Robot Learning

Winter Semester 2017/2018, Homework 1

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

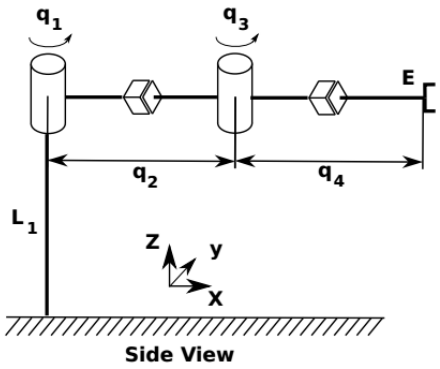
TECHNISCHE UNIVERSITÄT DARMSTADT

Total points: 68 + 10 bonus
Due date: Wednesday, 15 November 2017 (before the lecture)

Name, Surname, ID Number

Problem 1.1 Robotics in a Nutshell [12 Points]

You are considering to buy a new multi-purpose robot platform. Its kinematic chain has two rotational $q_{\{1,3\}}$ and two linear $q_{\{2,4\}}$ degrees of freedom (DoFs), as shown in the figure below. These four joints are actuated with forces and torques of u_i, $i \in \{1,2,3,4\}$. A gripper is mounted on the end of the robot, indicated by the letter E. The robot's base is mounted on a table. We assume that the base Cartesian coordinates at the mount are $x_{\text{base}} = [0,0,0]$.



a) Forward Kinematics [2 Points]

Compute the kinematic transformation in the global coordinate system from the base $\boldsymbol{x}_{\text{base}}$ to the end-effector E. Write the solution for the $\boldsymbol{x}_{\text{end-eff}} = [x,y,z]^T$ according to the joint values q_i, where $i \in \{1,2,3,4\}$.

Compute the position with rotation and translation matrices:

$$
\begin{aligned}
H_5 &= Trans_{z,l1} \cdot Rot_{z,q_1} \cdot Trans_{x,q_2} \cdot Rot_{z,q_3} \cdot Trans_{x,q_4} \\
H_5 &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & L1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} c_{q_1} & -s_{q_1} & 0 & 0 \\ s_{q_1} & c_{q_1} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & q_2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} c_{q_3} & -s_{q_3} & 0 & 0 \\ s_{q_3} & c_{q_3} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & q_4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
H_5 &= \begin{pmatrix} \cos(q_1+q_3) & -\sin(q_1+q_3) & 0 & q_4\cos(q_1+q_3)+q_2\cos(q_1) \\ \sin(q_1+q_3) & cos(q_1+q_3) & 0 & q_4\sin(q_1+q_3)+q_2\sin(q_1) \\ 0 & 0 & 1 & L1 \\ 0 & 0 & 0 & 1 \end{pmatrix}
\end{aligned}
\tag{5}
$$

So the position of the end-effector is:

$$
\begin{aligned}
x_e &= q_4\cos(q_1+q_3)+q_2\cos(q_1) \\
y_e &= q_4\sin(q_1+q_3)+q_2\sin(q_1) \\
z_e &= L1
\end{aligned}
\tag{6}
$$

One common method is to use the Denavit-Hartenberg table:

i	Θ_i	d_i	a_i	α_i
1	0	L	0	0
2	q1+pi/2	L	0	pi/2
3	0	q2	0	-pi/2
4	q3	0	0	-pi/2
5	0	q4	0	0

Now calculate: $0T1 = Rot(z,\Theta_i)\,Trans(0,0,d_i)\,Trans(0,0,a_i)\,Rot(x,\alpha_i), \qquad 0TE = 0T1 * 1T2 * \ldots * 4T5$

$$0T5 = \begin{bmatrix} \cos(q1+pi/2+q_3) & 0 & -\sin(q1+pi/2+q_3) & q_2\sin(q1+pi/2) - q_4\sin(q1+pi/2+q_3) \\ sin(q1+pi/2+q_3) & 0 & \cos(q1+pi/2+q_3) & -q_2\cos(q1+pi/2) + q_4\cos(q1+pi/2+q_3) \\ 0 & -1 & 0 & l_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (7)$$

Simplify: $sin(x+\pi/2) = cos(x) \quad cos(x+\pi/2) = -sin(x) \quad sin(x+y) = s_x c_y + s_y c_x \quad cos(x+y) = c_x c_y - s_x s_y$

$$0T5 = \begin{bmatrix} -\sin(q1+q_3) & 0 & -\cos(q1+q_3) & q_2\cos(q1) - q_4\cos(q1+q_3) \\ cos(q1+q_3) & 0 & -\sin(q1+q_3) & +q_2\sin(q1) - q_4\sin(q1+q_3) \\ 0 & -1 & 0 & l_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (8)$$

Direkt aus der Anschauung:
$x_e = q_2 cos(q_1) + q_4 cos(q_1+q_3)$
$y_e = q_2 sin(q_1) + q_4 sin(q_1+q_3)$
$z_e = l_1$

b) Inverse Kinematics [2 Points]

Define briefly in your own words the inverse kinematics problem in robotics. Can we always accurately model the inverse kinematics of a robot with a function?

Forward Kinematics:
with the given angles $\boldsymbol{q}$ you calculate the ${}^0\boldsymbol{T_E} = \begin{bmatrix} {}^0R_E & {}^0r_E \\ 0^T & 1 \end{bmatrix}$ matrix.

Inverse Kinematics:
with the given Pose (Orientation ${}^0\boldsymbol{R_E}$ and position ${}^0\boldsymbol{r_E}$) from the world coordinate system to the Endeffector coordinate system you can calculate the desired joint angles $\boldsymbol{q}$. The method therefore is called Inverse Kinematics (Inverse Euler or Inverse Roll Pitch Yaw method are possible.)

The INV KIN cannot always be modeled accurately because:

- if n<6 the INVKIN has no solution, $n = \sum_i \boldsymbol{q}_i$
- if n=6 the INVKIN has one solution
- if n>6 the INVKIN has infinity solutions (Redundancy)

The technical background of this is, that the robot can reach a end-effector position in task space with many different positions in joint space. So the inverse kinematics are not a function.

Although you can build robots with only a few degrees of freedom which have clearly invertible kinematics. This is often done for industrial robots.

c) Differential Kinematics [4 Points]

Compute the Jacobian matrix $\boldsymbol{J}(\boldsymbol{q})$ of the robot such that $\dot{\boldsymbol{x}} = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}}$, where $\dot{\boldsymbol{q}}$ is the first time derivatives of the state vector $\boldsymbol{q}$ of the robot. Explain in a sentence the physical meaning of the Jacobian.

Theory:

$\dot{\boldsymbol{x}} = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}}$, $\begin{bmatrix} {}^0v_n(t) \\ {}^0\boldsymbol{\omega}_n(t) \end{bmatrix} = {}^0\boldsymbol{J}_n(\boldsymbol{q}(t)) * \dot{\boldsymbol{q}}(t)$, ${}^0J_n = \begin{bmatrix} {}^0J_{n,v} \\ {}^0J_{n,\omega} \end{bmatrix}$, ${}^0J_n \in 6xn$ ${}^0J_n = [{}^0J_{n,1}, \ldots, {}^0J_{n,n}]$ (split into vectors) ${}^0\boldsymbol{R}_E = \left({}^0\boldsymbol{e}_x, {}^0\boldsymbol{e}_y, {}^0\boldsymbol{e}_z\right)$

There are basically 2 ways to calculate the Jacobian Matrix.

1. Get the Jacobian with the ${}^0T_{i-1}$

   If $q_i = \Theta_i$ revolute joint:

   $${}^0J_{n,i} = \begin{bmatrix} {}^0e_{z,i-1} x \left({}^0_e r_n - {}^0 r_{i-1}\right) \\ {}^0_{z,i-1} \end{bmatrix}$$

   If $q_i = \Theta_i$ prismatic joint:

   $${}^0J_{n,i} = \begin{bmatrix} {}^0e_{z,i-1} \\ \mathbf{0} \end{bmatrix}$$

2. Get the Jacobian with the ${}^0T_n$

   ${}^0v_n(t) = {}^0\dot{\boldsymbol{r}}_n(q(t))R(\dot{q})$

   ${}^0\omega_n(t) : B\left(q(t), \dot{q(t)}\right) = \frac{\partial R(q(t))}{\partial q(t)} * \dot{q(t)} * R^T(q(t))$, $B = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}$

Physical Meaning:

$$J = \begin{pmatrix} -q_4 \sin(q_1 + q_3) - q_2 \sin(q_1) & \cos(q_1) & -q_4 \sin(q_1 + q_3) & \cos(q_1 + q_3) \\ q_4 \cos(q_1 + q_3) + q_2 \cos(q_1) & \sin(q_1) & q_4 \cos(q_1 + q_3) & \sin(q_1 + q_3) \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{10}$$

The jacobian describes the relationship between the end-effector velocity an the velocity of the different joints.

d) Singularities [3 Points]

What is the kinematic singularity in robotics? How can you detect it? When does our robotic arm, which was defined above, enter a kinematic singularity?

Singularities occur if $det\left({}^0\boldsymbol{J}_n(q)\right) = 0$.
This can happen for instance if the rotational axis of two joints is the same in such a case one DOF is lost. So simply checkout if a certain row or column of the Jacobi matrix gets zero for special joint values.

To analyze the kinematic singularity of the robot, you only have to look at rotational joints. The translational joint cannot move into a singularity. So you can build a reduced jacobian:

$$J_{red} = \begin{pmatrix} -q_4 \sin(q_1+q_3) - q_2 \sin(q_1) & -q_4 \sin(q_1+q_3) \\ q_4 \cos(q_1+q_3) + q_2 \cos(q_1) & q_4 \cos(q_1+q_3) \end{pmatrix} \tag{14}$$

In a singularity the determinat of J_red equals zero. So we can calculate it and find the roots:

$$\det J_{red} = q_2 q_4 \sin(q_3) \tag{15}$$

The roots are:

$$\begin{aligned} q_2 &= 0 \\ q_4 &= 0 \\ q_3 &= 0 \vee \pi \end{aligned} \tag{16}$$

The robot hits singularities when the translational joints q_2 or q_4 are at the zeros position. Then the joints q_1 and q_3 are at the exactly same position in task space. However these singularities are never reached because of the technical limits of the robot. So the only important singularity is the third. The joints q_2 and q_4 are on the same axes and the robot loses one degree of freedom.

e) Workspace [1 Points]

If your task is to sort items placed on a table, would you buy this robot? Briefly justify your answer.

The robot can only reach positions with the coordinate $z = L1$. Because of this we would not buy the robot for this task.

Name, Surname, ID Number

Problem 1.2 Control [26 Points + 5 Bonus]

In robotic locomotion it is common to abstract from the robot by using inverted pendulum models. In this exercise we will use a planar double inverted pendulum to test different control strategies. Our robot can be controlled by specifying the torque $\mathbf{u} = [u_1, u_2]$ of its motors. Consider that in mechanical systems the torque $\boldsymbol{u}$ is a function of the joint positions $\boldsymbol{q}$, velocities $\dot{\boldsymbol{q}}$ and accelerations $\ddot{\boldsymbol{q}}$, as given by

$$\boldsymbol{u} = \boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{c}(\boldsymbol{q}, \dot{\boldsymbol{q}}) + \boldsymbol{g}(\boldsymbol{q}),$$

where $\boldsymbol{M}$ denotes the inertial matrix, $\boldsymbol{c}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ the Coriolis and centripetal forces, and $\boldsymbol{g}$ the gravity terms. In the following exercises assume that these terms are given.

For the programming exercises you will use the attached code. We provide skeletons for controlling the system either in joint space (`my_ctl.py`) or in task space (`my_taskSpace_ctl.py`) and a basic functionality for plotting. You can invoke either mode by running `jointCtlComp.py` or `taskCtlComp.py` respectively. Attach a printout with plots and a snippet of your source code for each programming exercise.

a) PID Controller [2 Points]

What is the form of a proportional-integral-derivative (PID) controller and how could you use it to control a robot, i.e. what physical quantities could you control? Name one positive and one negative aspect of PID controllers.

Form of a PID controller: $u(t) = K_P e(t) + K_I \int_{-\infty}^{t} e(\tau)\, d\tau + K_D \frac{de(t)}{dt}, \qquad e(t) = q_{desired} - q$

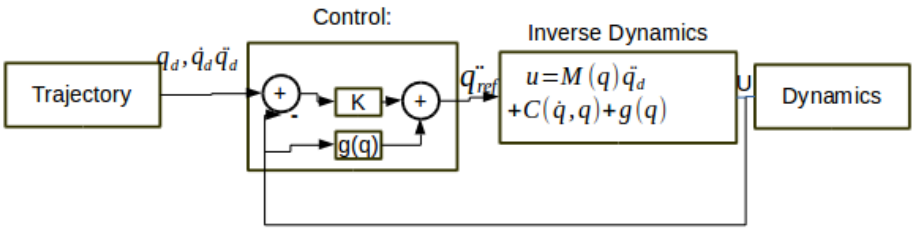
Quantities to control:

- KP Proportional part adjusts the gain of the control loop. Increasing the gain makes it faster.
- KI Integral part used to prevent any residual steady state error.
  Caution if the KI is to high overshooting the set point is likely
- KD Derivative part improves the settling time and the stability of the system.
  (Caution: an ideal derivative is not causal)

Compared to PD,P Controllers, PID Controllers have no steady state error. However if the model is not precisely known it is not advisable to use PID controllers especially not for tracking control. Moreover you should keep in mind, that a WINDUP might occur.

b) Gravity Compensation and Inverse Dynamics Control [4 Points]

Suppose that you would like to create a control law to set the joint angles on the double inverted pendulum model by controlling the torque of the motors. Write a feedback control law which additionally gravity compensates and then extend it to full inverse dynamics control.



$$\ddot{q}_{ref} = \ddot{q}_d + K_D(\dot{q}_{des} - \dot{q}) + K_P(q_{des} - q) + g(q)$$

c) Comparison of Different Control Strategies [12 Points]

In the following exercise you will investigate the differences of the following control algorithms, P, PID, PD with gravity compensation, and full inverse dynamics. The double pendulum is initiated hanging down, with state $\boldsymbol{q}_{\text{start}} = [-\pi, 0]$. We simulate the system with a time-step $dt = 0.002$ seconds using symplectic Euler integration and run the simulation for $t_{\text{end}} = 3s$.

Implement the control laws by filling the skeleton file `my_ctl.py`. Use the following feedback gains $K_P = 60, K_D = 10, K_I = 0.1$ for the first joint and $K_P = 30, K_D = 6, K_I = 0.1$ for the second one. The target state of the double pendulum is set to $\boldsymbol{q}_{\text{des}} = [-\pi/2, 0]$.

Create (max. 4) plots that compare the different control strategies and analyze the results. It is your choice how to illustrate your results. In your analysis you should include a discussion on the overall performance of each controller. Which controllers manage to go to the desired point, and how does the choice of a controller affects the behavior of the second joint of the pendulum? Additionally discuss which controller you would choose and why. The provided code is able to generate plot but feel free to modify it if you like. Points will be deducted for confusing plots. Do not forget to include your source code in your solutions.

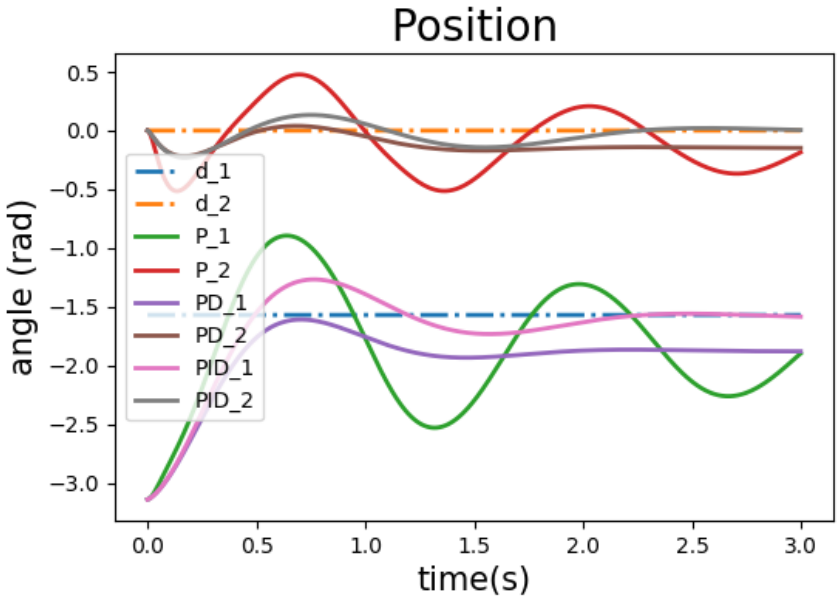
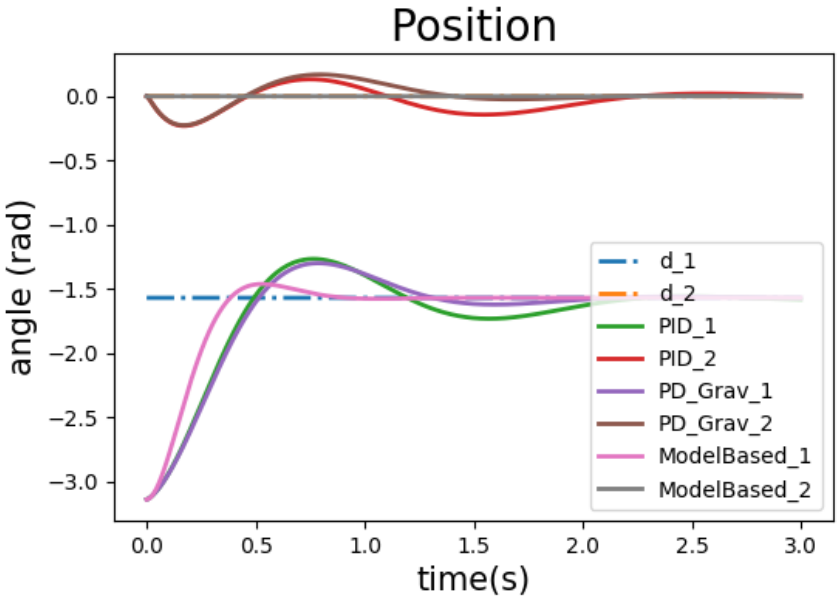


Figure 5: Comparing the position of the P,PD,PID controller

First of all we compare the positioning behavior of the P,PD and PID controller. In figure we can see, that the P controller oscillates around the desired position. The PD controller has a steady state error. Only the PID controller behaves nicely and has no steady state error and does not overshoot the set point too much. Moreover the PID controller is not oscillating.



The PD_Grav. controller reaches the desired position of joint2 faster than the PID controller. However this costs a little bit more overshooting. Comparing all 3 controllers in figure 6 it is clear that the model based controller has the optimal positioning behavior. For joint1 it reaches the desired position quickest with minimal overshooting. In case of joint2 the model based controller is able to not oscillate at all.

Figure 6: Comparison of the position of PID, PD_GRAV and MPC

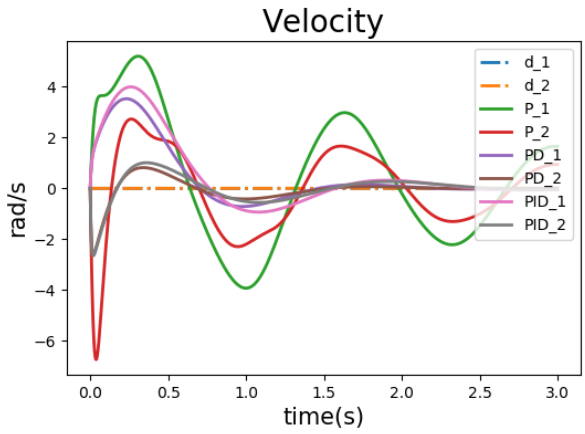


Figure 7: Comparing the velocity of the P,PD,PID controller

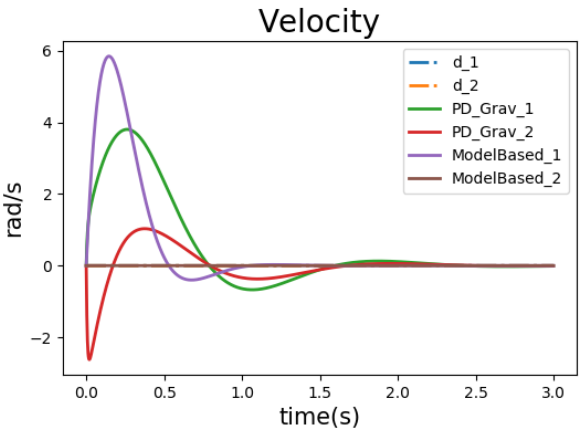


Figure 8: Comparison of the velocity of PD_GRAV and Model Based controller

Having a closer look to the velocity behavior of the controllers reveals that the all controllers (exept the P) reach the desired velocity of 0 rad/sec within 3 seconds. However again the model based controller is the fastest.

In total it can be said the model based controller has the fastest positioning and velocity behavior of all controllers. Moreover it is the only one which is able to just stay at the desired position if the current position is equal to the desired position (see position for joint2). Neglecting any actuator saturation ("Stellgrössenbeschränkung") the model based controller is the best one. However if you have to care about an actor saturation you will probably not choose the model based controller, as it has the highest requirements on the actuators (see e.g. the high overshooting of the velocity (purpel line)). So if you have to care about actuator saturation the PD_Grav is definitely better than the model based controller.

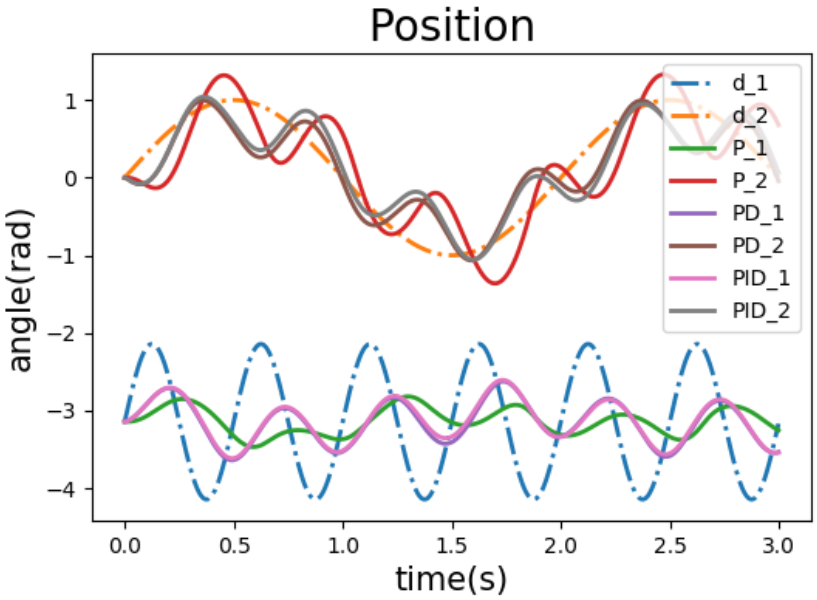
My_clt.py:

```
# my_ctl.py computes the torques u vector with the given values!

import numpy as np
```

```
Kp = np.array([[60,30]])
Ki = np.array([[0.1,0.1]])
Kd = np.array([[10,6]])

def my_ctl(ctl, q, qd, q_des, qd_des,
    qdd_des, q_hist, q_deshist, gravity, coriolis, M):
    u = np.zeros((2, 1))
    e = (q_des-q)
    ed = (qd_des-qd)
    ei=q_deshist-q_hist

    if ctl == 'P':
        u=Kp*e
    elif ctl == 'PD':
        u = Kp*e+Kd*ed
    elif ctl == 'PID':
        e_int = np.sum(ei,0)
        u = Kp*e+Kd*ed+Ki*e_int
    elif ctl == 'PD_Grav':
        u = Kp*e+Kd*ed+gravity
    elif ctl == 'ModelBased':
        qddref = qdd_des+Kd*ed+Kp*e
        u = np.dot(qddref,M)+coriolis+gravity
    return u.T
```

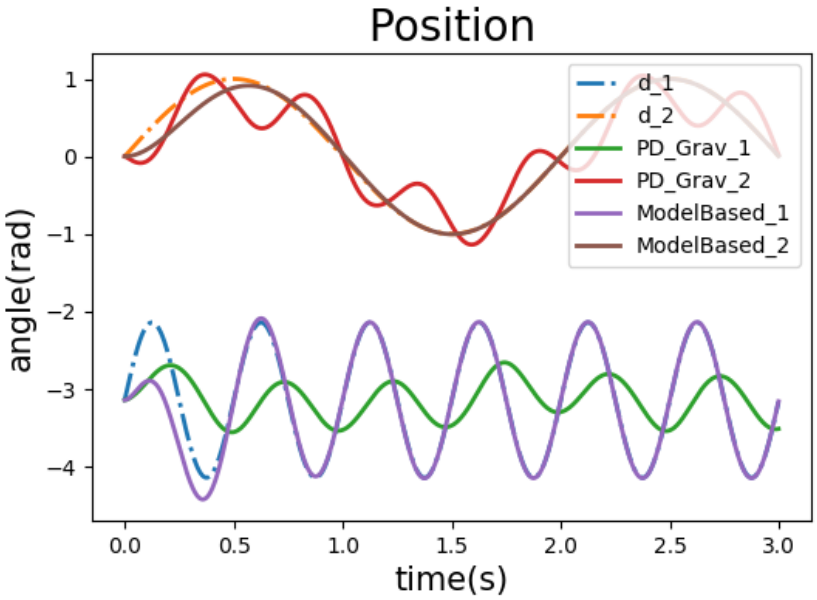
d) Tracking Trajectories [4 Points]

Repeat the same experiment but this time use the provided time-varying target trajectory. Create (max 4) plots that compare the different control strategies and analyze the results. In your analysis discuss the overall performance and which controllers track the desired trajectory nicely. Additionally discuss which controller you would choose and why.



In figure 13 you can see that the tracking behavior of the P,PD,PID controllers fails. The desired positions of joint1 cannot be reached. Especially for joint1 we also can see that there is a time offset between the desired trajectory and the actual trajectories of the controllers.

Figure 13: Comparison of the P,PD and PID positioning tracking behavior



In figure 14 you can see that the tracking behavior of the PD_Grav and model based controllers differs a lot. The desired positions of joint1 and joint2 can only be reached by the model based controller. The PD_Grav controller is faster than the P,PD,PID controller, however not fast enough to track the desired position of joint1 and joint2. It has a time offset and also does not reach the desired positions (see especially for joint1 green line).

Figure 14: Comparison of the PD_Grav. and model based controller's positioning tracking behavior

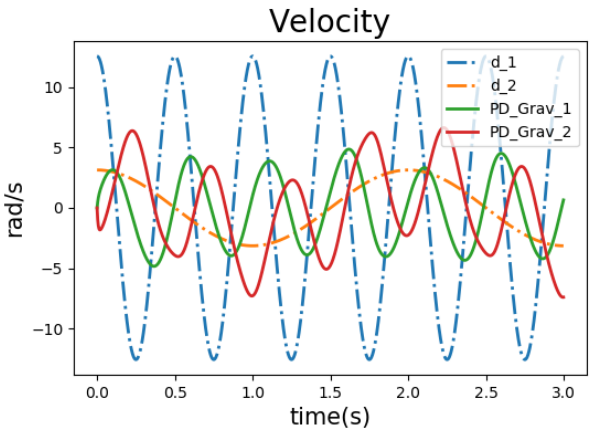


Figure 15: The PD_Grav tracking velocity behavior

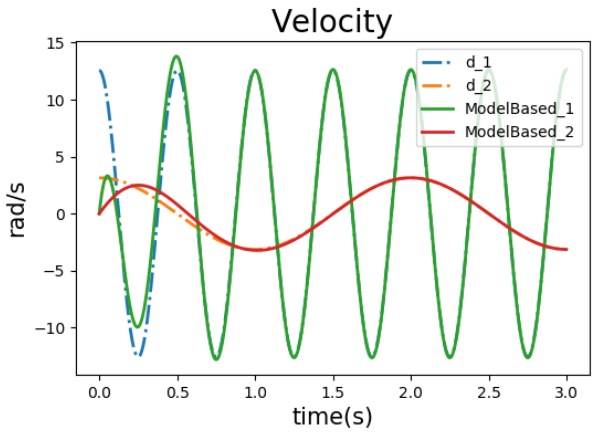


Figure 16: The model based controller's velocity tracking behavior

Having a look at the two best positioning tracking controller's (the PD_Grav and the model based) reveals, that their velocity tracking behavior also differs a lot. In 16 you can see that the velocity tracking behavior of the model based controller works perfectly well (with zero offset after some time), whereas the velocity behavior of the PD_Grav controller fails (see 15). In terms of tracking a trajectory the model based controller must be chosen, cause it is the only controller of the compared ones which is able to track a trajectory with zero offset. To track a trajectory a controller is required to reach a desired point fast enough with zero offset. This requirement is especially met by the model based controller.

e) Tracking Trajectories — High Gains [4 Points]

Repeat the same experiment (using the provided trajectory) but this time multiply the gains by ten. Create plots that compare the different control strategies and analyze the results. In your analysis discuss the overall performance and compare it to the previous case. Are there any drawbacks of using high gains?

$K_{P,new} = K_{P,old} * 10 = [600, 300]^T$
$K_{I,new} = K_{I,old} * 10 = [1, 1]^T$
$K_{D,new} = K_{D,old} * 10 = [100, 60]^T$

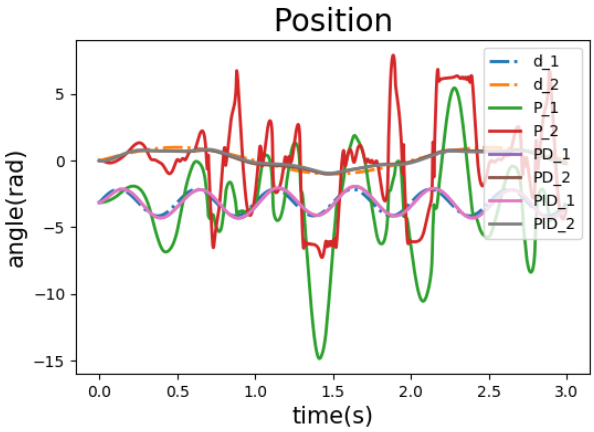


Figure 19: P,PD,PID position tracking behavior with high gains

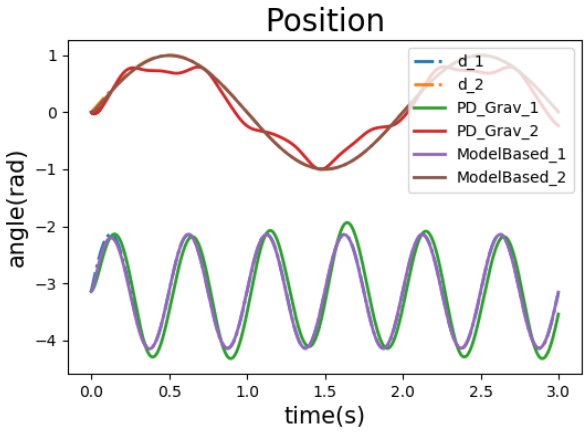


Figure 20: PD_Grav, Model based position tracking behavior with high gains

Having a look at figure 19 one can see, that higher gains improve the positioning tracking behavior of the PD and PID controller. However the P controller's tracking behavior is worse (compared to d)) cause it shows unstable behavior. So it can be concluded that higher gains might destabilize a system. The PD and PID controller however work better(compared to d)), because increasing their gains results in a faster step response and now both controllers are fast enough to track the desired trajectory. Due to the derivative part these two controllers are also stable. Having a close look on the PD and PID controllers one can investigate that both have still a small time offset.

Having a look at figure 20 one can see, that (neglecting any noise increasing effects and actuators saturation levels) the model based controller still behaves best. It as almost zero time offset and tracks the desired position for both joints perfectly. The PD_Grav. PD and PID controllers behave all very similar. All have a small time offset and the small positioning errors of joint1 are even increased for tracking positioning accuracy of joint2.

In total it should be said, that when using higher gain's the stability of the system has to be rechecked and also the actuators saturation level. Moreover increasing especially the K_D's gain results in the fact, that the noise of a system is also increased. For this reason it is in general dangerous to higher the gain's too much.

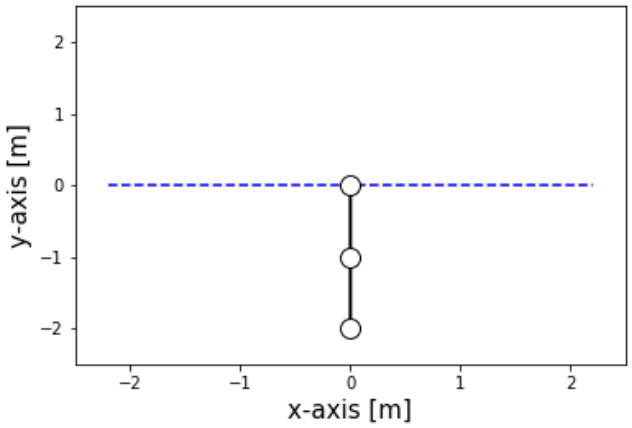
Neglecting any actuators saturation level's the Model based controller still has the best positioning tracking performance and should therefore be chosen.

f) Task Space Control [5 Bonus Points]

The robot must now reach a desired position in task space $\boldsymbol{x}_{\text{end}} = [-0.35, 1.5]$. In class we derived the Jacobian transpose, Jacobian pseudo-inverse, and Jacobian pseudo-inverse with damping methods. All of them are implemented in `my_taskSpace_ctl.py`. You are asked to implement also the null-space task prioritization method with a null-space resting posture $\boldsymbol{q} = [0, \pi]$. Run the simulation and plot the initial and final configuration of the robot. Then, change the resting posture to $\boldsymbol{q} = [0, -\pi]$ and redo the plots. Analyze in a couple of sentences your observation. Use the same damping coefficient 10^{-6} and include a code snippet to your solutions.

Note:
Task space is the cartesian space where the operation of robot is required. It has X,Y and Z ortho normal axes and Roll, Pitch and Yaw rotations about each axes. In other words, it is the space in which we live.

Joint/Configuration space as the name suggests describes a particular configuration of robot or also called Posture. These postures are defined by individual and independent actuation of the joints. That means, it is those joints (revolute, prismatic, spherical, cylindrical etc.,) which do not depend on any other joint. The other name of the NUMBER that signifies the independency which describes the posture of the robot in concrete terms is called Degrees of Freedom.



Comparing the actual end position with the desired end position(-0.35,1.5) in task space:

method	endx	endy
JacTrans	-0.366	1.476
JacPseudo	0.137	1.1264
JacDPseudo	-0.326	1.512
JacNullSpace[0,pi]	-0.3247	1.512
JacNullSpace[0,-pi]	-0.2985	1.4905

Figure 24: Task Space Control: Initial Position

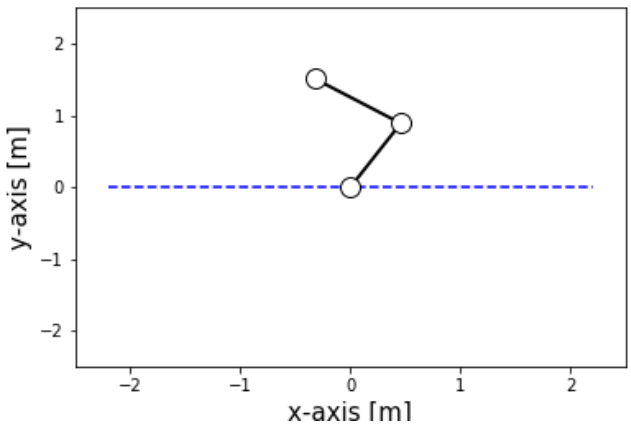


Figure 25: Task Space Control Jac Null Space with resting pose [0,pi]

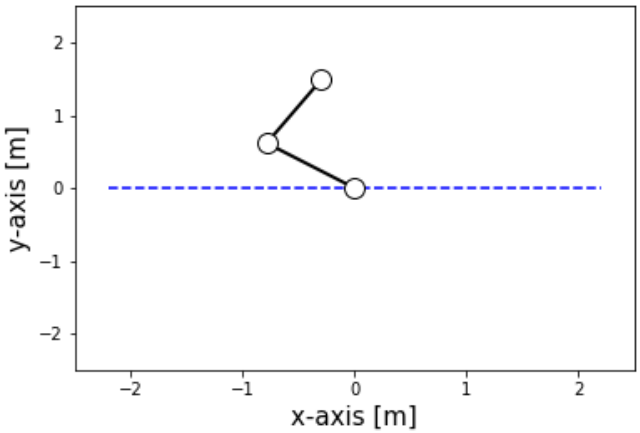


Figure 26: Task Space Control Jac Null Space with resting pose [0,-pi]

Using the JacDPseudo method the desired position is almost reached. However using the JacNullSpace method the desired position is approximated in two different ways one from the "left" (see 25) and one from the right (see 26)

My_taskspace.py:

```
#my_tastspace.py
import numpy as np
from math import pi

def my_taskSpace_ctl(ctl, dt, q, qd, gravity, coriolis,
 M, J, cart, desCart, resting_pos=None):
    KP = np.diag([60, 30])
    KD = np.diag([10, 6])
    gamma = 0.6
    dFact = 1e-6

    if ctl == 'JacTrans':
        #see 91 in script: qdot = gamma * J^T e
        qd_des = gamma * J.T * (desCart - cart)
        error = q + qd_des * dt - q
        errord = qd_des - qd
        #u=M qdotdotref +c +g
        #qdotdotref = KD*(qdotdes - qdot)+KP*(q+qd_des*dt -q)
        u = M * np.vstack(np.hstack([KP,KD])) * np.vstack([error,errord]) + coriolis + gravit
    elif ctl == 'JacPseudo':
        #see 91: qdot = eta*J^T Jpseudo *e
        qd_des = gamma * J.T * np.linalg.pinv(J * J.T) * (desCart - cart)
        error = q + qd_des * dt - q
        errord = qd_des - qd
        u = M * np.vstack(np.hstack([KP,KD])) * np.vstack([error,errord]) + coriolis + gravit
    elif ctl == 'JacDPseudo':
        #Is this the damped Pseudoinverse?
        #see 97
        qd_des = J.T * np.linalg.pinv(J * J.T + dFact * np.eye(2)) * (desCart - cart)
        error = q + qd_des * dt - q
        errord = qd_des - qd
        u = M * np.vstack(np.hstack([KP,KD])) * np.vstack([error,errord]) + coriolis + gravit
    elif ctl == 'JacNullSpace':
        #See 94: qdot = Jpseudo xdot +(I-Jpseudo J)*KP(qrest-q)
        #See 97!
        qrest=resting_pos
        qq0=KP*(qrest-q)
        pseudo=J.T *np.linalg.inv(J*J.T+dFact*np.eye(len(J)))
        qd_des = pseudo*(desCart-cart)+(np.eye(len(J))-pseudo*J)*qq0
        error = q + qd_des * dt - q
        errord = qd_des - qd
        u = M * np.vstack(np.hstack([KP,KD])) * np.vstack([error,errord]) + coriolis + gravit
    return u
```

Name, Surname, ID Number

Problem 1.3 Machine Learning in a Nutshell [30 Points + 5 Bonus]

For this exercise you will use a dataset, divided into training set and validation set (both attached). The first row is the vector $\boldsymbol{x}$ and the second row the vector $\boldsymbol{y}$.
Based on these data, we want to learn a function mapping from $\boldsymbol{x}$ values to $\boldsymbol{y}$ values, of the form $\boldsymbol{y} = \boldsymbol{\theta}^T \boldsymbol{\phi}(\boldsymbol{x})$.

For all questions requiring a plot, you also have to provide a snippet of your code!

You are allowed to use `scipy.spatial.distance.cdist` and `scipy.exp`.

a) Supervised vs Unsupervised Learning [2 Points]

Briefly explain the differences between supervised and unsupervised learning. Is the above a supervised or unsupervised learning problem? Why?

Supervised(predictive) Learning ("Überwachtes Lernen"):

Predict unknown values. The given data actually has a label already. Each datapoint of the given data has an input and an output value: $f(\mathbf{X}) = \mathbf{y}$
Examples are Regression and Classification.

Unsupervised(descriptive) Learning:

Understand given data.Find structures in data. This means labels have to be found first! Examples are Clustering, Dimensionality Reduction,Density Estimation.

In the given example we have already the "labels": We have for each x an y. For this reason it is supervised learning.

See also http://oliviaklose.azurewebsites.net/machine-learning-2-supervised-versus-unsupervised-learning/

b) Regression vs Classification [2 Points]

Supervised learning is typically divided into regression and classification tasks. Briefly explain what are the differences between regression and classification.

Regression:

Regression algorithms are algorithms that learn to predict the value of a real function for a single instance of data. Regression algorithms can incorporate input from multiple features, by deterng the contribution of each feature of the data to the regression function.

Once the regression algorithm has trained a function based on already labeled data, the function can be used to predict the label of a new (unlabeled) instance. For example, a housing price predictor might use a regression algorithm to predict the value of a particular house, based on historical data about regional house prices.

Examples:

Bayesian Linear Regression	Creates a Bayesian linear regression model
Boosted Decision Tree Regression	Creates a regression model using the Boosted Decision Tree algorithm
Linear Regression	Creates a linear regression model
Neural Network Regression	Creates a regression model using a neural network algorithm

Classification:

Is a method in machine learning of using data to determine the category, type, or class of an item or row of data.

c) Linear Least Squares [4 Points]

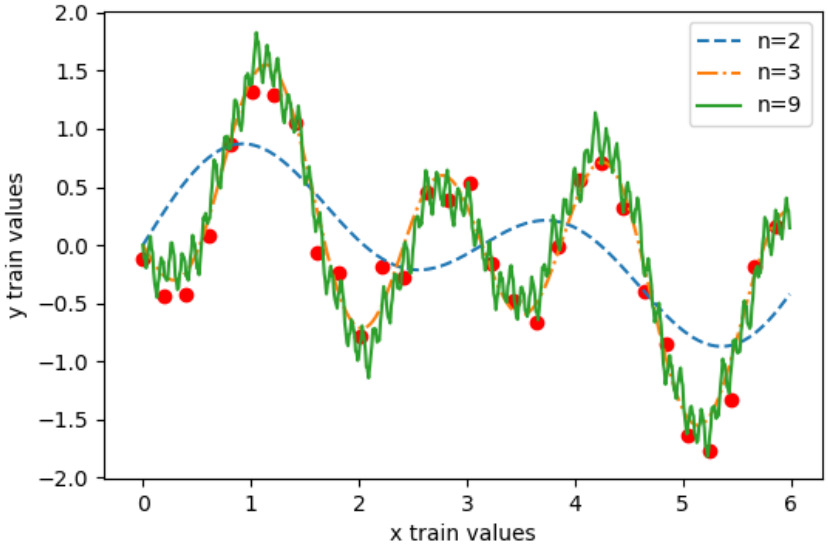
Consider the training set above to calculate features $\boldsymbol{\phi}(x)$ of the form $[\sin(2^i x)]_{i=0\ldots n-1}$. Compute the feature values when n is 2, 3 and 9 (i.e., when using 2, 3 and 9 features). Use the linear least squares (LLS) method to predict output values y for input values $x \in \{0, 0.01, 0.02, \ldots, 6\}$ using the different numbers of features. Attach a single plot showing the three resulting predictions when using 2, 3 and 9 features (i.e., having x and y as axes).

Some Theory(LLS) (example for 2 Features):

$$x = (A^T A)^{-1} A^T b \qquad \phi(x) = a_0 sin(x) + a_1 sin(2x) \qquad A = \begin{bmatrix} sin(x_1) & sin(2x_1) \\ sin(x_2) & sin(2x_2) \\ \ldots & \ldots \\ sin(x_n) & sin(2 * x_n) \end{bmatrix} x = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}, b = \begin{bmatrix} y_0 \\ \ldots \\ y_n \end{bmatrix}$$

In the notation of this lecture:

$$y_i = \phi(x_i)^T \boldsymbol{\Theta} + \epsilon, \quad Parameters: \Theta = \begin{bmatrix} a_0 \\ \ldots \\ a_n \end{bmatrix}, \quad Features: \phi(x_i) = \begin{bmatrix} sin(x) \\ \ldots \\ sin(2^n x) \end{bmatrix}, \Phi = [\phi_1, \ldots, \phi_n]^T, Y = [y_1, \ldots, y_n]^T, \quad \Theta = (\Phi^T \Phi)^{-1} \Phi^T Y$$

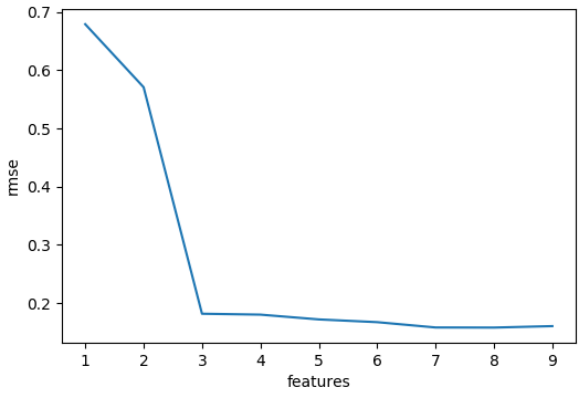



```
# A1.3c Linear Least Squares
def phi(x_t, n_t):
    # this function calculates the value of the feature function
    # x_t: x value, n_t number of features
    phi_t = np.zeros((n_t,np.size(x_t)))
    for i in xrange(n_t):
        phi_t[i]=np.sin(pow(2,i)*x_t)
    return phi_t

def calc_y(x_t, n_t, theta_t):
    # this function calculates the prediction y from given values for x and theta
    phi_t = phi(x_t,n_t)
    return np.dot(phi_t.T,theta_t)

def calc_theta(n,x_train,y_train):
    # this function calculates the theta parameter with the left pseudo-inverse
    N = np.size(x_train)
    Phi = np.zeros((N,n))
```

```
    Y = np.array([y_train])
    Y = Y.T

    for i in xrange(N):
        Phi[i,...] = phi(x_train[i], n).T

    # calculate left pseudo-inverse
    temp = np.dot(Phi.T, Phi)
    temp = np.dot(inv(temp), Phi.T)

    return np.dot(temp, Y)
```

d) Training a Model [2 Points]

The root mean square error (RMSE) is defined as RMSE $= \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(y_i^{\text{true}} - y_i^{\text{predicted}}\right)^2}$, where N is the number of data points. Using the LLS algorithm implemented in the previous exercise, train a different model for each of the number of features between 1 and 9, i.e., [1,2,3...,9]. For each of these models compute the corresponding RMSE for the training set. Attach a plot where the x-axis represents the number of features and the y-axis represents the RMSE.




```
# A1.3d Training a Model
def calc_rmse(y_true, y_pred):
    # this function calculates the RMSE from given y-values
    rmse_t = np.power(y_true-y_pred,2)
    return np.sqrt(np.mean(rmse_t))


nlist = np.array(range(1,10))
rmse_train = []

for i in nlist:
    # calc the rmse for n=1...9
    theta = calc_theta(i,x_train,y_train)
    y_pred = calc_y(x_train,i,theta)

    rmse_train.append(calc_rmse(Y_train,y_pred))
```

e) Model Selection [4 Points]

Using the models trained in the previous exercise, compute the RMSE of each of these models for the validation set.
Compare in one plot the RMSE on the training set and on the validation set. How do they differ? Can you explain what is the reason for these differences? (Hint: remember the plot from Exercise c)) What is the number of features that you should use to achieve a proper modeling?

Note: Use the parameters a_i from c) on the validation dataset!

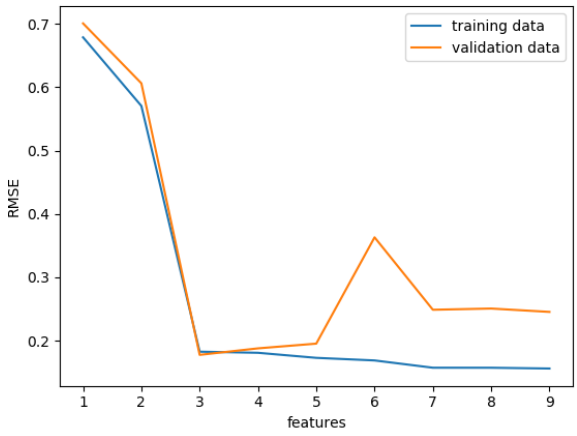


Figure 29: Comparing RMSE

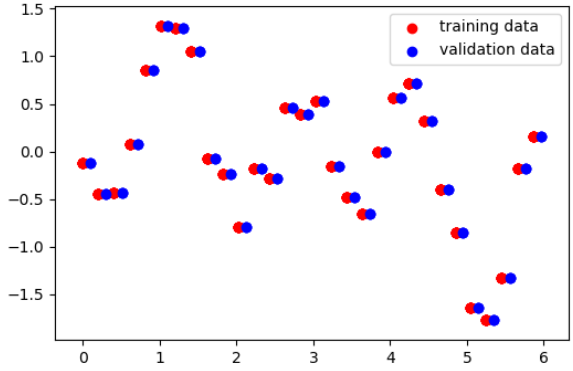


Figure 30: Data sets

How do they differ:
Using 3 features the training set and the validation set have have the same rmse. Afterwards the rmse of the training set is marginally decreasing and the rmse of the validation set is increasing.

What is the reason for these differences:
Using three features the model is sufficiently described (see e.g c). If you use more features the model is overfitted. Thus the rmse is increasing

What is the number of features that you should use to achieve a proper modeling:
As you can see in figure 29 the smallest rmse is reached for 3 features. For this reason 3 features should be used.

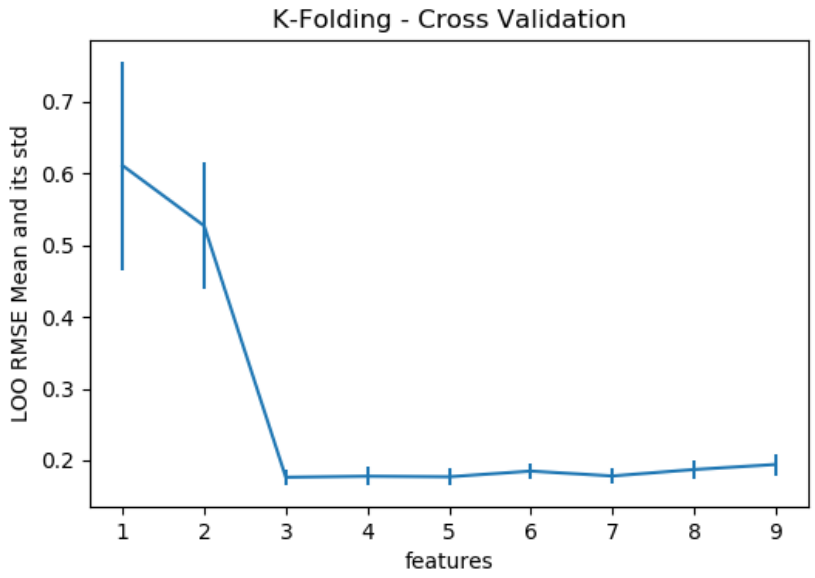
```
# A1.3e Model Selection

rmse_val = []

for i in nlist:
    # calculate the RMSE for n=1...9 of the training set
    theta = calc_theta(i,x_train,y_train)
    y_val = calc_y(x_val,i,theta)

    rmse_val.append(calc_rmse(Y_val,y_val))
```

f) Cross Validation [8 Points]

K-fold cross validation is a common approach to estimate the test error when the dataset is small. The idea is to randomly divide the training set into K different datasets. Each of these datasets is then used as validation set for the model trained from the remaining $K-1$ datasets. The resulting vector of errors $\boldsymbol{E} = [e_1 ... e_K]$ can now be used to compute a distribution (typically by fitting a Gaussian distribution). When K is equal to the number of data points, K-fold cross validation takes the name of leave-one-out cross validation (LOO).

Apply LOO using only the training set and compute the mean/variance of the RMSE for the learned models. Repeat for the models with the number of features between 1 and 9, i.e., [1,2,3...,9]

Attach a plot showing the mean/variance (as a distribution) of the RMSE computed using LOO and having on the x-axis the number of features and on the y-axis the RMSE. Which is the optimal number of features now? Discuss the results obtained and compare against model selection using train/validation set.




```
# A1.3f Cross Validation
def calc_loo(x, y, nlist):
    # this function uses the LOO algorithm for a given number of features
    rmse = np.zeros((np.size(x), np.size(nlist)))

    for i in xrange(np.size(x)):
        # choose one element as validation data, rest is training data
        x_train = np.delete(x,i)
        y_train = np.delete(y,i)
        x_val = x[i]
        y_val = y[i]

        for pos,j in enumerate(nlist):
            # calculate the RMSE for the different numbers of features
            theta = calc_theta(j,x_train,y_train)
            y_pred = calc_y(x_val,j,theta)
            rmse[i,pos] = calc_rmse(y_val,y_pred)

    means = []
    variance = []

    for i in xrange(np.size(nlist)):
        # calculate the mean and variance
        means.append(np.mean(rmse[:,i]))
```

```
        variance.append(np.var(rmse[:,i]))
```

g) Kernel Functions [2 Points]

A kernel function $k(x_i, x_j)$ is given by the inner product of two feature vectors. Write out the kernel function for the previous set of features where $n = 3$.

$$k(x_i, x_j)_{n=3} = \begin{bmatrix} sin(x_i) & sin(2x_i) & sin(4x_i) \end{bmatrix} \begin{bmatrix} sin(x_j) \\ sin(2x_j) \\ sin(4x_j) \end{bmatrix} = sin(x_i)sin(x_j) + sin(2x_i)sin(2x_j) + sin(4x_i)sin(4x_j)$$

References:
https://de.wikipedia.org/wiki/Kernel_(Maschinelles_Lernen)
https://en.wikipedia.org/wiki/Kernel_(statistics)

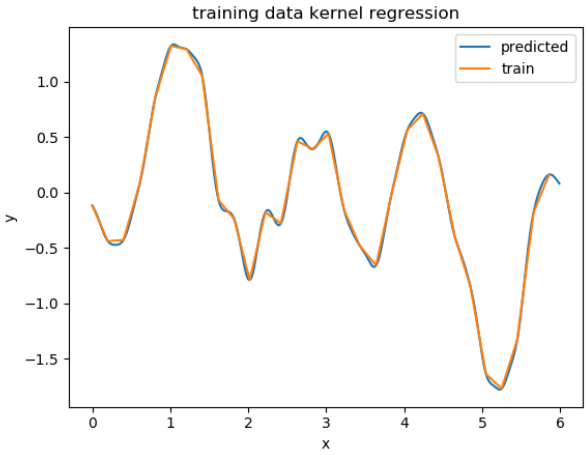
h) Kernel Regression [6 Points]

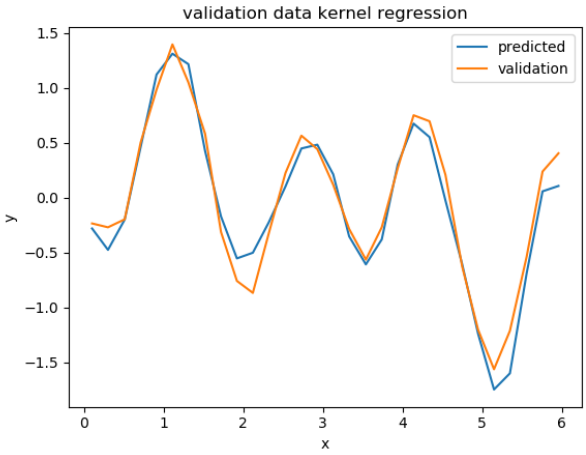
The kernel function in the previous question required explicit definition of the type and number of features, which is often difficult in practice. Instead, we can use a kernel that defines an inner product in a (possibly infinite dimensional) feature space.
Using the training set and an exponential squared kernel $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(-\frac{1}{\sigma^2}\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2\right)$ with $\sigma = 0.15$, predict output values y for input values $x \in \{0, 0.01, \ldots, 6\}$. Attach a plot of your results.
(Hint: use standard kernel regression: $f(\boldsymbol{x}) = \boldsymbol{k}^\top \boldsymbol{K}^{-1} \boldsymbol{y}$ with $\boldsymbol{K}_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ and $\boldsymbol{k}_i = k(\boldsymbol{x}, \boldsymbol{x}_i)$).

Compute the RMSE on the validation set for the kernel regression model. Compare it with the RMSE of the best LLS model you found.

Some Theory:

- squared Euclidean distance: $||a - b||^2 = \sqrt{(a_1 - b_1)^2 + \ldots + (a_n - b_n)^2}^2 = (a_1 - b_1)^2 + \ldots + (a_n - b_n)^2$
- Kernel function: $k(x_1, x_2) = \phi(x_1)^T \phi(x_2)$, (inner product)
- Kernel regression: $y_{pred} = f(x_{\text{pred}}) = k^T_{\text{pred, dataset}} K^{-1}_{\text{dataset}} y_{\text{dataset}}$
- $k_i = k(x, x_i) = [e^{-\frac{||x_{pred} - x_{0,dataset}||^2}{\sigma^2}}, \ldots, e^{-\frac{||x_{pred} - x_{n,dataset}||^2}{\sigma^2}}]$, Gaussian Radial Basis kernel function
- $K_{ij} = k(x_i, x_j) = \begin{bmatrix} e^{-\frac{||x_{0,dataset} - x_{0,dataset}||^2}{\sigma^2}} & \ldots & e^{-\frac{||x_{n,dataset} - x_{0,dataset}||^2}{\sigma^2}} \\ \ldots & \ldots & \ldots \\ e^{-\frac{||x_{0,dataset} - x_{n,dataset}||^2}{\sigma^2}} & \ldots & e^{-\frac{||x_{n,dataset} - x_{n,dataset}||^2}{\sigma^2}} \end{bmatrix}$





The prediction for the training data is nearly perfect. You can see no difference between the prediction and the rail data. But th prediction for the validation data is not so well, you can see big differences for $x = 2$. The calculated RMSE is 0.2422. So it is slightly bigger compared to the Linear Regression model with three features.

```
# A1.3g Kernel Regression
def kernel(xi,xj, sigma = 0.15):
    # this function calculates the kernel of the two x-values xi and xj
    return np.exp(-1/(sigma**2)*abs(xi-xj)**2)


def kernel_regression(x_train):
    # this function calculates the K-matrix for the kernel regression
    n = np.size(x_train)
    K = np.zeros((n, n))
    for i in xrange(n):
        for j in xrange(n):
            K[i, j] = kernel(x_train[i], x_train[j])
    return K


def kernel_predict(x, X, Y, K):
    # this function uses the K-matrix to predict new values
    f_x = np.zeros((np.size(x), 1))
    for i in xrange(np.size(x)):
        k = kernel(x[i],X)
        f_x[i,0] = np.dot(np.dot(k.T, inv(K)),Y)
    return f_x


K = kernel_regression(x_train)
# calculate the predicted values for the training data
y_pred_kernel = kernel_predict(x_plot, x_train, y_train, K)
# calculate the predicted values for the validation data
y_kernel_pred_val = kernel_predict(x_val, x_train, y_train, K)
# calculate the RMSE
kernel_rmse = calc_rmse(Y_val, y_kernel_pred_val)
```

Refererences:
https://en.wikipedia.org/wiki/Radial_basis_function_kernel

i) Derivation [5 Bonus Points]

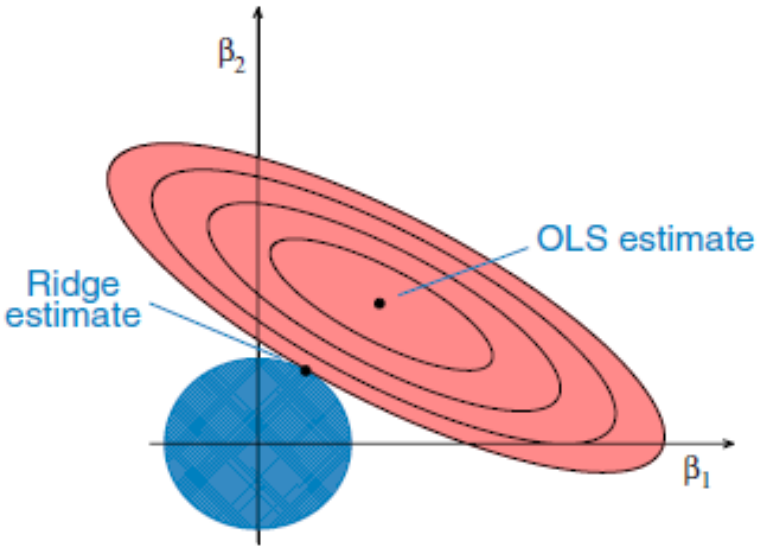
Explain the concept of ridge regression and why/when it is used. Derive its final equations presented during the lecture.
(Hint: remind that for normal linear regression the cost function is $J = \frac{1}{2}\sum_{i=1}^{N}(f(\boldsymbol{x}_i) - y_i)^2$)
(Hint 2: use matrix notation)

Ordinary Linear Squares (OLS):

- Parameters Θ are Best linear unbiased estimators (BLUE).
- The Parameters have the smallest variance out of all other unbiased estimators.
- If 2 or more parameters are highly correlated the variance is quite high and quite different solution's occur for different data sets. In such a case Ridge Regression should be used!
- $f_\Theta = \phi^T(x_i)\Theta$
- $J = \frac{1}{2}\sum_{i=1}^{N}(y_i - f_\Theta(x_i))^2, \quad J = \frac{1}{2}(Y - \Phi\Theta)^T(Y - \Phi\Theta)$
- $\Theta = \left(\Phi^T\Phi\right)^{-1}\Phi^T Y$

Ridge ("Kamm") Regression:

- Estimators are biased and can therefore have an even lower variance compared to OLS.
- Especially if the parameters are highly correlated ridge regression is a powerful method to restrict the correlation of the parameters using a lagrangian multiplier λ.
- $min_\Theta ||y - \Phi\Theta||_2^2 + \lambda||\Theta||_2^2$
- $J = \frac{1}{2}[\sum_i^N (y_i\phi(x_i)\Theta)^2 + \lambda\Theta^T\Theta], \quad J = \frac{1}{2}[(Y - \Phi\Theta)^T(Y - \Phi\Theta) + \lambda\Theta^T\Theta], \quad \frac{\partial J}{\partial\Theta} = 0$
- $J = \frac{1}{2}[Y^TY - Y^T\Phi\Theta - \Phi^T\Theta^TY + \Phi^T\Theta^T\Phi\Theta + \lambda\Theta^T\Theta] = \frac{1}{2}[Y^TY - 2Y^T\Phi\Theta + \Phi^T\Theta^T\Phi\Theta + \lambda\Theta^T\Theta]$
- $\frac{\partial J}{\partial\Theta} = Y^T\Phi - \Phi^T\Phi\Theta + \lambda\Theta = 0, \quad \Phi^TY = \left(\Phi^T\Phi + \lambda I\right)\Theta$
- $\Theta = \left(\Phi^T\Phi + \lambda I\right)^{-1}\Phi^TY$



https://www.youtube.com/watch?v=5asL5Eq2x0A
https://onlinecourses.science.psu.edu/stat857/node/155