1. Diagram, text, letter

   Description automatically generated
2. This code works on the framework of the exponential decay assignment done previously which uses the odeint function to solve a given differential equation. Here, we solve for the variable n (potassium gating variable) over different values of time in milliseconds. The first equation in the model function represents alpha, which is a function that depends on voltage. Here we set voltage to a single value outside the function. The next equation is beta which also varies with voltage, but here we set voltage to a single value. The differential equation for n (dn/dt) relies on alpha, beta, and n. We then supply the solver with an initial condition of n which I had set to .1. T is an array linearly spaced into 100 segments from 0 to 11 ms. N is then called as the solution of odeint which is supplied with the function model, initial n, and time. The result was plotted as n^4 (the gating portion of the potassium conductance- gk=n^4\*gkmax) over time to see the gating behavior. To visualize the conductance of potassium you would simply need to multiply this by a chosen gkmax value.
   1. I edited this code to work for multiple values of voltage over the same time period which can be plotted on the same figure. To do this, I added a vector called voltage with values ranging from 11 to 21 at each integer and again at 11 to 210 in 10 even steps. I iterated through these values with a for loop, keeping the same initial condition and time vector for each run. Then I continued to plot the n^4 graph for each value of v on a single figure to show how the conductance changes at different values of clamped voltage. I noticed that higher values of voltage made the conductance rise more quickly and to higher values of steady state conductance (steady state being where the conductance leveled out for long periods of time for each trace)
3. The full Hodgkin Huxley Model was coded as discussed in class. Our ODE solver was solve\_ivp instead of the odeint we had used previously. The inputs to this function were t and X, X being our state variables V, m, n, and h as indicated in the function. Then, all of our constants were initialized including: Nernst Potentials (sodium, potassium, and leakage), conductance (for the same ion channels), the applied current, and the capacitance of the membrane. Next, I copied in the voltage dependent alpha and beta equations for each gate (m,n,h). Next was the equations related to each ionic current based on their ohmic relationship (conductance times voltage difference) and the gating fractions as outlined by Hodgkin and Huxley (the maximum conductance for the channel, multiplied by the gating fractions m, n, and h. Lastly, I compute the derivatives of each changing variable (voltage, m, n, h) that I am tracking. V changes based on the membrane capacitance, applied current, and ionic currents while m,n, and h change based on alpha and beta (which are voltage dependent) and the prior state of m, n, and h. tspan was created to hold the timesteps over which this is being integrated. Then, the variable results is created to store the outcome of the function. The function solve\_ivp is called and the first argument is the name of our function dXdT, then a vector of the start and stop times is used for the next argument (the interval of integration), the next vector contains the initial conditions for each of our state variables, the next argument is the method solution (here I used LSODA), and the last argument t\_eval is set to be equal to tspan a vector containing all of the timesteps. From results, we can call results.t which contains the time vector or results.y. Results.y contains solutions for each of our 4 state variables at each point in time. These can then be plotted to view the solutions. You can plot voltage, m, n, and h individually, altogether, or any combination thereof.
   1. My alteration to this code was trying to see the behavior when you had more than one current pulse during the specified period. Instead of having one 3 mA applied current as I had done previously. However, the way the function works there was not a great way to switch the value of Iapp partway through the time period. In order to accomplish my task, I instead implemented a second function that started at the ending time of the first. All of the values in the function and the equations remained the same, but the Iapp value was increased to 6. In order to make sure this second function wasn’t just operating as its own HH code, I made sure to feed in the state variables from the end of the first run as the initial conditions for this second function. I got these values by selecting the value at the end of the results.y for that state variable. I then appended the results of my two functions together with np.append and plotted them over the full time period of the two functions. This code has potential to be used in the study of refractory periods as you could make the second current pulse come later to illicit action potential (in my test It occurred at 40 ms and did elicit a stereotypical action potential at the time of the change in Iapp) or come closer to the first pulse to see the resulting behavior. The Iapp of the second pulse could also be increased to see if it has effects on the ability to create action potential during the refractory period. This process can easily be repeated with a daisy chain of functions for more Iapp currents that could cause subsequent action potentials. This all makes sense in the context of what I know about the applied current in HH based on prior experience and use of the simulation software neurons in action.