Jay Lanzafane

RMP HW #3

Problem 1.)

1.1) This function just uses a quick conversion from polar coordinates to (x,y) in order to quickly trace out the outer edge of each sphere. Note that the function works for a single sphere or n spheres.

1.2) These are 2 quick functions. The bulk of the work was done in sphere_plot and, for sphereworld_plot, we simply call sphere_plot once for the obstacles and once for the sphere of influence around each of those obstacles. After that we mark the goal location with a simple plot command. Sphereworld_plot_test is even simpler, we load the provided sphereworld.mat file and then call sphereworld_plot with the (now loaded) variables world and xGoal.

Problem 2.)

2.1) sphere_potential and sphere_potentialGrad use almost exactly the same structure but the function to evaluate the potential at each point is slightly different. I will briefly walk through the structure of the code, though it is commented in-line as well. In both cases, we need to sum the effects of each obstacle at the position xEval. To do this, we iterate through each obstacle individually and then sum the result to the total value, which starts at zero. For each iteration, we first need to determine if xEval is inside the obstacle (or outside, in the case of the world's boundary). If we are, then we set the potential equal to NaN and immediately return (invalid location). If not, then we need to proceed and determine if we're inside that obstacle's zone of influence. If we are, then we compute U or gradU and sum that to the totalvalue (which we set to zero initially in both cases). The functions for U and gradU were provided and are relatively straightforward to compute. After we have totaled all the individual contributions to the total field, we then return that value.

2.2) attractive_potential is a quick function to determine the attractive potential of a field of type potential.type at a position xEval. The formula is given in the homework and is fast to compute, we simply need to determine whether we are using p = 1 (conic) or p = 2 (quadratic) by checking potential.type. potential_attractive is a similarly quick function, but in this case we need to compute the gradient. This formula was not given, but is a simple calculation away. We can easily show that, in the case p = 1, the gradient of $U_{attr} = d^p\left(x_{eval}, x_{goal}\right)$ becomes, where x,y are the coordinates of X

$$\nabla U_{attr} = \frac{x_{eval} - x_{goal}}{d(X_{eval}, X_{goal})}, \frac{y_{eval} - y_{goal}}{d(X_{eval}, X_{goal})}$$

Meanwhile, when p = 2, the gradient becomes

$$\nabla U_{attr} = 2 * \left(x_{eval} - x_{goal}, y_{eval} - y_{goal}\right)$$

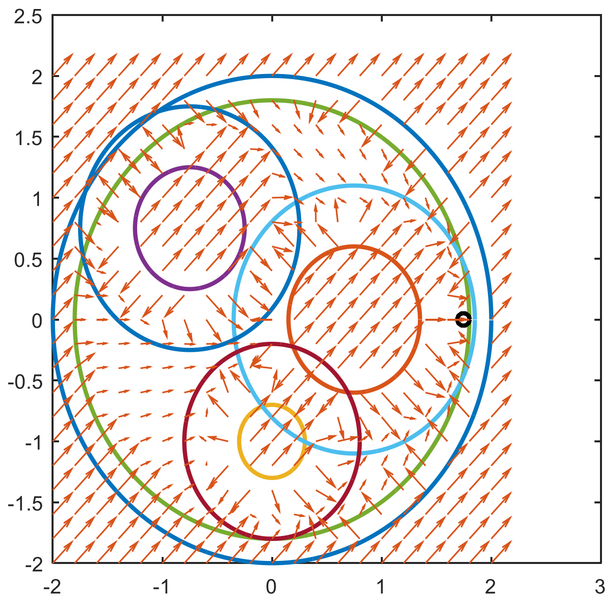
Which is, of course, easy to code into matlab.

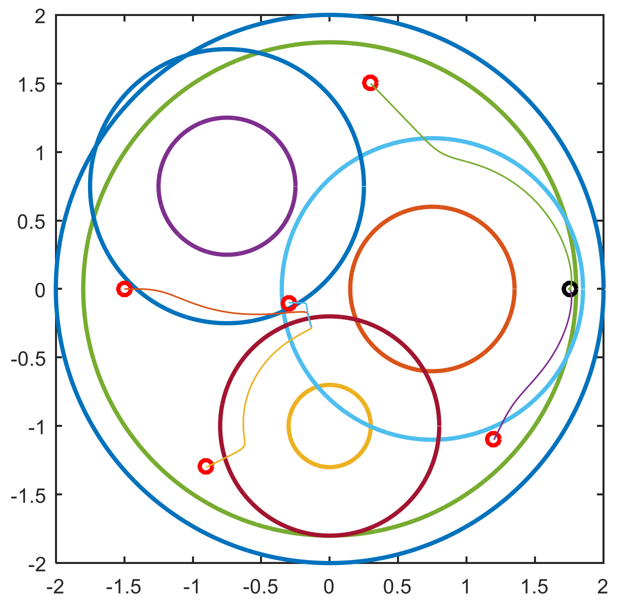
2.3) The functions potential_total and potential_totalGrad are extremely quick functions which call their respective generating functions for both the attractive and repulsive potentials at a position xEval and returns the sum of the attractive and repulsive potentials/gradients.

2.4) The potential planner is just a gradient descent algorithm which loops through, up to 1000 times, and evaluates the gradient at the current location and then moves in the (negative) direction of the gradient multiplied by a step size epsilon.

2.5) Test function for the potential planner procedure. Loads the file sphereworld, plots the world with sphereworld_plot, prepares a structure potential, plots the (-) gradient of sphereworld, and calls the potential_planner function 5 times with the initial start positions xStart. The trajectories are then overlaid.

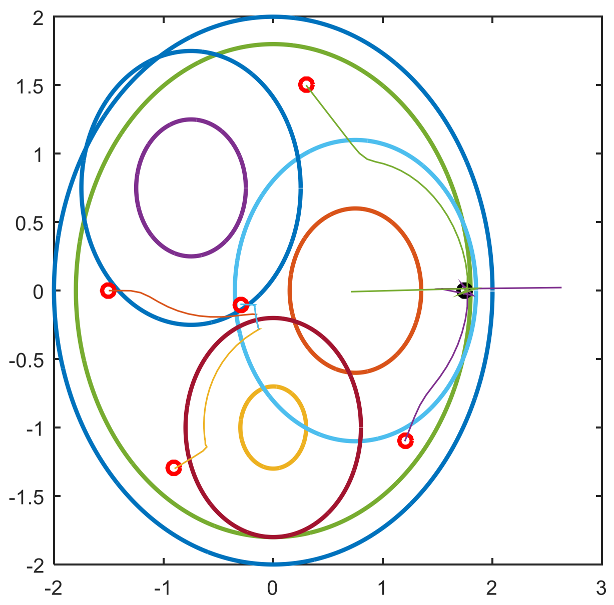
2.6) Using potential_planner_test we can quickly begin to understand some of the properties and pitfalls of using a gradient descent based solution. We start with alpha = 10, epsilon = 1e-3, and an initial potential type of conic. Here we see an example of the (-) gradient, which comes out as one would expect (pushing away from the obstacles within their sphere of influence, and generally pulling towards the goal location). When we plot the trajectories from the potential planner we get the somewhat expected results seen below:

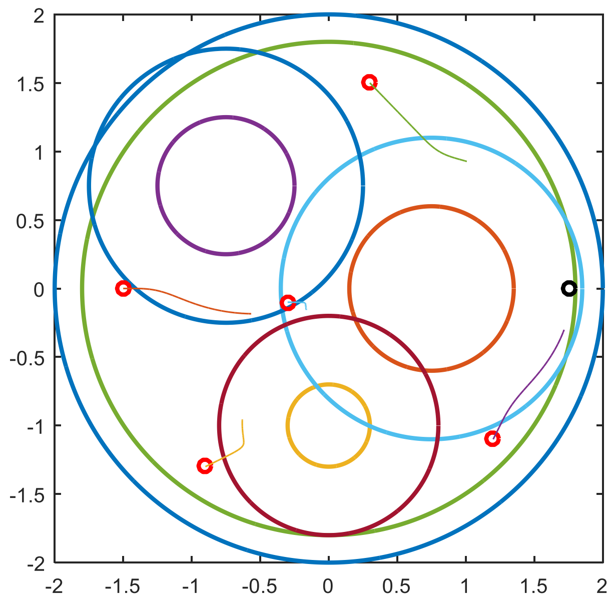




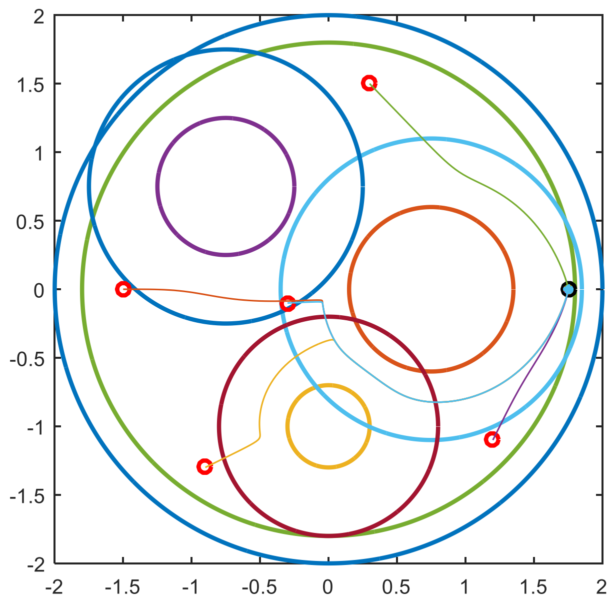
As we can see here, the 2 trajectories that approach the goal location don't end up converging, as they end up bouncing around the goal and running out of steps. The 3 other trajectories all find a local minima on the other side of the obstacle at the 3 o'clock position, which is obviously not the desired goal location. To get better results, we need to tune our parameters in order to reduce the number of local minima (ideally to 0, if possible) and thus get our paths to converge to the goal location.

The first parameter we will demonstrate changes with is the stepsize epsilon. If we were to run the exact same setup, but multiply the stepsize by a factor of 10 we see the result at right (epsilon = 10e-3). The step size has gotten too large, and instead of converging on the goal point, one of the trajectories becomes unstable and is bounced back and forth between the repulsive fields of the boundary and an obstacle. The size of these oscillations increases (because the step size is too large) and it then gets ejected from the workspace. Contrary to this we could reduce the stepsize to a value which is too small. In this case we never make enough progress within the given number of steps, which can be seen in the next result. Here we used an epsilon of 1e-4.



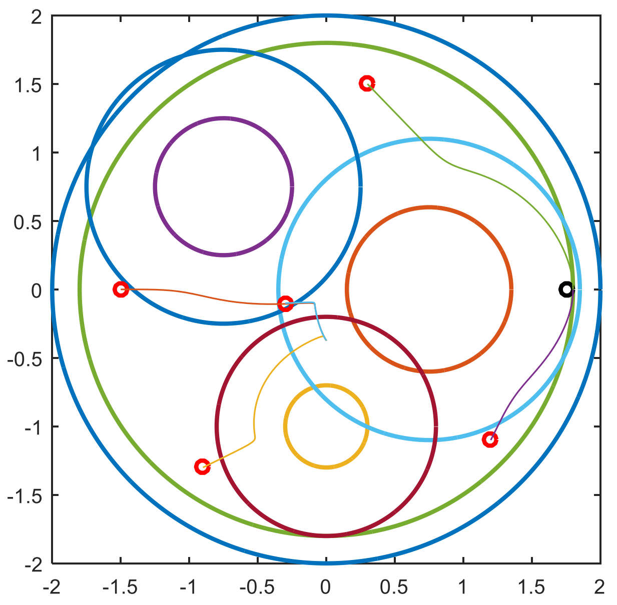


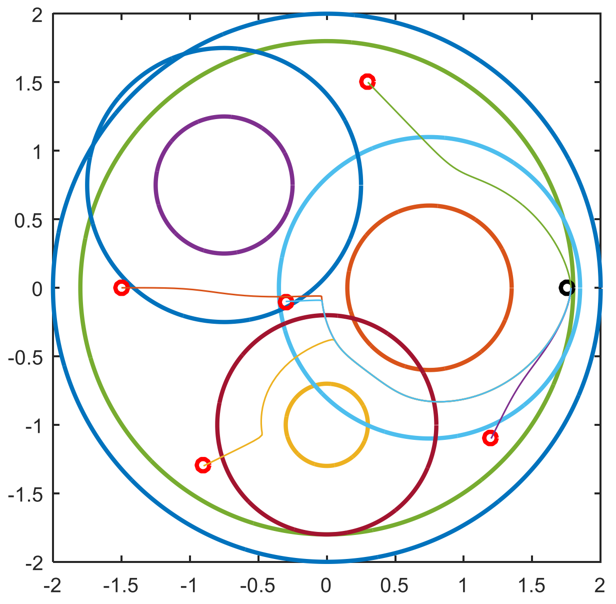
If we begin changing the value of alpha we can see that we increase the effect of the attractive potential. In an extreme case we can look at alpha = 75. We need to reduce the step size epsilon in this case, otherwise we will incite the same problem we had previously and some trajectories will escape the boundary, but if we adjust to epsilon = .5e-3 and set alpha = 75 we get:



This is significantly better. We see all 5 trajectories approach and almost settle on the goal location (though in this particular example they did not converge). By increasing the parameter alpha we can lessen the relative effect of the repulsive field around the obstacles (these trajectories get closer to the objects) as well as change/eliminate the number of local minima. If we were to go in the opposite direction and reduce alpha, we would see that the repulsive fields dominate and in many cases the goal will just never be reached, because it does lie inside of a repulsive field.

Finally we will examine a case where we change to a quadratic potential. This produces similar results to the first conic potential but with some slight variations. The 2 trajectories that approach the goal are quite similar, but upon closer inspection exhibit different behavior than in the conic case. The attractive potential is somewhat weaker in the vicinity of the goal and causes the trajectories to settle in a place which is not quite the real target goal. This problem persists even with higher values of alpha (with a lower step size here as well, to avoid the unstable back-and-forth that caused a few trajectories to escape). Here we turned the alpha value up to 25 and cut the step size in half to .5e-3. Still, however, when the trajectories get close to the goal the attractive potential is not strong enough to pull them through the repulsive field and as such they settle at a point very close to the goal, but not quite on it.





We can conclude that for each example, we would need to do quite a bit of tuning in order to get the behavior we want. This makes potential planner somewhat difficult to work with, as depending on the selected parameters they may or may not converge, and often times they are pulled to local minima if the potential fields are not well-defined.

3.1) To detect a collision in sphereworld_collision, we simply need to check if the distance from a point xEval to the center point of an obstacle xSphereAll(:,n) is less than the corresponding radius of that obstacle rSphereAll(n). If this is the case, then we know it is inside the obstacle. The one exception to this is, as before, the boundary. In the case that rInfluenceAll(n) is less than rSphereAll(n), then we know it is a boundary and we need to check if the distance is greater than the radius to determine if it is outside the boundary.

3.2) To make the grid we simply use the linspace function. We know the bounds will always be -2 to 2 and we can efficiently make the two vectors xx and yy with this functions. Not that these are the vectors we would use for calling a function meshgrid.

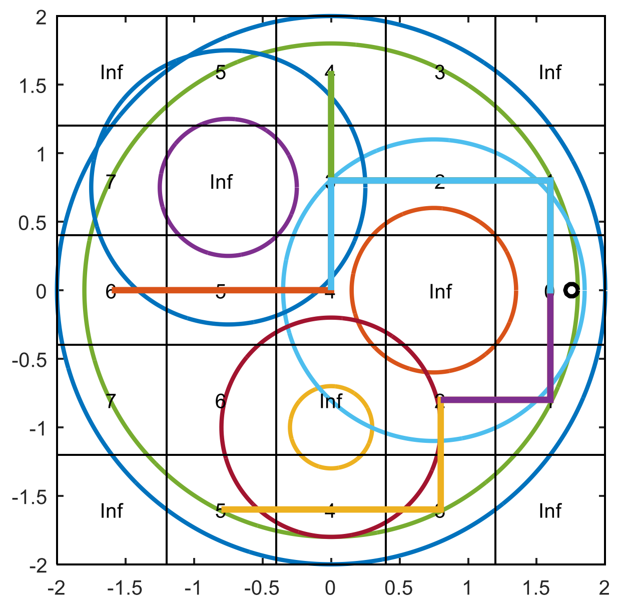
3.3) In order to initialize the grid with wavefront_initialize, we first start by initializing a grid of NaNs. From here, we need to set the goal location to zero and the cells inside of obstacles to inf. These are all relatively simple calculations to perform, but does require us to perform them for each cell.

3.4) The algorithm for wavefront_propagate depends on a relatively simple recursive function. This function does one rather simple thing: it sets the value of the cell it was called on to a provided distance dist, and then, where applicable, calls itself on all of that cell's neighbors with a value of dist+1. This will get the desired effect, as the values will increase by one every step away from the goal location. The only tricky bit is determining the "where applicable" condition. For each direction (up, down, left, right), we first check to see if the neighboring cell would be inside our workspace. If it is, then we also need to check to make sure it is not an obstacle and not equal to infinity. Then, once we know it is viable, we need to check to make sure it is either NaN or has a distance which is larger than the current distance + 1. If it meets all of these criteria, we will then propagate to that node, otherwise we do not. This keeps us from entering an infinite loop. It is important to check the distance because the distance depends on the path. As such, we always want to make sure we get the shortest path and thus need to overwrite poor solutions (paths) with better ones.

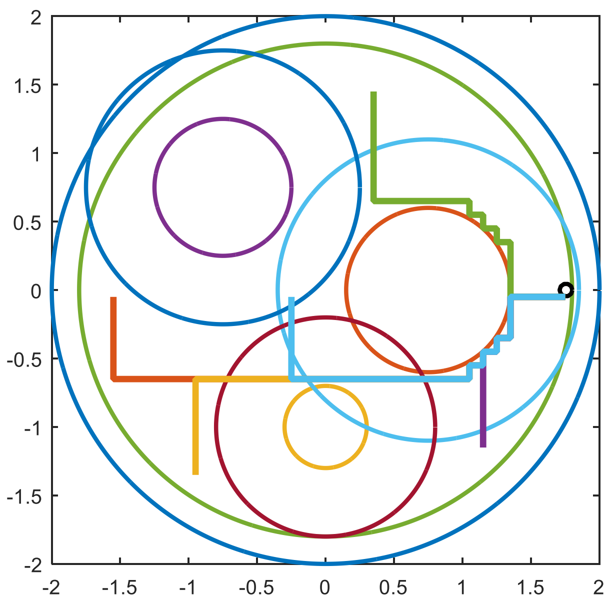
3.5) The wavefront planner starts at a given node (xStart) within val, determines which of its neighbors has a lower value (distance to goal), and then moves in that direction. Note that, because the wavefront propagator was implemented to always compute the shortest path, any time the planner finds a direction with a value lower than its current position it has found (one of) the shortest paths. As such we only need to check new directions until we find a lower value than the present value, and then move in that direction.

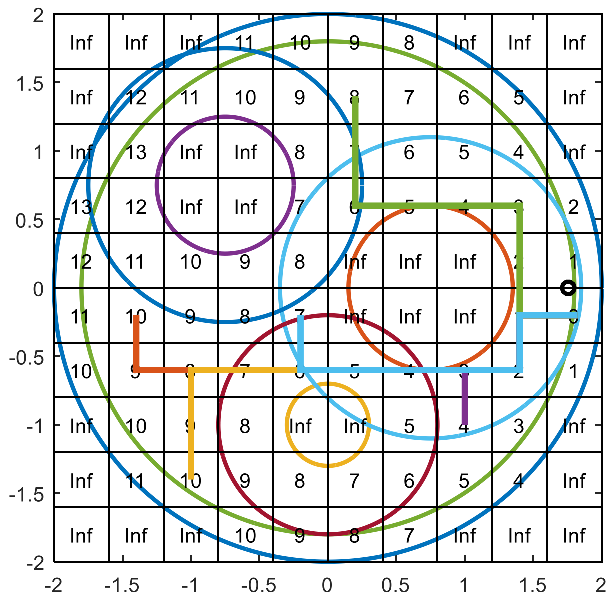
3.6) To test our wavefront code, we create the function wavefront_planner_test. This code utilizes all our previous wavefront code and only requires a scaler GridSize as an input. It then loads the sphereworld dataset and plots it. We then make and initialize the wavefront array val and propagate all of its values. We call the provided function to label the distances on the plot and then run the planner for each of the 5 staring locations in xStart.

3.7) Compared to the potential planner, there aren't nearly as many parameters to tune. In this case, we only have the the gridsize to play with. In general, making the grid too coarse will give us inaccurate representations of our obstacles (seen in the case where N = 4, at right) and could lead to collisions or a lack of a path when in fact one is actually present. Note the example at right portrays our circular obstacles as squares. Making the grid too fine will result in excessive computation and take a significant amount of time. Note that the next example took about 45 seconds to generate with n = 40, however increasing the grid size takes a large toll on the processing time. My trials put a 100x100 grid at about 3 minutes to perform the propagation. Also,



because of the finer granularity, the paths can become quite jagged and potentially impractical to implement. We see this in the case where gridsize = 40, especially around the obstacle at the 3 o'clock position. The paths are jagged and take the "one over, one down" approach, as seen at right. This is not ideal and not smooth, and in practice would require a robot to make many hard turns. Of course there are ways around this, for example we could prioritize always checking the direction we're currently moving in first, and only once that direction has been disqualified would we begin checking the other directions. With a smart planner, or with one assisted by the below suggestion, the major drawback to fine grids is simply their increased and unnecessary processing time, while coarse grids misrepresent our workspace and can lead to paths that don't accurately relate back to the physical world, which could cause collisions or impractical paths. Ideally one would pick a gridsize that is just fine enough to accurately represent the problem while being coarse enough to be quick to compute. Below is an example of N = 10 where we run a heavy risk of colliding with obstacles because the grid spacing is not too coarse (and somewhat unlucky).





3.8) One potential way to speed up the processing for the planner is to store the position of the node that got us there during the propagation function, in, say, a field called origin. In this way we don't need to make any checks for out of bounds or to check the values in the neighboring nodes. This would increase the amount of storage required (though not significantly) but would effectively do the planner's job for it, as it would only need to construct a chain of nodes by querying the current node's origin value, moving to that node, and repeating until it arrived at the goal.