Robot Learning

TECHNISCHE UNIVERSITÄT DARMSTADT

Winter Semester 2016, Homework 2 (45 points + 20 bonus)
Prof. Dr. J. Peters, F. Veiga, S. Parisi

Due Date: Thursday, 12 January 2017 (before the lecture)

Problem 2.1 Model Learning [23 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.

Your graphic could be here. It just wasn't included.

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 now want to learn the inverse dynamics of the robot in order 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 [1 Points]

What kind of machine learning problem is learning an inverse dynamics model?

It is a supervised learning problem. The reason being that it tries to map input to output on labelled data.

b) Assumptions [5 Points]

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

We are not learning a function with this. This is a problem as there can be many solutions to one problem resulting in difficulties when learning (using the mean is not useful as the mean of two solutions is itself far off).

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)

$$\boldsymbol{\phi} = \begin{bmatrix} \ddot{q}_1 & 0 & 0 \\ \ddot{q}_1 & 2\dot{q}_3\dot{q}_2 q_3 + q_3^2\ddot{q}_2 & \ddot{q}_3 - q_3\dot{q}_2^2 \\ 1 & 0 & 0 \end{bmatrix}$$

$$\boldsymbol{\theta} = \begin{bmatrix} m_1 & m_2 & g \cdot (m_1 + m_2) \end{bmatrix}$$

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.

$\theta = \left(\phi\phi^T\right)^{-1}\phi Y$
$\theta \in \mathbb{R}^{3\times 1}$,
$\phi \in \mathbb{R}^{3\times 300}$,
$\phi^T \in \mathbb{R}^{300\times 3}$,
$Y \in \mathbb{R}^{300\times 1}$

$$\boldsymbol{\theta} = \begin{bmatrix} -0.07297389 \\ 1.6496578 \\ 15.09510 \end{bmatrix}$$

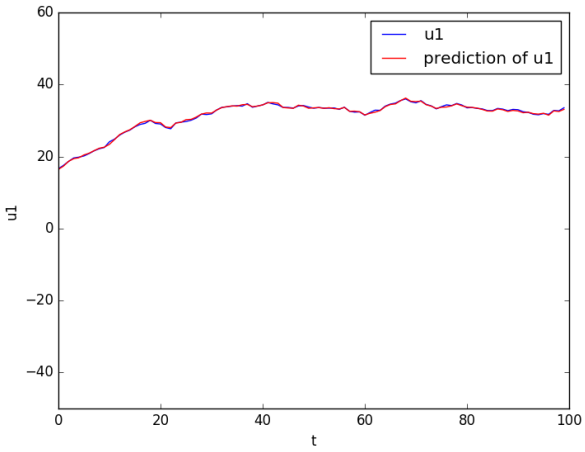
e) Recovering Model Information [4 Points]

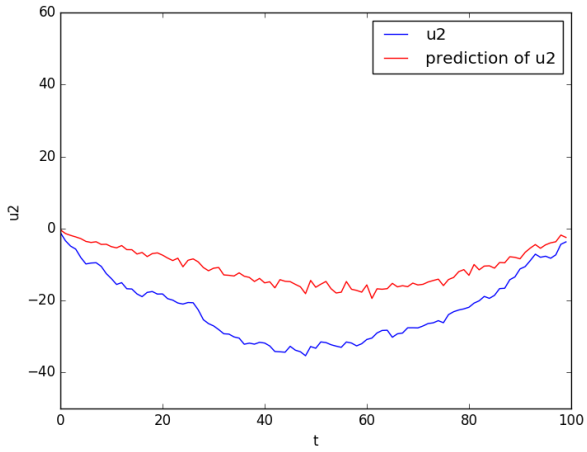
Can you recover the mass properties m_1 and m_2 from your learned parameters? Has the robot learned a plausible inverse dynamics model? Explain your answers.

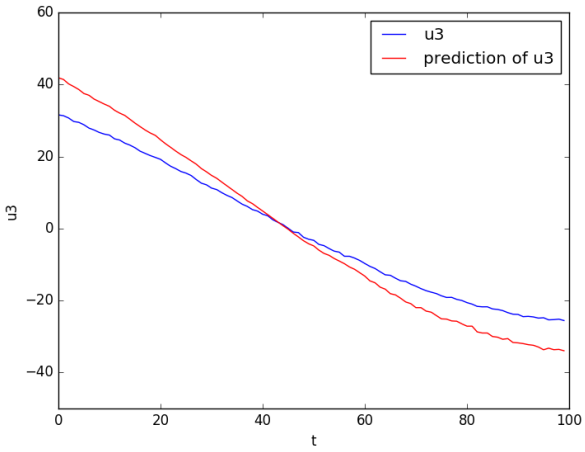
While we could retrieve some value from the learned values it does not make much sense. We learned a negative value for our first mass which might in simulation solve our problem but is not attainable in the real world. This is due to the fact that we aren't learning a function which would only allow one solution that makes sense with real life data since it directly correlates with the training data.

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.







The prediction for the first joint is very precise.
The second plot shows that the prediction for the second joint at least has the same trend as the real values, but differ especially at around half of the time.
The predictions of the third joint differ at the beginning and the end the most, and are better at around half of the time. All in all the models accuracy is not acceptable, since the calculated values for the second and third joints differ to much at some points.

To improve the model, it would be better if we could incorporate the gravity constant directly into the feature matrix, instead of learning it as a parameter, since it is a well known constant. Moreover, a well defined function would be advantageous as we can actually learn it.

g) Models for Control [4 Bonus Points]

Name and describe three different ways to use learned models in control laws or for generating control laws.

We can use Controller Falsification which uses our model to test random Control Laws and outputs the first one that does not fail.
We can also use it to optimize our control law by generating control laws and comparing them to our current best control law.
We can also use our learned model to generate a policy that simplifies our control.

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.

$$\ddot{y}_z = \tau^2 \alpha\beta \cdot (g - y) + \alpha \cdot \tau \cdot (0 - \dot{y}) + \tau^2 f_w(z)$$
$$K_p = \tau^2 \alpha\beta$$
$$K_d = \alpha \cdot \tau$$
$$y_z^{des} = g$$
$$\dot{y}_z^{des} = 0$$

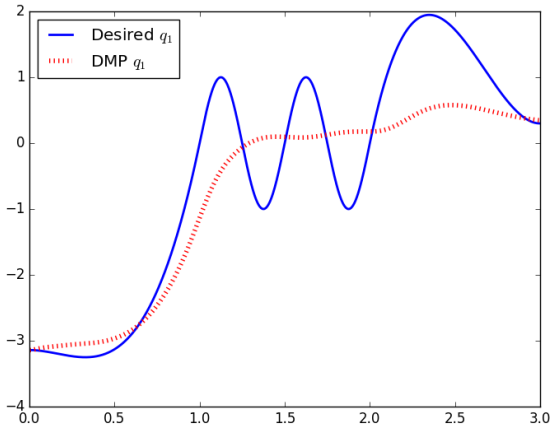
b) Stability [2 Points]

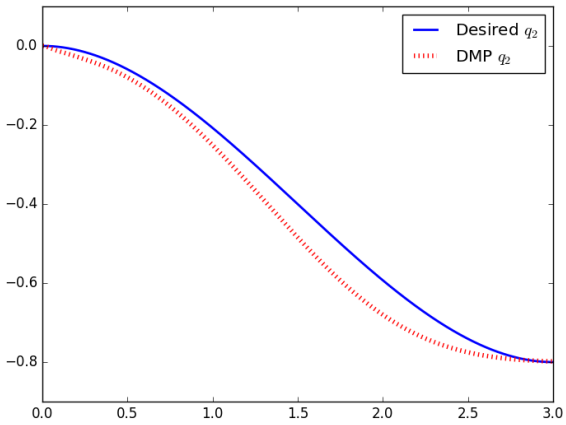
Show why the DMPs are stable when $t \to \infty$ and what would the equilibrium point be.

A high t means our forcing function goes to zero. This leaves us with a PD-controller that only controls $\ddot{y}_z$ to compensate for gravity. The equilibrium point would be at y_z^{des} which is g.

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.





the control code:

```
def dmpCtl (dmpParams, psi_i, q, qd):
    alpha = dmpParams.alpha
    beta = dmpParams.beta
    tau = dmpParams.tau
    goal = dmpParams.goal
    w = dmpParams.w

    KD = tau * tau * alpha * beta
    KP = alpha * tau
    qdd = KD * (goal - q) - KP * qd + (tau * tau) * np.matmul(psi_i, w)
    return qdd
```

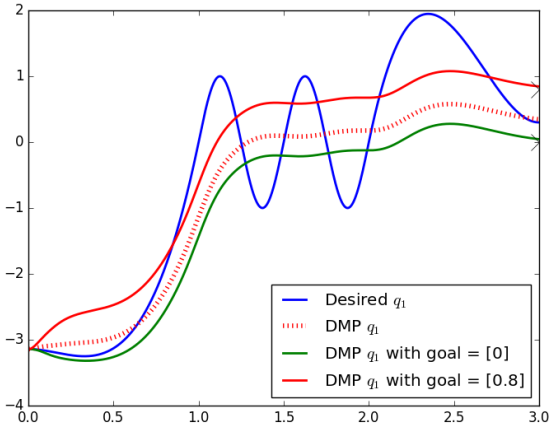
the training code:

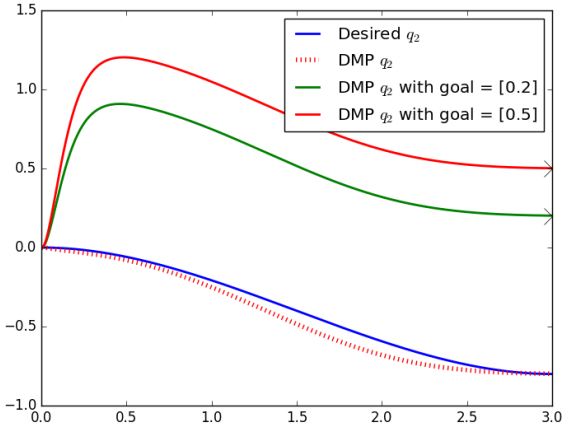
```
def dmpTrain(q, qd, qdd, dt, nSteps):

    params = dmpParams()
    # Set dynamic system parameters
    params.alphaz = 3 / (dt*nSteps)
    params.alpha = 25
    params.beta = 6.25
    params.Ts = (dt*nSteps)
    params.tau = 1
    params.nBasis = 50
```

```
    params.goal = np.transpose(q[:, -1])

    Phi = getDMPBasis(params, dt, nSteps)

    # shorthand for parameters
    tau = params.tau
    alpha = params.alpha
    beta = params.beta
    goal = params.goal
    # Compute the forcing function
    ft = np.zeros((nSteps, 2))
    ydd = np.zeros((2, 1))
    yd = np.zeros((2, 1))
    y = np.zeros((2, 1))
    for z in range(0, nSteps):
        ydd[0] = qdd[0][z]
        ydd[1] = qdd[1][z]
        yd[0] = qd[0][z]
        yd[1] = qd[1][z]
        y[0] = q[0][z]
        y[1] = q[1][z]
        temp = ydd/(tau*tau) - np.transpose(alpha * (beta * np.subtract(goal, np.transpo
        ft[z, 0] = temp[0]
        ft[z, 1] = temp[1]

    # Learn the weights
    params.w = np.matmul(np.linalg.inv(np.matmul(Phi.transpose(), Phi) + np.eye(50)), np

    return params
```

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

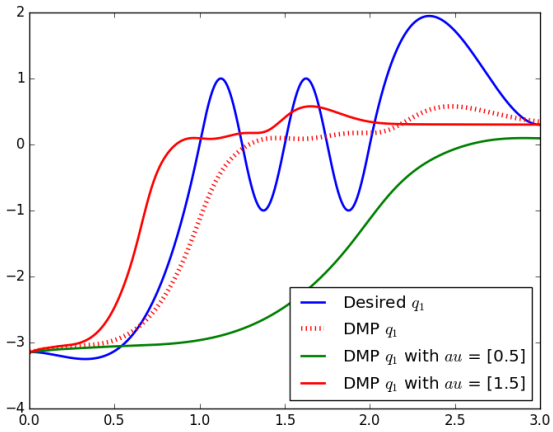
Using the trained DMPs from the previous question, simulate the system with different goal positions: first with $q_{t=\text{end}} = \{0, 0.2\}$ and then with $q_{t=\text{end}} = \{0.8, 0.5\}$. Generate one figure per DoF. In each figure, plot the demonstrated trajectory and the reproduced trajectories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

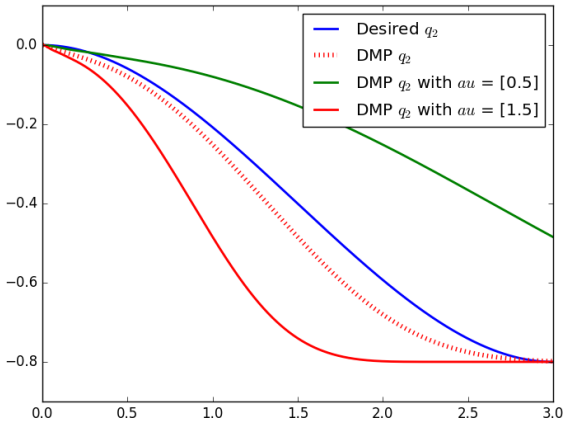




e) Double Pendulum - Temporal Modulation [3 Points]

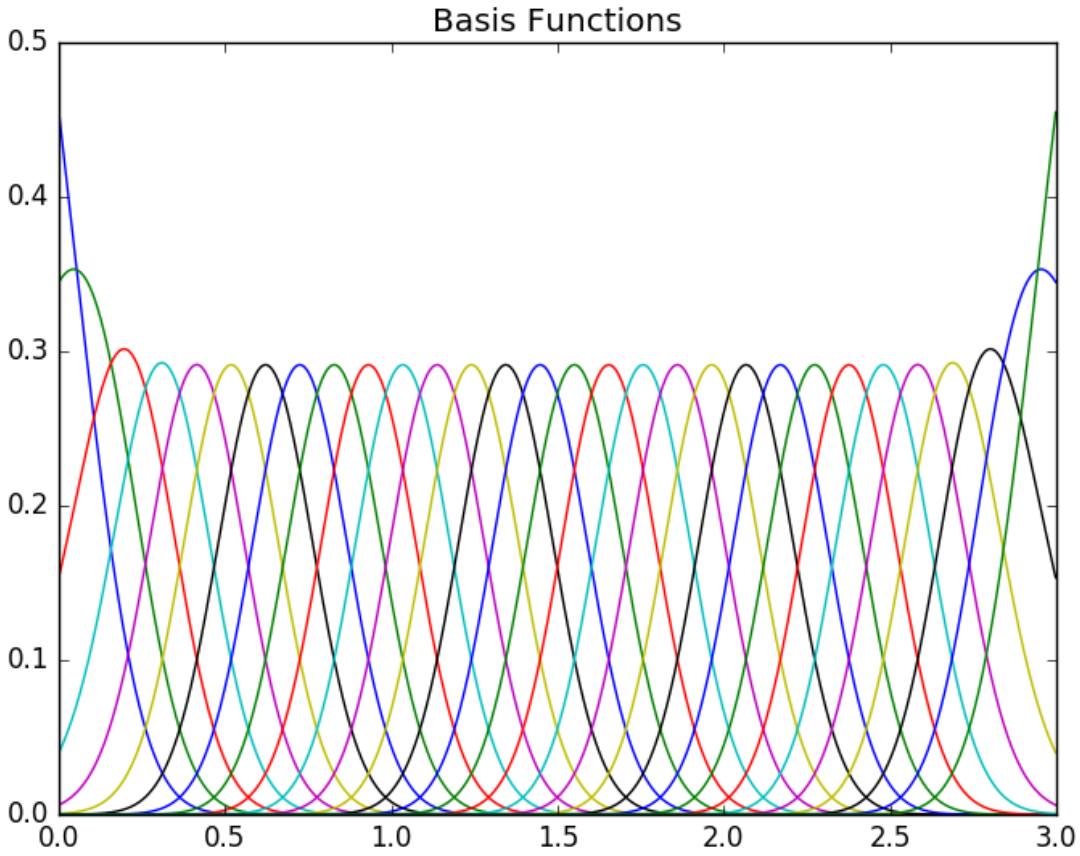
Using the trained DMPs from the previous question, simulate the system with different temporal scaling factors $\tau = \{0.5, 1.5\}$. Generate one figure per DoF and explain the result.





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.

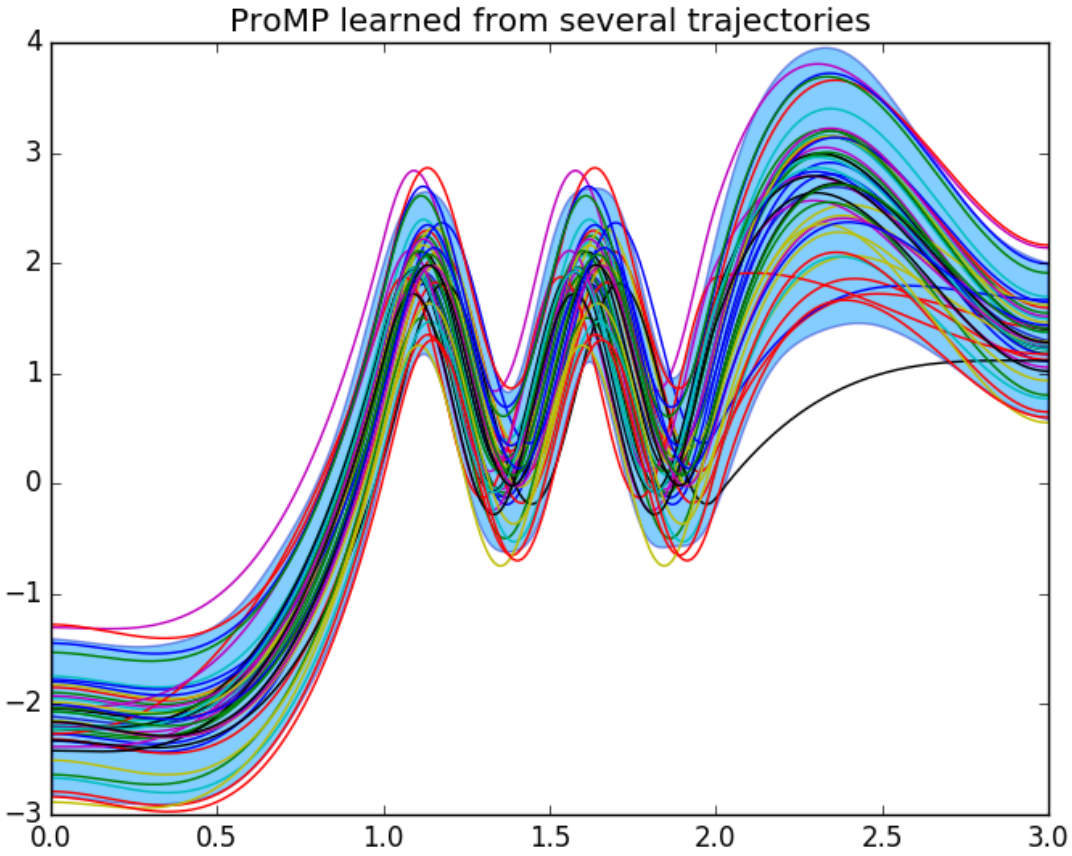



```
def getProMPBasis(dt, nSteps, n_of_basis, bandwidth):

    time = np.arange(dt, nSteps*dt, dt)
    Phi = np.zeros((len(time), n_of_basis))

    for i in range(len(time)):
        currentTime = time[i]
        phi_i = np.zeros(n_of_basis)
        centers = np.linspace(dt, nSteps * dt - dt, n_of_basis)
        for j in range(n_of_basis):
            phi_i[j] = np.exp(-((centers[j] - currentTime)*(centers[j] - currentTime))/(
        sum_phi_i = np.sum(phi_i)
        Phi[i, :] = phi_i / sum_phi_i
```

```
        plt.figure()
        plt.plot(time, Phi)
        plt.title('Basis Functions')

        return Phi.transpose()
```

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.

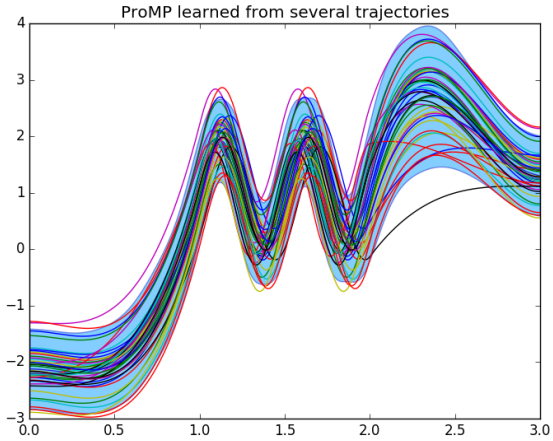


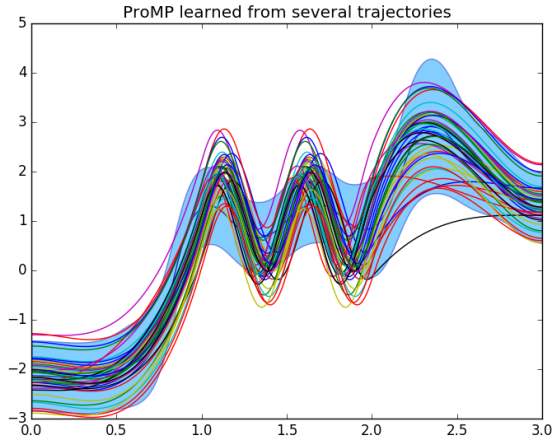

```
def proMP (nBasis):

    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)
    w = np.matmul(np.matmul(np.linalg.inv(np.matmul(Phi.transpose(), Phi) + (bandwidth*b
    mean_w = np.mean(w.transpose(), axis=0)
```

```
cov_w = np.cov(w)
plt.figure()
plt.hold('on')
plt.fill_between(time, np.dot(Phi.transpose(), mean_w) - 2 * np.sqrt(np.diag(np.dot(Ph
plt.plot(time, np.dot(Phi.transpose(), mean_w), color='#1B2ACC')
plt.plot(time, q.transpose())
plt.title('ProMP learned from several trajectories')
```

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.

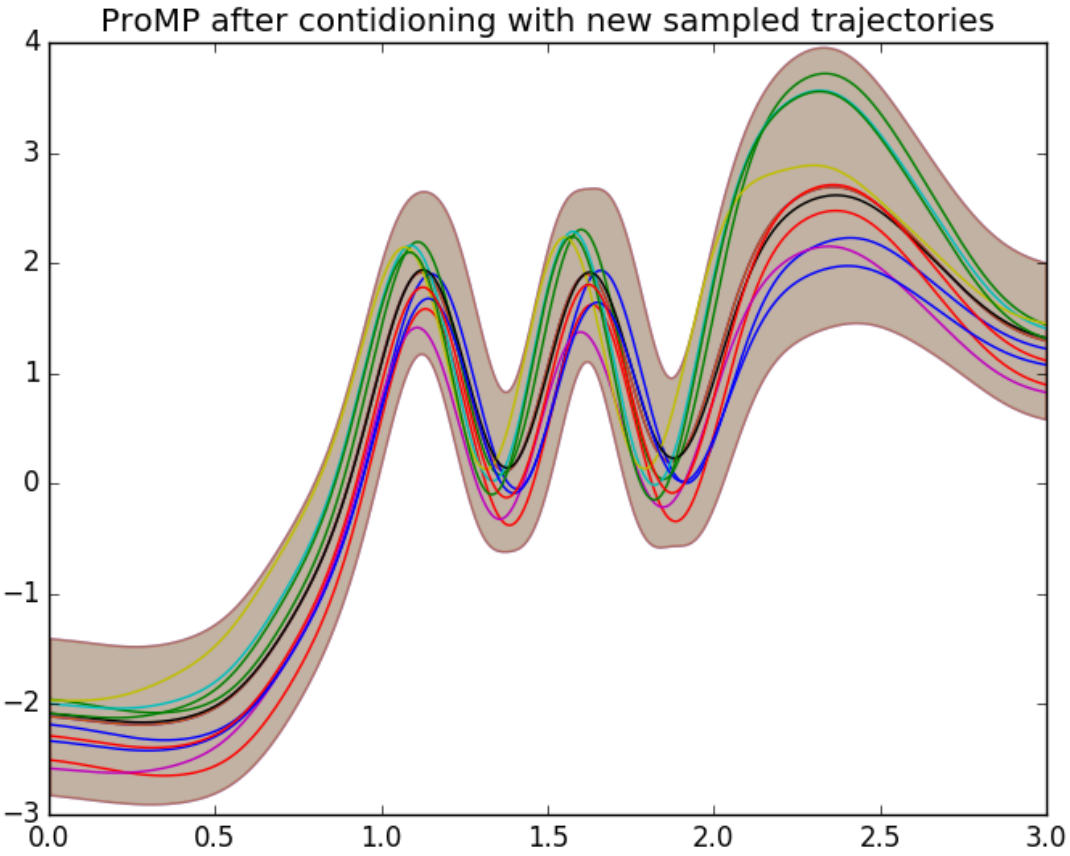




On the left side were 10 radial basis functions used. On the right side 20. One can easily see, that the reduction to N=20 basis functions does'nt affect the result too much, but the reductionn to N=10 does. The trajectory distribution doesn't fit the trajectories equally well anymore as with 20 functions.

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 $\Sigma_{y^*} = 0.0002$. 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.




```
#Conditioning
y_d = 3
Sig_d = 0.0002
t_point = np.round(2300/2)
#x = np.divide(y_d - np.matmul( Phi.transpose(), w), Sig_d)
#w_new = np.exp(- x*x) / (np.sqrt(np.pi) * Sig_d)
Phi_j = np.zeros((nSteps, nBasis))
for i in range(nSteps):
    phi_i = np.zeros(nBasis)
    for j in range(nBasis):
        phi_i[j] = np.exp(-((y_d - t_point) * (y_d - t_point)) / (Sig_d * Sig_d))
```

```
        sum_phi_i = np.sum(phi_i)
        Phi_j[i, :] = np.divide(phi_i, sum_phi_i)
w_new = np.matmul(np.matmul(np.linalg.inv(np.matmul(Phi.transpose(), Phi) + (bandwidth
mean_w_new = np.mean(w_new.transpose(), axis=0)
cov_w_new = np.cov(w_new)

plt.figure()
plt.hold('on')
plt.fill_between(time, np.dot(Phi.transpose(), mean_w) - 2*np.sqrt(np.diag(np.dot(Phi.
plt.plot(time, np.dot(Phi.transpose(), mean_w), color='#1B2ACC')
plt.fill_between(time, np.dot(Phi.transpose(), mean_w_new) - 2*np.sqrt(np.diag(np.dot(
plt.plot(time, np.dot(Phi.transpose(), mean_w_new), color='#CC4F1B')

sample_traj = np.dot(Phi.transpose(),np.random.multivariate_normal(mean_w_new, cov_w_n
plt.plot(time, sample_traj)
plt.title('ProMP after contidioning with new sampled trajectories')


plt.figure()
```