Homework 3 (Matlab version)

ME570 - Prof. Tron

2023-10-14

Problem 1: Drawing and collision checking for spheres

In this problem you will write functions that are similar to those in Problem 1 of Homework 1, but applied to 2-D spheres (i.e., circles).

Data structure. We represent a 2-D sphere with a structure `sphere` with three fields:

- `sphere.xCenter`, a $[2 \times 1]$ array containing the 2-D coordinate of the center of the sphere;
- `sphere.radius`, a scalar whose absolute value is equal to the geometric radius of the sphere, and the sign indicates whether the obstacle should be interpreted as filled-in (`radius` > 0) or hollow (`radius` < 0);
- `sphere.distInfluence`, a scalar containing the *influence* distance of the sphere (the exact meaning of this will become clear after we talk about potential-based planning).

Figure 1 demonstrates the meaning of these quantities.

Question provided 1.1. Implement a function that draws a sphere.

> `sphere_plot(sphere,color)`
> Description: This function draws the sphere (i.e., a circle) of the given radius, and the specified color, and then draws another circle in gray at the influence radius.
> Input arguments
> - `sphere` (dim. $[1 \times 1]$, type `struct`): a structure, as described above, defining a sphere.
> - `color` (dim. $[1 \times 1]$, type `string`): a color specification string (e.g., `'b'`, `'r'`, etc.)

Question optional 1.1. Vectorize the function `sphere_plot(,)`, so that if `sphere` is an array of structs, it plots multiple spheres.

Question code 1.1. Implement the following function.

> `[dPointsSphere]=sphere_distance(sphere,points)`
> Description: Computes the signed distance between points and the i-th sphere, while taking into account whether the sphere is hollow or filled in.
> Input arguments

- `sphere` (dim. [1 × 1], type `struct`): a structure, as described above, defining a sphere.
- `points` (dim. [2 × `NPoints`]): an array of 2-D points.

Output arguments

- `dPointsSphere` (dim. [1 × `NPoints`]): the distance between each point an the surface of the sphere. The distance is negative if `points` is inside the obstacle (i.e., inside the sphere for filled-in obstacles, and outside for hollow obstacles), and positive otherwise.

Requirements: Remember that the radius of the sphere is negative for hollow spheres.

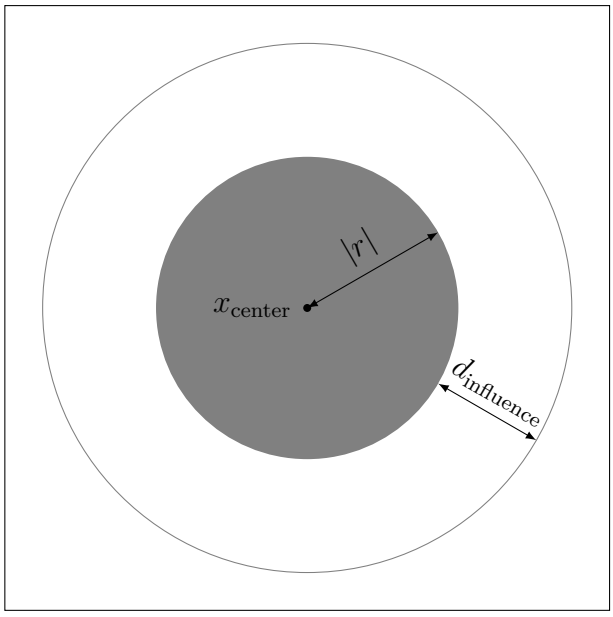
Question code 1.2. Implement a function to compute the gradient of the sphere.

[`gradDPointsSphere`]=`sphere_distanceGrad`(`sphere,points`)
Description: Computes the gradient of the signed distance between points and the sphere, consistently with the definition of `sphere_distance`(_).
Input arguments

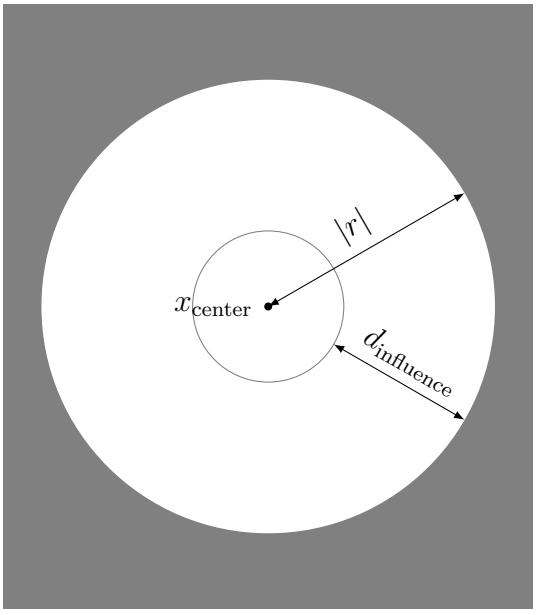
- `sphere` (dim. [1 × 1], type `struct`): a structure, as described above, defining a sphere.
- `points` (dim. [2 × `NPoints`]): an array of 2-D points.

Output arguments

- `gradDPointsSphere` (dim. [2 × `NPoints`]): the gradient of the distance. If a point corresponds to the center of the sphere, return the zero vector.



(a) Filled sphere (`radius` > 0)



(b) Hollow sphere (`radius` < 0)

Figure 1: Examples of spheres.

Question optional 1.2. Implement a function that shows collision checks with a sphere.

> `sphere_testCollision(_)`
> Description: Generates one figure with a sphere (with arbitrary parameters) and `NPoints=100` random points that are colored according to the sign of their distance from the sphere (red for negative, green for positive). Generates a second figure in the same way (and the same set of points) but flipping the sign of the radius `r` of the sphere.

Red and green points represent locations that are, respectively, in collision and not in collision with the spherical obstacle.

The world

The goal of this question is to prepare and visualize the *sphere world* workspace that will be used in the other problems. The workspace is described by the data stored in the file `sphereworld.mat` provided with this homework.

The meaning of the variables is the following:

- `world`, a [`NSpheres` × 1] vector of `sphere` structs (as defined in Problem 1) defining all the spherical obstacles in this sphere world.
- `xStart`, a [2 × `NStart`] vector of initial starting locations (one for each column).
- `xGoal`, a [2 × `NGoal`] vector containing the coordinates of different goal locations (one for each column).

Notice that `radius` < 0 for the first sphere in `world` (this obstacle defines the external boundary of the sphere world), while `radius` > 0 for the others.

Question provided 1.2. Implement a function to visualize a sphere world

> `sphereworld_plot(world,xGoal)`
> Description: Uses `sphere_plot(_)` to draw the spherical obstacles together with a `*` marker at the goal location.
> Input arguments
> - `world` (dim. [`NSpheres` × 1], type `struct array`), `xGoal` (dim. [2 × 1]): same as defined for the `sphereworld.mat` file.

Use this function to visualize the contents of the file.

Question provided 1.3. The following is a utility function.

> `grid_plotThreshold(fHandle,threshold,grid)`
> Description: The function evaluates the function handle `fHandle` on points placed on a regular grid.
> Input arguments
> - `fHandle` (type `function handle`): an handle to a function that takes a vector `x` of dimension [2 × 1] as a single argument, and returns a scalar or a vector. An

example of this function could be the function `norm(_)`.

- `threshold` **optional**, default `10`: if a vector `fEval`, obtained by evaluating `fHandle`, has `norm(fEval)>threshold`, then it is replaced by `fEval/norm(fEval)*threshold`. If omitted, `threshold` = 10.
- `grid` (type `struct`) **optional**: a structure with fields `xx`, `yy`, `F` as used in Homework 2. If omitted, the default is a regular grid from -10 to 10 for both x and y coordinates with `NGrid` = 61 points.

Requirements: The function is build upon the function `grid_eval(_)` from Homework 2.

Note that you can look at the source code of this function for a way to handle optional arguments.

Note on the use with functions that take extra parameters You can use `grid_plotThreshold(_)` also with functions that require additional parameters (e.g., `potential_attractive(_)`). This can be done by creating an *anonymous function* where the additional parameters are fixed. The best way to understand this is with an example:

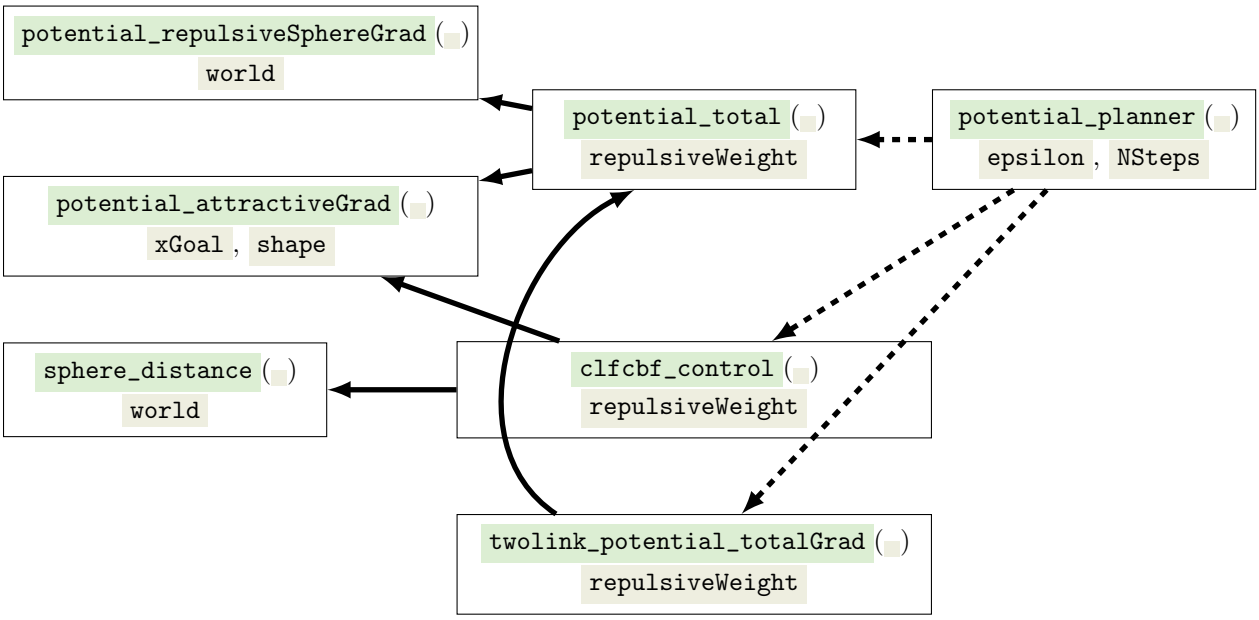
```
potential=struct('shape','conic','xGoal',[0;0],'repulsiveWeight',10);
fHandle=@(xInput) potential_attractive(xInput,potential);
grid_plotThreshold(fHandle,10)
```

You need to type the example, copying from this PDF document will not work.

The first line of the example sets a structure `potential`. The second line creates `fHandle` as an handle to an anonymous function; the construct `@(xInput)` says that what follows is a function with `xInput` as argument; since `potential` is not an argument, it is "captured" by the anonymous functions. What this means is that Matlab will see `fHandle` as a function that takes a single argument (i.e., you can try to call `fHandle([1;2])`), and the value of `potential` will be remembered from when the anonymous function was created (in fact, if you change `potential` but do not redefine `fHandle`, the old value of the struct will be remembered). As such, `fHandle` can be successfully passed to `grid_plotThreshold(_)` for plotting. Please search the Matlab help for *anonymous functions* for detailed information.

Organization of the reminder of the assignment

Below you can find below a diagram of the most important functions in the reminder of the homework. Solid arrows denote functions that directly call each other; dashed arrows denote functions that are called by passing them as arguments. Each box contains also the main parameters used by the function.



The code is written in such a way that many parts are reused. We will have a function that runs the potential planner, i.e., that applies Euler's method, for three different settings. First, the traditional potential methods (Problem 2), where the planner uses the total potential which is computed as the sum of repulsive and attractive potentials. Second, the CLF-CBF method (Problem 3), where the planner uses a vector field computed, from the attractive gradient and the sphere distance function. Third, an inverse kinematics solver for a two-link manipulator (Problem 4) where the planner uses the total potential attached to the end effector of the manipulator.

Problem 2: The potential-based planner

In this problem, you will implement and test the path planning strategy based on attractive-repulsive potential. Different functions will use similar structures, which are explained here for ease of reference:

- A struct array `world` with the same meaning as explained for the one in the file `sphereworld.mat`.
- A struct `potential` with fields:
  - `xGoal`: a $[2 \times 1]$ vector with the x, y coordinates of the goal location.
  - `repulsiveWeight`: a scalar containing the weight of the attractive term with respect to the repulsive term.
  - `shape`: a string that specifies the shape of attractive potential. It can be equal to `'conic'` or `'quadratic'`.

How these structures are actually used is explained in the questions below.

The attractive and repulsive potential. In the following, we use the shorthand notation $d_i(x)$ to denote the distance between the point x and the sphere (defined as seen in class).

The attractive potential we use for this assignment is given by:

$$U_{\text{attr}}(x) = d^p(x, x_{\text{goal}}) = \|x - x_{\text{goal}}\|^p, \tag{1}$$

where p is a parameter to distinguish between conic and quadratic potentials, and its corresponding gradient is:

$$\nabla U_{\text{attr}}(x) = p\, d^{p-1}(x, x_{\text{goal}}) \frac{x - x_{\text{goal}}}{\|x - x_{\text{goal}}\|} = p d^{p-2}(x, x_{\text{goal}})(x - x_{\text{goal}}), \tag{2}$$

The repulsive potential we use for this assignment is given by

$$U_{\text{rep},i}(x) = \begin{cases} \frac{1}{2}\left(\frac{1}{d_i(x)} - \frac{1}{d_{\text{influence}}}\right)^2 & \text{if } 0 < d_i(x) < d_{\text{influence}}, \\ 0 & \text{if } d_i(x) > d_{\text{influence}}, \\ \text{undefined} & \text{otherwise}, \end{cases} \tag{3}$$

and the corresponding gradient is

$$\nabla U_{\text{rep},i}(x) = \begin{cases} -\left(\frac{1}{d_i(x)} - \frac{1}{d_{\text{influence}}}\right) \frac{1}{d_i(x)^2} \nabla d_i(x) & \text{if } 0 < d_i(x) < d_{\text{influence}}, \\ 0 & \text{if } d_i(x) > d_{\text{influence}}, \\ \text{undefined} & \text{otherwise}. \end{cases} \tag{4}$$

Question provided 2.1. This function evaluates the repulsive potential.

> [`URep`]=`potential_repulsiveSphere`(`xEval,sphere`)
> Description: Evaluate the repulsive potential from `sphere` at the location $x =$ `xEval`. The function returns the repulsive potential as given by (3).
> Input arguments
> - `xEval` (dim. [2 × 1]): the location x at which to evaluate the total repulsive potential.
> - `sphere` (dim. [1 × 1], type `struct`): a structure, as described above, defining a sphere.
>
> Output arguments
> - `URep` (dim. [1 × 1]): the value of U_{rep} evaluated at $x =$ `xEval`.

Use the value *Not-a-Number* (`Nan` in Matlab) for the case where the potential is undefined.

Question code 2.1. For this question you need to implement the gradient for the repulsive potential for spheres.

> [`gradURep`]=`potential_repulsiveSphereGrad`(`xEval,sphere`)
> Description: Compute the gradient of U_{rep} for a single sphere, as given by (4).
> Input arguments
> - `xEval` (dim. [2 × 1]), `sphere` (type `struct`): see input arguments for `potential_repulsiveSphere`(_).
>
> Output arguments

- `gradURep` (dim. $[2 \times 1]$): the gradient of U_{rep} evaluated at $x =$ `xEval`.

Requirements: This function should use `sphere_distanceGrad(_)`

Use the value *Not-a-Number* (`Nan` in Matlab) for the case where the potential is undefined.

Question report 2.1 (0.5 points). Use the provided function `grid_plotThreshold(_)` to visualize `potential_repulsiveSphereGrad(_)` for the first two spheres in the sphere world, and overlap the plot of the sphere in each plot; use a different figure for each sphere. Include the figures in your report.

Question report 2.2 (0.5 points). In the figures generated with the previous question, should the arrows point toward or away from the obstacle? Why? Why are the arrow only visible around the contour of the obstacle?

Question provided 2.2. Implement the attractive potential

`[UAttr]=potential_attractive(xEval,potential)`

Description: Evaluate the attractive potential U_{attr} at a point `xEval` with respect to a goal location `potential.xGoal` given by the formula: If `potential.shape` is equal to `'conic'`, use $p = 1$. If `potential.shape` is equal to `'quadratic'`, use $p = 2$.

Input arguments

- `xEval` (dim. $[2 \times 1]$): the location x at which to evaluate the potential.
- `potential` (dim. $[1 \times 1]$, type `struct`): the structure with fields `xGoal`, `shape` and `repulsiveWeight` previously described (the value of the field `repulsiveWeight` is not used in this function).

Output arguments

- `UAttr` (dim. $[1 \times 1]$): the value of U_{attr} evaluated at $x =$ `xEval`.

Question code 2.2 (2 points). Implement a function to compute the gradient of the attractive potential.

`[gradUAttr]=potential_attractiveGrad(xEval,potential)`

Description: Evaluate the gradient of the attractive potential ∇U_{attr} at a point `xEval`. The gradient is given by the formula If `potential.shape` is equal to `'conic'`, use $p = 1$; if it is equal to `'quadratic'`, use $p = 2$.

Input arguments

- `xEval` (dim. $[2 \times 1]$): the location x at which to evaluate the potential.
- `potential` (dim. $[1 \times 1]$, type `struct`): the structure with fields `xGoal`, `shape` and `repulsiveWeight` previously described (the value of the field `repulsiveWeight` is not used in this function).

Output arguments

- `gradUAttr` (dim. $[2 \times 1]$): the value of ∇U_{attr} evaluated at $x =$ `xEval`.

Question code 2.3. The following two functions should combine the attractive and repulsive potentials into the total potential.

[UEval]= potential_total (xEval,world,potential)
Description: Compute the function $U = U_{\text{attr}} + \alpha \sum_i U_{\text{rep,i}}$, where α is given by the variable potential.repulsiveWeight .
Input arguments

- xEval (dim. [2 × 1]), world (dim. [NSpheres × 1], type struct): see input arguments for potential_repulsiveSphere ( ).
- potential (type struct): the structure with fields xGoal , shape and repulsiveWeight previously described. This function uses the field repulsiveWeight , and then the structure is passed to potential_attractive ( )).

Output arguments

- UEval : The value of the potential U evaluated at xEval .

[gradUEval]= potential_totalGrad (xEval,world,potential)
Description: Compute the gradient of the total potential, $\nabla U = \nabla U_{\text{attr}} + \alpha \sum_i \nabla U_{\text{rep,i}}$, where α is given by the variable potential.repulsiveWeight
Input arguments

- xEval (dim. [2 × 1]), world (dim. [NSpheres × 1], type struct), potential (dim. [1 × 1], type struct): same as potential_total ( ).

Output arguments

- gradUEval (dim. [2 × 1]): The gradient of the potential, ∇U, evaluated at xEval .

Question optional 2.1. Use the function grid_plotThreshold ( ) to visualize the attractive and repulsive potentials (first separately for a goal and obstacle, and then combined in the total potential), and their gradients, for different situations (try both conic and quadratic potentials, and both filled-in and hollow spherical obstacles). For the total potential, overlap the output of grid_plotThreshold ( ) on top of the output of sphereworld_plot ( ).

Question code 2.4. Implement a function for your main potential planner. We will collect all the parameters describing the behavior of the planner into the struct plannerParameters , which has the following fields:

- U : handle to the function for computing the value of the potential function; its arguments are the same as potential_total ( );
- control : handle to the function for computing the direction in which the planner should move (for gradient-based methods, this is the negative gradient of the potential function); its arguments are the same as potential_totalGrad ( );
- epsilon : value for the step size;
- NSteps : total number of steps.

[xPath,UPath]= potential_planner (xStart,world,potential,plannerParameters)

Description: This function uses a given function (given by plannerParameters.control) to implement a generic potential-based planner with step size plannerParameters.epsilon , and evaluates the cost along the returned path. The planner must stop when either the number of steps given by plannerParameters.NSteps is reached, or when the norm of the vector given by plannerParameters.control is less than $5 \cdot 10^{-3}$ (equivalently, 5e-3).

Input arguments

- xStart (dim. [2 × 1]): starting location.
- world (dim. [NSpheres × 1], type struct array), potential (type struct): descriptions of the world and of the potential as previously described. These are passed to the function plannerParameters.control .
- plannerParameters (type struct): a struct with four fields as described above.

Output arguments

- xPath (dim. [2 × NSteps]): sequence of locations generated by the planner, such that x[:,1]= xStart and x[:,k+1]=x[:,k]+epsilon*controlCurrent , where controlCurrent is obtained by evaluating control using x[:,k] , world and potential as arguments. If the planner stops for some k<NSteps , the remaining entries should be filled with NaN (the special value *Not-a-Number*, which can be generated with the NaN ( ) function in Matlab).
- UPath (dim. [1 × NSteps]): The array of values of the potential U evaluated at each of the points on the path (that is, UPath[k] is the value of U evaluated at xPath[k]). Again, use NaN 's to fill the array if the planner stops before NSteps .

Note: Questions report 2.3–report 2.7 should be considered together.

Question report 2.3. Implement a function that runs the planner on the provided data, and visualizes the results to explore and illustrate the effect of the different parameters in the planner. This question will generate many figures. Include all figures in your report, trying to have all of them organized in one or two pages, for ease of comparison.

potential_planner_runPlotTest ( )

Description: The function should call potential_planner ( ) for every combination of start and goal location in world , and for different interesting combinations of the parameters. See below for the specific steps.

This function performs the following steps for every combination of epsilon , shape , NSteps :

1) Load the problem data from the file sphereworld.mat .

2) For every goal location in xGoal :

(a) Create a new figure, with two subplots. In the subplot on the left, use the function `sphereworld_plot(_)` to plot the world.

(b) For every start location in `xStart`:

i. Prepare the `potential` stucture with the following fields set for the current test combination.
- `xGoal`
- `repulsiveWeight`
- `shape`

ii. Create function handles to `potential_total(_)` and `potential_totalGrad(_)` using the `potential` stucture above.

iii. Prepare the structure `plannerParameters` with the following fields:
- field `U`, set to `@potential_total`
- `control`, set to the negative of `@potential_totalGrad`.
- `epsilon`.
- `NSteps`.

iv. Calls the function `potential_planner(_)` with the problem data and the input arguments.

v. Plot the resulting trajectory superimposed to the world in the first subplot.

vi. Plot the corresponding value of the potential `UPath` (using the same color and using the `semilogy` command) in a second subplot.

(c) For the current goal, you should have one figure with two subplots with the data for five paths.

(d) If some of the trajectories converge, add another figure zoom in closely around the final equilibrium (to see the behavior of the planner near it).

3) For the current combination of parameters, you should have two figures, each one with two subplots.

Suggestions for tuning parameters. Start with `repulsiveWeight` in the inteval `0.01` – `0.1`, ϵ in the range `1e-3` – `1e-2`, for `plannerParameters.NSteps`, use `100`, and explore from there. Typically, adjustments in `repulsiveWeight` require subsequent adjustments in `epsilon`; if the trajectories seem to behave well but stop short of the goal, increase `plannerParameters.NSteps`. In general, you should be able to tune the parameters such that almost all the trajectories reach the respective goal (or its vicinity). Additional hints are available for this question.

Question optional 2.2. Make functions `sphere_distanceClip(_)` and `sphere_distanceClipGrad(_)` that are the same as `sphere_distance(_)` and `sphere_distanceClipGrad(_)`, except that they clip their outputs to `distInfluence` and `[0;0]`, respectively, when $d_i(x) > d_{\text{influence}}$, and make a function `clfcbf_controlClip(_)` that uses them. Test the planner.

Question report **2.4.** Use the function `grid_plotThreshold(_)` to visualize the total potential U, and its gradient ∇U in two separate figures for each one of the combinations `repulsiveWeight` and `shape` included in the previous question (report 2.3) (for this question, it is sufficient to consider only one of the two goals). Include the images in your report.

Question report **2.5.** Comment on the effects of varying each one of the parameters `repulsiveWeight` and `epsilon`. Explain why the planner behaves in the way you see, explicitly explaining what happens for the four cases where the two parameters are, respectively, small/small, small/large, large/small, and large/large. Additionally, comment on whether the results you see in practice are consistent with what discussed in class.

Question report **2.6 (0.5 points).** Comment on the relation between the value of the potential U toward the end of the iterations, versus the fact that the planner correctly succeeded or failed. Explain different reasons why the planner might fail, and why changing the various parameters can improve the situation.

Question report **2.7.** What is the difference between the two goals included in the provided dataset? In relation to this, what is the effect of the parameter `shape`?

Problem 3: CLF-CBF formulation

This problem is similar to Problem 2, except that you will use a CLF-CBF formulation instead of a traditional gradient-based formulation. We will use U_{attr} in (1) as the CLF, and $d_i(x)$ (implemented in `sphere_distance(_)`) as the CBF for each obstacle. Note that, since we control the position of our (point) robot directly, the dynamics of our system is input-affine with the form $\dot{x} = f(x) + g(x)u = u$, i.e., $f(x) = 0$ and $g(x) = I$, the identity matrix.

Preparation. First, we need to write the Quadratic Program (QP) that uses the minimum-effort CLF-CBF formulation (the supervisor version) to return a control $u^*(x)$ for any given x (see also the class notes). In particular, the QP will have the form

$$u^*(x) = \underset{u}{\text{argmin}} \|u - \clubsuit\|^2 \tag{5a}$$

$$\text{subject to } \spadesuit \leq 0, \qquad (\text{CBF constraint, } i\text{-th obstacle}) \tag{5b}$$

Question report **3.1.** In the report, write the expressions for $\clubsuit$ (involving U_{attr}) and $\spadesuit$ (involving $d_i(x)$). For this and the questions below, denote the constant appearing in the constraint (5b) as c_h. Please pay attention to the directions of the inequalities.

Question provided **3.1.** As detailed in the Week 6 in-class activity, the provided file `qp_supervisor.m` will solve the following QP.

$$\begin{aligned} u_{\text{opt}} &= \underset{u \in \mathbb{R}^2}{\text{argmin}} \|u - u_{\text{ref}}\|^2 \\ &\text{subject to } A_{\text{barrier}} u + b_{\text{barrier}} \leq 0 \end{aligned} \tag{6}$$

Question report **3.2 (0.5 points).** Comparing (6) with (5), and knowing that there are 4 obstacles in the environment, give the dimensions of A_{barrier} and b_{barrier}.

Question code 3.1. Write a function to compute u^*.

> [uOpt]= clfcbf_control (xEval,world,potential)
> Description: Compute u^* according to (5).
> Input arguments
> - xEval (dim. [2 × 1]), world (dim. [NSpheres × 1], type struct), potential (type struct): same as potential_total ( ).
>
> Output arguments
> - uOpt (dim. [2 × 1]): The solution to the QP (5).
>
> Requirements: This function should use qp_supervisor ( ) from Question provided 3.1, by building A_{barrier}, b_{barrier} according to ♠ in (5b), using
> $c_h =$ potential.repulsiveWeight .

Question provided 3.2.

> clfcbf_control_test_singleSphere ( )
> Description: Use the provided function grid_plotThreshold ( ) to visualize the CLF-CBF control field around a single filled-in sphere with the goal set at $x = \left[\begin{smallmatrix} 0 \\ -6 \end{smallmatrix}\right]$.

Question report 3.3 (0.5 points). Run the function from provided 3.2 above, and include the produced image in the report. You should see that the field induces a flow that goes around and has a stable equilibrium around the point $\left[\begin{smallmatrix} 0 \\ -6 \end{smallmatrix}\right]$.

Question report 3.4 (0.5 points). Explain why the field is essentially zero at the origin (i.e., at the top of the edge of the sphere). To get full points, your explanation should be based on 5 and what we have seen in class.

Question report 3.5. This question is analogous to report 2.3, but using the CLF-CBF framework instead of the gradient of the potential.

> clfcbf_planner_runPlotTest ( )
> Description: Use the function potential_planner_runPlot ( ) to run the planner based on the CLF-CBF framework, and show the results for one combination of repulsiveWeight and epsilon that makes the planner work reliably.
> Requirements: The function should follow these steps *for each goal*:
>
> *1)* Load the sphere world in the variable world
>
> *2)* Create the struct potential .
>
> *3)* Create the plannerParameters struct.
>
> *4)* Call potential_planner_runPlot ( )
>
> Use the following for the arguments in the various steps above:
> - plannerParameters.U : use the function potential_attractive ( );

- `plannerParameters.control`: use `@clfcbf_control`;
- `plannerParameters.NSteps`: use `20` (but increase if necessary);
- `potential.shape`: use `'conic'`.

Include all figures in your report, trying to have all of them in one or two pages, for ease of comparison.

Question report 3.6. Use the function `grid_plotThreshold(_)` to visualize the control field $u^*(x)$ for the value of `repulsiveWeight` included in the previous question (report 2.3). Make sure to superimpose the field on top of the `world` map. Note that the computation of $u^*(x)$ will take significantly longer than just the gradient, hence you might want to reduce the size of the grid passed to `grid_plotThreshold(_)` (e.g., use a 10×10 grid). Include the images in your report.

Question report 3.7. Comment on the trade-off between traditional gradient-based methods and a CLF-CBF formulation.

Question optional 3.1. Repeat the previous questions with `shape` set to `'quadratic'`.

Problem 4: Jacobian-based Inverse Kinematics (IK) for the two-link manipulator

In this problem, you will combine the attractive potential from Problem 2 with the Jacobian of the two-link manipulator from the previous homework to create a way to move the end effector of the manipulator to a specific configuration by acting on its joint angles. This procedure is commonly called Inverse Kinematics.

Question report 4.1. Recall from Homework 2 that we denote the position of the end effector in the world reference frame as ${}^{\mathcal{W}}p_{\text{eff}}$. Based on the results of Homework 2, write an expression for the Jacobian matrix J such that $\frac{\mathrm{d}}{\mathrm{d}t}({}^{\mathcal{W}}p_{\text{eff}}) = J(\theta)\dot{\theta}$ (this was optional in Homework 2, but it is now mandatory). To obtain full marks, derive the expression in matrix form using the derivative of rotations (which involves rotations and skew-symmetric matrices), and show all the steps of the derivation.

Question code 4.1. Implement a function for computing the matrix J.

[`Jtheta`]=`twolink_jacobianMatrix(theta)`

Description: Compute the matrix representation of the Jacobian of the position of the end effector with respect to the joint angles as derived in Question report 4.1.

Input arguments

- `theta` (dim. [2 × 1]) An array containing $\begin{bmatrix}\theta_1 \\ \theta_2\end{bmatrix}$, the two joint angles for the two-link manipulator.

Output arguments

- `Jtheta` (dim. [2 × 2]): The matrix $J(\theta)$ defined in Question report 4.1.

If you already solved this question in Homework 2, please copy the relevant portion of your previous report here.

Question optional **4.1.** Compare the results of `Jtheta*thetaDot`, where [`Jtheta`]=`twolink_jacobianMatrix`(`theta`), with the results of `twolink_jacobian`(`theta,thetaDot`), for arbitrary values of `theta` and `thetaDot` (they should be the same).

Question code **4.2.** In this question you will adapt the potential planner from Problem 2 to work with the two-link manipulator, by pulling back the total potential and its gradient from $\mathbb{R}^2$ to the configuration space of the two-link manipulator.

[`UEvalTheta`]=`twolink_potential_total`(`thetaEval,world,potential`)
Description: Compute the potential U pulled back through the kinematic map of the two-link manipulator, i.e., $U({}^{\mathcal{W}}p_{\text{eff}}(\vec{\theta}))$, where U is defined as in Question code 2.3, and ${}^{\mathcal{W}}p_{\text{eff}}(\theta)$ is the position of the end effector in the world frame as a function of the joint angles $\vec{\theta} = \begin{bmatrix}\theta_1\\ \theta_2\end{bmatrix}$.
Input arguments

- `thetaEval` (dim. [2 × 1]), `world` (dim. [`NSpheres` × 1], type `struct`): see input arguments for `potential_repulsiveSphere`(_).
- `potential` (dim. [1 × 1], type `struct`): the structure with fields `xGoal`, `shape` and `repulsiveWeight` previously described. This function uses the field `repulsiveWeight`, and then the structure is passed to `potential_attractive`(_)).

Output arguments

- `UEvalTheta`: The value of the potential U evaluated at `xEval`.

[`gradUEvalTheta`]=`twolink_potential_totalGrad`(`thetaEval,world,potential`)
Description: Compute the gradient of the potential U pulled back through the kinematic map of the two-link manipulator, i.e., $\nabla_{\vec{\theta}} U({}^{\mathcal{W}}p_{\text{eff}}(\vec{\theta}))$.
Input arguments

- `thetaEval` (dim. [2 × 1]): an array containing the two joint angles $\begin{bmatrix}\theta_1\\ \theta_2\end{bmatrix}$ at which the gradient should be evaluated.
- `world` (dim. [`NSpheres` × 1], type `struct`), `potential` (dim. [1 × 1], type `struct`): same as `potential_total`(_).

Output arguments

- `gradUEvalTheta` (dim. [2 × 1]): The gradient of the pulled-back potential, evaluated at `thetaEval`.

Note that for computing the gradient of the pulled-back function, you will need to use the Jacobian matrix given by `twolink_jacobianMatrix`(_) (see also the material from class).

Question provided **4.1.** Make a function `twolink_plotAnimate`(`thetaPath,fps`) that uses the function `twolink_plot`(_) from Homework 2 to show the sequence of configurations on the path specified by `thetaPath`. Put a `pause` of $\frac{1}{\texttt{fps}}$ seconds between each draw

operation (so that the function shows a sequence of figures as a movie), and use the command `hold on` to make all the drawing overlap (so that you can see all the configurations in a single picture). If you feel there are too many configurations, you can draw only a subset of the configurations (e.g., one configuration every 15).

Question provided **4.2.** Implement a function that tests the potential planner applied to the two-link manipulator. This is similar to Question **??**, except that we are planning the trajectory of the end effector of the manipulator, instead of a free point.

> `twolink_planner_runPlot(potential,plannerParameters)`
>
> Description: This function performs the same steps as `potential_planner_test(_)` in Question **??**, except for the following:
>
> - In step *2)*(b)iv: `plannerParameters.U` should be set to `@twolink_total`, and `plannerParameters.control` to the negative of `@twolink_totalGrad`.
> - In step *2)*(b)iv: Use the contents of the variable `thetaStart` instead of `xStart` to initialize the planner, and use only the second goal `xGoal(:,2)`.
> - In step *2)*(b)v: Use `twolink_plotAnimate(_)` to plot a decimated version of the results of the planner. Note that the output `xPath` from `potential_planner(_)` will really contain a sequence of join angles, rather than a sequence of 2-D points. Plot only every 5th or 10th column of `xPath` (e.g., use `xPath(:,1:5:end)`). To avoid clutter, plot a different figure for each start.
>
> Input arguments
>
> - `potential` (type `struct`): same as in `potential_total(_)`.
> - `plannerParameters` (type `struct`): same as in `potential_planner(_)`, except that `plannerParameters.U` and `plannerParameters.control` can be left unset.

Question report **4.2.** Show the results of `twolink_planner_runPlot(_)` for (one) combination of `repulsiveWeight` and `epsilon` that makes the planner work reliably. For the argument `plannerParameters.NSteps`, use `400`, and for `potential.shape`, use `'quadratic'`. Note that `plannerParameters.U`, and `plannerParameters.control` are set by the function, so they do not need to be set in the argument. In your report, try to have all figures on the same one or two pages for ease of comparison.

If it is too difficult to get the planner work reliably for all goal configurations, it is fine to include the results using different parameters for different goals, or just for a few out of the five goals.

Note that, in this problem, we are considering only collisions between the spheres and the end effector of the manipulator; we are not considering collisions between the spheres and the links of the manipulator; in other words, it is normal to have overlap between the manipulator and the spheres, as long as the end effector is not inside an obstacle.

Hint for question report 2.3: For the argument `plannerParameters.control`, you can use
`@(x) -potential_totalGrad(x,world,potential)`. If everything works, the plot of `URep` should be monotonically decreasing, at least for some very small `epsilon`, even if the planner does not get to the goal. As a rule of thumb, the values of `repulsiveWeight` and `epsilon` can be significantly different between the `'conic'` and `'quadratic'` shapes, and when `repulsiveWeight` is increased, the value of `epsilon` that works tends to decrease (given our discussion in class, can you explain this?) Generally, if the planner "goes crazy", it is because either the repulsive gradient or the step size are too large. If the planner never enters the region of influence of the obstacles, the weight for the repulsive term is too large. If the planner does not make enough progress, the step size is too small.

Hint for question report 4.1: Note that expressions such as $av + bw$, where $a, b \in \mathbb{R}$ are scalars, and $v, w \in \mathbb{R}^2$ are vectors, can be equivalently written as the matrix-vector multiplication $\begin{bmatrix} v & w \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$.

Hint for question code 2.2: Recall that, in Matlab, you need to use `strcmp(_)` to compare two strings

Hint for question code 2.3: You can refer to as an example of how to iterate over the `sphereworld` structure.

Hint for question report 3.2: Knowing that $u \in \mathbb{R}^{2\times 1}$, the dimensions of A_{barrier} and b_{barrier} should be consistent with matrix-vector multiplication rules.