Jay Lanzafane

ME 570

Robot Motion Planning

Homework #2

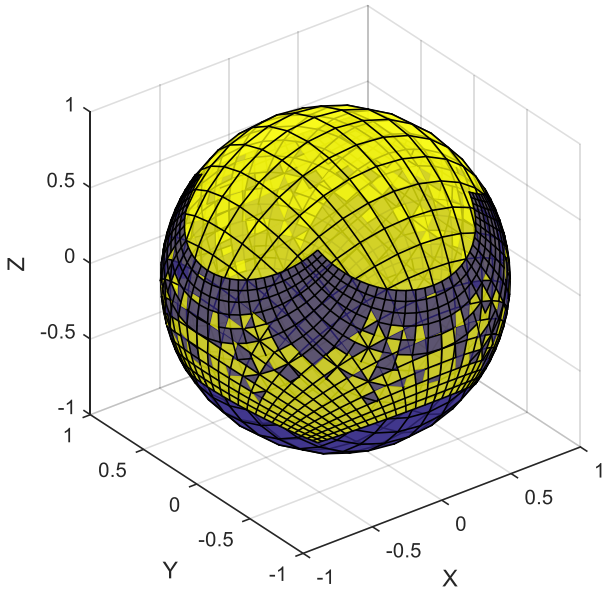
Problem 1:

1.1) This is a straightforward application of the provided mappings with the slight caveat that they are applied entry-wise. This is achieved by using the . (dot) operator before multiplication/division/etc. in order to indicate that it should be done entry-wise and not via matrix multiplication, for example.

1.2) A simple function that runs our mapping functions through a few quick tests. Note that the error checking expects the answers to be correct and checks for the variation to be under some allowable threshold epsilon. This is not a robust error checking method, but having the expected values print to screen makes it a little more obvious. I added this functionality in after already verifying it, so I didn't think it was especially important to check both the values and the epsilons, if they exist (it appears they do not).

1.3) Comments included in code. This quickly implements our mapping functions and then plots the results using the surf command.

1.4) The result is shown below. The (x,y) pairs are projected onto the sphere as expected, each covering a hemisphere and then some. We know that the phiS function can never achieve the mapping onto the south pole, because it would have the same coordinates as the north pole in the (x,y) plane. Similarly, the phiN function can never achieve the mapping onto the south pole. This is why we require 2 different mappings for the sphere, one can not encompass all possible locations on the sphere. The overlap is useful for making our paths smooth.



1.5) We can compute the Jacobian of phiNinv, which is done below. Here I used the symbolic toolbox from MATLAB to quickly derive the Jacobian, which results as:

$$J_{\phi_N^{-1}}(XN, YN)$$

$$= \begin{bmatrix} \frac{2}{(XN^2+YN^2+1)} - \frac{4*XN^2}{(XN^2+YN^2+1)^2} & \frac{-4*XN*YN}{(XN^2+YN^2+1)^2} \\ \frac{-4*XN*YN}{(XN^2+YN^2+1)^2} & \frac{2}{(XN^2+YN^2+1)} - \frac{4*YN^2}{(XN^2+YN^2+1)^2} \\ \frac{2*XN*(XN^2+YN^2-1)}{(XN^2+YN^2+1)^2} - \frac{2*XN}{(XN^2+YN^2+1)^2} & \frac{2*YN*(XN^2+YN^2-1)}{(XN^2+YN^2+1)^2} - \frac{2*YN}{(XN^2+YN^2+1)^2} \end{bmatrix}$$

1.6) JphiNinv is a little tedious to code up, but relatively simple. We just implement the above computation for the Jacobian and store the values in the appropriate places. We can show JphiNinv has rank 2 by evaluating JphiNinv(0,0) as we get

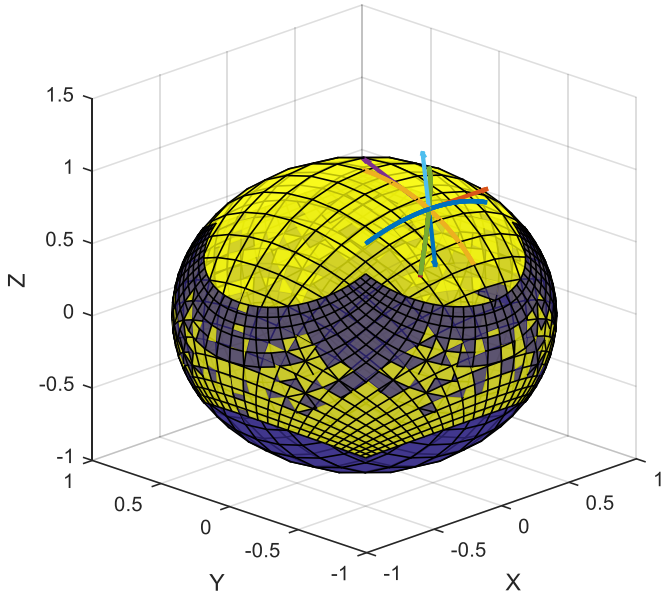
ans =

2 0
0 2
0 0

Which has full-row rank.

1.7) phiPushCurveForward requires a few major steps to work properly. We begin by first calculating the curve c(t) = (ax*t+bx, ay*t+by) with our given parameters ax, bx, ay, and by. Recall this lies within the z=0 plane. We can take the derivative of this quite easily to get dc/dx = ax and dc/dy = ay, which we will use later on. Next we compute the values of gamma, which lives in R3, which we attain using our phiNinv function on our path c(t). Finally, we want to compute gammadot at t = 0. We evaluate our Jacobian using JphiNinv(XN, YN), where c(0) = (XN, YN). Lastly, we multiply this value, the Jacobian evaluated at that point, by the derivative of c(t). This final result will give us the tangent to our projected curve. We can then plot these results.

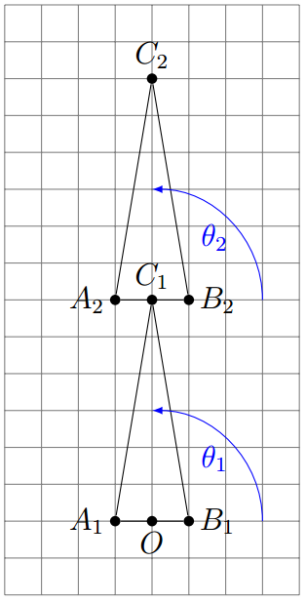
1.8) We combine our efforts from previous problems to get our culminating figure:



We can see here that our 4 sets of parameters for c(t) map to 4 distinct curves (note that 2 of them overlap, but are travelling in opposite directions, which we can see by the tangents). Each of these curves has its' tangent plotted as well. As expected, it goes in the direction the curve is propagating in but in a plane that is tangent to the sphere at the point c(t = 0). All of the paths cross at c(0), so all of the tangents should be within the same plane. By inspection, this is in fact the case, as we would hope. Because 2 of the curves have the same points, but propagate in opposite directions, we have 2 tangent vectors which originate in the same location but point completely opposite one another, as expected.

Problem 2:

2.1) This function is performed by using two solid-body rotations and a single translation. We start the first arm at theta = 0 and rotate it to the desired theta1. Next, we again start at the origin, and rotate the second arm to the desired theta2. We perform this rotation before the translation because all rotations using our R matrix will rotate about the origin. If the attachment point is not at the origin, it too will be rotated about the origin (which would not make physical sense, as it would detach from the first arm). After the second arm has been rotated to the angle theta2, we then translate it to the end of arm1, which is just point C1.



2.2) Next we need to compute the Jacobian of $X_{C2}(\theta_1, \theta_2)$. Here we call $\theta = (\theta_1;\ \theta_2)$

$$\emptyset(q) = \begin{bmatrix} \emptyset_1(\theta) \\ \emptyset_2(\theta) \end{bmatrix} = \begin{bmatrix} L1 * \cos(\theta_1) + L2 * \cos(\theta_2) \\ L1 * \sin(\theta_1) + L2 * \sin(\theta_2) \end{bmatrix}$$

$$J(\theta_1, \theta_2) = \begin{bmatrix} \frac{\partial \emptyset_1}{\partial \theta_1} & \frac{\partial \emptyset_1}{\partial \theta_2} \\ \frac{\partial \emptyset_2}{\partial \theta_1} & \frac{\partial \emptyset_2}{\partial \theta_2} \end{bmatrix} = \begin{bmatrix} -L1 * \sin(\theta_1) & -L2 * \sin(\theta_2) \\ L1 * \cos(\theta_1) & L2 * \cos(\theta_2) \end{bmatrix}$$

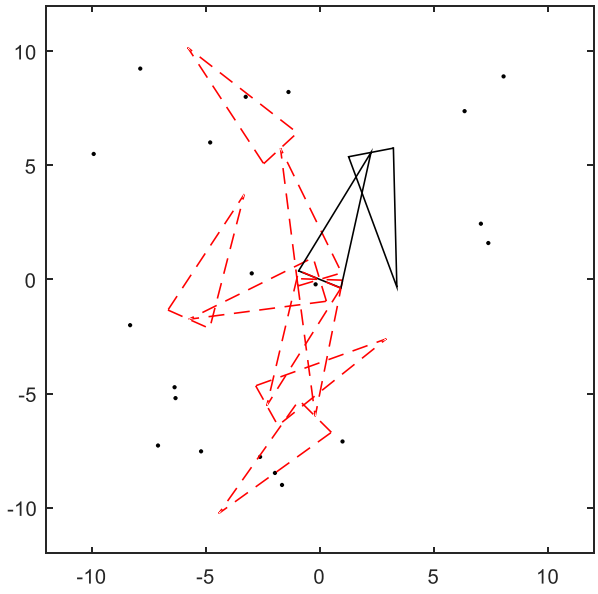
As such, we could now compute the velocity of the end effector $\dot{X}_{C_2}$ by:

$$\dot{X}_{C_2} = J(\theta)\dot{\theta}$$

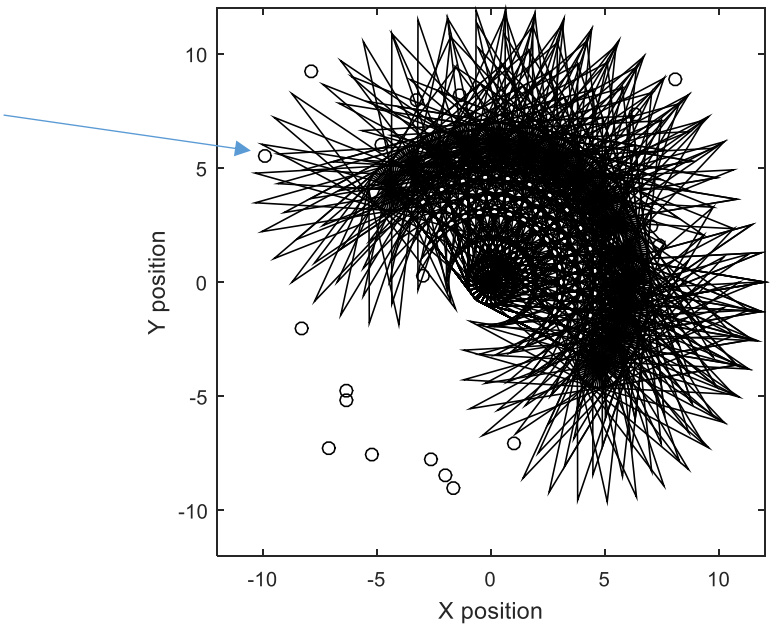
2.3) twoLinkDraw leverages the function defined above, twolinkKinematicMap, as well as triangleDraw from the previous assignment, to draw the position of the manipulator. triangleDraw has been modified to also accept a color parameter which will determine the color the manipulator is drawn in. We call triangleDraw twice, once for each manipulator arm, passing it the points A, B, and C.

2.4) For twolinkCheckCollision we can again leverage code from the last homework, specifically the triangleCheckCollision function. Here we follow the same first step of twolinkDraw where we compute the vertices of the 2 arms using the twolinkKinematicMap function. Pass the first set of A, B, and C (for arm1) into our triangleCheckCollision function, and then repeat for the second set of A, B, and C (for arm2). If either of these had a collision, we flag the configuration as a collision and return flag = 1. Otherwise, flag = 0.

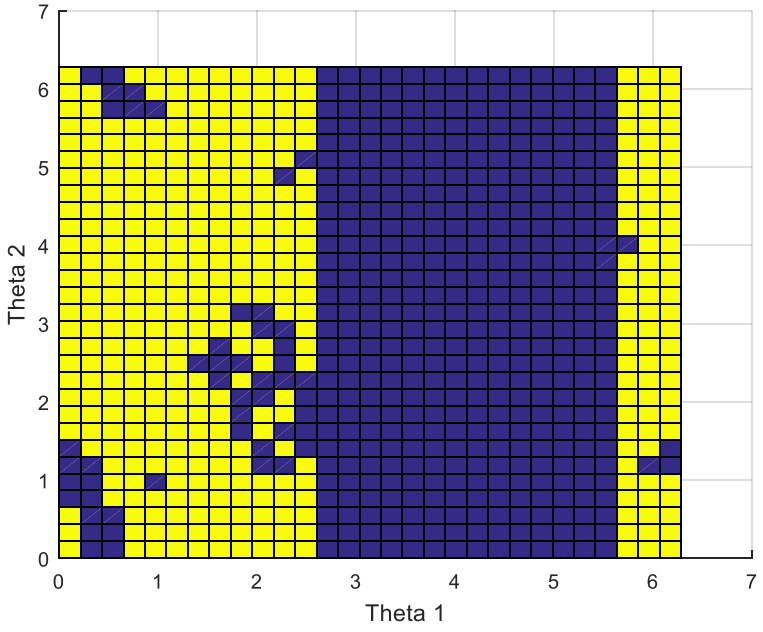
2.5) We begin by writing a quick random point generator twolink_testData that saves the generated points to file. For twolink_test we begin by loading in the random obstacles, or points. We then generate 5 random configurations for the manipulator and check for collisions with the points. Finally we plot all the points along with our 5 generated configurations. The configuration in drawn in red if it collides with an obstacle, and in black if it is a permissible configuration with no collisions. Note there is a point (obstacle) almost directly below the origin that causes multiple configurations to be rejected.



2.6) Here we are basically just using a meshgrid function to define the configuration space we are interested in and then just keep track of which positions are flagged as a 1 (collision) and mark those as invalid configurations. We can draw all possible configurations, as seen below:



Note that again, there is an obstacle almost directly below the origin that invalidates a huge number of configurations. This massively cuts our number of usable configurations, as can be seen below (yellow is permissible states, purple is invalid). This is really cool! IF we didn't need to worry about the granularity being a problem, we could use this map to start doing path planning. In this particular example, we would never be able to rotate around 360 degree, as theta1 has a gap in it from around [2.6 5.6] radians. We WOULD need to worry about the granularity in this case, however. One example can be seen at the highlighted point in the above image. That obstacle would clearly cause interference between the two states, and is why granularity would matter for path planning.



We can also do a 3d plot, just for fun (it's not any more enlightening, if anything it's harder to read accurately.