Learning for Adaptive and Reactive Robot Control
Instructions for Practical 2

Professor: Aude Billard
Assistants: Harshit Khurana,
Lukas Huber and Yang Liu
Contacts:
aude.billard@epfl.ch, harshit.khurana@epfl.ch,
lukas.huber@epfl.ch, yang.liuu@epfl.ch

Spring Semester 2022

Introduction

In this practical you will develop dynamical systems for 3D obstacle avoidance, and control the motion of a 7 degree-of-freedom panda robot.
All the ressources are in the folder `practical_2` that you can download and place in your `matlab_exercises` folder, like the other exercises folders.

1 Part 1: Defining 3D shapes

In this section, we will derive implicit equations for various 3D shapes, and use them to define the gamma function of each type of obstacle.
A generic gamma function can be constructed using the following formula:

$$\Gamma(\mathbf{x}, \mathbf{x}_c) = \frac{||\mathbf{x} - \mathbf{x}_c||}{R(\mathbf{x}, \mathbf{x}_c)} \tag{1}$$

with $R(\mathbf{x}, \mathbf{x}_c)$ the radius from the center of the obstacle $\mathbf{x}_c$ to the boundary of the obstacle, in the direction of the point $\mathbf{x}$.
In this practical, all gamma functions are defined in the referential frame of the obstacle center, so Eq. (1) becomes:

$$\Gamma(\mathbf{x}) = \frac{||\mathbf{x}||}{R(\mathbf{x})} \tag{2}$$

TASK 1 Program the gamma function and its gradient for a vertical, infinite length cylinder.

The cylinder is vertical, so its cross section in the x-y plane is a circle of constant radius r. Because the cylinder is of infinite length, the height of its center point (z component) doesn't matter, and we can define the reference direction to be always in the x-y plane. Thus $R(\mathbf{x}, \mathbf{x}_c) = r$. We also don't need to model its top and bottom surfaces.

1. Program the `gamma` function in the class `Library/VerticalCylinder.m`, following the convention of Eq. (2).
2. In the same class, program the `gradientGamma` function by computing the gradient of the cylinder equation $x^2 + y^2 = r^2$.
3. Create some cylinders to verify your implementation. You can do so in section 1 of the file `practical2_main.m` by calling the function `myWorld.addCylinder()` with the first argument being the cylinder radius in meter and the second one the 3D position vector.

```
% Create a cylinder with a radius of 0.2m and a position
% in the X-Y plane of [-0.4, -0.4]:
myWorld.addCylinder(0.2, [-0.4, -0.4, 0]);
```

TASK 2 Program the gamma function and its gradient for a 3D ellipsoid.

We will use the same model as in task 1. To find the function $R(\mathbf{x})$ for an ellipsoid, we first consider a simple ellipse as in figure 1.

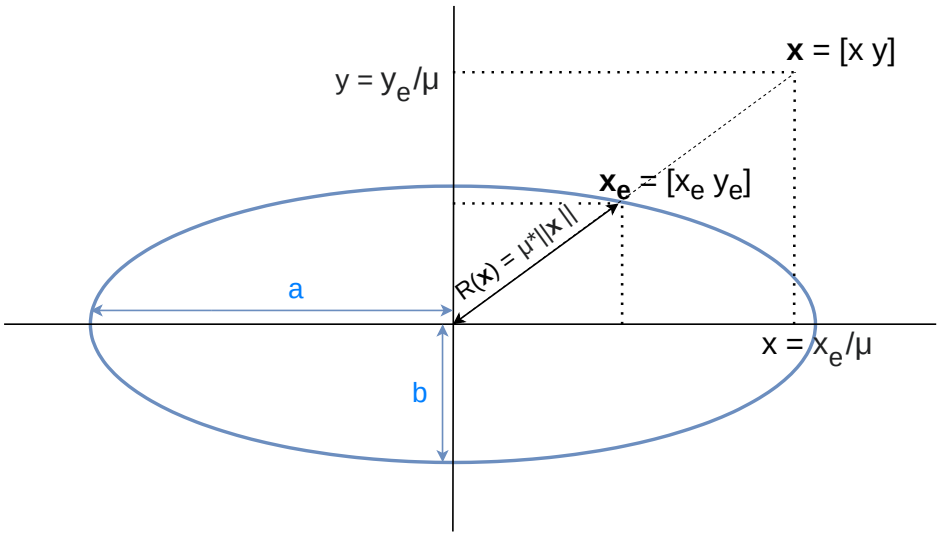

Figure 1: Radius $R(\mathbf{x}, \mathbf{x}_c)$ from center of ellipse to the point $\binom{x_e}{y_e}$, with equation $(\frac{x_e}{a})^2+(\frac{y_e}{b})^2 = 1$

The point $\mathbf{x}_e = \binom{x_e}{y_e}$ is on the boundary of the ellipse and the vector $\binom{x_e}{y_e}$ is colinear with $\binom{x}{y}$. To find $R(\mathbf{x})$, we solve the following equation:

$$R(\mathbf{x}) = ||\mathbf{x}_e|| = \mu * ||\mathbf{x}|| \qquad \text{with} \quad (\frac{x_e}{a})^2 + (\frac{y_e}{b})^2 = 1 \tag{3}$$

which gives $\mu = \frac{1}{\sqrt{(\frac{x}{a})^2+(\frac{y}{b})^2}}$, with a and b the two semi-axes of an ellipse.

1. Derive an expression for $R(\mathbf{x})$ and $\Gamma(\mathbf{x})$ using μ. Adapt this formula for a 3D ellipsoid, then program the `gamma` function in `Library/VerticalEllipsoid`
2. In the same class, program the `gradientGamma` function by computing the gradient of the ellipsoid equation $(\frac{x}{a})^2 + (\frac{y}{b})^2 + (\frac{z}{c})^2 = 1$.

3. Create some ellipsoids to verify your implementation. You can do so in section 1 by calling the function `myWorld.addEllipsoid()` with the first argument being a 3D vector with the three semi-axes in meter and the second one the 3D position vector of the ellipsoid center.

```
% Create a cylinder with semi-axes a=10cm, b=10cm, c=20cm,
% and a center position at [0, 0.2, 0.5].
myWorld.addEllipsoid([0.1, 0.1, 0.2], [0, 0.2, 0.5]);
```

TASK 3 Study the effect of using this formulation for the gamma function. What are the shortcomings of this method for multiple obstacles with different sizes and shapes ?

1. Place in your world only one ellipsoid with semi axes of 10cm, 20cm and 30cm respectively. To plot isosurfaces of your gamma fuctions, use the function `myWorld.showGammaIsosurfaces()` with an array of isosurfaces values that you want to plot.
   Hint: An isosurface value of 1 correspond to the surface of your obstacle, and values above 1 are the outside of the obstacle
   How does the gamma values increases in different directions of the ellipsoid ? Why could this be an issue if you want to avoid this obstacle ?
2. Add a cylinder of radius 30cm in a different part of your workspace. Plot again the isosurface to compare gamma values of the cylinder and ellipsoid. What do you observe ?
3. Propose another way of constructing the gamma function based on euclidean distance that would solve these issues. You can drop the constraint of having negative gamma values inside the obstacle, but keep a gamma value of 1 at the boundary.
   Hint: Define a gamma function whose values have a physical meaning

TASK 4 Implement your new gamma function formulation

1. Implement the gamma function of a cylinder with this new formulation in the function `gammaDistance` of the file `Library/VerticalCylinder.m`
2. Do the same for the ellipsoid in the file `Library/VerticalEllipsoid.m`
3. Do the same for an infinite plane with normal $\mathbf{n}$.
   Hint: Use the dot product of your vector $\boldsymbol{x}$ and the normalized vector $\boldsymbol{n}$ to get the distance of $\boldsymbol{x}$ to the plane
4. Verify that your new gamma function works by plotting the gamma values outside the obstacle. You can set the optional second parameter of the `showGammaIsosurfaces()` function to 1 to use your new function `gammaDistance`.

2 Part 2: Modulating DS for obstacle avoidance

In this section, a nominal DS is modulated using the gamma function derived in the previous section.
We use the following formulation to modulate a nominal DS $\dot{\mathbf{x}} = f(\mathbf{x})$:

$$\dot{\mathbf{x}} = \left(\prod_{k=1}^{K} \mathbf{M}^k \right) f(\mathbf{x}) \qquad \text{with} \quad \mathbf{M}^k = \mathbf{E}^k \, \mathbf{D}^k \, {\mathbf{E}^k}^{-1} \tag{4}$$

with M^k the modulation for each obstacle using the change of basis method. E^k is an orthogonal basis with the first vector being the local normal to the obstacle. Then D^k is a diagonal matrix with eigenvalues $\lambda_{1,2,3}$ defined as:

$$\lambda_1(\mathbf{x}) = 1 - \frac{1}{\Gamma^k(\mathbf{x})} \quad \text{and} \quad \lambda_i(\mathbf{x}) = 1 + \frac{1}{\Gamma^k(\mathbf{x})} \quad 2 \leq i \leq d \tag{5}$$

TASK 1 Implement the simple modulation method proposed in Eq. (4) and (5) in the function `modulatedDS` at the end of the file `practical2_main.m`.

1. Go over each obstacle stored in the list `world.listOfObstacles`, compute the change of basis E^k using the gradient of gamma `gradientGamma`, then compute D^k from the gamma function `gammaDistance`

```
% Get the gamma value of obstacle k at point x
world.listOfObstacles(k).gammaDistance(x(1), x(2), x(3))

% Get the gradient of obstacle k at point x
world.listOfObstacles(k).gradientGamma(x(1), x(2), x(3))
```

2. Compute $\mathbf{M}^k$ from $\mathbf{E}^k$ and $\mathbf{D}^k$, then use it to modulate the variable `xdot_modulated` at each obstacle.
3. Place one obstacle in section 1 of `practical2_main.m`, then run section 2 to display the robot and the path integrals of the modulated DS. Repeat with multiple obstacles. How does the modulated DS adapt to these new obstacles ?
4. Run section 3 (or the whole file at once) to see the modulated DS being followed by the panda robot. What type of collision cannot we avoid with this formulation ?

TASK 2 Improve the modulation method to better handle multiple obstacles, change the modulation sensitivity for different obstacles, and take into account the robot end effector size.
We can improve the modulation by adding two more parameters in equation 5.

$$\lambda_1(\mathbf{x}) = 1 - \frac{w^k}{\Gamma^k(\mathbf{x})^{\frac{1}{\rho}}} \quad \text{and} \quad \lambda_i(\mathbf{x}) = 1 + \frac{w^k}{\Gamma^k(\mathbf{x})^{\frac{1}{\rho}}} \quad 2 \leq i \leq d \tag{6}$$

The first parameter is the sensitivity ρ which changes the region of modulation around an obstacle. Small values of ρ reduce the modulation region so the original DS will be less affected by the obstacle. Large values of ρ increase the modulation region, and are thus

helpful for fragile or dangerous obstacles that should not be approached.
The second parameter is the weight w^k which depends on the distance to the k-th obstacle, and thus increase the modulation effect of the closest obstacles.

$$w^k = \prod_{i=1, i\neq k}^{K} \frac{\Gamma^i(\mathbf{x}) - 1}{(\Gamma^k(\mathbf{x}) - 1) + (\Gamma^i(\mathbf{x}) - 1)} \tag{7}$$

Using our new gamma function, $\Gamma^k(\mathbf{x}) - 1$ is the euclidean distance to the obstacle k. When close to an obstacle, w^k approaches 1 and the eigenvalues of equation 6 are unchanged. When far away from this obstacle, w^k is less than 1 and the modulation is reduced.

1. In the main loop of the `modulatedDS()` function, compute the weight w_k for each obstacle and update the modulation M_k accordingly.
   Hint: You can use the `gamma` *array to more efficiently compute the scalar w^k for each obstacle k using the MATLAB function* `prod()`
2. Use the parameter ρ to modulate each obstacle with different sensitivity. This value is accessible with `world.listOfObstacles(k).rho`. You can change this value by passing it as a third (optional) argument when creating an object.

```
% Create a sphere of radius 10cm with default rho value of 1.
myWorld.addEllipsoid([0.1, 0.1, 0.1], [0, 0, 0]);

% Create the same sphere with a rho value of 0.5
myWorld.addEllipsoid([0.1, 0.1, 0.1], [0, 0, 0], 0.5);
```

3. Use the end effector radius $r_{ee} =$ `world.endEffectorRadius` to create a safe spherical space around the end effector. To do this, you add a margin r_{ee} on your gamma value $\Gamma^k(\mathbf{x})$:

$$\Gamma^k_m(\mathbf{x}) = \begin{cases} \Gamma^k(\mathbf{x}) - r_{ee} & \text{if } \Gamma^k(\mathbf{x}) > 1 + r_{ee} \\ 1 & \text{if } (\mathbf{x}) \leq 1 + r_{ee} \end{cases} \tag{8}$$

4. Test your new modulation method by adding multiple obstacles of different shapes and sizes, and try different values of ρ (typically between 0.1 and 10) for each obstacle to simulate more or less fragile obstacles.
   Use the simulation environment to simulate the closed loop behaviour of the robot, and use the keyboard to generate disturbances.
5. Add a very large sphere centered on the robot base to represent the robot workspace boundary. The robot end-effector should be inside this sphere. Does this modulation method works for this scenario ?
6. In section 2 of the file `practical2_main.m`, you can change the flag `usingMPC_ds` to `true` to use our MPC-based DS learned using LPVDS like in practical 1. Does the modulation method still works for non-linear DS ?
   Optional: Try implementing your own linear or non-linear DS and passing it as an argument to the function `modulatedDS()`, like it was done for the MPC-based DS called `ds_control`.

TASK 3 (Optional) Make the obstacles move by defining their velocities, and adapt your modulation to take into account this velocity.

You can set a velocity as a function for each obstacle when instantiating them. All velocity functions will take the obstacle position `x` and the time `t` as inputs and output the velocity as a **column vector**. You do not need to use the position and time in your function, but must keep this general form.

```
% Define a constant velocity in Z axis
vel_func = @(x,t) [0; 0; 1];

% Define the obstacle velocity as a DS
vel_func_2 = @(x,t) (A_obs * x + b_obs);

% Define a time-varying velocity vector
vel_func_3 = @(x,t) [10*cos(50*t); 0; 0 ];
```

This velocity function can be passed as an optional fourth argument when creating an obstacle

```
% Create a sphere with 10cm radius at the origin, a rho value of 0.5, ...
    and a velocity function
myWorld.addEllipsoid([0.1, 0.1, 0.1], [0, 0, 0], 0.5, vel_func);
```

1. Define a linear DS function and use it to make some obstacles move. Run section 3 to see the obstacle move with the robot.
2. Implement a constant and linear velocity function so that your obstacle goes through the initially planned trajectory. Is there a velocity for which the robot can no longer avoid the obstacle ? How can you remedy this ?
3. Include the obstacle velocities in your obstacle-avoidance modulation. To take into account the multiple obstacle velocities, use a weighted average of the velocities, using the weight computed in equation 7. You can get the current velocity of obstacle k with `world.listOfObstacles(k).velocity`.

References

[1] Aude Billard, Sina Mirrazavi, and Nadia Figueroa. *Learning for Adaptive and Reactive Robot Control: A Dynamical Systems Approach*. MIT press, 2022.