Applied Math: Assignment 02

Satchel Schiavo, Braden Vaccari, Alton Coolidge

October 3rd, 2024

Contents

1 Time Worked on Assignment

- Satchel Schiavo: 22 hours
- Braden Vaccari: 13 hours
- Alton Coolidge: 24 Hours

2 Newton's Method for Multiple Dimensions

We first went about implementing multi-Newton's method by using the one dimensional Newton's method as a template. Many of the early termination conditions remain in spirit, but now instead of given derivative value we must compute the Jacobian, or a matrix of partial derivatives instead of single derivative value. Therefore, the first challenge lied in computing the Jacobian:

Programming the approximate Jacobian function began with understanding how to compute vector multiplication across an array of each partial derivative, and then storing each of those derivatives in a matrix. Just as one-dimensional Newton's method computes the line tangent to a guess with x-guess - (function(x)/derivative(x)) until function(x) results a value of close to zero, we now go through each column at a time but with our newly acquired Jacobian.

It was challenging to understand the Jacobian at first, especially shifting from full derivatives to partial and single value to matrices simultaneously.

Some of the biggest issues we ran into were changing the set number of times and realizing we had to change the while loops to if loops. Initially, we were hesitant to implement a for loop instead of a while loop as we were essentially estimating the number of iterations we would need to compute, but we eventually ran into an error case which never exited the while loop. We instead put the while condition in an if statement, which computed properly. Another issue we ran into was properly implementing the boolean check for an approximate Jacobean versus a numerical Jacobean, as at first we hard-coded a numerical into Newton's solver function, which didn't work once we tested the collision function. However, back-tracking to the previous project for how we computed the derivative gave us the answer for the correct implementation.

We verified our implementation beyond the main test of a "Simulation of a Projectile Shot on Target" given in class by plotting an acceptable range of guesses for both theta and t and seeing if an acceptable amount ran. The difficult thing about testing multi-newton's method lies in the main failure mode of newton's method: an initial guess too far removed from the root will not compute. This problem is exacerbated by now having two inputs, theta and t, which can break the solver. However, through using the run simulation function, we could test a range of both theta (from pi/3 to pi/6) and t (from 0 to 10) and we would see that any cases the code would fail would be due to newton's solver breaking and the simulation would not run, as opposed to computing and resulting in an incorrect root. Therefore, we made the reasonable assumption

that our multi-newtons-solver would work for any acceptable range, and for usage in the Strandbeest.

The test we performed was changing the range of t to see where the roots are the same, and if there is a successful connection. We noticed that when theta is larger, pi/3, and t is in the range of 2.6 to 3.9, the connection is true while anything outside that range would fail. By changing the theta to make it smaller, pi/6, the t range changed to have a wider range.

3 Simulating the Strandbeest

Having a working multidimensional Newton's solver our next step was to initiate our constraints for the strandbeest linkage. The provided constraints defined the lengths of each leg, at which vertices each leg was connected, the length and axis of rotation of the crankshaft, and one fixed point at (0,0). With the legs fully constrained the next step was to find a set of points for the vertices that could exist inside the assigned constraints.

The leg link constraints were addressed first. The goal was to be able to solve for a set of vertex points that would define the strandbeest such that the lengths would remain constant at every value theta of the rotation of the crank. To solve this we used the equation $(xb - xa)^2 + (yb - ya)^2 = d^2$ where d was the given constraining length of the leg. By setting this equation equal to zero it created a function for which the roots were values of x and y that satisfied the leg length constraints. With four unknowns in this function, we needed to define our function further to solve for a single set of vertices.

To do this we implemented the fixed point constraints and the crankshaft constraints. The fixed point constraint was simply a matter of setting the position of the vertice number to always be (0,0). For setting the value of the vertice on the end crankshaft we used trigonometry to define the x and y position as a function of theta, the leg of the crank arm, and the given fixed position of the axis of rotation. With these known positions we could again create functions where the roots of the function would be the values of x and y we were looking for by checking that the difference between the known values and the calculated values were zero.

By combining this defined system of equations into a single function we could then use our multidimensional Newton's solver to solve for the roots of this function which because of the way we set it up were the vertex positions such that all the constraints were satisfied and we had a real set of vertex points. This could be calculated for any value of theta allowing us to find a real set of vertex positions at any point of the linkages movement. By calculating and plotting the vertex positions over a full rotation of the value theta we were able to animate the full movement of the strandbeest linkage.

Furthermore, we also wanted to calculate the velocity at the tip of the linkage. Again we make use of the fact that we set up our constraining function so that the roots of the equations are the values of x and y that create a real set of linkages and set the function equal to zero. With this function, we can

derive two things. Firstly, and more simply, because the leg length constraining function is equal to zero when given a real set of vertex positions the derivative of the function for a real set of vertices is also zero. Secondly, when symbolically taking the derivative of our leg length constraint function we see that we are taking the partial derivative of each index of the function with respect to the and x and y values for each vertex point multiplied by the derivative of the same values of x and y with respect to theta. This matrix can be represented by two separate processes. The derivative of position with respect to theta is velocity, and taking the partial of the index of a function with respect to each of the input positions is the Jacobian of the function. This means we can represent the derivative of our leg length constraint function as being equal to the Jacobian of the function times the velocity of the vertices. With a real set of vertex positions, this would mean that the Jacobian times the velocity of the vertices would be equal to zero because the derivative of the function is zero for real vertices. However, this would only solve for the velocities of 5 of the vertices so to solve for the velocity of all the vertices we need to also consider the fixed point and crank.

Similarly to finding the real positions of the vertices we consider the fixed point and crank separately. In this case, the derivative of the function does not equal zero when solved numerically and is not represented by the Jacobian multiplied by the velocity when solved symbolically. However, similarly to before because the fixed point and tip of the crank have constant movements the positions and derivatives are much easier to calculate with the derivative of the fixed point simply being zero, and the tip of the crank being the derivative of our earlier trig functions. By creating a 4x4 identity matrix followed by enough zeros to match the length of the Jacobian and spending both this matrix and the derivatives of the fixed point and crank tip to the Jacobian matrix and column vector representing the derivative or the linkage error function respectively we get the equation that the identity matrix appended to the Jacobian times the velocity of each of the vertices is equal to the derivative of the constraining function. By then using matrix division to divide the derivative of the constraining function by the identity matrix and Jacobian matrix we can compute the velocity.

4 What We Would Do Differently

If we were to start over we think it would be beneficial to find more ways to test the individual parts of the project. When developing the strand beast constraining functions and the Jacobian we didn't have a way to test them individually which meant when debugging we had to look through multiple functions and often the function erroring out wasn't the function causing the issues. Because of this, we think it would have been beneficial to create a more simplified system or some more accurate guess data to test each function individually.

5 What We Learned

A learning point or more so a realization point for me was the way we set up equations to be solved using root-finding algorithms. I imagine this is commonplace for simulation problems like the strandbeest problem and is likely even something that has come up before in more simple calculations that I have just not noticed until its implementation in this project. Again, there's not much in the way of new material here, but more so a new way of framing things that I hadn't fully thought through. Outside of this specific use case, I think I'm most likely to apply this method to solving systems of equations in opposition to substitution or use of the symbolic toolbox.

A place where I was introduced to a new concept in this project was with the Jacobian, especially when solving for the velocity. While it took some time to understand why we were able to solve for the velocity using the Jacobian, walking through the steps for this write-up helped me understand how solving using the Jacobian worked, and what kind of systems can be solved this way. From my understanding, solving for the velocity using the Jacobian applies to any function in which the length of the linkages remains constant. With this understanding, I realized just how applicable this method was as this applies to the vast majority of linkages such as mechanical/robotic arms, suspension systems, and so on. With more complex systems this would no doubt become more complicated to apply but even then I imagine it would still be doable and even if not its usefulness with simplified systems is still substantial.

One final learning was how we define the strandbeest's position. At first it felt really intuitive to model the strandbeest as a set of coordinates instead of using a struct that includes the position of each vertices, which linkages are connected to what, etc. However, taking our time through the strandbeest and seeing how effective it is to have a struct - how we can later find velocity, check for error lengths, and how it builds off the root-finding method we built earlier in class - helped me rethink my definition of intuitive into a more sophisticated one.

6 Plots

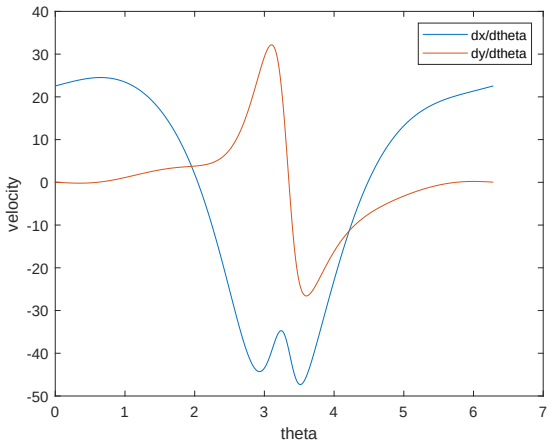

Figure 1: A graph representing the velocities in the x and y directions as a function of theta.