Homework 3

ME 570 - Prof. Tron

2020-10-15

In this homework we will first introduce a special type of environment, a *sphere world*. You will then implement a potential-based planner (with two shapes for the attractive potential), and an Inverse Kinematics solver for the two-link manipulator based on the same planner.

General instructions

Programming For your convenience, together with this document, you will find a `zip` archive containing Matlab files with stubs for each of the questions in this assignment; each stub contains an automatically generated description of the function and the function header. You will have to complete these files with the body of each function requested. The goal of this files is to save you a little bit of time, and to avoid misspellings in the function or argument names. The functions from the parts marked as **provided** (see also the *Grading* paragraph below) contain already the body of the function.

Please refer to the guidelines on Blackboard under Information/Programming Tips & Tricks/Coding Practices.

Homework report Along the programming of the requested functions, prepare a small PDF report containing one or two sentences of comments for each question marked as **report**, and including:

- Embedded figures and outputs that are representative of those generated by your code.
- All analytical derivations if required.

You can include comments also on the questions marked as **code**, if you feel it necessary (e.g., to explain any difficulty you might have encountered in the programming), but these will not be graded. In general, *do not* insert the listing of the functions in the report (I will have access to the source files); however, you can insert *short* snippets of code if you want to specifically discuss them.

A small amount of *beauty points* are dedicated to reward reports that present their content in a professional way (see the *Grading criteria* section in the syllabus).

Analytical derivations To include the analytical derivations in your report you can type them in LaTeX(preferred method), any equation editor or clearly write them on paper and use a scanner (least preferred method).

Submission

The submission will be done on Gradescope through two separate assignments, one for the questions marked as **code**, and one for those marked as **report**. See below for details. You can submit as many times as you would like, up to the assignment deadline. Each question is worth 1 point unless otherwise noted. Please refer to the Syllabus on Blackboard for late homework policies.

Report Upload the PDF of you report, and then indicate, for each question marked as **report**, on which page it is answered (just follow the Gradescope interface). Note that some of the questions marked as **report** might include a coding component, which however will be evaluated from the output figures you include in the report, not through automated tests. Many of these questions are intended as checkpoints for you to visually check the results of your functions.

Code questions Upload all the necessary `.m` files, both those written by you, and those provided with the assignment. The questions marked as **code** will be graded using automated tests. Shorty after submission, you will be able to see the results for *some* of the tests: green and red mean that the test has, respectively, passed or not; if a test did not pass, check for clues in the name of the test, the message provided on Gradescope, and the text of this assignment. You can post questions on Blackboard at any time.

Note 1: The final grade will depend also on additional tests that will become visible only after all the grades have been reviewed as a whole. It is therefore important that you examine the results in the figures that you will generate for the report.

Note 2: The automated tests use Octave, a open-source clone of Matlab. For the purposes of this class, you should not encounter any specific compatibility problem. If, however, you suspect that some tests fail due to this incompatibility, please contact the instructor.

Optional and provided questions. Questions marked as **optional** are provided just to further your understanding of the subject, and not for credit. Questions marked as **provided** have already a solution provided in the file accompanying this assignment, which you can directly use unless you want to answer it by yourself. If you submit an answer for optional or provided questions I will correct it, but it will not count toward your grade.

Maximum possible score. The total points available for this assignment are 38.5 (37.5 from questions, plus 1.0 that you can only obtain with beauty points).

Hints

Some hints are available for some questions, and can be found at the end of the assignment (you are encouraged to try to solve the questions without looking at the hints first). If you use these hints, please state so in your report (your grading will not change based on this information, but it is a useful feedback for me).

Use of external libraries and toolboxes All the problems can be solved using Matlab's standard features. You are **not allowed** to use functions or scripts from external libraries or toolboxes (e.g., mapping toolbox), unless specifically instructed to do so (e.g., CVX).

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:

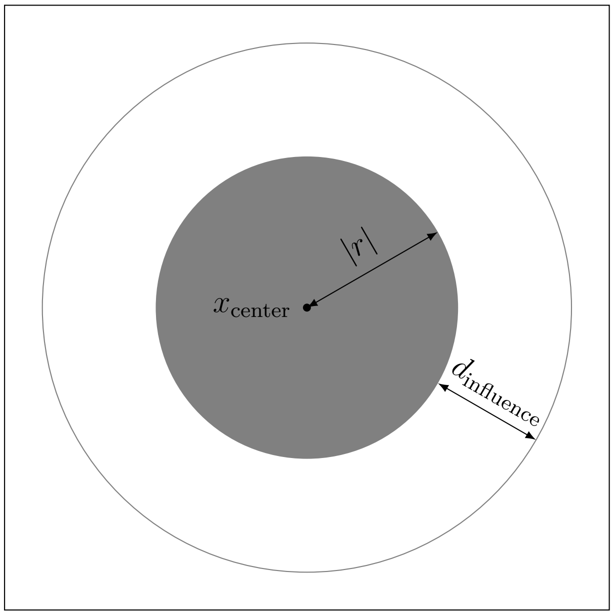
- `sphere.xCenter`, a $[2 \times 1]$ array containing the 2-D coordinate of the center of the sphere;
- `sphere.radius`, a scalar whose absolute value is equal to the geometric radius of the sphere, and the sign indicates whether the obstacle should be interpreted as filled-in (`radius` > 0) or hollow (`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.

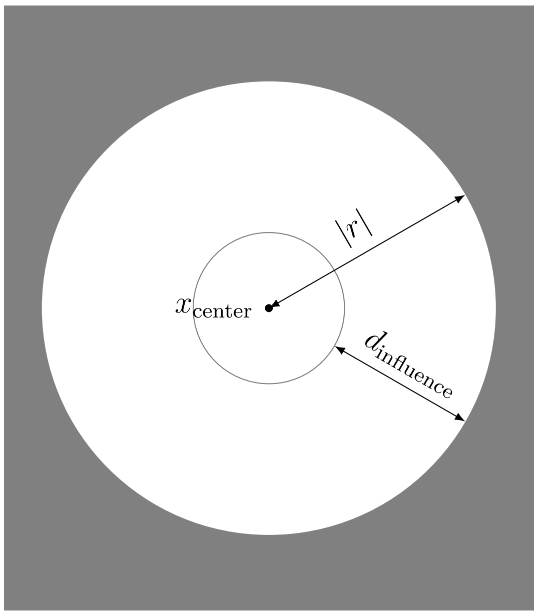
Question provided 1.1. Implement a function that draws a sphere.

> `sphere_plot` (`sphere,color`)
>
> Description: This function draws the sphere (i.e., a circle) with radius `sphere.r` and the specified color, and another circle with radius `sphere.rInfluence` in gray.
>
> 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.



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



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

Figure 1: Examples of spheres.

Question **code** **1.1.** Implement the following function.

[`dPointsSphere`]= `sphere_distance` (`sphere,points`)
Description: Computes the signed distance between points and the 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.

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

`sphere_testCollision` ( )
Description: Generates one figure with a sphere (`xCenter=[0;0]` , `r=1`) 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 with `r=-1` . For each sampled point, plot also the result of the output `pointsSphere` .

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 variables 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 in each column).
- `xGoal`, a [`2` × `NGoal`] vector containing the coordinates of different goal locations (one in 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.

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 convenience:

- 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` × `1`] vector with the (x, y) coordinates of the goal location.
  - `repulsiveWeight`: a [`1` × `1`] 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. It is recommended that you approach this problem holistically. Read all the questions first. You should work on the function `potential_planner( )` from the last question while you work and debug the other functions.

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

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

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

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

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

The repulsive potential we use for this assignment is given by

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

and the corresponding gradient is

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

Question provided 2.1. This function evaluates the repulsive potential.

[`URep`]= `potential_repulsiveSphere` (`xEval,sphere`)

Description: Evaluate the sum repulsive potentials 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 × 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 provided 2.2. Implement the attractive potentials

[`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 × 1]): the location x at which to evaluate the potential.
- `potential` (dim. [1 × 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 × 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 `potential.shape` is equal to `'quadratic'`, use $p = 2$.

Input arguments

- `xEval` (dim. [2 × 1]): the location x at which to evaluate the potential.
- `potential` (dim. [1 × 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 × 1]): the value of U_{attr} evaluated at $x =$ `xEval`.

Question provided 2.3. The following is a utility functions.

`field_plotThreshold`(`fHandle,threshold,grid`)

Description: The function evaluates the function handle `fHandle` on points places on a regular grid; at any point where the norm of the result is larger than `threshold`, the result is thresholded (normalized); the function then uses `surf`( ) or `quiver`( )

to display the result (depending on whether the output of `fHandle`( ) is one- or two-dimensional); in the former case, you can rotate the plot to get a 3-D view of the function's profile.

Input arguments

- `fHandle` (, type `function handle`): an handle to a function that takes a vector `x` of dimension $[2 \times 1]$ as a single argument, and returns a scalar or a vector. An example of this function could be the function `norm`( ).
- `threshold` **optional**: 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 `field_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);
field_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 `field_plotThreshold`( ) for plotting. Please search the Matlab help for *anonymous functions* for detailed information.

Question **optional** **2.1.** Use the function `field_plotThreshold`( ) to visualize the attractive and repulsive potentials, and their gradients, for different situations (try both conic and quadratic potentials, and both filled-in and hollow spherical obstacles). Overlap the output of `field_plotThreshold`( ) on top of the output of `sphereworld_plot`( ).

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` $\times$ 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

- `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` $\times$ 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 code **2.4.** Implement a function for your main potential planner.

[`xPath`,`UPath`]=`potential_planner`(`xStart`,`world`,`potential`,`plannerParameters`)
Description: This function uses a given control field (`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 10^{-3} (equivalently, `1e-3`).
Input arguments

- `xStart` (dim. [2×1]): starting location.
- `world` (dim. [`NSpheres` $\times$ 1], type `struct array`), `potential` (dim. [1×1], type `struct`): descriptions of the world and of the potential as previously described. These are passed to the function `fHandles.control`.
- `plannerParameters` (, type `struct`): a struct with four members:
  - `U`: handle to the function for computing the value of the potential function;

  - `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);
  - `epsilon`: value for the step size;
  - `NSteps`: total number of steps.

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`.

Question provided 2.4. Implement a test function for the planner.

`potential_planner_runPlot`(`potential,plannerParameters`)
Description: This function performs the following steps:

1) Loads the problem data from the file `sphereworld.mat`.

2) For each goal location in `world.xGoal`:

   (a) Uses the function `sphereworld_plot`( ) to plot the world in a first figure.

   (b) Sets `plannerParameters.U` to `@potential_total`.

   (c) Calls the function `potential_planner`( ) with the problem data and the input arguments. The function needs to be called five times, using each one of the initial locations given in `xStart` (also provided in `sphereworld.mat`).

   (d) After each call, plot the resulting trajectory superimposed to the world in the first subplot; in a second subplot, show `UPath` (using the same color and using the `semilogy` command).

Input arguments

- `potential` (, type `struct`): same as in `potential_total`( ).
- `plannerParameters` (, type `struct`): same as in `potential_planner`( ), except that `plannerParameters.U` can be left unset.

Requirements: To avoid too much clutter, use separate pairs of figures for different goal locations (but keep all the different paths from the different starting points to the same goal location in the same pair of figures).

Note: Questions report 3.2–report 2.5 should be considered together.

Question report 2.1. Show the results of `potential_planner_runPlot`() for different interesting combinations of `potential.repulsiveWeight`, `potential.shape`, and `plannerParameters.epsilon`. For the `plannerParameters.control` argument, pass a function computing the negative of the gradient, and for `plannerParameters.NSteps`, use `1000`. Start with α in the inteval `0.01` $-$ `0.1` and ϵ in the range `1e-3` $-$ `1e-2`, and explore from there. Typically, adjustments in `repulsiveWeight` require subsequent adjustments in `epsilon`. When the planner converges, zoom in closely around the final equilibrium. In your report, try to have all figures on the same one or two pages for ease of comparison. 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 2.2. Use the function `field_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 3.2) (for a single goal is sufficient). Include the images in your report.

Question report 2.3. Comment on the effects of varying each one of the parameters `repulsiveWeight` and `epsilon`. Explain why the planner behaves in the way you see, stating what happens when each value is too small, too large, or just right, and whether the results you see in practice are consistent with what discussed in class.

Question report 2.4 (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.

Question report 2.5. 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 robust CLF-CBF formulation to return a control $u^*(x)$ for any given x (see also the class notes for Week 7). In particular, the QP will have the form

$$u^*(x) = \underset{u,\delta}{\operatorname{argmin}} \|u\|^2 + m\delta^2 \tag{5a}$$
$$\text{subject to } \clubsuit \leq \delta, \qquad \text{(CLF constraint)} \tag{5b}$$
$$\spadesuit \leq 0, \qquad \text{(CBF constraint, } i\text{-th obstacle)} \tag{5c}$$

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, assume that the constants appearing in the constraints are $c_V = c_h = 1$. Please pay attention to the directions of the inequalities.

Question provided 3.1. The function `qp_minEffortFix`( ) will solve the following QP. Note that this is equivalent to a question from Homework 1, although the variable δ is moved on the other side of the first constraint. Please use the file provided with this homework.

$$\begin{aligned} u_{\text{opt}}, \delta_{\text{opt}} = &\underset{u\in\mathbb{R}^2,\delta\in\mathbb{R}}{\operatorname{argmin}} \|u\|^2 + m\delta^2 \\ \text{subject to } &A_{\text{attr}}u + b_{\text{attr}} \leq \delta \\ &A_{\text{barrier}}u + b_{\text{barrier}} \leq 0 \end{aligned} \tag{6}$$

[`uOpt,DeltaOpt`]=`qp_minEffortFix`(`AAttr,bAttr,ABarrier,bBarrier,mPenalty`)

Description: Solves the optimization problem in (6) for the given parameters. *Note:* $\|u\|^2$ can be written in CVX as `u'*u`, however (due to a problem of CVX on Octave), this expression will not work in the autograder; instead, please use `pow_pos(norm(u,2),2)`.

Input arguments

- `AAttr` (dim. [1 × 2]), `bAttr` (dim. [1 × 1]): parameters of a linear constraint that involves both u and δ.
- `ABarrier` (dim. [nbObstacles × 2]), `bBarrier` (dim. [nbObstacles × 1]): parameters of `nbObstacles` linear constraints on u; `nbObstacles` can be an arbitrary non-negative integer (even zero).
- `mPenalty` : a weight for the cost function.

Output arguments

- `uOpt` (dim. [2 × 1]), `DeltaOpt` : vector and scalar with the optimal solution.

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

[`uOpt`]=`clfcbf_control`(`xEval,world,potential`)

Description: Compute u^* according to (5), where m 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

- `uOpt` (dim. [2 × 1]): The solution to the QP (5).

Requirements: This function should use `qp_minEffortFix`( ) from Question provided 3.1, by building A_{attr}, b_{attr} according to ♣ in (5b), A_{barrier}, b_{barrier} according to ♠ in (6).

Question report 3.2. Show the results of `potential_planner_runPlot`( ) for (one) combination of `repulsiveWeight` and `epsilon` that makes the planner work reliably. For the argument `plannerParameters.control`, use `@clfcbf_control`, for `plannerParameters.NSteps`, use `20`, and for `potential.shape`, use `'quadratic'`. In your report, try to have all figures on the same one or two pages for ease of comparison.

Question report 3.3. Use the function `field_plotThreshold`( ) to visualize the control field $u^*(x)$ for each one of the combinations `repulsiveWeight` included in the previous question (report 3.2). 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 `field_plotThreshold`( ) (e.g., use a 10×10 grid). Include the images in your report.

Question report 3.4. 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 `'conic'`.

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 equivalent to computing an inverse of the kinematic map, and it is therefore 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).

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., $U({}^{\mathcal{W}}p_{\mathrm{eff}}(\vec{\theta}))$, where U is defined as in Question code 2.3, and ${}^{\mathcal{W}}p_{\mathrm{eff}}(\theta)$ 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., $\nabla_{\vec{\theta}}U({}^{\mathcal{W}}p_{\mathrm{eff}}(\vec{\theta}))$.
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).

Question provided 4.2. Implement a function that tests the potential planner applied to the two-link manipulator. This is similar to Question provided 2.4, 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 provided 2.4, except for the following:
>
> - In step *2)*c: `plannerParameters.U` should be set to `@twolink_total`, and `plannerParameters.control` to the negative of `@twolink_totalGrad`.
> - In step *2)*c: Use the contents of the variable `thetaStart` instead of `xStart` to initialize the planner, and use only `xGoal(:,2)` for the goal.
> - In step *2)*d: 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.

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.

Problem 5: Homework feedback

Question report 5.1. Indicate an estimate of the number of hours you spent on this homework (possibly broken down by problem). Explain what you found hard/easy about this homework, and include any suggestion you might have for improving it.

Hint for question report 3.2: 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 $\begin{bmatrix} v & w \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$.