# Learning Robots Exercise 4

TECHNISCHE UNIVERSITÄT DARMSTADT

**Dominik Marino - 2468378, Victor Jimenez - 2491031, Daniel Piendl - 2586991**
**February 1, 2021**

## Contents

## 1 Trajectory Generation with Dynamical Systems

### 1.1 Similarities to a PD controller

$$
\begin{aligned}
\ddot{y} &= \tau^2(\alpha(\beta(g-y) - \frac{\dot{y}}{\tau}) + f_w(z)) = \\
&= \tau^2(\alpha\beta(g-y) - \alpha\frac{\dot{y}}{\tau} + f_w(z)) = \\
&= \alpha\beta\tau^2(g-y) - \alpha\tau^2\frac{\dot{y}}{\tau} + \tau^2 f_w(z) = \\
&= \alpha\beta\tau^2(g-y) - \alpha\tau\dot{y} + \tau^2 f_w(z) \\
&= \alpha\beta\tau^2(g-y) + \alpha\tau(0-\dot{y}) + \tau^2 f_w(z) = \\
&= \underbrace{\alpha\beta\tau^2}_{K_P}(\underbrace{g}_{y_z^{des}} - \underbrace{y}_{y_z}) + \underbrace{\alpha\tau}_{K_D}(\underbrace{0}_{\dot{y}_z^{des}} - \underbrace{\dot{y}}_{z}) + \underbrace{\tau^2 f_w(z)}_{u_{ff}} \\
&= K_P(y_z^{des} - y_z) + K_D(\dot{y}_z^{des} - \dot{y}_z) + u_{ff}
\end{aligned}
$$

### 1.2 Stability

- A DMP (Dynamic Motor Primitives) is a stable estimator of a dynamic systems to smooth convergence to the goal state. DMPs are able to represent nonlinear movements without losing stability of the behavior[1]. The equilibrium point would be the goal position. We can show this, if we solve the second equation:

$$
\begin{aligned}
\dot{z} &= -\tau\alpha_z z \\
z(t) &= z_0 e^{\lambda t} \overset{t\to\infty}{=} z_0 e^{-\tau\lambda t} \longrightarrow f_w(z) \overset{t\to\infty}{=} 0
\end{aligned}
$$

  the third equation from the exercise sheet goes direktly to zero for $t \longrightarrow \infty$ or set $\dot{y}$ and $\ddot{y}$ to zero. Now we can solve the equation on y to get equilibrium point. From the lecture we also know that "the DMP is stable per construction as the forcing function vanishes" and the DMP are a normal PD controller and thus is stable for $t \longrightarrow \infty$.

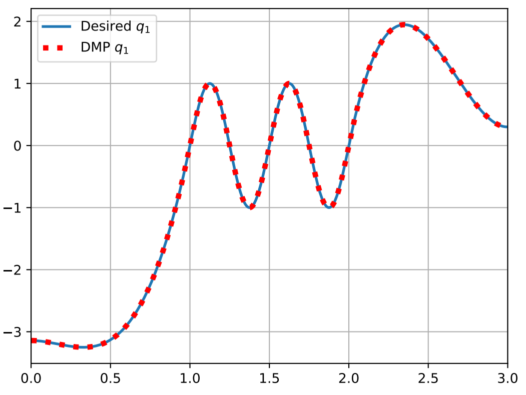
$$
\Longrightarrow t \longrightarrow \infty, f_w \longrightarrow 0 \text{ as } z \longrightarrow 0
$$

[1] Book: An Algorithmic Perspective on Imitation Learning, Page 66 - 69

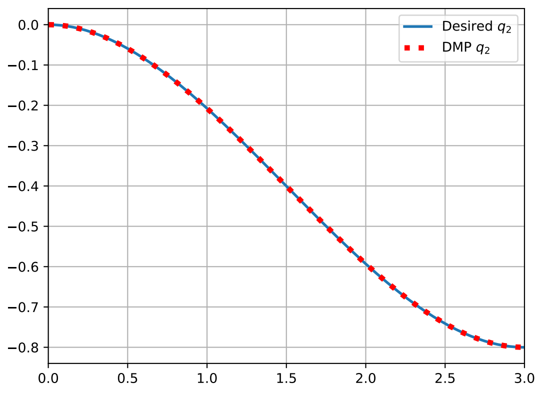
## 1.3 Double Pendulum - Training

- To calculate $f_z(w)$ we have to rewrite the formula:

$$\ddot{y} = \tau^2(\alpha(\beta(g-y) - \frac{\dot{y}}{\tau}) + f_w(z)) =$$
$$\frac{\ddot{y}}{\tau^2} = \alpha(\beta(g-y) - \frac{\dot{y}}{\tau}) + f_w(z)$$
$$f_w(z) = \frac{\ddot{y}}{\tau^2} - \alpha(\beta(g-y) + \frac{\dot{y}}{\tau})$$



(a) $y = x$



(b) $y = x$

```python
def dmpTrain(q, qd, qdd, dt, nSteps):
    params = dmpParams()
    # Set dynamic system parameters
    params.alphaz = 3 / (nSteps * dt - dt)
    params.alpha = 25
    params.beta = 6.25
    params.Ts = nSteps * dt - dt
    params.tau = 1
    params.nBasis = 50
    params.goal = np.asarray(q[:, -1])  # np.asarray([0.3, -0.8])

    # Daniel: This should actually be Psi, right?
    Phi = getDMPBasis(params, dt, nSteps)

    # Compute the forcing function
    ft = 1 / params.tau ** 2 * qdd - 1 / params.tau * (0 - qd) \
         - params.alpha * params.beta * (params.goal.reshape(2, 1) - q)  #

    # Learn the weights
    sigma = 10 ** -12
    params.w = np.linalg.inv(Phi.transpose().dot(Phi) + sigma ** 2 * np.identity(Phi.shape[1])).dot(Phi.transpose()) \
        .dot(ft.transpose())
```

Listing 1: dmpTrain

```python
def dmpCtl(dmpParams, psi_i, q, qd):
    f_w = psi_i.transpose().dot(dmpParams.w)

    K_d = dmpParams.tau * dmpParams.alpha
    K_p = dmpParams.tau ** 2 * dmpParams.alpha * dmpParams.beta
    u_ff = dmpParams.tau ** 2 * f_w
    q_des = dmpParams.goal

    qdd = K_p * (q_des - q) + K_d * (-qd) + u_ff

    return qdd
```

Listing 2: cmpCtl

## 1.4 Double Pendulum - conditioning on the Final Position

In Figure 2 we see how the angles $q_1$ and $q_2$ of the double pendulum behave for the different goals $q_{t=end} = (0, 0.2)$ and $q_{t=end} = (0.8, 0.5)$. In both cases the behaviour of the angle $q_1$ resembles the imitated data only that one both cases $q_1$ is shifted by a small amount. For both goals $q_2$ has a similar behaviour to one another but in both cases the values don't resemble the values of the imitation data. At the beginning the values rise fast and after 0.5 seconds the curves look smooth, just as the imitation data. Also in this case the goal is reached in the given time. From the results one can say, that using the force function with radial basis function works the way it should. One thing to be aware of are the rapid changes the angles $q_1$ and $q_2$ experience, this could lead to damages in the robot.

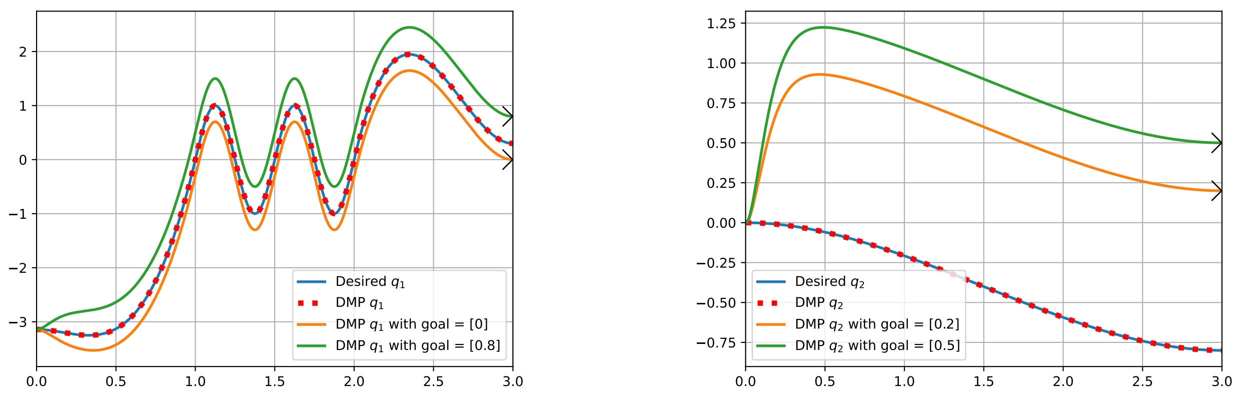


Figure 2: Values of $q_1$ and $q_2$ for different goals

## 1.5 Double Pendulum - Temporal Modulation

The behaviour of the double pendulum using different temporal scaling factors are shown in Figure 3. They all share the same goal and depending on the scale factor the values over time changes differently. For a higher $\tau$ the pendulum reaches faster the goal but on the other hand $q_1$ oscillates more during the simulation. For a $\tau = 0.5$ the goal isn't reach within 3 seconds but the values for both angles behave much more smoothly than in the other 2 cases.

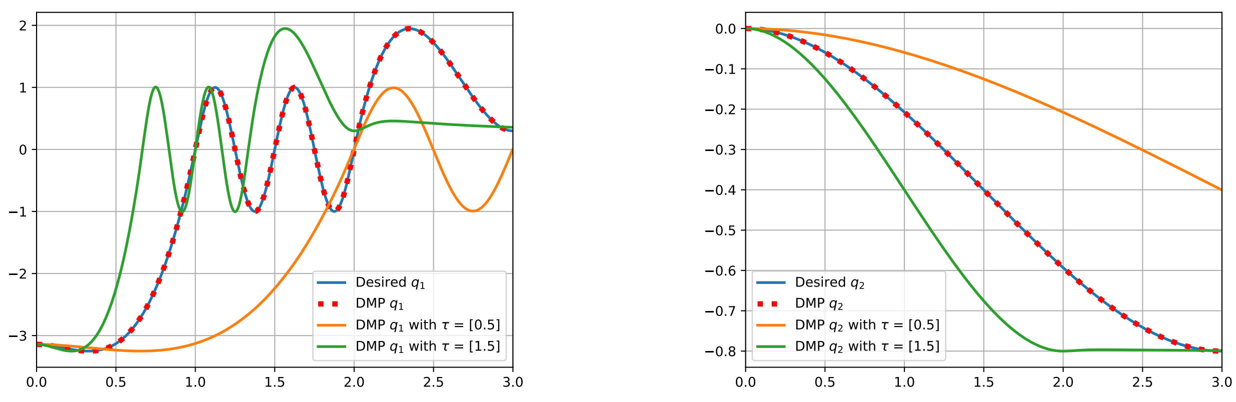


Figure 3: Values of $q_1$ and $q_2$ for different time scaling

## 1.6 Probabilistic Movement Primitives - Radial Basis Function

Figure 4 shows $N = 30$ radial basis functions for the probabilistic movements primitives. They are distributed uniformly in the time interval $[0 - 2b, T + 2b]$

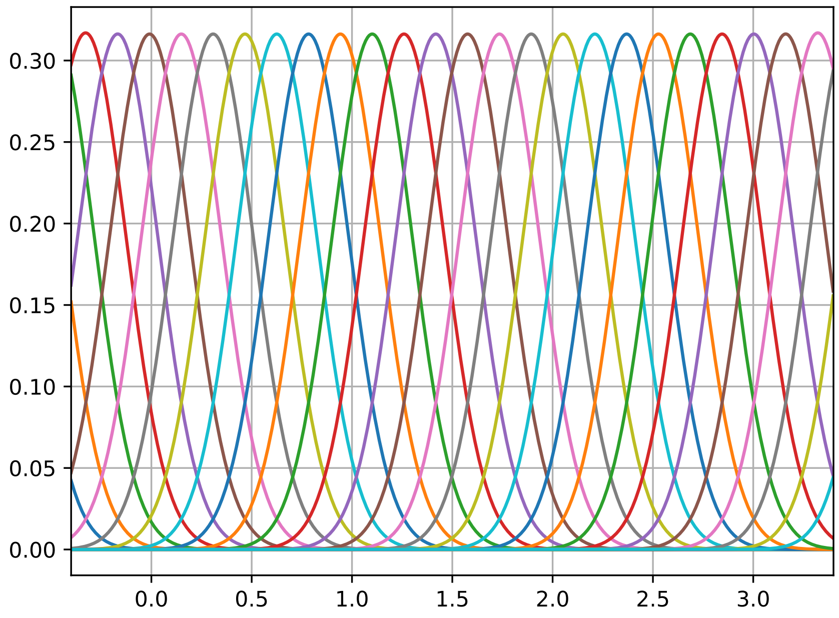


Figure 4: Radial basis functions for the probabilistic movements primitives

```python
def getProMPBasis(dt, nSteps, n_of_basis, bandwidth):
    nBasis = n_of_basis
    time = np.arange(dt, nSteps * dt, dt)
    assert nSteps == len(time)
    Ts = nSteps * dt - dt

    C = np.zeros(nBasis)  # Basis function centres
    H = np.zeros(nBasis)  # Basis function bandwidths

    # for i in range(nBasis):
    #     C[i] = -2 * bandwidth + (Ts + 4 * bandwidth) / nBasis * i
    C = np.linspace(0 - 2 * bandwidth, Ts + 2 * bandwidth, nBasis)
    # for i in range(nBasis):
    #     H[i] = bandwidth ** 2

    Phi = np.zeros((nBasis, nSteps))

    for k, time_k in enumerate(time):
        for j in range(nBasis):
            Phi[j, k] = np.exp(-.5 * (time_k - C[j]) ** 2 / bandwidth ** 2)  # Basis function activation over time
    for k in range(Phi.shape[1]):
        Phi[:, k] = (Phi[:, k]) / np.sum(Phi[:, k])  # Normalize basis functions and weight by canonical state

    return Phi
```

Listing 3: getProMPBasis.py

## 1.7 Probabilistic Movement Primitives - Training

Figure 5 shows the 45 sampled trajectories, their mean (it is mostly hidden under the imitated trajectory) and the standard deviation. It also shows the imitated trajectory when using 30 basis functions. The imitated trajectory almost exactly matches the mean observed trajectory.

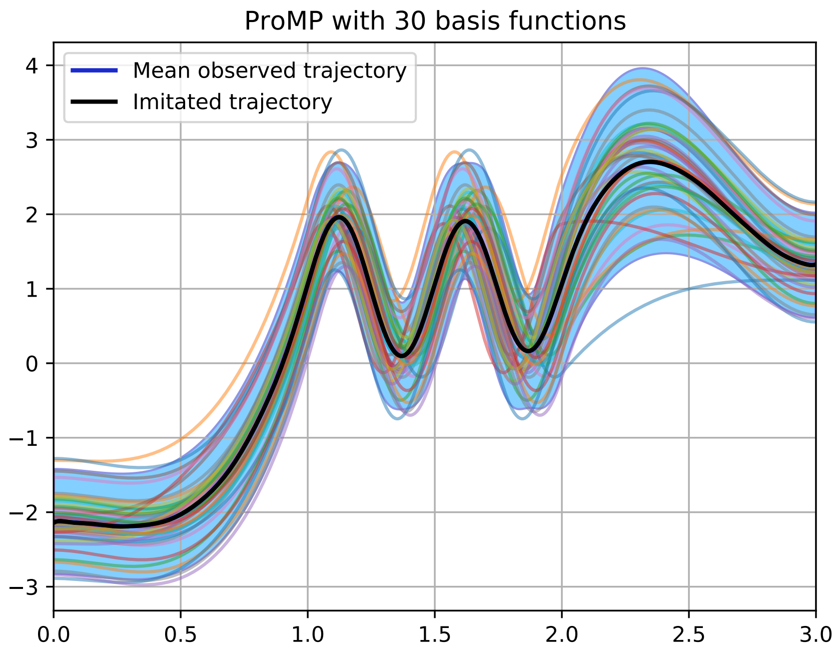


Figure 5: ProMP with 30 basis functions. X-axis shows $t$ and the y-axis shows $q$.

```python
def proMP(nBasis, condition=False):
    dt = 0.002
    time = np.arange(dt, 3, dt)
    nSteps = len(time)
    data = getImitationData(dt, time, multiple_demos=True)
    q = data[0]
    qd = data[1]
    qdd = data[2]

    bandwidth = 0.2
    Phi = getProMPBasis(dt, nSteps, nBasis, bandwidth)

    sigma = 10 ** -12

    # https://www.ias.informatik.tu-darmstadt.de/uploads/Team/AlexandrosParaschos/promps_auro.pdf
    # Eq. 13
    w = np.linalg.inv(Phi @ Phi.transpose() + sigma ** 2 * np.identity(Phi.shape[0])) @ Phi @ q.transpose()

    mean_w = np.mean(w, axis=1)
    cov_w = np.cov(w, rowvar=True)
    mean_traj = np.mean(q, axis=0)
    std_traj = np.std(q, axis=0)

    if not condition:
        plt.figure()
        plt.fill_between(time, mean_traj - 2 * std_traj, mean_traj + 2 * std_traj, alpha=0.5, edgecolor='#1B2ACC',
                         facecolor='#089FFF')

        # Plot all trajectories
        plt.plot(time, q.T, alpha=.5)

        plt.plot(time, mean_traj, color='#1B2ACC', label='Mean observed trajectory', linewidth=2)
        # Plot learned trajectory
        plt.plot(time, Phi.transpose() @ mean_w, label="Imitated trajectory", linewidth=2, color='black')

        title = 'ProMP with ' + str(nBasis) + ' basis functions'
        plt.title(title)
        plt.legend()
        plt.grid()
        plt.xlim((0, 3))

    # Conditioning
    else:
        y_d = 3
        Sig_d = 0.0002
        t_point = np.int(2300 / 2)

        tmp = np.dot(cov_w, Phi[:, t_point]) / (Sig_d + np.dot(Phi[:, t_point].T, np.dot(cov_w, Phi[:, t_point])))

        cov_w_new = cov_w - tmp.reshape((30, 1)) @ (Phi[:, t_point].dot(cov_w)).reshape((1, 30))
        mean_w_new = mean_w + tmp * (y_d - Phi[:, t_point].T.dot(mean_w))
        mean_traj_new = Phi.T @ mean_w_new
        std_traj_new = np.sqrt(np.diagonal(sigma * np.eye(Phi.shape[1]) + Phi.T @ cov_w_new @ Phi))

        sample_traj = np.dot(Phi.T, np.random.multivariate_normal(mean_w_new, cov_w_new, 10).T)

        plt.figure()
        plt.fill_between(time, mean_traj - 2 * std_traj, mean_traj + 2 * std_traj, alpha=0.5, edgecolor='#1B2ACC',
                         facecolor='#089FFF')
        plt.fill_between(time, mean_traj_new - 2 * std_traj_new, mean_traj_new + 2 * std_traj_new, alpha=0.5,
                         edgecolor='#CC4F1B', facecolor='#FF9848')
        plt.plot(time, sample_traj, alpha=.5)
        plt.plot(time, mean_traj, color='#1B2ACC', linewidth=2, label='Mean old trajectory')
        plt.plot(time, mean_traj_new, label='Mean new trajectory', linewidth=2, color='black')

        plt.scatter(t_point * dt, y_d, label="Via point")

        plt.xlim((0, 3))
        title = 'ProMP after conditioning with new sampled trajectories'
        plt.title(title)
        plt.grid()
        plt.legend()

    plt.savefig(title + ".pdf")
    plt.draw_all()
    plt.pause(0.001)
```

Listing 4: proMP

## 1.8 Probabilistic movement Primitives - Number of Basis Functions

Figure 6 and figure 7 show the trained ProMPs with 20 and 10 RBFs, respectively. When using the 20 RBFs, the fit is slightly worse, as the imitated trajectory deviates from the mean trajectory at $t = 1.1$ more than it did with 30 RBFs. Nonetheless, the trajectory is still imitated very precisely. However, when using 10 RBFs, the imitated trajectory does not fit the observed trajectory very well. The low number of basis functions is not able to reproduce the periodic movement from $t = 1$ to approx. $t = 2.3$. Most likely, due to the low bandwidth, not enough basis functions are active in those time steps to reproduce the complex trajectory. To improve the results while still using 10 RBFs, higher bandwidths could be tested and their performance evaluated.

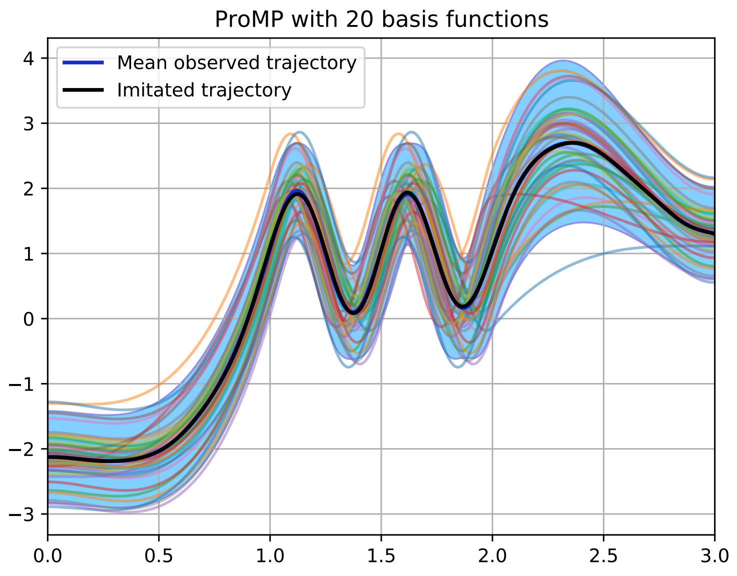


Figure 6: ProMP with 20 basis functions. X-axis shows $t$ and the y-axis shows $q$.

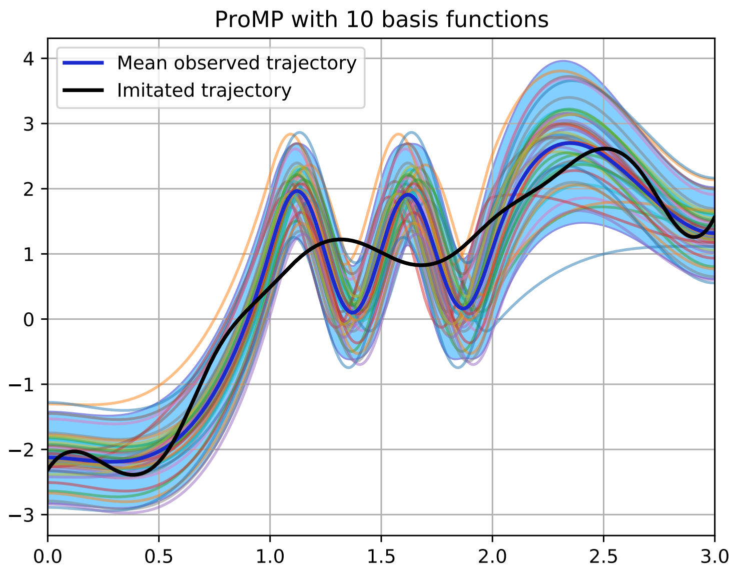


Figure 7: ProMP with 10 basis functions. X-axis shows $t$ and the y-axis shows $q$.

## 1.9 Probabilistic Movement Primitives - Conditioning

Figure 8 shows the mean old observed trajectory that was already utilized in the previous two tasks as well as the new imitated trajectory that has a new via-point at $t = 2.3$ and $y = 3$. The standard deviation of the new trajectory generally follows the standard deviation of the old trajectory but is slightly narrower. Also the mean of the two trajectories is very similar until $t = 2$. In the vicinity of the via-point, the standard deviation of the new trajectory becomes very small and the imitated trajectory deviates from the old observed trajectory as it tries to meet the desired point.

For the computation of the new mean and standard deviation, formulas from Paraschos et al.: Using Probabilistic Movement Primitives in Robotic[2], were utilized.

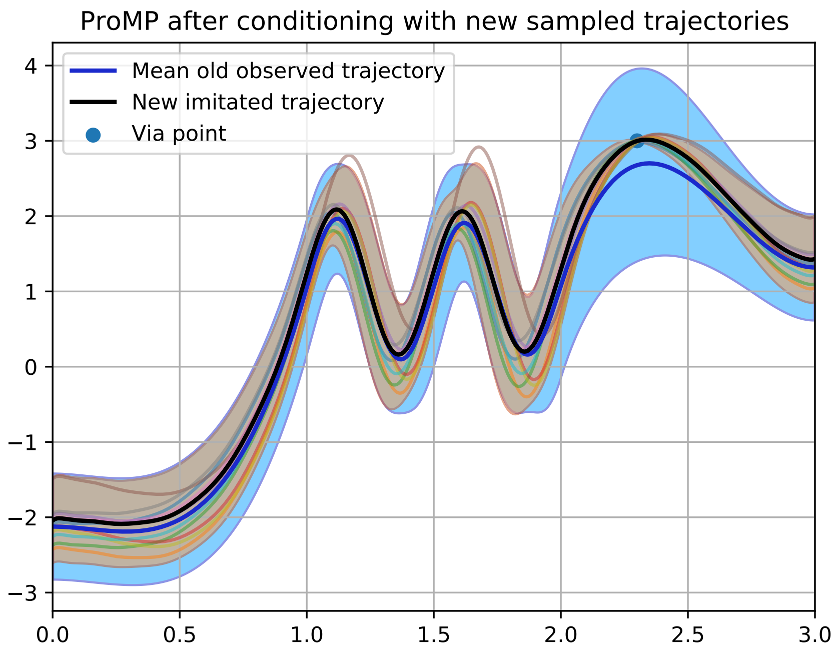


Figure 8: ProMP after conditioning with new sampled trajectories. X-axis shows $t$ and the y-axis shows $q$.

[2]Page 11, `https://www.ias.informatik.tu-darmstadt.de/uploads/Team/AlexandrosParaschos/promps_auro.pdf`