Robot Learning

Winter Semester 2017/2018, Homework 2

Prof. Dr. J. Peters, D. Tanneberg, M. Ewerton

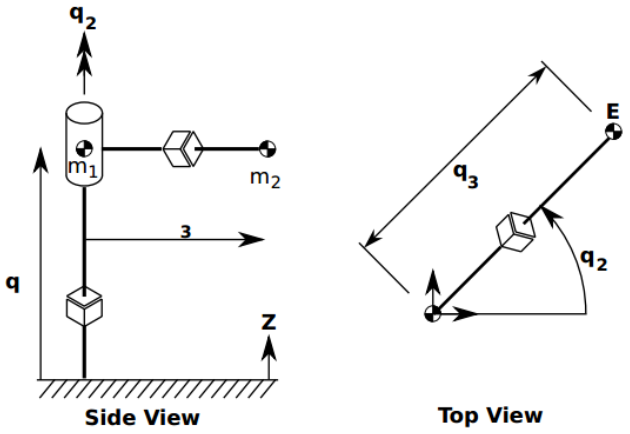
TECHNISCHE UNIVERSITÄT DARMSTADT

Total points: 46 + 20 bonus
Due date: Wednesday, 20 December 2017 (before the lecture)

Name, Surname, ID Number **Markus Lamprecht, 2424163** **Moritz Knaust, 2430801**

Problem 2.1 Model Learning [24 Points + 4 Bonus]

The new and improved Spinbot 2000 is a multi-purpose robot platform. It is made of a kinematic chain consisting of a linear axis q_1, a rotational axis q_2 and another linear axis q_3, as shown in the figure below. These three joints are actuated with forces and torques of u_1, u_2, and u_3. Different end effectors, including a gripper or a table tennis racket, can be mounted on the end of the robot, indicated by the letter E. Thanks to Spinbot's patented SuperLight technology, the robot's mass is distributed according to one point mass of m_1 at the second joint and another point mass of m_2 at the end of the robot E.



The inverse dynamics model of the Spinbot is given as

$$
\begin{aligned}
u_1 &= (m_1 + m_2)(\ddot{q}_1 + g), \\
u_2 &= m_2\left(2\dot{q}_3\dot{q}_2 q_3 + q_3^2\ddot{q}_2\right), \\
u_3 &= m_2\left(\ddot{q}_3 - q_3\dot{q}_2^2\right).
\end{aligned}
$$

We now collected 100 samples from the robot while using a PD controller with gravity compensation at a rate of 500Hz. The collected data (see `spinbotdata.txt`) is organized as follows

	t_1	t_2	t_3	...
$q_1[m]$				
$q_2[rad]$				
$q_3[m]$				
...				
$\ddot{q}_3[m/s^2]$				
$u_1[N]$				
$u_2[Nm]$				
$u_3[N]$				

Given this data, you want to learn the inverse dynamics of the robot to use a model-based controller. The inverse dynamics of the system will be modeled as $\boldsymbol{u} = \boldsymbol{\phi}(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}})^{\mathsf{T}} \boldsymbol{\theta}$, where $\boldsymbol{\phi}(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}})$ are features and $\boldsymbol{\theta}$ are the parameters.

a) **Problem Statement [2 Points]**

What kind of machine learning problem is learning an inverse dynamics model? What kind of information do you need to solve such a problem?

Theory:
Learning Inverse Dynamics is quite interesting, as rigid body("Starrkörper") dynamics are lacking of good friction models and are therefore incomplete. Moreover dynamic parameters are difficult to estimate. The task of an inverse model is to predict the action needed to reach a desired state. This means a Inverse Matrix can be learned directly.
Inverse Dynamics: $u = f\left(q, \dot{q}, \ddot{q}_{ref}\right)$, $\ddot{q}_t^{des} = K_P\left(q_t^{des} - q_t\right) + K_D\left(\dot{q}_t^{des} - \dot{q}_t\right)$ *(PD-Controller)*
Task:
Kind of machine learning problem:
Learning an inverse dynamics model is a kind of supervised learning (in model learning) which can be solved by linear regression. It should be noted that only if the system is an invertible function an inverse dynamics model can be learned.
Kind of information:
To learn an inverse dynamics model you need many measurements of the output values and input values. For the inverse model above the output values are: (u_i) *and the input values are:* $(q_i, \dot{q}_i, \ldots)$.

b) **Assumptions [5 Points]**

Which standard assumption has been violated by taking the data from trajectories?

The data of the trajectories should be Independent and identically distributed(IID).
A collection of random variables is IID if each random variable has the same probability distribution as the others and all are mutually independent.

c) **Features and Parameters [4 Points]**

Assuming that the gravity g is unknown, what are the feature matrix $\boldsymbol{\phi}$ and the corresponding parameter vector $\boldsymbol{\theta}$ for $\boldsymbol{u} = [u_1, u_2, u_3]^\mathsf{T}$? (Hint: you do not need to use the data at this point)

What are the parameters?

$$
\begin{aligned}
u_1 &= (m_1 + m_2)(\ddot{q}_1 + g), \\
u_2 &= m_2\left(2\dot{q}_3\dot{q}_2 q_3 + q_3^2\ddot{q}_2\right), \\
u_3 &= m_2\left(\ddot{q}_3 - q_3\dot{q}_2^2\right).
\end{aligned}
$$

$$
\underset{3\times 1}{y_1} = \phi^T\theta = \underbrace{\begin{bmatrix} \ddot{q}_1 & 0 & 1 \\ 0 & 2\dot{q}_3\dot{q}_2 q_3 + q_3^2\ddot{q}_2 & 0 \\ 0 & \ddot{q}_3 - q_3\dot{q}_2^2 & 0 \end{bmatrix}}_{3\times 3} \underbrace{\begin{bmatrix} m_1 + m_2 \\ m_2 \\ m_2 g + m_1 g \end{bmatrix}}_{3\times 1}
$$

For n measurements:

$$
\underset{3n\times 1}{Y} = \begin{bmatrix} y_1 \\ \dots \\ y_n \end{bmatrix}
$$

$$
\underset{3n\times 3}{\Phi} = \begin{bmatrix} \phi_1^T \\ \dots \\ \phi_n^T \end{bmatrix}
$$

Linear Regression:

$$
\underset{3\times 1}{\theta} = \left(\underset{3\times 3n}{\Phi^T}\ \underset{3n\times 3}{\Phi}\right)^{-1}\ \underset{3\times 3n}{\Phi^T}\ \underset{3n\times 1}{Y}
$$

d) **Learning the Parameters [2 Points]**

You want to compute the parameters $\boldsymbol{\theta}$ minimizing the squared error between the estimated forces/torques and the actual forces/torques. Write down the matrix equation that you would use to compute the parameters. For each matrix in the equation, write down its size.
Then, compute the least-squares estimate of the parameters $\boldsymbol{\theta}$ from the data and report the learned values.

Task:

$$
min_\theta \sum_{i=1}^{n}\left(u_{meas,i} - u_{pred,i}\right)^2 = min_\theta \sum_{i=1}^{n}\left(u_{meas,i} - \phi_i^T\theta\right)^2
$$

$$
\underset{3\times 1}{\theta} = \left(\underset{3\times 3n}{\Phi^T}\ \underset{3n\times 3}{\Phi}\right)^{-1}\ \underset{3\times 3n}{\Phi^T}\ \underset{3n\times 1}{Y}
$$

theta: [1.57600862 1.65617664 15.09760938]

g: 9.57964898326

m1: -0.0801680184959

m2: 1.65617663934

e) **Recovering Model Information [4 Points]**

Can you recover the mass properties m_1 and m_2 from your learned parameters? Has the robot learned a plausible inverse dynamics model? Explain your answers.

Yes the masses and the g could be derived see d). No the robot has apparently not learned a plausible inverse dynamics model, cause all masses should be at least higher than zero and g should be around 9.81 .

f) **Model Evaluation [7 Points]**

Plot the forces and torques predicted by your model over time, as well as those recorded in the data and comment the results. Is the model accuracy acceptable? If not, how would improve your model? Use one figure per joint.

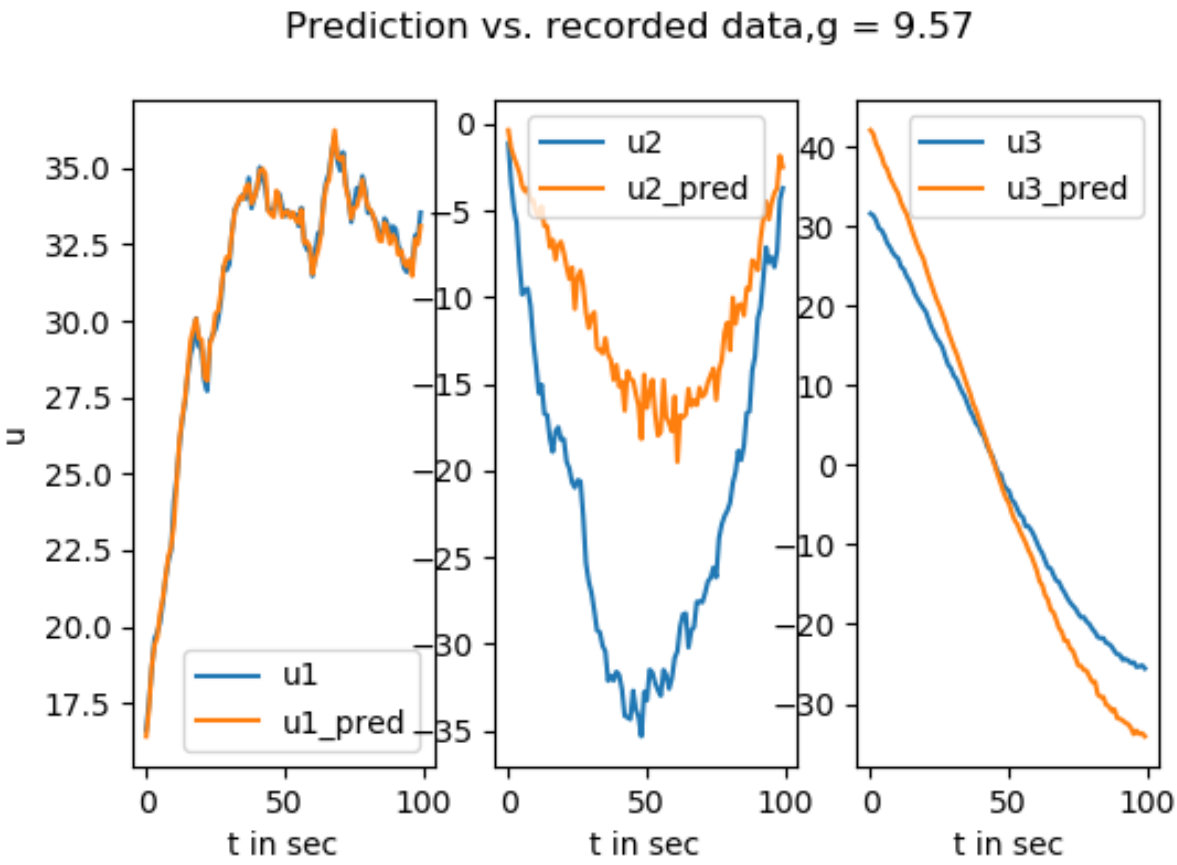


Figure 2: Predicted values for u

The Model accuracy is especially for the torque u_2 not acceptable (see middle figure above). The reason therefore is that the mass m_1 is obviously wrong. That is why the trajectory of the torque u_2 is overestimated.

Model Improvement: The model could be improved, by assuming g to be 9.81. Next the masses can be predicted again using:

$$\underset{3\times1}{y_1} = \phi^T\theta = \underset{3\times2}{\begin{bmatrix} \ddot{q_1}+g & 0 \\ 0 & 2\dot{q_3}\dot{q_2}q_3+q_3^2\ddot{q_2} \\ 0 & \ddot{q_3}-q_3\dot{q_2}^2 \end{bmatrix}} \underset{2\times1}{\begin{bmatrix} m_1+m_2 \\ m_2 \end{bmatrix}}$$

Another method may be to use many different trajectories and use them all to predict the parameters. Than you have better distributed data and less over-fitting.

g) **Models for Control [4 Bonus Points]**

Name and describe three different learned models types and their possible applications.

Model Type	*Description*	*Possible Application*
Forward Model	*Predict the future state:* $s_{k+1} = f_D(s_k, u_k) + \epsilon$ *s:state, u:action Can be used for direct action geneartion.*	*Usefull for long-term predictions. Can be used as Simulator.*
Inverse Model	*Predict the action needed to reach a desired state or other outcome.* $u = \pi(s_t) = f\left(s_t, s_{t+1}^{des}\right)$*. An Inverse can be learned directly (e.g. inverse dynamics control). Only if system is an invertable function, inverse model learning is useful.*	*In inverse dynamics control: The action u to reach a desired state with* $q_t^{des}, \dot{q}_t^{des}, \ddot{q}_t^{des}$ *should be predicted. Applications are Model Based Control and Fast Forward control.*
Mixed Model	*Predict required task elements with a forward model and use an inverse model for control.*	*Systems with Hysteresis, Inverse Kinematics.*
Multi-Step Prediction Model	*Predict a task variable long term (far in the future). Especially useful for system's which have a lot of latency in the control loop.*	*Throw a ball into a cup at a given position (like the example in the lab). Or Systems with a lot latency like the Mars Rover when controlling it from earth.*

Problem 2.2 Trajectory Generation with Dynamical Systems [22 Points + 16 Bonus]

In this exercise we will use the Dynamic Motor Primitives (DMPs), described by the following dynamical system,

$$\ddot{y} = \tau^2 (\alpha (\beta (g - y) - (\dot{y}/\tau)) + f_w(z)), \tag{1}$$
$$\dot{z} = -\tau \alpha_z z, \tag{2}$$

where y is the state of the system, $\dot{y}$ and $\ddot{y}$ are the first and second time derivatives, respectively. The attractor's goal is denoted by g and the forcing function by f_w. The parameters α and β control the spring-damper system. The phase variable is denoted by z and the temporal scaling coefficient by τ. The forcing function f_w is given by

$$f_w(z) = \frac{\sum_{i=0}^{K} \phi_i(z) w_i z}{\sum_{j=0}^{K} \phi_j(z)} = \boldsymbol{\psi}(z)^T \boldsymbol{w}, \quad \text{with} \quad \psi_i(z) = \frac{\phi_i(z) z}{\sum_{j=1}^{K} \phi_j(z)}, \tag{3}$$

where the basis functions $\phi_i(z)$ are Gaussian basis given by

$$\phi_i(z) = \exp\left(-0.5 (z - c_i)^2 / h_i\right), \tag{4}$$

where the centers c are equally distributed in the phase z, and the width h is an open parameter. For the programming exercises a basic environment of a double link pendulum is provided, as well as the computation of the $\boldsymbol{\psi}_i(z)$.

a) **Similarities to a PD controller [2 Points]**

Transform Equation (1) to have a similar structure to a PD-controller,

$$\ddot{y}_z = K_P \left(y_z^{des} - y_z\right) + K_D \left(\dot{y}_z^{des} - \dot{y}_z\right) + u_{ff} \tag{5}$$

and write down how the following quantities K_p, K_d, y_z^{des} and $\dot{y}_z^{des}$ look like in terms of the DMP parameters. Do not expand the forcing function $f_w(z)$ at your solutions.

$\ddot{y} = \tau^2 \alpha\beta (g - y) + (-\tau\alpha)(0 - \dot{y}) + f_w(z)$
$K_P = \tau^2 \alpha\beta, \quad K_D = -\tau\alpha, \quad y_z^{des} = g, \dot{y_z}^{des} = 0$

b) **Stability [2 Points]**

Show why the DMPs are stable when $t \to \infty$ and what would the equilibrium point be.

If you solve the diff. equation: $\dot{z} = -\tau\alpha_z z, \quad z(t) = z_0 e^{\lambda t} = z_0 e^{-\tau\alpha_z t} \overset{t\to\infty}{=} 0 \to f_w(z) \overset{t\to\infty}{=} 0.$
If you insert $f_w(z) = 0$ *in eq(1) you can calculate the equilibrium point by transforming eq. (1) with Laplace and then using the "Endwertsatz" or the steady state error.*
$G(s) = \frac{Y(s)}{g(s)} = \frac{\tau^2\alpha\beta}{s^2 + s\tau\alpha + \tau^2\alpha\beta}, \quad G(0) = 1, \quad Y(s) = g(s)$
So your transfer function for $t \to \infty$ *is* $G(s) = 1$ *and you have bo steady state error. This means your equilibrium point is your goal position:*
$y_{eqp} = g(goal)$
You can also set $\dot{y}$ *and* $\ddot{y}$ *to zero and solve this equation on y. Then you get the same equilibrium point.*

c) **Double Pendulum - Training [12 Points]**

Implement the DMPs and test them on the double pendulum environment. In order to train the DMPs you have to solve Equation (1) on the forcing function. Before starting the execution, set the goal g position to be the same as in the demonstration. Then, set the parameters to $\alpha = 25, \beta = 6.25, \alpha_z = 3/T, \tau = 1$. Use $N = 50$ basis functions, equally distributed in z. Use the learned DMPs to control the robot and plot in the same figure both the demonstrated trajectory and the reproduction from the DMPs. You need to implement the DMP-based controller (`dmpCtl.py`) and the training function for the controller parameters (`dmpTrain.py`). To plot your results you can use `dmpComparison.py`. Refer to `example.py` to see how to call it. Attach a snippet of your code.

At first you have to solve Equation (1) *on the forcing function. This results in the following:*

$$f_W(z) = \frac{\ddot{y}}{\tau^2} + \alpha\left(\frac{\dot{y}}{\tau} - \beta(g - y)\right) \tag{7}$$

The next step is the training of the DMP. The end position of the demonstration is $[0.3, -0.8]$*, so this is the goal Position g. Then you can use the equations of lecture 5 too train the DMP:*

```
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.array([[0.3, -0.8]])

    Phi = getDMPBasis(params, dt, nSteps)

    #Compute the forcing function
    ft = qdd/(params.tau**2) - params.alpha*(params.beta*(params.goal.T-q) - qd/params.tau)
    #Learn the weights
    sigma = 1e-8
    pseudo_inv = inv((np.dot(Phi.T, Phi) + sigma*np.eye(Phi.shape[1])))
    params.w = np.dot(np.dot(pseudo_inv, Phi.T),ft.T)

    return params
```

Listing 3: dmpTrain.py

After this you have to implement the DMP controller:

```
def dmpCtl (dmpParams, psi_i, q, qd):
    ...
    fw = np.dot(psi_i.T, w)
    qdd = tau**2*(alpha*(beta*(goal-q)-(qd/tau))+fw)
    return qdd
```

Listing 4: dmpCtl.py

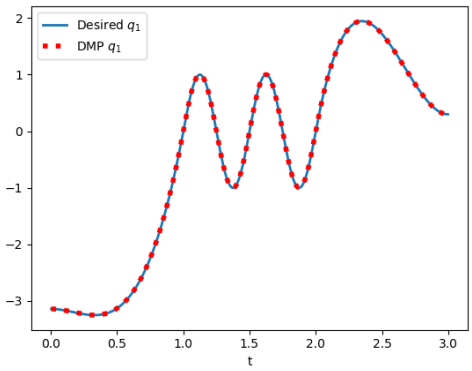


Figure 5: The desired and reproduced trajectory of q_1 are nearly equal.

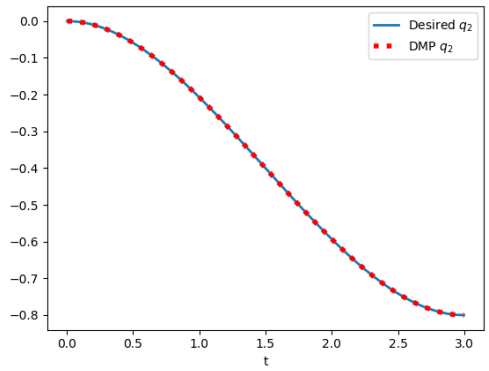


Figure 6: The desired and reproduced trajectory of q_2 are also nearly equal.

You can use this functions to reproduce the demonstration trajectory. In figure 5 and figure 6 you can see, that the desired trajectory and the reproduced trajectory from the DMP are nearly the same. This is a good result and shows that we use enough basis functions.

d) **Double Pendulum - Conditioning on the Final Position [3 Points]**

Using the trained DMPs from the previous question, simulate the system with different goal positions: first with $q_{t=\text{end}} = \{0, 0.2\}$ and then with $q_{t=\text{end}} = \{0.8, 0.5\}$. Generate one figure per DoF. In each figure, plot the demonstrated trajectory and the reproduced trajectories with different goal positions. How do you interpret the result?

In this task we use the trained DMP from question c) with a different goal position. You get the trajectories in figure 9 and figure 10. The trajectories of q1 are most of the time nearly the same, the new trajectory is only shifted a bit up or down. The only really difference is the start behavior. All trajectories start at the same point, but then the reach the offset to the trained DMP really fast. This is not an optimal behavior, because it would be more useful, if the new trajectory follows the trained one at the beginning and than slowly moves away from it.

The q_2-trajectory looks really different compared to the trained DMP. It rises very fast at the beginning and after this it shows the same course like the trained DMP, but has a big offset. It is hard to interpret the graph without some more graphical simulation, but this high gradient at the beginning cannot be good for the robot.

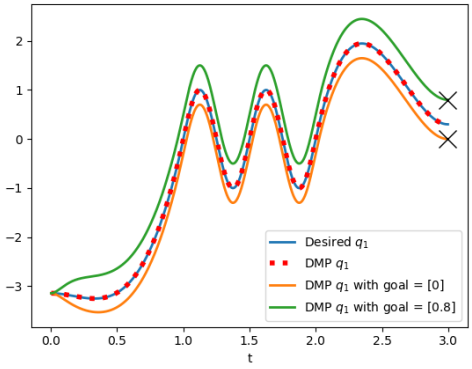


Figure 9: q_1 with goal position $q_{t=\text{end}} = \{0, 0.2\}$.

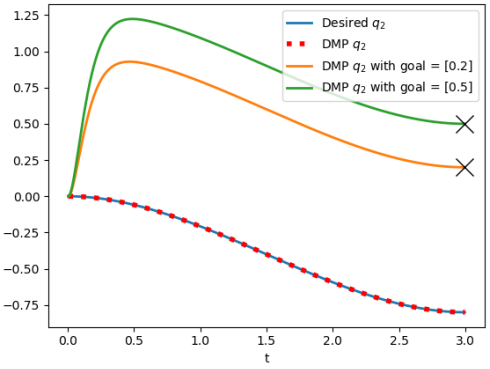


Figure 10: q_2 with goal position $q_{t=\text{end}} = \{0.8, 0.5\}$.

e) **Double Pendulum - Temporal Modulation [3 Points]**

Using the trained DMPs from the previous question, simulate the system with different temporal scaling factors $\tau = \{0.5, 1.5\}$. Generate one figure per DoF and explain the result.

Now the DMP is rescaled in the time. One trajectory is faster as the trained DMP, one is slower. You can see the result in figure 13 and figure 14. The trajectories follow the same course, but have different time scalings. It looks like a linear rescaling of the trained DMP. Note that in this example the trained and temporal scaled trajectories have the same goal as the desired trajectory. If the temporal scaling factor is smaller than 1 the goal position is not reached. If the temporal scaling factor is higher than one the goal position is reached faster.

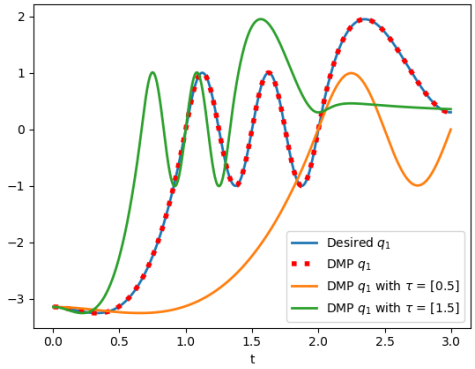


Figure 13: q_1 with different end times.

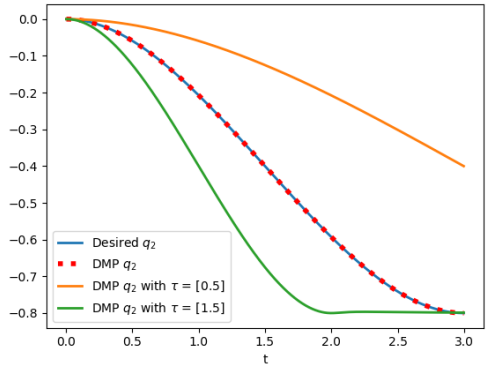


Figure 14: q_2 with different end times

f) **Probabilistic Movement Primitives - Radial Basis Function [3 Bonus Points]**

We now want to use ProMPs. Before we train them, we need to define some basis functions. We decide to use $N = 30$ radial basis functions (RBFs) with centers uniformly distributed in the time interval $[0-2b, T+2b]$, where T is the end time of the demonstrations. The bandwidth of the Gaussian basis (std) is set to $b = 0.2$. Implement these basis functions in `getProMPBasis.py`. Do not forget to normalize the basis such at every time-point they sum-up to one! Attach a plot showing the basis functions in time and a snippet of your code.

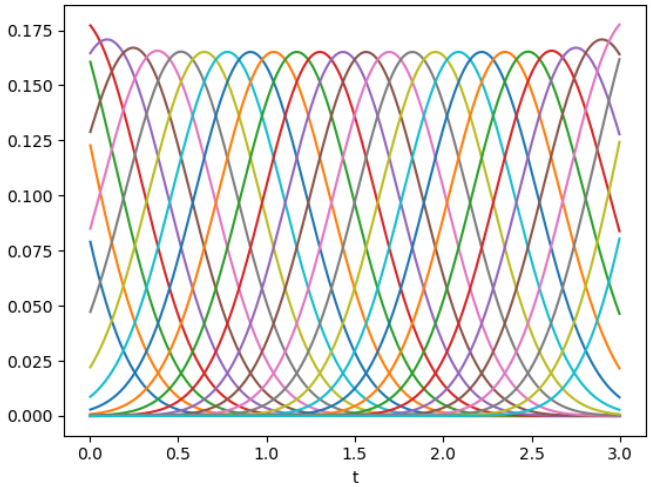


Figure 16: 30 uniformly distributed gaussian basis functions.

```python
def getProMPBasis(dt, nSteps, n_of_basis, bandwidth):

    time = np.arange(dt,nSteps*dt,dt)
    T = dt * nSteps - dt
    nBasis  = n_of_basis
    b = bandwidth

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

    H[:] = b
    C = np.linspace(0-2*b, T+2*b, nBasis)

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

    for i in xrange(nSteps):
        for j in xrange(nBasis):
            Phi[i,j] = np.exp(-1/2*((time[i]-C[j])**2)/H[j])
        # normalize the basis functions
        Phi[i, :] = Phi[i, :]/np.sum(Phi[i,:])

    return Phi
```

code/getProMPBasis.py

g) **Probabilistic Movement Primitives - Training [7 Bonus Points]**

In this exercise you will train the ProMPs using the imitation learning data from `getImitationData.py` and the RBFs defined in the previous question. Modify the `proMP.py` in order to estimate weight vectors w_i reproducing the different demonstrations. Then, fit a Gaussian using all the weight vectors. Generate a plot showing the desired trajectory distribution in time (mean and std) as well as the trajectories used for imitation. Attach a snippet of your code.

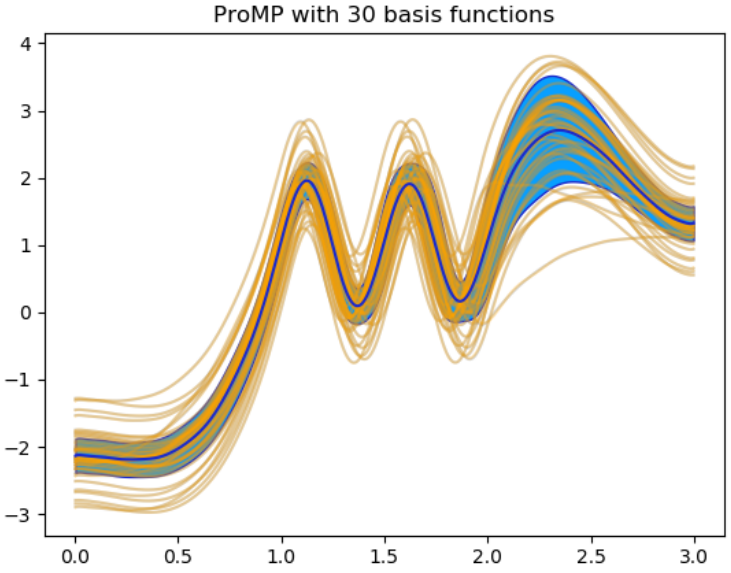


Figure 18: In this figure you can see the trained ProMP in blue. The dark blue line shows the mean, the light blue area is the uncertainty. The grey lines are the training data and the orange lines are the reproduced trajectories.

```python
def proMP (nBasis, condition=False):
    ...
    # estimate weight vectors w for all demonstrations
    W = np.eye(Phi.shape[1])*1e-12
    pseudo_inv = inv(np.dot(Phi.T, Phi) + W)
    ws = np.dot(np.dot(pseudo_inv, Phi.T), q.T)

    # fit a Gaussian over all weight vectors
    mean_w = np.mean(ws, axis=1)
    cov_w = np.cov(ws)
    mean_traj = np.dot(Phi, mean_w)
    std_traj = np.diag(np.dot(np.dot(Phi, cov_w), Phi.T))
```

code/proMP_h.py

h) **Probabilistic Movement Primitives - Number of Basis Functions [2 Bonus Points]**

Evaluate the effects of using a reduced number of RBFs. Generate two plots showing the desired trajectory distribution and the trajectories used for imitation as in the previous exercise, but this time use $N = 20$ and $N = 10$ basis functions. Briefly analyze your results.

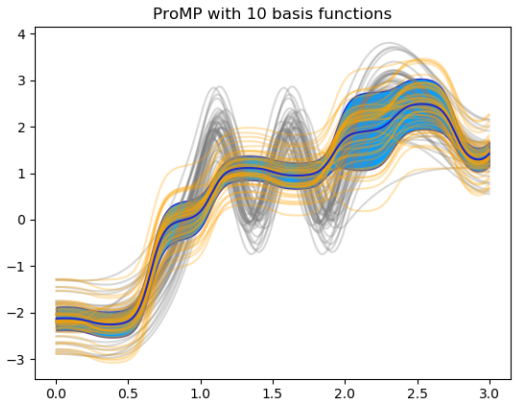


Figure 21: ProMP with 10 basis functions: The reproduced trajectories cannot follow the training trajectories, they only show the mean. This is not a usefull model.

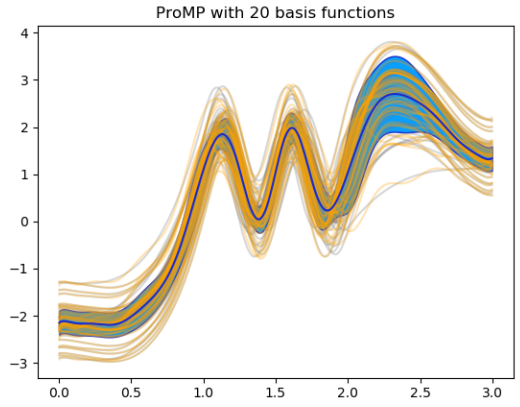


Figure 22: ProMp with 20 basis functions: This ProMP is very simluar to the ProMP with 30 basis function. So this is also a good model. Then you look at the single training trajectories, you can see some dfferences between the training and the reproduced trajectory. Because of this some more basis functions would be usefull.

i) **Probabilistic Movement Primitives - Conditioning [4 Bonus Points]**

Using Gaussian conditioning calculate the new distribution over the weight vectors w_i such as the trajectory has a via point at position $y^* = 3$ at time $t_{\text{cond}} = 1150$ with variance $\Sigma_{y^*} = 0.0002$. Use again 30 basis functions. Assuming that the probability over the weights is given by $\mathcal{N}\left(\boldsymbol{w}|\boldsymbol{\mu}_w, \boldsymbol{\Sigma}_w\right)$ and the probability of being to that position is given by $\mathcal{N}\left(y^*|\boldsymbol{\Phi}\boldsymbol{w}, \Sigma_{y^*}\right)$, show how the new distribution over $\boldsymbol{w}$ is computed (how does the mean and variance look like)?
Then, in a single plot, show the previous distribution (learned from imitation) and the new distribution (after conditioning). Additionally, sample $K = 10$ random weight vectors from the ProMP, compute the trajectories and plot them in the same plot. Analyze briefly your results and attach a snippet of your code.

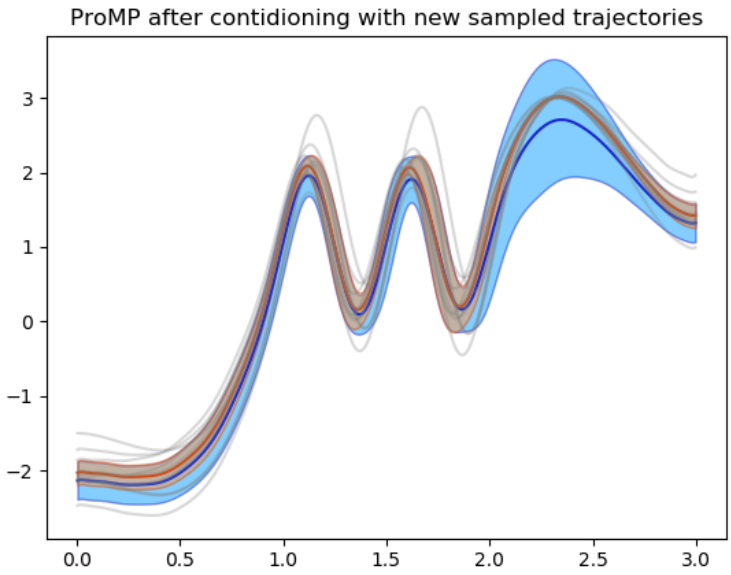


Figure 24: New distribution after reconditioning: The bule line shows the original ProMp, the orange one is the conditioned ProMp. The grey functions are example trajectories.

Figure 24 shows the conditioned ProMP. You can see original ProMp in blue and the new one in orange. The mean of the new ProMp goes exactly through the given point at $t_{cond} = 1150$. The uncertainty at this point is very low. Then you look at other points of the ProMp the uncertainty is much bigger. The new trajectory follows the original ProMp most of the time and only at the given position it moves away from it. This is a much better behavior as the DMP in the last exercise.

```
def proMP (nBasis, condition=False):
    ...
    #Conditioning
    if condition:
        y_d = 3
        Sig_d = 0.0002
        t_point = np.round(2300/2)

        Phi_t = np.reshape(Phi[:,t_point],(30,1))
        tmp = np.dot(cov_w, Phi_t) / (Sig_d + np.dot(Phi_t.T,np.dot(cov_w,Phi_t)))

        tmp = np.reshape(tmp,(30,1))
        cov_w_new = cov_w - np.dot(np.dot(tmp, Phi_t.T), cov_w)
        mean_w_new = mean_w + np.dot(tmp, y_d-np.dot(Phi_t.T, mean_w))
        mean_traj_new = np.dot(Phi.T, mean_w_new)
        std_traj_new = np.diag(np.dot(np.dot(Phi.T, cov_w_new), Phi))

    ...
```

code/proMP_i.py