Learning for Adaptive and Reactive Robot Control
Instructions for Practical 3 - Simulation

Professor: Aude Billard
Contact: aude.billard@epfl.ch

Introduction

This practical takes place entirely in simulation, using pybullet to simulate the physics of a Franka Emika Panda robot with 7 degrees of freedom. In this practical you will teach a task to a Panda robot using simulated kinesthetic demonstrations and the Dynamical Systems formulation taught throughout the class. You will then control the robot using the learned DS, and play with simple obstacle avoidance. Finally, you will test joint space obstacle avoidance, along with more complex obstacles.

The software structure for the practical is as follows: we are using ROS2, with a node in MATLAB sending cartesian velocity commands computed from a dynamical system, and a C++ node converting these commands into joint torques for the robot. You will work mainly with the MATLAB part of the application, but you will also use the terminal to access ROS functionalities. In the final part of the practical, another ROS node is launched which will enable joint space obstacle avoidance using a variation of Model Predictive Path Integral (MPPI).

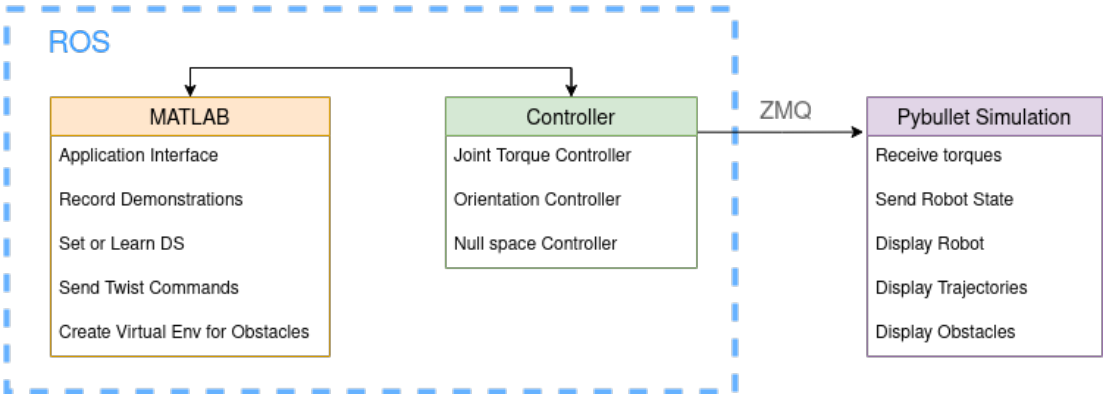


Figure 1: Software Structure

The robot is controlled by sending joint torques commands. Our controller tracks the desired twist (linear and angular velocity) by feedforward linearization and a correction on twist error. We also control the null space of the robot to stabilize the posture around the default configuration $\mathbf{q}_0$:

$$\boldsymbol{\tau} = \boldsymbol{\tau}_f + \mathbf{G}(\mathbf{q}) + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} - \mathbf{D}\dot{\tilde{\mathbf{q}}} + \boldsymbol{\tau}_0 \qquad \text{with} \quad \boldsymbol{\tau}_0 = k_0 * \mathbf{N}^{\#} * (\mathbf{q} - \mathbf{q}_0) \tag{1}$$

with the feedforward terms being $\boldsymbol{\tau}_f$ the estimated joint torques frictions, $\mathbf{G}(\mathbf{q})$ the gravity torques, and $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}}$ the Coriolis torques.

Application Interface

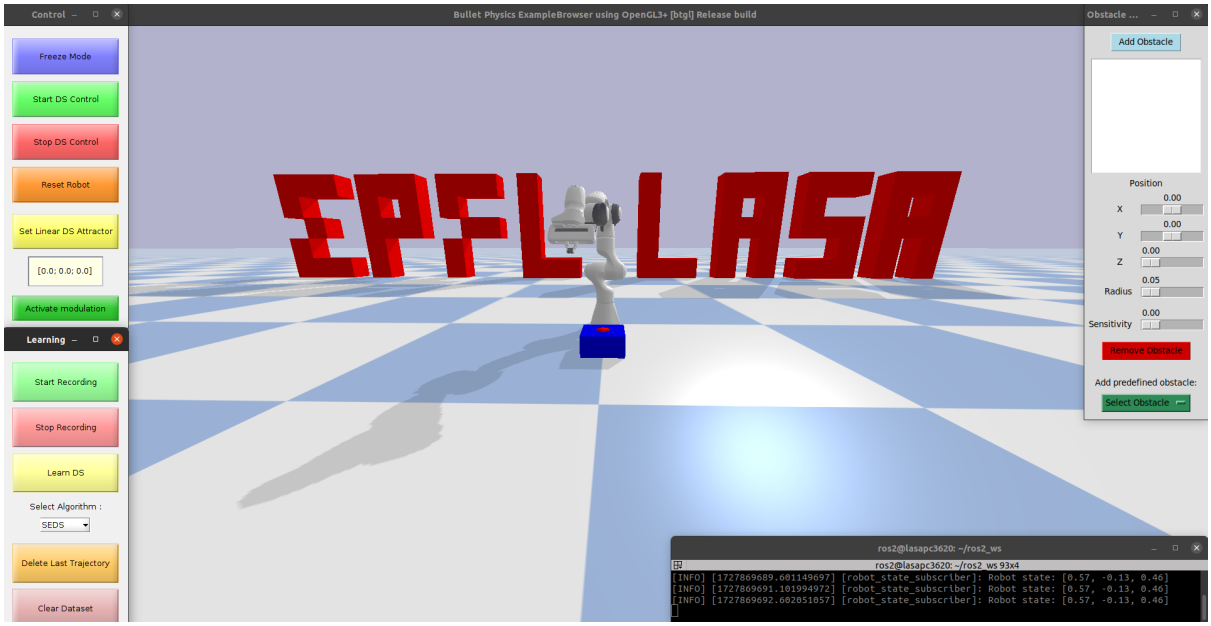


Figure 2: Example of Complete App Interface

During this practical, you will be controlling everything using three graphical interfaces which will be explained throughout the practical. The interfaces allow you to respectively manage the `control` of the robot, the `learning` and the `obstacles` in the environment. The first two are created when launching the `practical3_main.m` script. You will start the last one using a bash terminal when instructed.

Important notes :

- Matlab will display information in the command window whenever a new action begins, feel free to use this to verify that your are using the GUIs correctly.
- You only need to run the main script once, as all functions can be called directly from the GUI and code can be edited without needing to close the app.
- Several ROS commands will also be used in a bash terminal. These commands are specified both in the instructions and the `practical3_main.m` script. You can use CTRL+Shift+V to paste them in the terminal, and CTRL+C to stop the command.
- The simulator might freeze when adding/removing obstacles and trajectories. This is expected, real-time behavior returns promptly afterwards.

Part 0: Setup the practical

Detailed instructions can be found in the `setup.md` file in the practical_3_sim folder. All following commands assume you are already in the practical folder.

To run this practical, you will need :

- A PC with Ubuntu 20.04 or above (or equivalent VM)
- Docker with post-install steps for Linux
- Matlab 2019 or above
- Code provided in the repository

Once, you've downloaded the code from Github, Docker can easily be installed by running the script install_docker.sh located in the docker folder. You can also install it yourself via their website.

```
$ bash docker/install_docker.sh
```

You can verify docker isntalled correctly by running :

```
$ docker run hello-world
```

Build the docker image for the simulation. This can take a couple of minutes :

```
$ cd dependencies/simulator-backend/pybullet_zmq
$ bash build-image.sh
```

Build the docker image for the controller. This should be instantaneous as the image is actually being pulled from Github. You can also build it locally with the option '–local' :

```
$ bash docker/build_docker.sh
```

Once this is done, you should open MATLAB to the *practical_3_sim* folder.

Part 1: Experience compliant robot behavior in simulation

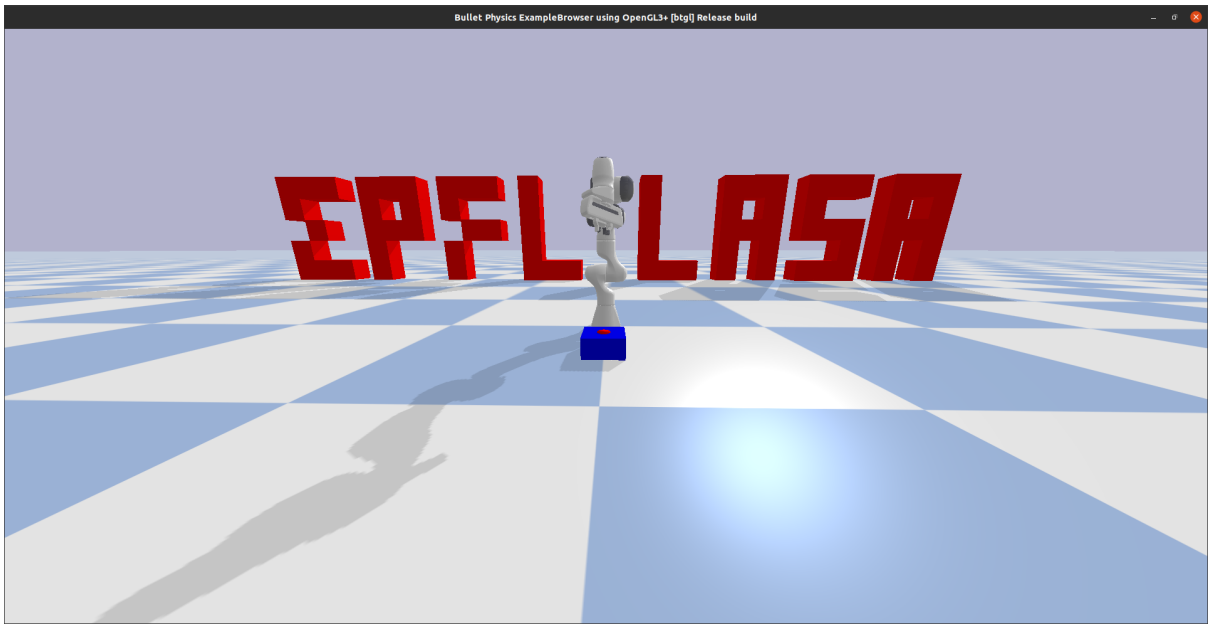


Figure 3: Simulation Start

TASK 1 - Interact with the simulator

1. Open a new bash terminal in the *practical_3_sim* folder and start the Pybullet simulation in its docker environment. Use the following commands to navigate to the correct folder, start the container, then start the simulator :

```
$ cd dependencies/simulator-backend/pybullet_zmq
$ bash run.sh
$ zmq-simulator
```

2. *Notes* :
   - The blue target box in front of the robot is transparent and will not affect the robot's motion.
   - Shadows and lights can be removed/added by pressing 'S'.
3. Zoom with the mouse wheel to move the camera closer to the robot.
4. Move the robot around by clicking and dragging it using your mouse. Answer the following questions :
   - Can you grab the end-effector to move the robot?
   - Can you grab another link to move the robot?
   - Can you drag the robot beyond its physical limits?
   - What happens when you let go of the robot whilst moving it at high speeds? What makes it stop?
5. Move the camera by pressing 'Ctrl' and click-and-dragging with your mouse. Change the camera angle to get a top-down view of the robot.

TASK 2 - Interact with the controlled robot

1. Start the controller which will send torque commands to your robot. Open a new bash terminal in the *practical_3_sim* folder and use the following commands to start the docker container, then launch the required ROS nodes :

```
$ bash docker/start-docker.m
$ ros2 launch matlab_bridge matlabe_bridge.launch.py
```

2. You should immediately notice your robot will start moving without your interaction. Can you explain why?

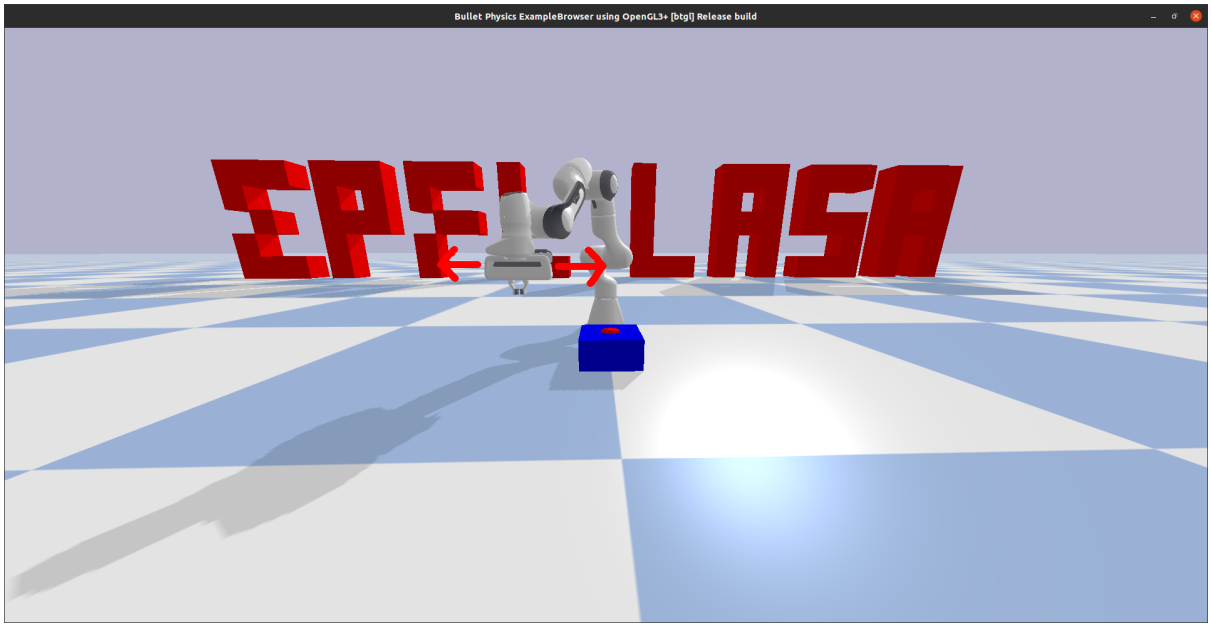


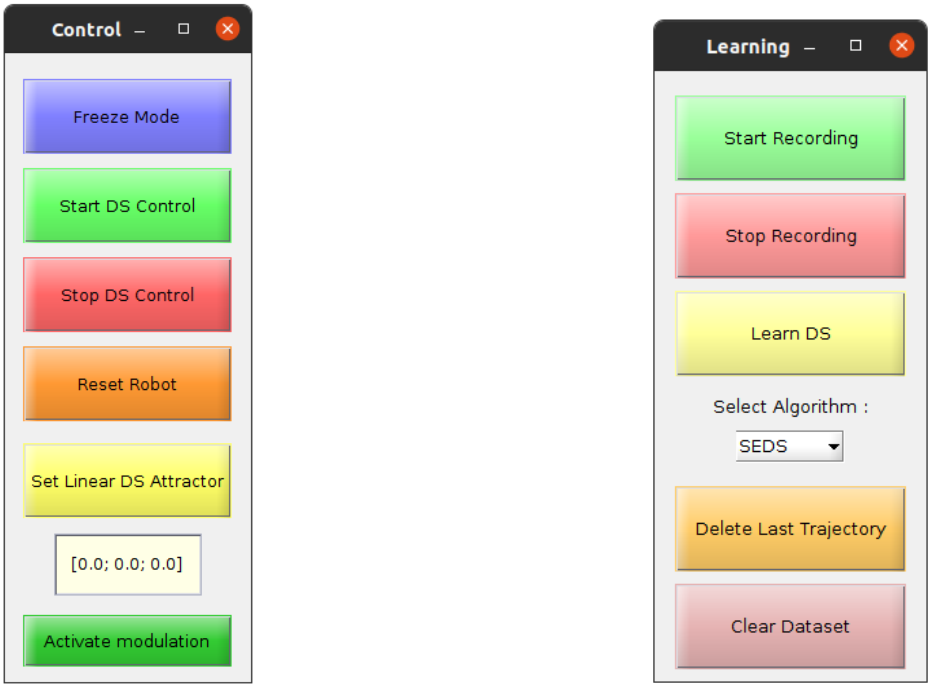
Figure 4: Simulation

3. Position your camera to be facing the robot as shown above and drag the end-effector side to side. Answer the following questions:
   - What happens to the other links?
   - What happens when you release the end-effector?
   - Which degrees of freedom are controlled, and which ones are free to move ?

When not receiving any commands, the controller is in IDLE mode, which controls the robot orientation at the end-effector, along with the null-space of the robot to stabilize the posture around a default configuration. This is active constantly and is done so the robot keeps a similar configuration when moving it by hand or when executing a trajectory.

TASK 3 - Program a linear DS in task space

1. Open a new bash terminal and open Matlab by typing "`matlab`". Then start the script practical3_main.m and wait until the GUI windows opens. To facilitate usage on Ubuntu you can left click the top of each window and select 'Always on Top'.



(a) Control Interface

(b) Learning Interface

Figure 5: Application Interfaces

2. Two MATLAB graphical interfaces will open. Minimize the right-hand one, titled 'Learning'. We will start by using only the 'Control' Interface.
3. A FREEZE mode exists in the controller which will freeze the robot to its current configuration. You can still move it with your mouse, which makes placing the robot to your desired position much easier. Activate the Freeze mode by pressing the corresponding button in the MATLAB window.

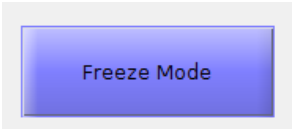


Figure 6: MATLAB GUI - Freeze Button

4. The RESET button will make the robot move back to its default configuration. Once it has reached it, it will return to the IDLE controller from Task 2. Use this button to remove the FREEZE mode.

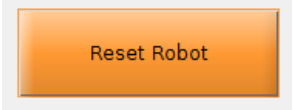


Figure 7: MATLAB GUI - Reset Button

5. Open a new bash terminal in the *practical_3_sim* folder, connect to the docker container and display the end-effector position using the following commands :

```
$ bash docker/start-docker.sh -m connect
$ ros2 run matlab_bridge print_robot_state
```

This runs a small script to display the 3D end effector position in the terminal.

6. Move the end effector of the robot to define the limits of its workspace in the x,y, and z axis.
7. In the Control Interface, set an attractor of your choosing by changing the values in the bottom field. This is an [x,y,z] position vector in the robot reference frame. Once the correct values are entered, press the 'Set Linear DS Attractor' button to create a linear DS with the desired target.

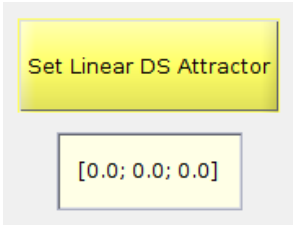


Figure 8: MATLAB GUI - Linear DS

8. Press the "Start DS Control" to start the position controller. A red line will appear, showing the trajectory planned by the DS. The robot will then follow your DS to reach the specified target. Once it is close enough, a green line will appear, showing the trajectory taken by the robot.

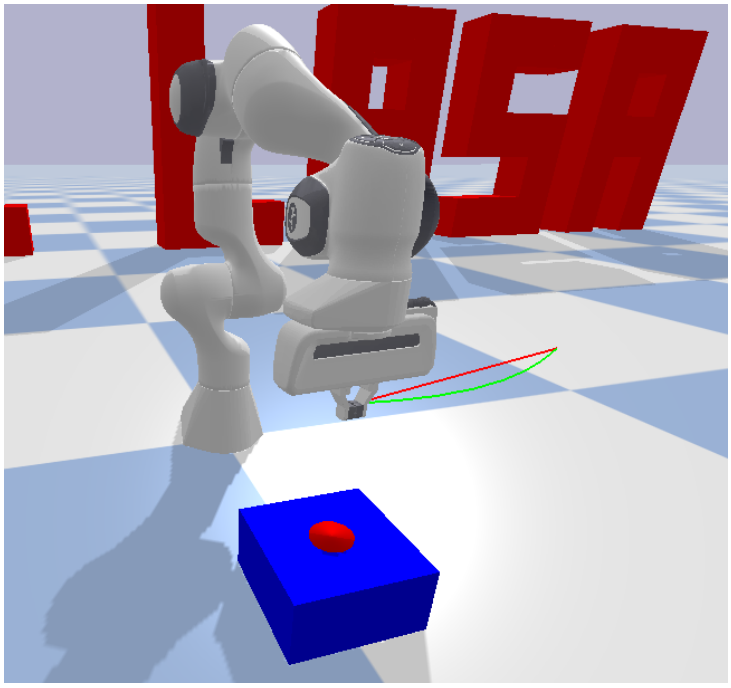


Figure 9: Example of linear DS trajectories: planned (red) and actual (green)

9. Press the "Stop DS Control" button to stop commanding the robot with the DS. This will also make the red and green lines disappear. You can now move the

robot to another position and press the start button again to activate the DS. Use the FREEZE mode to make the robot start exactly where you want it to, it will deactivate automatically when controlling the robot with the DS.

10. Play around with the robot to answer the following questions:
    - Does this control method cover the entire workspace of the robot? How well does it respond to perturbations?
    - What happens if you place the end effector of your robot on the side opposite your attractor, then start the DS control? How could you remedy this issue ?
    - Can the robot reproduce the exact planned trajectory initally given by the DS (shown in red)?
    - What steady-state error does the robot achieve with this controller? Where does this error come from?
    - Is the robot compliant when you perturb it? Can it still reach the target?

Part 2: Learning from Kinesthetic Demonstrations

TASK 1 - Learn a DS with your own demonstrated trajectories

You will now record trajectories using the Matlab interface. To do so, you need to maximize the Learning Interface window.

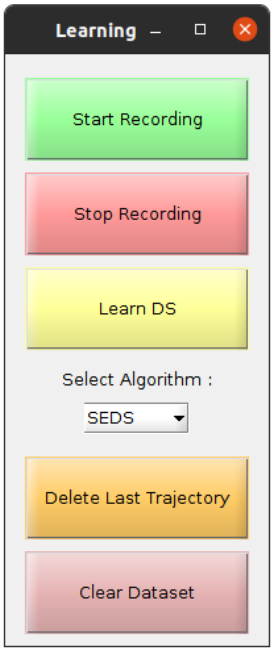


Figure 10: MATLAB GUI - Learning

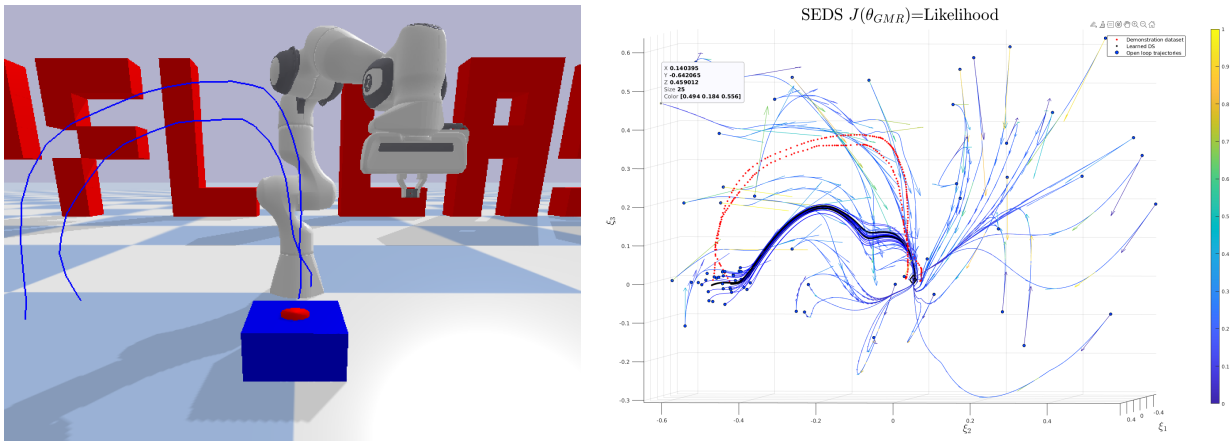
You can record a trajectory by pressing the "Start Recording" button when starting, then the "Stop Recording" button when you finish you demonstration. When recording, the Cartesian position controller is de-activated to allow the user to move the robot around freely. You need to stop the recording and click on 'Start DS Control' to turn the DS position control back on.

1. Record two trajectories starting from close initial positions, in a path similar to the provided example, ending on the red target. When you're done, you can press the "Learn DS" button to start learning with SEDS.

   Notes :

   - Remember, your demonstrations take in the **position** and the **velocity** of the end effector!
   - If necessary, you can remove the last recorded trajectory or every trajectory recorded using the last two buttons of the interface.
   - If you need to modify SEDS parameters, you can do so in the function `learnSEDS()` of the file `DS.m`, and press the "Learn DS" button again to learn a DS with the new parameters until you're satisfied with the results.
   - During the learning, you can follow the progress in the MATLAB Command Window. The algorithm can take a few minutes to learn.
   - You can visualize the learned DS with the figures which appear during the learning; Namely the figure below which displays your demonstrations, along with random trajectories following the learned DS.
   - You can rotate MATLAB plots by clicking on the 'rotate' logo (in blue Fig. 11) in the plot's toolbar. Note the toolbar only becomes visible when your mouse hovers on the figure.

Figure 11: MATLAB Plot - How to rotate



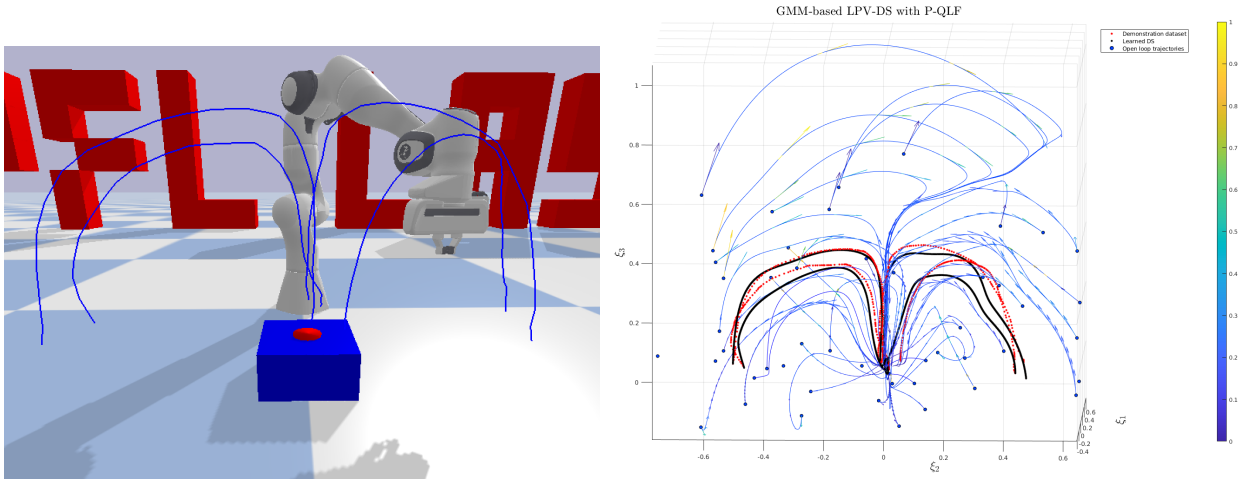
(a) Demonstrations Example (b) SEDS Results

Figure 12: Learning with two trajectories

   - Does SEDS fit your demonstration well?
   - Does it generalizes well on the whole workspace? What would happen if your start from another starting position?

2. Move the robot to the start of your demonstration.

3. In the 'Control' Interface, press the "Start DS Control" button to start following your DS and observe the result. You can see in red the open loop trajectory from your starting point and in green the robot's actual path.
4. Move you robot to a position outside of your demonstrations.
   - Does the planned trajectory reach the target position?
   - Does the robot reach the target position?
5. Without deleting your previous dataset, record new trajectories starting from different initial conditions to span the whole workspace (see figure 13). Change the learning algorithm by selecting 'LPVDS' in the MATLAB 'Learning' Interface, then start the learning again. Similarly, you can change the LPVDS parameters in the function `learnLPVDS()` in the file `DS.m`.



(a) Demonstrations Example (b) LPVDS Results

Figure 13: Learning with several trajectories

6. Press the "Start DS Control" button to start following your DS. Answer the following questions:
   - Can the robot reproduce your demonstrations? What is the main challenge?
   - How many trajectories are needed for generalization? What other factor affects generalization?
   - How does the quality of your DS affect the tracking performance? What could happen if the motion is not generalized over the whole workspace?
7. Several pre-made datasets are provided in the datasets folder.
   - To use them, clear your current dataset and click on 'Learn DS'.
   - To use another dataset, go to l.125 in mainHub.m and modify the name of the loaded dataset. Then do the previous step again.
   - Do you notice improvements in the path planning of your DS?
   - Do the provided demonstrations span the entire workspace ?
   - Did the learning generalize well ?
   - Based on the MATLAB plots, does the planned motion (red line) match the expected outcome from the learned DS?

TASK 2 - Add an obstacle to your environment and modulate your DS to avoid it

1. We will now add an obstacle to the simulation. Open a new bash terminal in the *practical_3_sim* folder, connect to the docker container and start the 'Obstacle Manager' Interface with the followoing commands :

```
$ bash docker/start-docker.sh -m connect
$ ros2 run matlab_bridge ObstacleManagerGUI.py
```

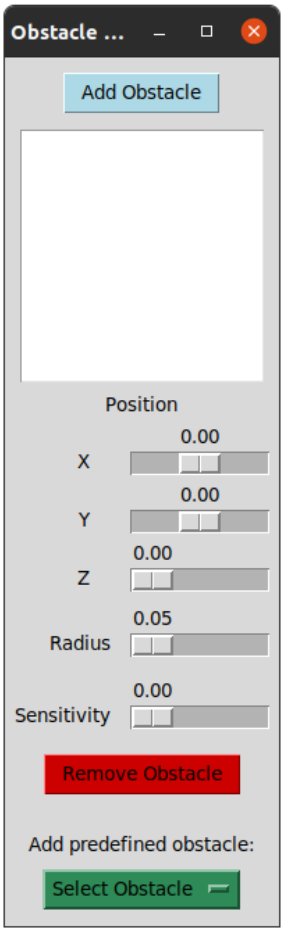


Figure 14: 'Obstacle Manager' Interface

With this interface, you can add, modify and remove as many obstacles as you want in the simulation. Note that spheres are tangible in the simulation, meaning the robot will collide with them without going inside them. Do not mistake collision for avoidance.

2. Familiarize yourself with the obstacle interface :
   - Click the 'Add Obstacle' button.
   - Select the new obstacle in the list and change its position.
   - Change its radius.
   - Remove the obstacle by clicking the corresponding button

3. Your screen should now look like the display in Fig. 2.
4. Add an obstacle and place it on one of your recorded trajectories as shown in Fig. 15.

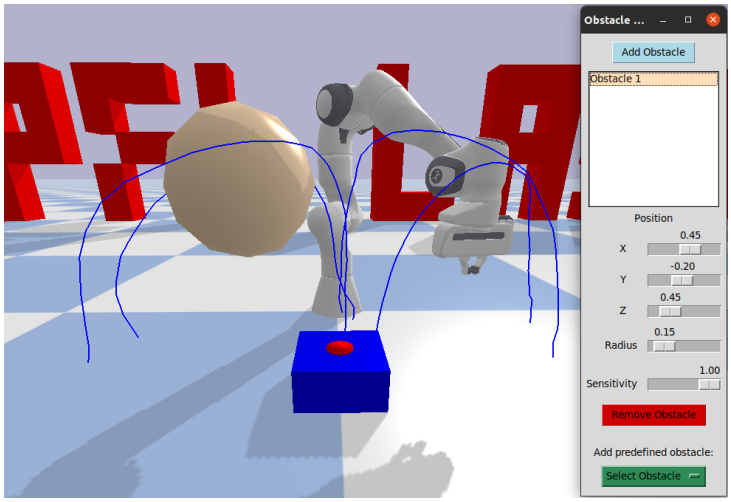


Figure 15: Obstacle for learning trajectories

5. Deactivate the DS modulation so that the planned trajectory ignores the obstacle. The lowest button of the 'Control' Interface should read 'Activate Modulation'

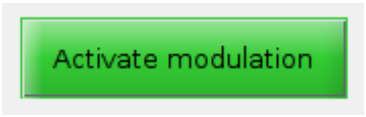


Figure 16: Control Interface - Activate modulation button

6. Place the robot at the start or your trajectory and start the controller. Observe how well it tracks your DS.
   - Does the planned path avoid the obstacle?
   - Can the robot follow that path and avoid the obstacle at the same time?
7. Re-activate the modulation, place the robot back at the start and start the controller again. The lowest button of the 'Control' Interface should now read 'Deactivate Modulation'.

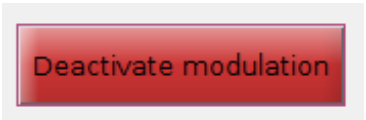


Figure 17: Control Interface - Dectivate modulation button

   - Is the planned path the same as before?
   - Can the robot now follow that path and avoid the obstacle at the same time?
   - What happens when avoiding the obstacle makes the robot leave the workspace of the learned DS? Does it still follow a path similar to what you have taught? Does it reach the target?

8. Reduce the sensitivity of your obstacle and try again.
    - Is the planned path the same as before?
    - Can the robot now follow that path and avoid the obstacle at the same time?
9. Record more trajectories and learn a new DS to improve your path planning. This way, even if the robot moves far from the initial demonstrations to avoid the obstacle, it will manage to reach the target.

Part 3: Obstacle Avoidance

TASK 1 - Avoidance of a single obstacle with a linear DS

1. To better differentiate between the robot behavior due to obstacle avoidance and the one due to the learned DS, we will now remove the learned DS and use a Linear DS, which spans the entire workspace with similar behavior.

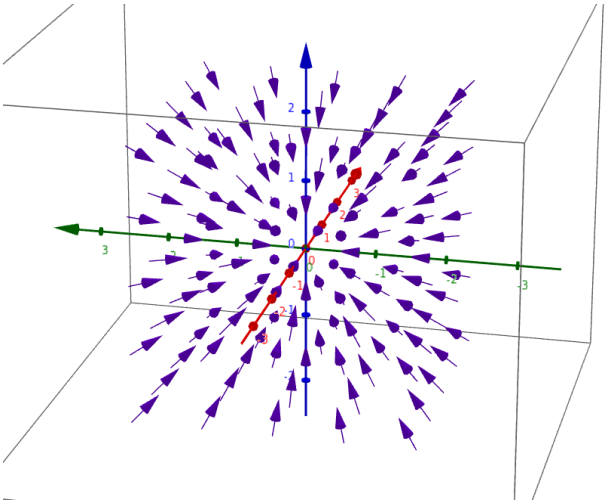


Figure 18: Representation of a 3D linear DS, converging to the origin

2. Set your DS to be linear by using the "Set Linear DS Attractor" button.
3. Place the robot far from the attractor, and move your obstacle to be between the robot and the attractor. De-activated the modulation and start the DS control.
    - Does the planned path avoid the obstacle?
    - Can the robot follow that path and avoid the obstacle at the same time?
4. Now activate its modulation, place the robot back in the same place and start the DS control again.
    - What changes occur in the robot behavior?
    - Does the planned path avoid the obstacle?
    - Can the robot follow that path and avoid the obstacle at the same time?
5. Change the sensitivity `rho` of the obstacle. How does this parameter affect the planned trajectory (in red)?
6. What happens if you place the obstacle on the attractor? Can the robot reach the attractor?

7. You can move an obstacle along an axis using the position sliders. Start your linear DS and slowly move the obstacle in the way of your robot.
   - How does the robot react?
   - Does it avoid the obstacle?
   - Can it still avoid it if you move the obstacle faster?

TASK 2 - Avoidance of multiple obstacles

1. Place your obstacle on the path of the arm of the robot (not the end-effector). Does the robot avoid it? Can you explain why?
2. Add one more obstacle and place it on the robot's path. Can it correctly avoid the obstacles and still reach the target?
3. Add and place more obstacles one by one and verify that the robot can still reach the target and avoid them correctly.

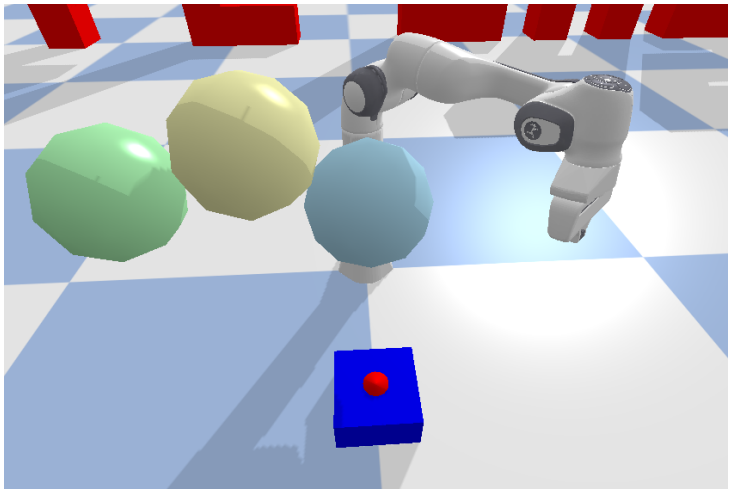


Figure 19: Example of multiple obstacles

4. How many obstacles did you need to stop the robot from reaching the target?

TASK 3 - Avoidance of complex obstacles

For this last task, you will use a controller which avoids obstacles in joint space from [1].

Explanation : We taught the robot collision mesh to a neural network which can be called in real time to give the distance between an obstacle and the robot's closest joint. Combined with joint space control, we can use this to make sure all the joints of the robot avoid the obstacle. To deal with more complex concave obstacles, an MPC planner can be added to test different joint configurations via predictions and find one which can escape concave obstacles. More details can be found in the ICRA poster at the end of this document. A visual explanation can be found in this video.

1. We will not use MATLAB for the end of this practical, so you can close the 'Control' and 'Learning' Interfaces.
2. First, go to the terminal where you launched the ROS controller in Part 1 and kill it using Ctrl+C. Then run this command to launch the joint space controller :

```
$ ros2 launch matlab_bridge joint_space_avoidance.launch.py
```

This will start a joint space controller which automatically switches between two DSs so that the robot arm moves back and forth in a circular arc.

3. Can the new controller handle the set of previous obstacles? What is the difference between the previous cartesian space controller and the current joint space controller?
4. What happens now if you place an obstacle on the trajectory of the arm? Does it manage to avoid it?
5. To start fresh, remove all obstacles. You can also restart both the 'Obstacle Manager' Interface and the simulator.
   4 predefined obstacles are available, respectively 'line','wall', 'ring' and 'tshape'. Unlike the previous obstacles, these have a fixed position and can only be called one at a time. Add a predefined line obstacle by selecting it in the drop down menu at the bottom of the 'Obstacle Manager' Interface.

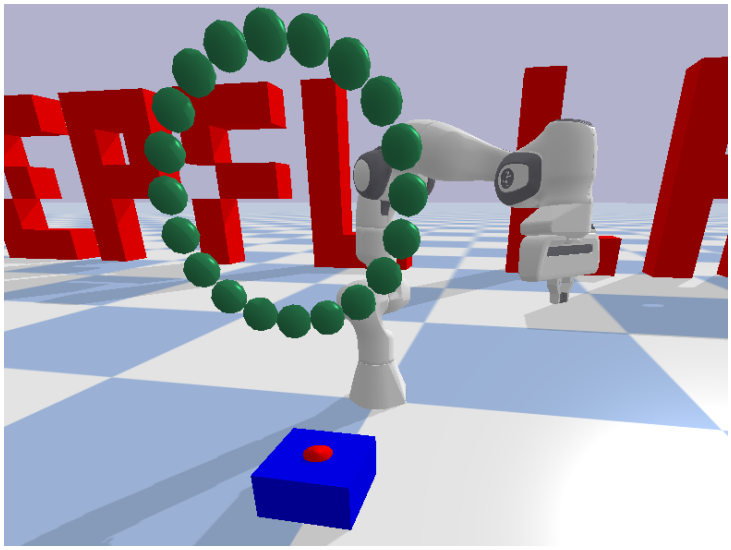


Figure 20: Predefined Obstacle example

6. How does the robot behave around the obstacle? Does it avoid it correctly?
7. Try out the other predefined obstacles. Is there one which the robot does not manage to avoid? Why do you think that is?
8. To handle concave and complex obstacles, you can run the MPC planner to explore possible configurations to escape those situations. Open a new terminal in the *practical_3_sim* folder, connect to the docker container and run the planner with the following commands:

```
$ bash docker/start-docker.m -m connect
$ ros2 run matlab_bridge MppiPlanner.py
```

You will see potential configurations in blue in the simulator

9. Does the robot manage to avoid the concave obstacle and reach its target now?
10. What happens if you add a new sphere (or several) on the robot trajectory? How many new obstacles do you need to trap the robot?

EPFL
LEARNING ALGORITHMS AND SYSTEMS LABORATORY (LASA)

ICRA 2022
IEEE International Conference on Robotics and Automation

Implicit Distance Functions: Learning and Applications in Control

Mikhail Koptev[1], Nadia Figueroa[2], and Aude Billard[1]

[1] EPFL, Lausanne, Switzerland [2] University of Pennsylvania, Philadelphia, Pennsylvania, USA

OVERVIEW

In our approach, we learn the signed distance field as a function of the robot state. That allows replacing traditional minimal distance computation with queries to the learned model. The gradients of the learned model provide repulsive potential directly in the joint space. We demonstrate the application of the learned model to improve the performance of QP and MPC controllers.

DISTANCE EVALUATION & REPULSION VECTOR

Direct computation of minimal distance between the k-th link and point y:

$$d^k_{min}(q,y) = \min_{x \in \mathcal{B}_k} ||f(q,x) - y||$$

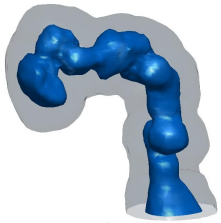
q – joints configuration (m-dim)
$k = 1..K$ – links of the robot
$\mathcal{B}_k$ – set of points on k-th link
$f(q,x)$ – forward kinematics
y – point in the workspace (3-dim)

Precise distances are computationally hard for online evaluation; thus robot geometry is usually approximated conservatively with bounding spheres, capsules, or convex hulls.
Joint-space repulsion requires solving IK: $\Delta q_{rep} \propto J^{\dagger}(x - y)$

Our approach includes learning minimal distance as a function of the robot state q and query workspace point y:

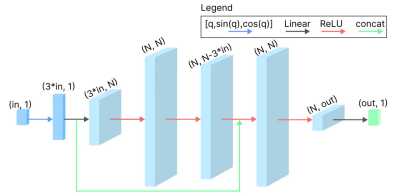
$$\Gamma(q,y) : \mathbb{R}^m \times \mathbb{R}^3 \to \mathbb{R}, \qquad \Delta q_{rep} \propto \nabla_q \Gamma(q,y) = \frac{\partial \Gamma(q,y)}{\partial q}$$

Seamless parallelisation improves the performance (compared to mesh approximations) in certain scenarios.
Repulsive gradient computation is straightforward using backpropagation.



Implicit distance isosurfaces for $\Gamma(q,y)=1$ (solid) and $\Gamma(q,y)=10$ (transparent).

Some examples of the learned distance field are visualised as isosurfaces of 1 and 10 cm levels and compared to the exact robot mesh.



Neural Network architecture. N=256, in=10, out=9. Total weights: 199,915.

NETWORK ARCHITECTURE & LEARNING

We use Multi-Layer Perceptron with five hidden layers. Positional encoding and skip-connection are introduced to improve the regression results.
To collect the dataset, we uniformly sample robot configurations q. For each configuration, we generate 1000 workspace points so that half of them are closer than 3cm to the robot (including interior points), and the other half have a minimal distance from 3 to 100 cm. Final dataset contains 3,000,000 pairs (q,y).
Prediction RMSE on test data: 0.77 (±0.86) cm.
Classification accuracy of *sign(Γ(q,y)-1)* is 0.94 for points closer than 5 cm.

COLLISION-AVOIDANCE CONSTRAINT IN QP IK

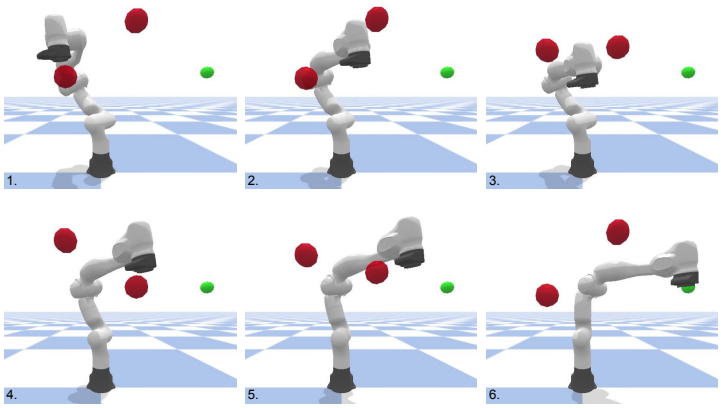
We use learned function $\Gamma(q,y)$ as a collision-avoidance constraint in Quadratic programming inverse kinematics optimization [1]:

$$\min_{\Delta q, \delta} \delta^T Q \delta + \Delta q^T R \Delta q$$

$$\text{s.t.} \quad \begin{cases} f(q) + \dfrac{\partial f(q)}{\partial q} \Delta q = x + \delta \\ q_i^- < q_i + \Delta q_i < q_i^+, \quad i = 1..m \\ -\dfrac{\partial \Gamma_k(q, y_s)}{\partial q}^T \Delta q \leq \Gamma_k(q, y_s) - r_s, \quad k = 1..K. \end{cases}$$

Δq – joint displacement
x – cartesian tasks
δ – slack variable
Q, R – weights
q^+, q^- – joint-limits
y_s – centers of spherical obstacles ($s = 1..S$)
r_s – radii of obstacles ($s = 1..S$)
$\Gamma(q, y_s)$ – learned signed-distance function

The last constraint guarantees collision avoidance, forcing the solver to project joint displacement in a plane locally tangential to the collision boundary.
This QP problem is solved with the code-generation tool CVXGEN [2], with an average frequency of ~200Hz (4.2GHz CPU, collision constraint evaluated on GPU).



A sequence of snapshots for goal (green sphere) reaching scenario while avoiding collisions with moving obstacles (red spheres).

SAMPLING HEURISTIC IN MPPI

We leverage the seamless parallelization of our method by using it in a sampling-based MPC control scheme. We use the gradient of the learned function to reproject samples leading to collisions and inject corrected samples into the MPPI update rule for the policy:

$$\mu_{t,h} = (1 - \alpha_\mu)\mu_{t-1,h} + \alpha_\mu \frac{\sum_{i=1}^N w_i u_{i,h}}{\sum_{i=1}^N w_i}.$$

Knowing repulsion in joint-space, we can obtain projection and the tangential component of the sampled accelerations:

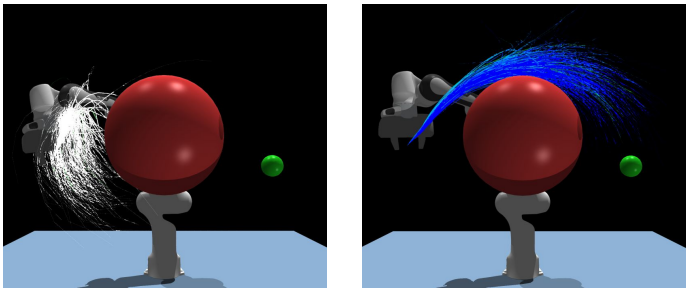
$$u^{proj}_{i,h} = \frac{\nabla_q \Gamma \cdot u_{i,h}}{||\nabla_q \Gamma|| \, ||u_{i,h}||} \nabla_q \Gamma, \qquad u^{\tau}_{i,h} = u_{i,h} - u^{proj}_{i,h}.$$

Finally, reprojected sampled accelerations are defined as:

$$u^*_{i,h} = (1 - f_c) u_{i,h} + f_c u^{\tau}_{i,h}.$$

Above, f_c is a parameter that depends on the distance to collision and cosine similarity between joint velocity and repulsive vector in joint space.
Injecting such samples allows for better exploration in the vicinity of obstacles and faster trajectory convergence. The comparison of two methods is provided below:

Method	Success rate	Iterations	Time, s	Freq., Hz
Original [3]	0.96	663	5.43	**121**
Our modification	**1.00**	**296**	**2.78**	106



An example comparing sampling in the standard MPPI implementation (left) and our modification with guided sampling via reprojecting heuristic (right).

[1] M. Koptev, N. Figueroa, and A. Billard, "Real-time self-collision avoidance in joint space for humanoid robots," IEEE Robotics and Automation Letters, vol. 6, no. 2, pp. 1240–1247, 2021.
[2] J. Mattingley and S. Boyd, "CVXGEN: A code generator for embedded convex optimization," Optimization and Engineering, vol. 13, March 2012.
[3] M. Bhardwaj, B. Sundaralingam, A. Mousavian, N. D. Ratliff, D. Fox, F. Ramos, and B. Boots, "STORM: An integrated framework for fast joint-space model-predictive control for reactive manipulation," in 5th Annual Conference on Robot Learning, 2021.

References

[1] Mikhail Koptev, Nadia Figueroa, and Aude Billard. Neural joint space implicit signed distance functions for reactive robot manipulator control. *IEEE Robotics and Automation Letters*, 2022.

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