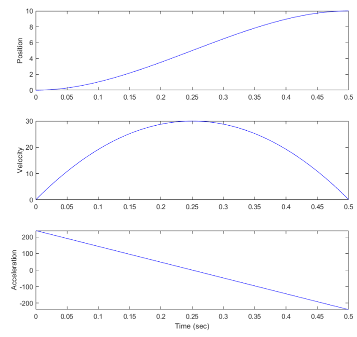
ES 249 Problem Set 3

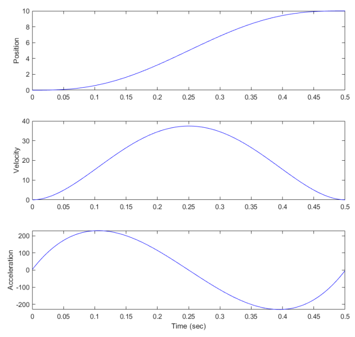
Andrew T. Sullivan

18 April 2019

1 Question 1: Minimum Acceleration, Jerk, and Snap

1.1 a)


(a) Acceleration.


(b) Jerk.

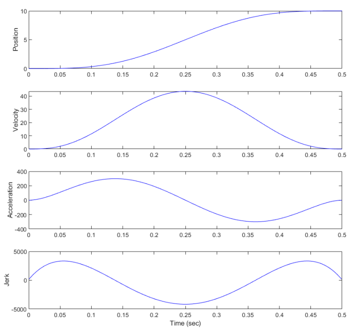
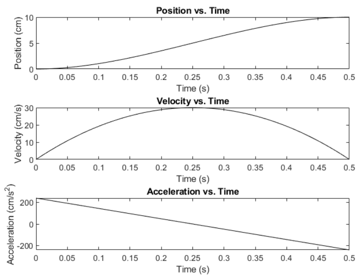
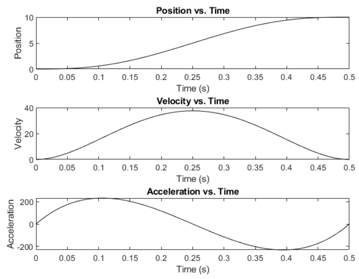

(c) Snap.

Figure 1: Trajectories of Minimized Cumulative Squared Derivatives of Position Using Numerical Nonlinear Optimization.


(a) Energy.


(b) Convergence of Energy.

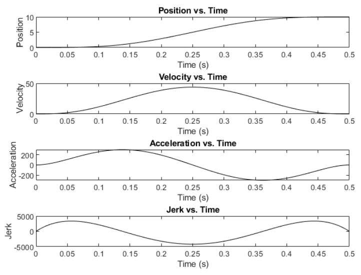
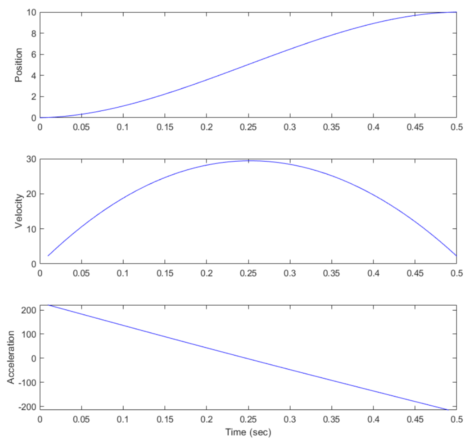

(c) Calculation Time.

Figure 2: Trajectories of Minimized Cumulative Squared Derivatives of Position Using Calculus of Variations.

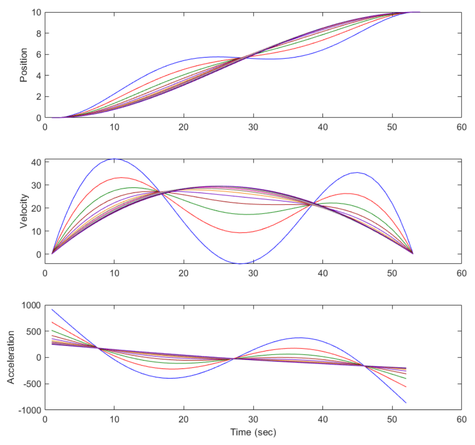
In this section, we use the "lsqnonlin" function in Matlab to solve the minimum acceleration, jerk, and snap problems previously solved analytically using the calculus of variations. Results from the numerical simulations are presented in Figure 1 and can be directly compared to analytical results in Figure 2. Clearly, the numerical methods produce the same behaviors as the analytical methods. Numerical simulations were performed by allowing 1000 points to vary and passing the desired cumulative squared property into the nonlinear least squares function with an initial condition of all zeros. This initial condition was padded with zeros on the left-hand side and the desired value of the final position (10) on the right-hand side such that the values were still present after taking the desired number of derivatives (i.e. 2 for acceleration, 3 for jerk, and 4 for snap). After solving for the minimized trajectory, it was scaled such that the total time along the horizontal axis was 0.5 seconds, and the function and its derivatives were plotted (velocity and acceleration, as well as jerk for the minimzed cumulative squared snap).

1.2 b)

A gradient descent function was constructed by hand in Matlab and was used to perform the same optimization for the minimal cumulative squared acceleration case. Initial conditions were chosen randomly between 0 and 10 so that the gradient was non-zero to begin, and the cumulative squared function was initialized by appending zeros and final values to the initial conditions, rather than during the function call. The gradient descent function features a number of parameters that had to be properly optimized to produce the correct behavior, including the number of points tested (N), the convergence threshold (thresh), the number of consecutive points that needed to be below the convergence threshold for termination (countthresh), and the learning rate (alph). Using the initial conditions, the second derivative of position is first calculated as the value $y1$. The central 1000 points are iterated through and



(a) Acceleration.



(b) Trajectory Evolution.

Figure 3: Convergence to Minimized Cumulative Squared Acceleration Using Handwritten Gradient Descent.

a gradient is approximated using a second-order finite difference equation, which can be shown to be equal to the partial derivative of the function with respect to some coordinate x_i.

We begin this justification by examining the form of the calculated derivatives passed into the optimization function. The problem is initialized with a vector of coordinates, x. The first step in the cumulative squared acceleration function approximates the velocity as the difference between consecutive x values, that is $v_i = x_{i+1} - x_i$. We note that x_i also appears in the equation for $v_{i-1} = x_i - x_{i-1}$. The acceleration is estimated as the difference between consecutive velocity values, that is $a_i = v_{i+1} - v_i = (x_{i+2} - x_{i+1}) - (x_{i+1} - x_i) = x_{i+2} + x_i - 2x_{i+1}$. We can see that, through this method, the acceleration is estimated using a second-order central difference formula. Each acceleration value is then passed into the gradient descent function, which seeks to minimize the cumulative squared acceleration, that is $\Sigma_i(a_i)^2 = \Sigma_i(x_{i+2} + x_i - 2x_{i+1})^2$. Based on the above identities, we note that a specific position variable x_i appears in the expression for three acceleration values, a_i, a_{i-1}, and a_{i-2}. When we differentiate the cost function with respect to this position variable, we will obtain an expression of the form:

$$\frac{d}{dx_i}\Sigma_i(x_{i+2}+x_i-2x_{i+1})^2 = 2((x_{i+2}+x_i-2x_{i+1}) - 2(x_{i+1}+x_{i-1}-2x_i) + (x_i+x_{i-2}-2x_{i-1})) = 2(a_i + a_{i-2} - 2a_{i-1})$$

Therefore, the gradient for each x value is determined from the corresponding acceleration and the previous two acceleration values using a finite-difference method, and this gradient is then used to update the position. The central 1000 points in x are then updated by subtracting the product of the learning rate and the gradient at each point. In this way, the outer bounds remain unchanged, satisfying the boundary conditions. Once the function has been initialized, a while loop is used to iterate until convergence. For each loop, the new x is passed into the cumulative squared acceleration and the result is saved as $y2$. The sum of squared acceleration is evaluated for the previous vector, $ss1$, and for the updated vector, $ss2$. If the difference between the two is below the convergence threshold, the count variable is incremented by one. Otherwise, it is set back to zero. If the count variable is above the cutoff, the loop ends and the converged vector is returned. Otherwise, the gradient update is performed again, and the loop is repeated. In order to examine the evolution of the convergence, the trajectory and its derivatives are plotted every 10000 iterations using the modulo operator with a variable that is incremented every loop to keep track of the number of iterations. The final trajectory and its derivatives are plotted as before.

From this method, we can see that the correct behavior can be properly reproduced, yielding a linearly decreasing acceleration, a parabolic velocity, and a cubic position. While this result is largely independent of the initial conditions for the vector, it is quite susceptible to changes in other system parameters. Previously, a second convergence criterion for the absolute value of the sum of squared acceleration was included, but its value varied substantially with changes in the number of points or the learning rate. Instead, the number of consecutive points below the convergence threshold was increased to 100,000 so that the function is sure to have nearly approached the minimum by the time it ends. Still, it is difficult to say with certainty if the true value of the minimum has been reached, since very small gradients will lead to small increments in these regions of the phase space. Increasing the density of points to values comparable to those used in the previous part causes the trajectory to converge to non-optimal solutions, since the space to explore is too large. Given much longer times, these trajectories should converge to the same behavior, but due to the high dimensionality, this will not be accomplished easily. Reducing the number of points below 50 produces the same qualitative behavior with less smooth features. Similarly, altering the learning rate away from 0.05 changes the converged behavior as well. Reducing the rate produces less optimal results that, given more time and a stricter convergence cutoff (below 10E-9), should ultimately produce the same behavior. Increasing the learning rate substantially (to any value above 0.06) causes divergence and misses the minimum. It seems likely that other combinations of convergence threshold, number of points, and learning rate should produce similar solutions, but these three variables all appear to be interrelated. For example, substantially increasing the number of tested points would likely require a reduction in the convergence threshold and the learning rate to reproduce the proper behavior.

During the iteration, we can see that the trajectory essentially evolves as we would expect. In the first 10,000 iterations, the position approaches some form of sigmoidal function, initially with an apparent inflection point around the center of the plot. This produces a triphasic velocity and a clearly nonlinear acceleration. As the function continues, the humps in the position profile are progressively reduced until the trajectory converges to the proper form of a cubic polynomial. Consequently, the velocity and acceleration become quadratic and linear, respectively. Notably, the trajectories, each spaced apart by 10,000 iterations, become more closely spaced as the algorithm continues. This is expected, as the magnitude of the gradient will be progressively reduced as the minimum is approached.

2 Question 2: The Brachistochrone Problem

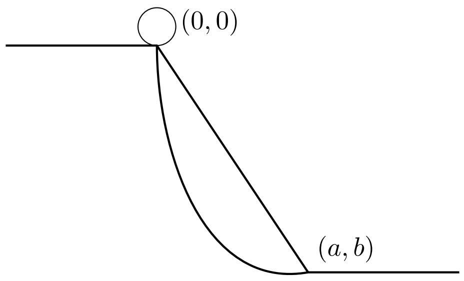


Figure 4: The Brachistochrone Problem.

2.1 a)

The brachistochrone problem is a common example which highlights the utility of calculus of variations. The problem is depicted in the figure above: a ball is initially located at the top of a hill at some position, set to be $(0, 0)$. The ball will travel to a second position, (a, b) by rolling down a curve of arbitrary shape. The problem seeks to find the shape of the curve which minimizes the time necessary for the ball to reach the second position, i.e.:

$$\arg\min_{y(x)} \int_{(0,0)}^{(a,b)} dt$$

Before applying the calculus of variations method, we must first find a functional form that relates the positional variables to the time. To do so, we neglect friction and consider conservation of energy as the ball moves down the ramp. Assuming an initial velocity of zero, the total energy is equal to the potential energy at the starting point, $PE = mgb$, which must equal the kinetic energy at the final point, $KE = \frac{1}{2}mv_f^2$. At all intermediate points, the sum of these two quantities, $E_{tot} = mgy(x) + \frac{1}{2}mv(x)^2$ must be conserved. If we instead use the coordinate system defined above, the initial total energy is equal to zero (since the object begins at y = 0), so at the final position, we have $\frac{1}{2}mv_f^2 - mgb = 0$, and this extends to all other points along the trajectory, so $mgy(x) + \frac{1}{2}mv(x)^2 = 0$, where $y(x)$ begins at zero and ends at b. Then, we can derive the relationship $v = \sqrt{-2gy(x)}$, which we can rewrite as $v = \sqrt{2gy(x)}$ if we flip the positive y direction in our coordinate system.

To obtain a relationship between position and time, we consider the definition of velocity in 2 dimensions, $v = \frac{ds}{dt}$, where s is the arc-length. By simple trigonometry at any point, the infinitessimal increment in the arc-length, ds is the sum in quadrature of the increments in the two positional variables, so $ds = \sqrt{dx^2 + dy^2} = dx\sqrt{1 + \frac{dy}{dx}^2} = dx\sqrt{1 + y'^2}$. From the definition of velocity, $dt = \frac{ds}{v}$, and using our above identities, we can write the time increment as $dt = dx\sqrt{\frac{1+y'^2}{2gy}}$, so our goal is then:

$$\arg\min_{y(x)} \int_{(0,0)}^{(a,b)} dx\sqrt{\frac{1 + y'^2}{2gy}}$$

To minimize this integral, we invoke the method of the calculus of variations, in which we consider an arbitary perturbation to our system, $y(x) \to y(x) + \varepsilon\eta(x)$, and minimize our cost function by setting the derivative with respect to the perturbation parameter $\varepsilon = 0$:

$$\frac{d}{d\varepsilon}\int_0^a dx\sqrt{\frac{1 + (y + \varepsilon\eta)'^2}{2g(y + \varepsilon\eta)}} = 0$$

In general, we can express the above as some arbitary function of x, y, and y', and we can derive the Euler-Lagrange equation. We begin by expanding our derivative, noting that the derivative with respect to the perturbation parameter can be expressed through the chain rule as:

$$\frac{d}{d\varepsilon}\int_0^a f(x, y + \varepsilon\eta, y' + \varepsilon\eta')dx = \int_0^a \frac{\partial}{\partial y}f(x, y, y')\eta dx + \int_0^a \frac{\partial}{\partial y'}f(x, y, y')\eta' dx$$

We can perform integration by parts on the second term on the right, noting that we require the perturbation to vanish at the boundaries to maintain our boundary conditions, so:

$$\int_0^a \frac{\partial}{\partial y'}f(x, y, y')\eta'(x)dx = \frac{\partial}{\partial y'}f(x, y, y')\eta(x)|_0^a - \int_0^a \frac{d}{dx}\frac{\partial}{\partial y'}f(x, y, y')\eta(x)dx = -\int_0^a \frac{d}{dx}\frac{\partial}{\partial y'}f(x, y, y')\eta(x)dx$$

Equating this to zero yields the following equation:

$$\int_0^a (\frac{\partial}{\partial y}f(x, y, y') - \frac{d}{dx}\frac{\partial}{\partial y'}f(x, y, y'))\eta dx = 0$$

The fundamental theorem of the calculus of variations states that if the integral of the product of a fixed function and any arbitrary continuous function is equal to zero, this implies that the fixed function must be zero over the entire interval, therefore:

$$\frac{\partial}{\partial y}f(x, y, y') - \frac{d}{dx}\frac{\partial}{\partial y'}f(x, y, y') = 0$$

And so,

$$\frac{\partial}{\partial y}f(x, y, y') = \frac{d}{dx}\frac{\partial}{\partial y'}f(x, y, y')$$
$$\frac{\partial}{\partial y}\sqrt{\frac{1 + y'^2}{2gy}} = \frac{d}{dx}\frac{\partial}{\partial y'}\sqrt{\frac{1 + y'^2}{2gy}}$$

Evaluating the partial differentials on either side,

$$\frac{-1}{2}\sqrt{\frac{1 + y'^2}{2gy^3}} = \frac{d}{dx}\sqrt{\frac{y'^2}{2gy(1 + y'^2)}} = \frac{y''}{\sqrt{2gy(1 + y'^2)}} - \frac{1}{2}\frac{y'^2}{\sqrt{2gy^3(1 + y'^2)}} - \frac{y'^2 y''}{\sqrt{2gy(1 + y'^2)^3}}$$

It should be noted that this final identity can be shown to be equal to the result from the familiar quotient rule by multiplying through by the common denominator $\sqrt{2gy^3(1 + y'^2)^3}$. From this point, we can multiply through on either side by $\sqrt{2gy(1 + y'^2)}$, which yields:

$$\frac{-1}{2}\frac{1 + y'^2}{y} = y'' - \frac{1}{2}\frac{y'^2}{y} - \frac{y'^2 y''}{1 + y'^2}$$

Moving over the central term reduces the left-hand side to $\frac{-1}{2y}$, and the right-hand side can be simplified to $\frac{y''}{1+y'^2}$, so we have:

$$-(1+y'^2) = 2yy''$$

Thus, we obtain a final differential equation by moving everything to one side and multiplying through by y':

$$2yy'y'' + y' + y'^3 = 0$$

We note that this is the derivative of the expression $y + yy'^2$ with respect to y, implying that the former expression is equal to a constant, C, to satisfy this equation. We can solve this for y' as:

$$\frac{dy}{dx} = \sqrt{\frac{C-y}{y}}$$

Therefore, we have a relationship between our two positional variables,

$$x = \int_0^b \sqrt{\frac{y}{C-y}}dy + A$$

Rather than explicitly considering y as a function of x, we can parameterize both spatial variables as functions of an angle, θ, such that $y = C\sin^2\theta$. Then using the relation $dy = 2C\sin\theta\cos\theta$,

$$x = \int \sqrt{\frac{C\sin^2\theta}{C - C\sin^2\theta}}dy = \int \frac{\sin\theta}{\cos\theta}2C\sin\theta\cos\theta d\theta = 2C\int \sin^2\theta d\theta$$

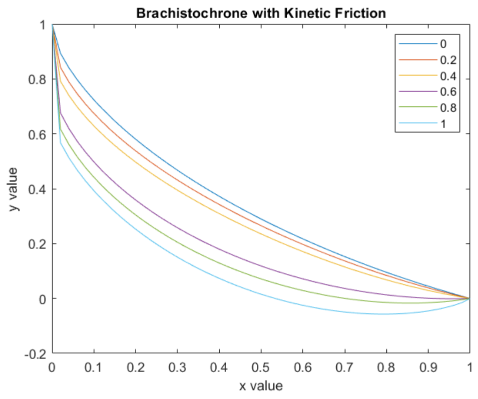
We invoke the trigonometric identity $\sin^2\theta = \frac{1}{2} - \frac{1}{2}\cos 2\theta$ and integrate:

$$x = 2C\int \frac{1}{2} - \frac{1}{2}\cos 2\theta d\theta = C(\theta - \frac{1}{2}\sin 2\theta) + A$$

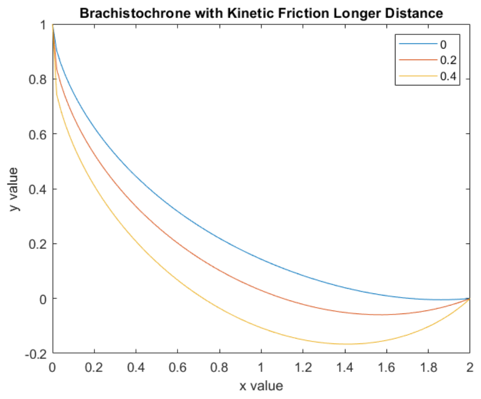
From here, the remainder of the problem is simply solving based on the boundary conditions. As mentioned previously, we begin at the point (0,0). Solving our y equation for θ at this point yields $\theta = \frac{1}{2}\arccos 1 = 0$. Plugging this into the equation for x yields $A = 0$, satisfying our first boundary condition. For the second boundary condition, the final point (a, b), requires some numerical estimation as the equations for x and y are not easily invertible for non-zero values of θ. We can divide the two equations, evaluated at the point of interest, and estimate the value of θ that yields the correct ratio $\frac{a}{b} = \frac{\theta - \frac{1}{2}\sin 2\theta}{\sin^2\theta}$. From this value, we can then solve either the x or y equation for the value of C which yields the correct a or b. This solves the brachistochrone problem. From these results, we can observe that x and y form a cycloidal path, i.e. the path formed by the edge of a circle as it rolls along a horizontal line.

2.2 b)

With the analytical solution to the brachistochrone problem solved, we next turn to the issue of introducing kinetic friction to the system. Friction induces the dissipation of energy, thereby preventing our previous use of conservation of energy during the transport. Rather than consider energy differences arising from the trajectory, we instead consider forces acting at each point and invoke Newton's Second Law to derive an analogous form to the kinematic equations and linearize the velocity within each interval x_i to x_{i+1} (i.e. assume constant acceleration to use the kinematic equations). For the first case, we assume the coefficient of kinetic friction is zero and ensure that we obtain the same trajectory as the analytical case. Our function, brach2.m, takes in a vector corresponding to the y-coordinates of a line beginning at y=0.98 and ending at y=0.02 in increments of 0.02. An x vector is constructed within the function to yield equally spaced points with a length equal to the augmented y vector. Within our function, the vector of y-values is first augmented by appending the desired initial and final conditions to either side so these boundaries are enforced. Since our algorithm uses a Levenberg-Marquardt algorithm for optimization, we cannot input upper and lower bounds and must enforce our conditions this way. The tangential acceleration provides the forward force and is given by $a_i = 9.81\sin\theta_i$, where the angle θ is obtained as the arctangent of the local change in y over the local change in x, i.e. as $\theta = atan2(diff(y_i)/diff(x_i))$. We note that here the atan2 function will return a negative angle if the argument is less than zero, and so for our case where the y-coordinate is decreasing along the path, we will still obtain the correct acceleration in the downward direction. We also initialize the velocity and time vectors to be zero, and only update the velocity from position 2 so the boundary condition of $v(0) = 0$ is enforced. For each interval along the trajectory, we evaluate the distance traveled as $\Delta = -\sqrt{(x_{i+1}-x_i)^2 + (y_{i+1}-y_i)^2}$. We can then obtain the time taken to traverse the interval as $\Delta t = \frac{-v_i - \sqrt{v_i^2 + 2a_i\Delta_i}}{a_i}$. The velocity at the end of the interval, and therefore the initial velocity for the next interval, is then found as $v_{i+1} = v_i + a_i \Delta t_i$. A value of 0 is appended onto the first position of t to correspond to the initial conditions. The function returns the sum of the dt vector, corresponding to the total time, i.e. what we wish to minimize. In order to introduce kinetic friction, we note that the acceleration is computed as the tangential component of the acceleration due to gravity, and that by multiplying by the mass, we may obtain Newton's Second Law, $ma_i = mg\sin\theta_i$. Including friction is equivalent to adding a force on the right-hand side of this equation which opposes the downward-going acceleration and has a magnitude equal to $uN = umg\cos\theta_i$, in which u is the coefficient of kinetic friction and N is the normal force. Thus, we can obtain the effective acceleration at each point along the trajectory as $a_i = -g\sin\theta_i + ug\cos\theta_i$. The options for the lsqnonlin optimization are altered so that the number of iterations is not constrained and will continue until convergence, and the function value convergence cutoff is set to 1E-7. We perform this calculation for six values of the coefficient of kinetic friction, between 0 and 1 in increments of 0.02. However, the optimizer appears to diverge at a value of 1, and so we approximate this final point as 0.999. From this plot, we can see that increasing the coefficient of kinetic friction serves to increase the magnitude of the initial slope, likely providing the object with a larger initial velocity and acceleration to counteract the friction. This is more evident in the case where the final distance is 2 instead of 1, as even for the $u = 0.2$ case, we observe the presence of a local minimum in the trajectory. For larger values of the coefficient of friction in this case, the calculated angle, θ takes on values such that the magnitude of the acceleration is in the positive y direction, thus providing an unphysical solution which suggests the ball will roll uphill. Instead, it seems as though these cases, at least for the initial conditions presented, the object will not move down the ramp with these large coefficients of friction and relatively small inclinations, and will instead remain at rest, balanced by static friction and gravity. There are other potential reasons why these large-friction cases seem to diverge, including an error in the implementation of the algorithm or its initialization. For high-friction cases, initializing the y vector as a line between x values of 0 and 2 produces angles which yield positive initial accelerations at all points, which in turn produce complex time values. It may be possible to use different initial conditions which yield more appropriate trajectories, but preliminary exploration including randomly generated values, zeros, or using the previous trajectory as the initial configuration have been unsuccessful. Within the algorithm, the definition of acceleration and the Δ quantity may also be made more complex, as for the former, the current implementation always assumes that the frictional component points upwards, which may not be true, e.g. at points after the minimum of the trajectory for the 0.4 case. The Δ quantity is also assumed to be negative, but for cases where the trajectory is pointing uphill, this is not necessarily accurate. In any case, we see the same qualitative behavior, i.e. that higher friction conditions shift away from a linear trajectory to provide larger initial velocities and accelerations to ultimately counteract friction. As the issues at high friction show, trajectories closer to linearity increase the probability that friction will counteract gravity and prevent motion altogether.



(a) For Lateral Distance of 1.



(b) Lateral Distance of 2.

Figure 5: Numerical Solutions to the Brachistochrone Problem with Kinetic Friction.

3 Question 3: The Hodgkin-Huxley Model

In order to simulate the Hodgkin-Huxley model of action potential propagation, two functions were constructed in Matlab. The first takes in a time variable, an input vector containing the state of the system in terms of membrane potential and the values of the three gating variables, and the injected current. This function uses these variables, along with specified conductances, Nernst potentials, and forms of the a and b equations for each gating variable to evaluate the differential equations governing how each state variable changes in time. The output of the function contains the time derivatives of all four state variables. The second function is broken up into seven parts. In the first, the potential is set equal to the resting membrane potential of the cell of -61.2 mV, the time vector is produced over which the dynamics are evaluated, and the initial conditions for the gating variables are introduced, based on solving the asymptotic voltage/infinity curves for each variable at the resting membrane potential. The conductances and Nernst potentials are also defined here for use in subsequent parts.

3.1 a)

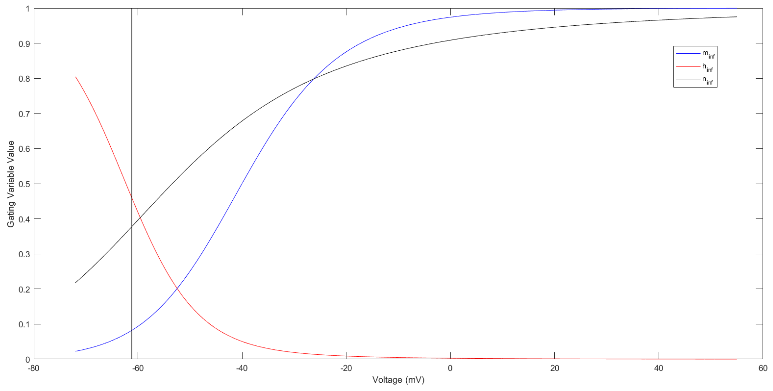


Figure 6: Voltage Dependence of m_∞, h_∞, and n_∞ Curves.

The first figure for the Hodgkin-Huxley model contains the voltage dependence of the asymptotic curves for each of the three gating variables, evaluated using the functional forms specified above. The voltage range is chosen to be between the Nernst potential for potassium, -72 mV, and for sodium, +55 mV. As expected, the activation variables, m and n, increase as a function of cell depolarization, while the inactivation variable, h, decreases. The resting membrane potential is indicated by the black vertical line. At this state, most sodium channels are closed or inactivated, reflecting the substantially lower membrane conductance for sodium at rest compared to potassium.

3.2 b)

The next figure shows the influence of injected current on the model. We evaluate this response by initializing the current as an anonymous function of time with fixed magnitude for each iteration, and initialize the state variables as defined above. For each current, a fourth-order Runge-Kutta method is used to evaluate the system of four coupled differential equations at all time points, and the results are plotted. Each subplot shows the membrane response to a different injected current density, ranging from 1 to 10 $\mu A/cm^2$. In the top-left subplot, it is apparent that the membrane voltage fluctuates briefly, but this low current density is insufficient to produce a suprathreshold depolarization and will not initiate an action potential. Subsequent larger current densities all produce action potentials, with the time of the first spike progressively decreasing as threshold is reached faster, and the firing frequency increasing for subsequent spikes as well. Though this is the expected qualitative behavior, constant current input is typically not a biologically relevant condition.

3.3 c)

The third figure for this model explores the response to a pulse of current, rather than a sustained injection, bringing the situation closer to the reality in *in vivo* neural systems. The current is initialized such that it has a value of $10\mu A/cm^2$ between 5 and 6 msec, and 0 elsewhere.The shape of the action potential is similar to those in the previous

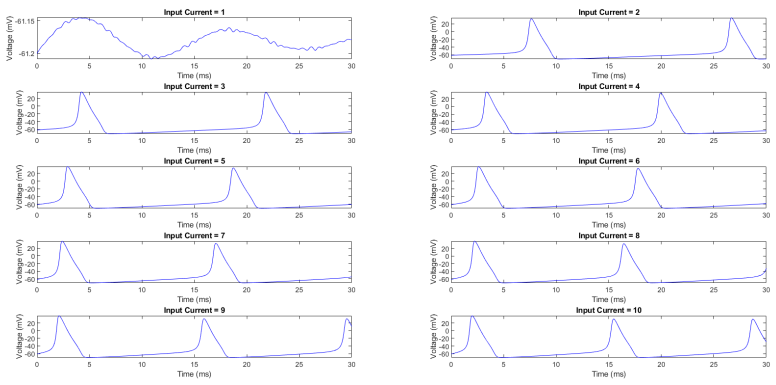


Figure 7: Evolution of Membrane Potential in Response to Constant Current Densities.

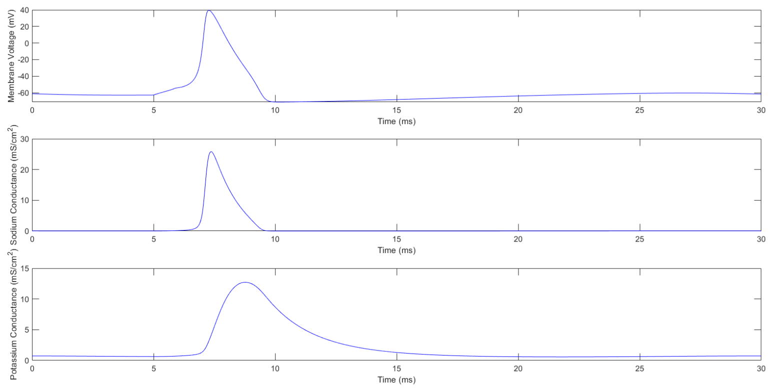


Figure 8: Evolution of Membrane Potential and Conductances in Response to 1 msec Current Pulse of Magnitude 10 $\mu A/cm^2$.

figure, with a slow initial increase as threshold is approached, followed by a transient upstroke, prolonged downstroke, and afterhyperpolarization. Examining the following two subfigures reveals the contribution of conductances to these behaviors. Here, the conductance is plotted as $G_{Na}m^3h$ for sodium and G_Kn^4 for potassium. We can see that, like the action potential, the sodium conductance rapidly increases to a maximum. Around this time, the action potential also reaches its maximal value, approaching but not reaching the sodium equilibrium potential. The sodium conductance then begins to decrease as the slower inactivation variable decreases, and the potassium conductance increases, counteracting the sodium influx with potassium efflux. The sodium conductance returns approximately to its baseline value after a few msec, but the potassium remains elevated for several milliseconds later, producing the sustained afterhyperpolarization and refractory period characteristic of action potentials.

3.4 d)

We next explore the activation function of the Hodgkin-Huxley neuron, characterized by the relationship between peak membrane current and the input current magnitude of a 1 msec pulse. We initialize the current using the same scheme as the previous part, but changing the magnitude of the pulse for each iteration of the loop. The maximal value is extracted from each voltage vs. time trace and stored to produce the activation function. This function is sigmoidal in nature and is approximately a logistic function, consistent with other models of neural activation. Below input current pulses of approximately $4\mu A/cm^2$, small subthreshold depolarizations are all that result from input. Above this value, the neuron fires an action potential with nearly constant amplitude. While the action potential is typically regarded as an all-or-none phenomenon, this function reveals a small deviation from step-function behavior at current densities very close to the threshold. It should be noted that these are currents of fixed density and duration, and thus changes in area or time will produce different thresholds, all of which correspond to approximately the same voltage threshold.

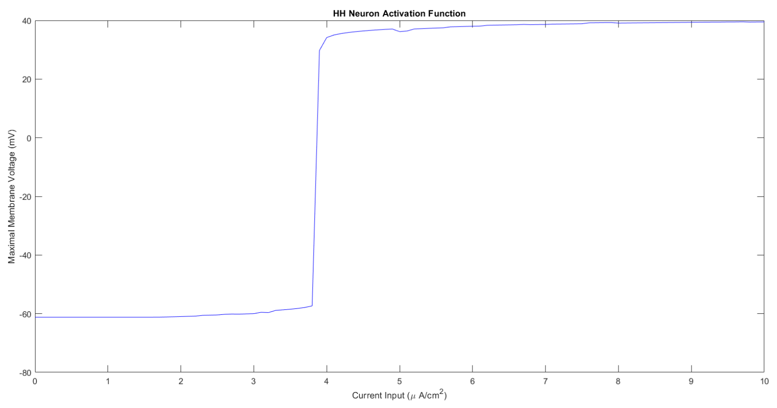


Figure 9: Relationship Between Current Magnitude and Peak Membrane Voltage for 1 msec Pulse.

3.5 e)

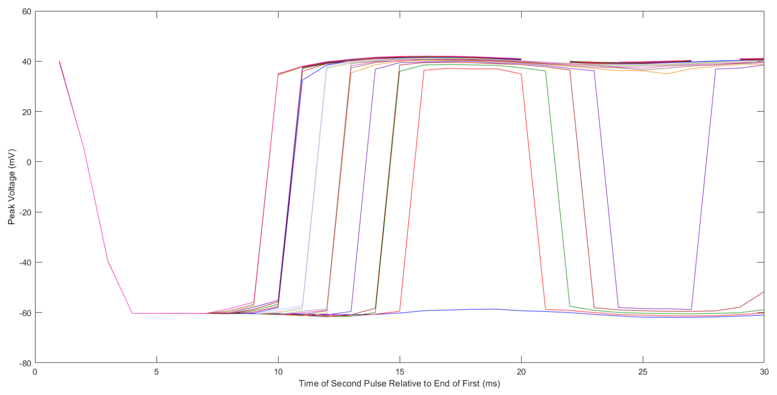
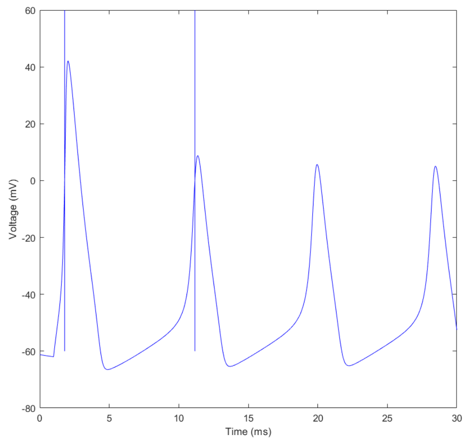


Figure 10: Relationship Between Time After First 1 msec-10 $\mu A/cm^2$ Pulse and Peak Membrane Voltage for 1 msec Pulses of Varying Magnitude.

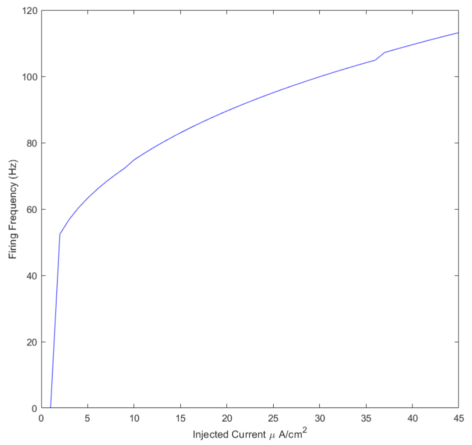
In order to explore the influence of the refractory period, we begin all simulations with a 1 msec-10 $\mu A/cm^2$ pulse, as seen before. We add to this a second pulse of varying magnitude at a varying time after the first pulse. For a given magnitude, we simulate the resultant voltage to a total of 40 msec with the second pulse coming between 1 and 30 msec after the end of the first pulse. We perform this simulation for current magnitudes between 1 and 20 $\mu A/cm^2$. From each trace, we extract the maximum voltage attained after the end of the first pulse, and plot each peak voltage-time after the first pulse trace for all magnitudes. In this plot, the smallest magnitude is the curve which does not ever pass above -50 mV, except within the first 3 msec, since these maximal voltages instead reflect the first action potential. As currents get larger, the curves increase in magnitude along with the portion of the curve reflecting suprathreshold depolarization. The next largest curve only shows the initiation of a second action potential with pulses 15 msec after the first, while the largest magnitude can initiate a second spike only 9 msec after the first pulse ends, reflecting the so-called "relative refractory period." Interestingly, lower-amplitude pulses show a return to subthreshold depolarization around 20-25 msec after the first pulse. Investigation of the voltage traces shows the presence of small potential oscillations around this value, which are sufficient to counteract the influence of the input current if this current is small enough in magnitude.

3.6 f)

The final figure in the Hodgkin-Huxley model captures the voltage dependence of neuronal firing rate. We simulate this by first declaring a threshold above which we qualify a membrane depolarization as a spike (here, set to 0 mV). The relevant current density range is chosen to be 1 to 45 $\mu A/cm^2$, since values above this range no longer



(a) Method for Calculating Firing Frequency.



(b) Firing Frequency.

Figure 11: Relationship Between Firing Rate and Current Magnitude for Constant Current Densities.

produce meaningful results (no spikes are seen and the model appears to diverge, which may be a consequence of the solver being used). For each current density, current is injected beginning at 1 msec through the end of the simulation (30 msec). We extract spikes from the voltage trace by finding all values of the voltage that are above the threshold. From these indices (if they exist), we find all indices where the difference to the next index is not equal to 1, i.e. corresponding to the end of one spike and the beginning of the next. For each of these indices, we take the one beginning the following spike. We also add on the first value in the array, corresponding to the onset of the first spike. For each spike for a given density, we extract the interspike intervals and average their values. This is schematically depicted in the first subfigure above for the first two spikes. If there are zero or one detected spike in the range, we set the frequency equal to zero. For all other cases, the firing frequency is taken as the reciprocal of the average interspike interval. This is plotted as a function of input current density, revealing the presence of an initial threshold, as seen before, and a sublinear relationship between firing rate and input current above the initial linear regime. Most neurons do not fire at such high frequencies (above 100 Hz), but instead stay closer to the most sensitive dynamical regime (around 60 Hz or less).

4 Appendix

The first codes presented are those which use Matlab's lsqnonlin function to optimize the minimal cumulative squared acceleration, jerk, and snap trajectories.

Acceleration:

```
N=1000;
x0=zeros(1,N);

csj = @(x,xf) diff([0,0,x,xf,xf],2);

x=lsqnonlin(@(x) csj(x,10), x0);
x_ = [0,x,10];
Tf=0.5;
t=[0:(N+1)]/(N+1)*Tf;
dt=Tf/(N+1);
figure;

subplot(3,1,1);
plot(t,x_);
ylabel('Position');
hold on;

subplot(3,1,2);
plot(t(2:end),diff(x_)/dt);
ylabel('Velocity');
hold on;

subplot(3,1,3);
plot(t(2:end-1),diff(x_,2)/dt^2);
ylabel('Acceleration');
xlabel('Time (sec)');
hold on;

hold off;
```

Jerk:

```
N=1000;
x0=zeros(1,N);

csj = @(x,xf) diff([0,0,0,x,xf,xf,xf],3);

x=lsqnonlin(@(x) csj(x,10), x0);
x_ = [0,x,10];
Tf=0.5;
t=[0:(N+1)]/(N+1)*Tf;
dt=Tf/(N+1);
figure;

subplot(3,1,1);
plot(t,x_);
ylabel('Position');
hold on;

subplot(3,1,2);
plot(t(2:end),diff(x_)/dt);
ylabel('Velocity');
hold on;
```

```
subplot(3,1,3);
plot(t(2:end-1),diff(x_,2)/dt^2);
ylabel('Acceleration');
xlabel('Time (sec)');
hold on;

hold off;
```

Snap:

```
N=1000;
x0=zeros(1,N);

csj = @(x,xf) diff([0,0,0,0,x,xf,xf,xf,xf],4);

x=lsqnonlin(@(x) csj(x,10), x0);
x_ = [0,x,10];
Tf=0.5;
t=[0:(N+1)]/(N+1)*Tf;
dt=Tf/(N+1);
figure;

subplot(4,1,1);
plot(t,x_);
ylabel('Position');
hold on;

subplot(4,1,2);
plot(t(2:end),diff(x_)/dt);
ylabel('Velocity');
hold on;

subplot(4,1,3);
plot(t(2:end-1),diff(x_,2)/dt^2);
ylabel('Acceleration');
hold on;

subplot(4,1,4);
plot(t(2:end-2),diff(x_,3)/dt^3);
ylabel('Jerk');
xlabel('Time (sec)');
hold on;

hold off;
```

The following code is used to initialize the acceleration condition for the hand-written gradient descent function:

```
N=50;
x0=10*rand(1,N);
xf = 10;

x0 = [0,0,x0,xf,xf];
csj = @(x) diff(x,2);

x=grad_desc(@(x) csj(x), x0);
x_ = x(2:end-1);
Tf=0.5;
t=[0:(N+1)]/(N+1)*Tf;
```

```
dt=Tf/(N+1);
figure;

subplot(3,1,1);
plot(t,x_);
ylabel('Position');
hold on;

subplot(3,1,2);
plot(t(2:end),diff(x_)/dt);
ylabel('Velocity');
hold on;

subplot(3,1,3);
plot(t(2:end-1),diff(x_,2)/dt^2);
ylabel('Acceleration');
xlabel('Time (sec)');
hold on;

hold off;
```

The following code is used to perform gradient descent for the minimal cumulative squared acceleration:

```
function [val] = grad_desc(fcn, x0)
%This function takes in a user-defined function and a vector of initial
   conditions
%It returns a vector of the optimized points to minimize the L2 norm of
%acceleration

thresh = 10E-9;           %convergence threshold
count = 0;                 %count variable to keep track of the consecutive
   number of below-threshold iterations
countthresh = 100000;           %number of required consecutive iterations below
   threshold
y1=fcn(x0);                %evaluate at the initial point
alph=0.05;                 %learning rate
x=x0;

num = 0;

N=50;
Tf=0.5;
dt=Tf/(N+1);
figure;
for i = 3:length(x0)-2
   j=i-2;
   g(j) = 2*(y1(i-2)-2*y1(i-1)+y1(i));
end

for i = 1:length(g)
    x(i+2)=x0(i+2)-alph*g(i);
end

while count < countthresh
    y2=fcn(x);
    ss1 = sum(y1.^2);
    ss2 = sum(y2.^2)
    converge = abs(ss2-ss1);
    if (converge < thresh)
        count=count+1;
    else
        count = 0;
    end
    if count < countthresh
        y1 = y2;
        for i = 3:length(x0)-2
            j=i-2;
            g(j) = 2*(y1(i-2)-2*y1(i-1)+y1(i));
        end

        for i = 1:length(g)
            x(i+2)=x(i+2)-alph*g(i);
        end
    end
    if mod(num,10000) == 0 && num ~= 0
        subplot(3,1,1);
        plot(1:length(x),x);
        ylabel('Position');
        hold on;

        subplot(3,1,2);
        plot(1:length(x)-1,diff(x)/dt);
        ylabel('Velocity');
        hold on;

        subplot(3,1,3);
        plot(1:length(x)-2,diff(x,2)/dt^2);
        ylabel('Acceleration');
        xlabel('Time (sec)');
        hold on;
    end
    num=num+1;
end
val = x;
disp(ss2);
disp(num);
```

This code is used to solve the kinematic equations and is passed into the lsqnonlin function for trajectory optimization:

```
function time = brach2(y,u)
y=[1;y(:);0]';
x=[0:(2/(length(y)-1)):2];
theta = atan2(diff(y),diff(x));
a = 9.81*(sin(theta))+u*(9.81*cos(theta));
%a = 9.81*(sin(theta));
v = zeros(1,length(x));
t = zeros(1,length(x)-1);

for i = 1:length(x)-1
    del(i) = -sqrt((x(i+1)-x(i))^2+(y(i+1)-y(i))^2);
    %disp(sign(v(i)^2+2*a(i)*(del(i))));
    t(i) = (1/a(i))*(-v(i)-sqrt((v(i)^2+2*a(i)*(del(i)))));
    v(i+1) = v(i)+a(i)*t(i);
end
t=[0,t];
```

```
if (min(t)<0)||(sum(~isreal(t)) > 0), time=inf; disp('Ball gets stuck!'); else
    time=sum(t); end
% plot(cumsum(t),v);
% figure;
% plot(x,y);
end
```

This script is used to initialize and call the nonlinear optimization for the brachistochrone problem with kinetic friction:

```
N = 99;
y0 = [.99:-.01:.01];

figure;
u=[0:0.2:0.8,0.999];
options = optimoptions(@lsqnonlin,'FunctionTolerance',1E-7,'
    MaxFunctionEvaluations',Inf,'MaxIterations',Inf);
for i = 1:6
    y(i,:) = lsqnonlin(@(y) brach2(y,u(i)), y0, [], [], options);
    ytot(i,:) = [1,y(i,:),0];
    plot([0:(2/(length(ytot)-1)):2],ytot(i,:));
    disp('next...');
    hold on
end

xlabel('x value');
ylabel('y value');
title('Brachistochrone with Kinetic Friction');
legend('0','0.2','0.4','0.6','0.8','1');
hold off;
```

The following code is the Hodgkin-Huxley function evaluator, which evaluates the four coupled differential equations based on the state of the system:

```
function [Output] = HH(t,in,I)
E = in(1);
m = in(2);
h = in(3);
n = in(4);
Output = zeros(4,1);

G_Na = 120;
G_K = 36;
G_L = 0.3;
E_Na = 55;
E_K = -72;
E_L = -50;

Output(1) = -G_Na.*m^3.*h.*(E-E_Na)-G_K.*n^4.*(E-E_K)-G_L.*(E-E_L)+I(t);

am = @(x) -0.1*(40+x)/(exp(-(40+x)/10)-1);
bm = @(x) 4*exp(-(65+x)/18);

ah = @(x) 0.07*exp(-(x+65)/20);
bh = @(x) 1/(exp(-(35+x)/10)+1);

an = @(x) -0.01*(55+x)/(exp(-(55+x)/10)-1);
bn = @(x) 0.125*exp(-(x+65)/80);
```

```
Output(2) = am(E)*(1-m)-bm(E)*m;
Output(3) = ah(E)*(1-h)-bh(E)*h;
Output(4) = an(E)*(1-n)-bn(E)*n;

tm = 1/(am(E)+bm(E));
minf=am(E)/(am(E)+bm(E));
th = 1/(ah(E)+bh(E));
hinf = ah(E)/(ah(E)+bh(E));
tn = 1/(an(E)+bn(E));
ninf = an(E)/(an(E)+bn(E));
```

Finally, the following script is used to solve all six parts of the Hodgkin-Huxley model question:

```
E = -61.2;
figure;
m0=0.0820;
h0=0.4603;
n0=0.3772;
tinc=0.001;
tspan=0:tinc:30;

G_Na = 120;
G_K = 36;
G_L = 0.3;
E_Na = 55;
E_K = -72;
E_L = -50;

%% Part 1
Vrange = [-72:0.01:55];
for i = 1:length(Vrange)
    am = -0.1*(40+Vrange(i))/(exp(-(40+Vrange(i))/10)-1);
    bm = 4*exp(-(65+Vrange(i))/18);

    ah = 0.07*exp(-(Vrange(i)+65)/20);
    bh = 1/(exp(-(35+Vrange(i))/10)+1);

    an = -0.01*(55+Vrange(i))/(exp(-(55+Vrange(i))/10)-1);
    bn = 0.125*exp(-(Vrange(i)+65)/80);

    tm = 1/(am+bm);
    minf(i)=am/(am+bm);
    th = 1/(ah+bh);
    hinf(i) = ah/(ah+bh);
    tn = 1/(an+bn);
    ninf(i) = an/(an+bn);
end

plot(Vrange,minf);
hold on;
plot(Vrange,hinf);
plot(Vrange,ninf,'k');
yl = ylim;
line([E E],yl,'Color','k');
xlabel('Voltage (mV)');
ylabel('Gating Variable Value');
legend('m_{inf}','h_{inf}','n_{inf}');
hold off;
```

```
%% Part 2

figure;
for k = 1:10
    I = @(t) k;
    y0 = [E m0 h0 n0];
    [T,Y] = ode45(@(t,in) HH(t,in,I), tspan, y0);
    subplot(5,2,k);
    plot(T,Y(:,1));
    xlabel('Time (ms)');
    ylabel('Voltage (mV)');
    plottitle = ['Input Current = ',num2str(k)];
    title(plottitle);
end

%% Part 3

figure;
I = @(t) (t>=5 && t<6)*10;
[T,Y] = ode45(@(t,in) HH(t,in,I), tspan, y0);

subplot(3,1,1);
plot(T,Y(:,1));
xlabel('Time (ms)');
ylabel('Membrane Voltage (mV)');

for i = 1:length(T)
    g_Na(i) = G_Na*Y(i,2)^3*(Y(i,3));
    g_K(i) = G_K*Y(i,4)^4;
    %g_Na(i) = Y(i,3);
    %g_K(i) = Y(i,4);
    IT(i)=I(T(i));
end

% subplot(3,1,1);
% plot(T,IT);
% xlabel('Time (ms)');
% ylabel('Input Current (\mu A/cm^2)');

subplot(3,1,2);
plot(T,g_Na);
xlabel('Time (ms)');
ylabel('Sodium Conductance (mS/cm^2)');

subplot(3,1,3);
plot(T,g_K);
xlabel('Time (ms)');
ylabel('Potassium Conductance (mS/cm^2)');

%% Part 4
figure;

Irange = 0:0.1:10;
for k = 1:length(Irange)
    I = @(t) (t>=5 && t<6)*Irange(k);
    [T,Y] = ode45(@(t,in) HH(t,in,I), tspan, y0);
    Vmax(k) = max(Y(:,1));
end

plot(Irange,Vmax);
xlabel('Current Input (\mu A/cm^2)');
ylabel('Maximal Membrane Voltage (mV)');
title('HH Neuron Activation Function');

%% Part 5
figure;

Irange2 = 1:20;
trange = 1:30;
step = 0.001;
tspan2 = 0:step:40;
for k = 1:length(Irange2)
    for l = 1:length(trange)
        I = @(t) (t>=5 && t<6)*10 + (t>=6+trange(l) && t<6+trange(l)+1)*Irange2(
           k);
        [T,Y] = ode45(@(t,in) HH(t,in,I), tspan2, y0);
        Vmax2(k,l) = max(Y(((6+trange(l))/step):end,1));
%          if k == 2
%              plot(T,Y(:,1));
%              hold on;
%              disp(l);
%              pause;
%
%          end
    end
    plot(Vmax2(k,:));
    xlabel('Time of Second Pulse Relative to End of First (ms)');
    ylabel('Peak Voltage (mV)');
    hold on;
end
%% Part 6

figure;
thresh = 0;
Irange3 = 1:1:45;
spikes = cell(1,length(Irange3));
ind = spikes;
ISI = spikes;
ISIavg = zeros(1,length(Irange3));
freq = ISIavg;
for k = 1:length(Irange3)
    I = @(t) (t>=1)*Irange3(k);
    [T,Y] = ode45(@(t,in) HH(t,in,I), tspan, y0);
    V = Y(:,1);
    plot(T,V);
    hold on;
    ind{k} = find(V>thresh);
    if isempty(ind{k})
        freq(k) = 0;
        continue;
    end
    inds = find(diff(ind{k}) ~= 1);
    spikes{k} = [ind{k}(1);ind{k}(inds+1)];
    line([T(spikes{k}(1)) T(spikes{k}(1))],[-60 60]);
    if length(spikes{k}) <= 1
        freq(k) = 0;
        print('empty')
    else
        for i = 1:length(spikes{k})-1
            ISI{k} = [ISI{k},(spikes{k}(i+1)-spikes{k}(i))];
        end
        ISIavg(k) = tinc*mean(ISI{k});
        freq(k) = 1000/ISIavg(k);
        line([T(spikes{k}(2)) T(spikes{k}(2))],[-60 60]);
    end
    xlabel('Time (ms)');
    ylabel('Voltage (mV)');
    hold off;
    if k>46
        pause;
    end
end

figure;
plot(Irange3,freq);
xlabel('Injected Current \mu A/cm^2');
ylabel('Firing Frequency (Hz)');
```