probability (depending on how you write it) of moving from the *i* state to the *j* state. In this way, you can pull all possible transitions from the matrix if you know the state of your system (Possible_Rates = All_Rates(state, :); This gives all possible rates FROM state equal to 'state'). The Singh model has a few tricks that may be best implemented with conditional statements, and keep in mind that the transition matrix isn't constant (depends upon force/position). So while this approach may be faster or more elegant, you won't all find it easier to code.

If you don't know how to translate a Markov chain into a transition matrix, try rewriting it as a decision tree. What questions do you have to ask of the system to decide what transitions are possible? These questions can be used to translate the model into a series of conditional statements. For example, if you break up the possible transitions as being cases A) where site 1 has ATP bound and B) where site 1 does not have ATP bound, you can write code like this:

```
if site_one_bound_flag
    % Calculate transition rates
    % Calculate time step
    % Randomly sample to choose transition
    % Execute chosen transition
else % Calculate transition rates, given site one is not bound
    % Calculate time step
    % Randomly sample to choose transition
    % Execute chose transition
end
```

How does the uniform random number used to pick a random reaction work?
Assume we have three possible transitions A, B, and C with rates 5 s^{-1}, 5 s^{-1}, and 10 s^{-1} respectively. Then in a given time step the three transitions have probability A=B=0.25 and C=0.5 of occurring. We need to create a probability distribution with three unique values of the random variable with probabilities 0.25, 0.25, and 0.5. We can do this using a uniform distribution from 0 to 1. In a uniform distribution, P(a<X<b) = b-a, that is, the probability of picking a value between two numbers is equal to the difference between the two numbers. Therefore, if you "stack" your events by probability along a number line, you can pick from a uniform distribution as if you were throwing darts, since the "area" on the number line corresponds to the probability of that transition occurring. For us (see below), we'll execute A if the random number is between 0 and 0.25, we'll execute B if the random number is between 0.25 and 0.5, and we'll execute C if the random number is greater than 0.5. We've given each of our events an "area" equal to their probability within the uniform distribution, allowing us to sample from the three state distribution we were interested in.