ROB 599 HW 2

Prof. Johnson-Roberson and Prof. Vasudevan

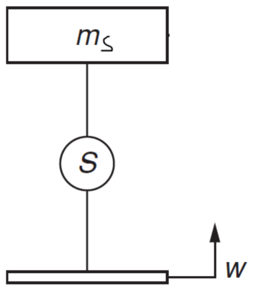
Due: 19 Oct, 2017 at 2:00 PM EST

Submission Details

Use this PDF only as a reference for the questions. You will find a code template for each problem on Cody. You can copy the template from Cody into MATLAB on your personal computer to write and test your own code, but *final code submission must be through Cody*.

Problem 1: State Estimation [25 Points]

Consider the single degree of free model of active suspension depicted below:



with the following state space model:

$$\frac{d}{dt}\begin{bmatrix}x_1\\x_2\end{bmatrix}=\begin{bmatrix}0 & -1\\0 & 0\end{bmatrix}\begin{bmatrix}x_1\\x_2\end{bmatrix}+\begin{bmatrix}0\\\frac{-1}{m_s}\end{bmatrix}u+\begin{bmatrix}1\\0\end{bmatrix}w \tag{1}$$

$$y=\begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}x_1\\x_2\end{bmatrix}+v \tag{2}$$

where x_1 is the suspension stroke (positive in extension), x_2 is the sprung-mass velocity (positive downwards), $m_s = 400$ [kg] is the mass, u is the suspension force, w is the ground velocity that can be modeled as a zero-mean white-noise disturbance, and v is measurement noise that can also be modeled as a zero-mean white-noise disturbance.

1.1 Observability [2 points] Compute the observability matrix, O of the system and its rank. Recall the observability matrix for an n-dimensional system is defined as:

$$O = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{n-1} \end{bmatrix} \tag{3}$$

and the system is considered <u>observable</u> if rank$(O) = n$.

1.2 Control and Observer Design

1.2.1 Feedback Design [2 points] Recall for an n-dimensional system, one can design gains, K, to place the poles of a closed loop system at the zeros of a polynomial function, Φ_c, according to Ackerman's formula:

$$K = [0\ 0 \cdots 0\ 1] \cdot [B\ AB \cdots\ A^{n-2}B\ A^{n-1}B]^{-1} \cdot \Phi_c(A) \tag{4}$$

Use this formula to design a feedback, K, to place both closed loop poles of the active suspension at -1 to ensure that a stroke of 0 and velocity of 0 are stable. K should be a 1x2 matrix.

1.2.2 Observer Design [2 points] Recall for an n-dimensional system, one can design observer gains, G, to place the poles of an observer at the zeros of a polynomial function, Φ_0, according to Ackerman's formula:

$$G = \Phi_o(A) \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{n-2} \\ CA^{n-1} \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \tag{5}$$

Use this formula to design a feedback, G, to place both closed loop poles of the observer to the active suspension model at -3. G should be a 2x1 matrix.

1.2.3 Simulate [5 points] Suppose the active suspension model has states x and the observer to this model has states $\hat{x}$. Assume that the active suspension model has additive zero-mean Gaussian noise, v, with covariance $\sigma_v = 0.0005$, in its system dynamics. In addition, assume that the measurement of the active suspension model has additive zero-mean Gaussian noise, w, with covariance

$\sigma_w = 0.0005$. In other words, the full system dynamics can be written down as:

$$\frac{dx}{dt}(t) = Ax(t) - BK\hat{x}(t) + \begin{bmatrix} w(t) \\ 0 \end{bmatrix} \tag{6}$$

$$y(t) = Cx(t) + v(t) \tag{7}$$

$$\frac{d\hat{x}}{dt}(t) = [A - GC]\hat{x}(t) + Gy(t) - BK\hat{x}(t) \tag{8}$$

$$\tag{9}$$

Use Euler's method to forward simulate this full system from the initial condition $x_1 = \begin{bmatrix} 0.1 \\ 0.1 \end{bmatrix}$ and $\hat{x}_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ for 10 seconds at 100hz (a time span of $t = [0 : 0.01 : 10]$). Note in MATLAB, the noise can be simulated at each timestep using $w = \text{randn}\sqrt{\sigma_w}$ and $v = \text{randn}\sqrt{\sigma_v}$. Plot the trajectories of each state. Save the trajectories as 2x1001 matrices called "x" and "x_hat."

1.3 LQR Design

Next you will use LQR to design both the feedback controller and observer.

1.3.1 Feedback Design [3 points]

Find the feedback matrices $K_{lqr} : [0, 10] \to \mathbb{R}^{1\times 2}$ to minimize the cost function

$$J_c = \int_0^{10} (x^T Qx + u^T Ru)dt, \tag{10}$$

where $Q = \begin{bmatrix} 100 & 0 \\ 0 & 100 \end{bmatrix}$ and $R = 5 \cdot 10^{-5}$. K_{lqr} can be found by solving the following set of equations:

$$\begin{aligned} -\dot{P}(t) &= A^T P(t) + P(t)A - P(t)BR^{-1}B^T P(t) + Q \\ K_{lqr}(t) &= R^{-1}B^T P(t) \end{aligned} \tag{11}$$

Use the provided lqr_LTV function (check the code from Lecture 4) and a timespan $t = [0 : 0.01 : 10]$ to find K_{lqr}. Note that A, B are the same for every timestep in this case. Your answer should be a 1x1001 cell where each element of the cell is a 1x2 matrix.

1.3.2 Observer Design [3 points]

Now use LQR to select gains $G_{lqr} : [0, 10] \to \mathbb{R}^{2\times 1}$ for the observer that minimize the cost function:

$$J_o = \int_0^{10} (e^T Qe + z^T Rz)dt, \tag{12}$$

where $e = x - \hat{x}$ and $z = y - \hat{y}$. With $Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $R = 1$. Notice how the matrix Q and R balance the observer's focus on the process model (states) and measurements, respectively. The cost, J_o, is minimized by solving the following set of equations:

$$\begin{aligned} -\dot{P}(t) &= AP(t) + P(t)A^T - P(t)C^T R^{-1} CP(t) + Q \\ G_{lqr}(t) &= P(t)CR^{-1} \end{aligned} \tag{13}$$

Solve for G_{lqr} with a timespan of $t = [0 : 0.01 : 10]$. Notice the duality between the observer and feedback problems. You can use the same method that you used in part 1.3.1, just with $A \implies A^T$, $B \implies C^T, K \implies G^T$. Your answer should be 1x1001 cell where each element of the cell is a 2x1 matrix.

1.3.3 Simulation [4 points] Using the gain matrices K_{lqr}, G_{lqr}, simulate the system response using the same timespan, initial conditions, and noise from part 1.2.3. Plot the trajectories of each state. Save the trajectories as 2x1001 matrices called "x_lqr" and "x_hat_lqr".

1.4 Comparison [4 points]

Plot the differences between the state and observer $x - \hat{x}$ and $x_{lqr} - \hat{x}_{lqr}$ over time for each state. Which observer converges faster? Which observer is less noisy? Enter 'acker' or 'lqr' for your answers in the autograder (include the apostrophes)

Problem 2: LQR Bicycle Model [25 points]

Consider the Kinematic Bicycle Model:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} \frac{v}{\ell}(\ell \cos(\psi) - d \sin(\psi) \tan(\gamma)) \\ \frac{v}{\ell}(\ell \sin(\psi) + d \cos(\psi) \tan(\gamma)) \\ \frac{v}{\ell} \tan(\gamma) \end{bmatrix}, \tag{14}$$

where x and y are the coordinates of the center of mass in the ground plane, ψ is the heading angle, l is the length of the wheel base, d is the distance to the center of mass from the rear axle, γ is the steering angle, and v is the longitudinal velocity.

2.1 Simulate Trajectory [5 points]

Let $d = 1.5$ [m], and $\ell = 3$ [m]. Simulate the system for the initial condition $x = y = \psi = 0$ and the timespan $T = [0 : 0.01 : 5]$. Let $\gamma = \arctan(0.3)$ and $v = 2$ [m/s].

2.2 Linearize [5 points] Treating v and γ as inputs, linearize the system about the circular trajectory generated by $\gamma = \arctan(0.3)$, $v = 2$, $\psi(t) = 0.2t$. The linearized system should take the form:

$$\begin{bmatrix}\delta\dot{x}\\ \delta\dot{y}\\ \delta\dot{\psi}\end{bmatrix} = A(t)\begin{bmatrix}\delta x\\ \delta y\\ \delta\psi\end{bmatrix} + B(t)\begin{bmatrix}\delta v\\ \delta\gamma\end{bmatrix} \tag{15}$$

On Cody coursework, write functions for $A(t)$ and $B(t)$.

2.3 LQR Feedback [5 points] Using the provided function lqr_LTV (check the code from Lecture 4), find the optimal feedback gains K when $R = I_{2\times2}$ and $Q = I_{3\times3}$.

2.4 Simulate LQR Control [10 points] Simulate the linearized system for the time span $T = [0 : 0.01 : 5]$ using the feedback gains from the previous step. For the simulation use the provided function ode1. Use the initial condition of

$$\begin{bmatrix}\delta x\\ \delta y\\ \delta\psi\end{bmatrix} = \begin{bmatrix}0.10\\ 0.80\\ 0.01\end{bmatrix} \tag{16}$$

Remember you will have to add back the trajectory that we linearized about. Since we use the same time step here as in 2.1 just add them together. (To visualize the results it may be helpful to plot the trajectory found in 2.1 and the result of this simulation).

Optional Ungraded problems

2.5 Use ode1 to simulate the nonlinear system using the LQR feedback found in 2.3. Use the same initial conditions as 2.4 and compare your response.

2.6 Using the results to 2.5, try adjusting the initial condition farther away from the desired trajectory for both the linear and non-linear simulations. What happens as you move farther away?

2.7 Re-write the the lqr_LTV function so that the inputs *A* and *B* are functions in time instead of time step, and replace the Euler integration with ode45. Run your new function in place of the old one for problem 2.3. Compare the initial feedback gain K found using each function.

Problem 3: Iterative Closest Point (ICP) [30 points]

3.1 ICP in 2D Figure 1 shows two point clouds, target p_t (blue) and source p_s (red). We want to align them by estimating the relative rigid body motion, rotation R and translation t. The target cloud p_t is assumed to be fixed in the world frame, and the source cloud p_s can be align with p_t with rigid body motions.

$$p_s \sim Rp_t + t$$

You can use the provided `get_pnt_2d()` to load the two point clouds.

```
[p_t, p_s] = get_pnt_2d();
plot(p_t(1, :), p_t(2, :), 'r+', p_s(1, :), p_s(2, :), 'bx');
axis equal
```

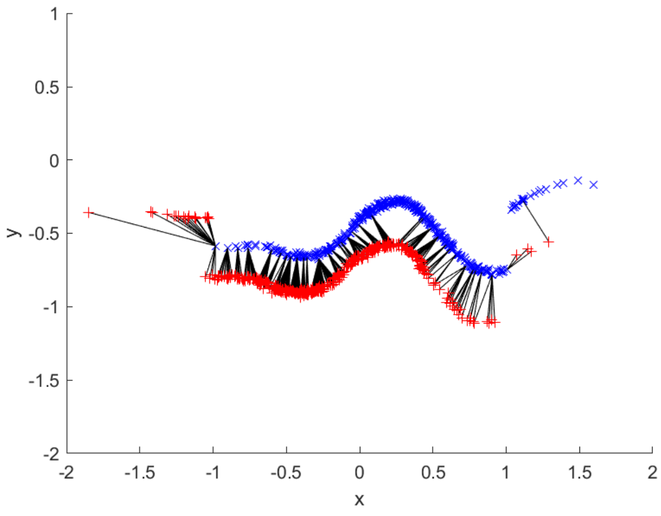


Figure 1: Initial point clouds.

3.1.1 Matching points [5 points] Let n_t and n_s be the number of points in p_t and p_s, respectively. Write a function `match_pnt()` to associate each point in p_s to point(s) in p_t.

```
function [match, min_dist] = match_pnt(p_t, p_s)
% Your code here ...
end
```

The first output `match` is a $n_s \times 1$ vector, representing the indices of the closest points for each point in p_s. In other words, among all the points in the target cloud, `p_t(:, match(i))` should be the closest point to `p_s(:, i)`. The second output `min_dist` is also a $n_s \times 1$ vector, and `min_dist(i)` stores the Euclidean distance between `p_t(:, match(i))` and `p_s(:, i)`. Figure 1 shows the matching pairs indicated by the black lines.

3.1.2 Naive ICP [10 points] Start with the provided code `icp_2d.m`, complete the function `icp_2d()` which computes the total rotation and translation needed to align p_t and p_s.

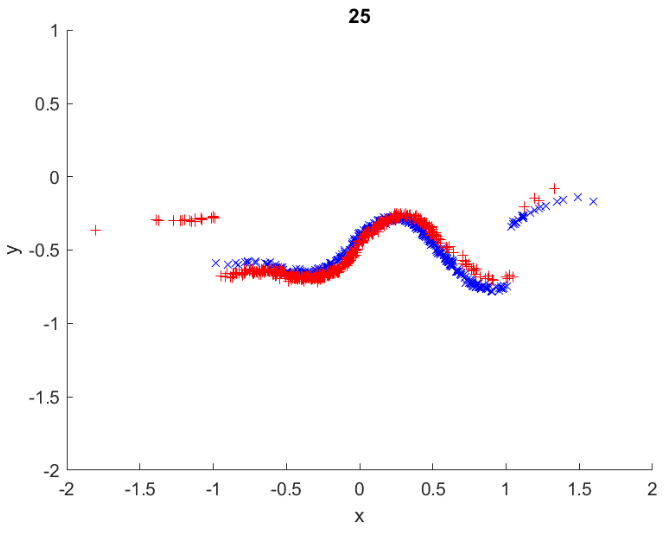


Figure 2: Result from naive ICP.

3.1.3 Weighted ICP [5 points] Weight every point in p_1 and p_2 with a vector $w \in \mathbb{R}^{n_s}$ using `min_dist`.

$$
\begin{aligned}
d_i &= \frac{\texttt{min\_dist(i)}}{\texttt{std(min\_dist)}} \quad \forall i \in \{1,2,\ldots,n_s\} \\
w_i &= \frac{e^{-d_i}}{\sum_{j=1}^{n_s} e^{-d_j}} \quad \forall i \in \{1,2,\ldots,n_s\} \\
\mu_1 &= p_1 w,\ \mu_2 = p_2 w \\
[\bar{p}_1]_i &= ([p_1]_i - \mu_1) w_i \quad \forall i \in \{1,2,\ldots,n_s\} \\
[\bar{p}_2]_i &= ([p_2]_i - \mu_2) w_i \quad \forall i \in \{1,2,\ldots,n_s\}
\end{aligned}
$$

where $[p]_i$ is the $i-$th column of p. Applying the weight w effectively makes ICP focuses more on ”good” pairs thus be more robust to outliers.

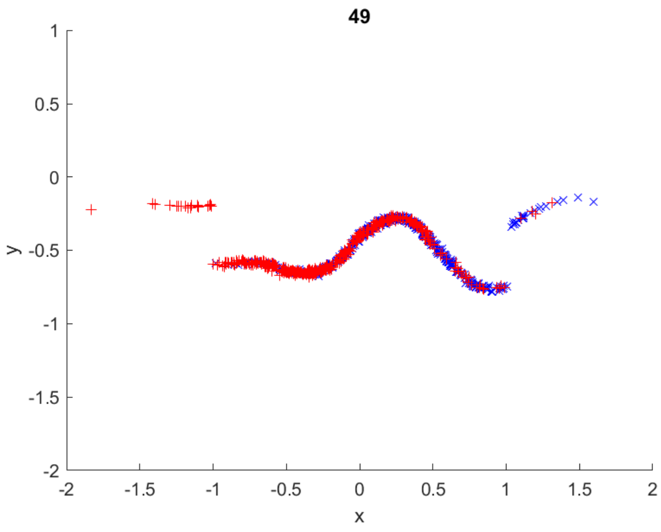


Figure 3: Result from weighted ICP.

3.2 ICP in 3D In this problem, you will apply your (weighted) ICP algorithm to real world LiDAR data from the KITTI dataset. You can use the following code to load the LiDAR data:

```
velodyne = load('velodyne.mat');
p = velodyne.data;
```

where `p` is a 11×1 cell array, and every element is a point cloud.

3.2.1 Modify `icp_2d.m` [5 points] Start from your 2D ICP code, write a function `icp_3d()` which takes two point clouds in 3D and calculates R and t needed to align them. Remember to remove all the visualization code in your function.

```
function [R, t] = icp_3d(p_t, p_s)
N = 500;
p_t = downsample(p_t, N);
p_s = downsample(p_s, N);
% Your code here ...
end
```

Use your `icp_3d()` to calculate the relative motion R_i and t_i needed to align the two consecutive clouds.

$$\forall i \in \{2, 3, \ldots, 11\} : \texttt{p\{i\}} \sim \texttt{R\{i\}p\{i - 1\} + t\{i\}}$$

You can do this for $R_2, R_3, \ldots, R_{11}$ and $t_2, t_3, \ldots, t_{11}$, and set $R_1 = I$, $t_1 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^\top$
Note: For this problem, you don't have to submit individual relative motions to Cody. The only thing you have to submit is your `icp_3d.m`.

3.2.2 Total transformation [5 points]

Write a function `get_total_transformation()` which takes the relative motions R_i and t_i to calculate the total transformations needed to align `p{1}` and `p{i}`.

```
function [R_total, t_total] = get_total_transformation(R, t)
% Your code here ...
end
```

where `R_total`, `t_total`, `R` and `t` are all 11×1 cell arrays. Further more,

$$\forall i \in \{1, 2, \ldots, 11\} : \texttt{p\{i\}} \sim \texttt{R\_total\{i\}p\{1\} + t\_total\{i\}}$$

Figure 4 shows `p{1}` and `p{11}` after alignment using ICP with 2000 points per cloud.

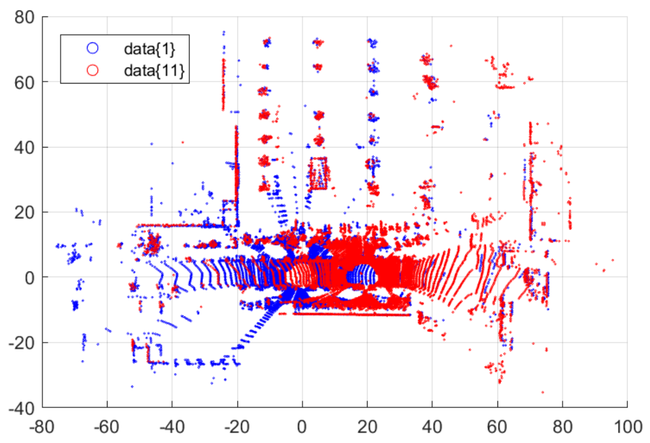


Figure 4: Aligned point clouds.

Hints:

1. `p{1}` is always assumed to be fixed in the world frame. When visualizing, all the transformations should only be apply to other clouds. In other words, to visualize the results we want to move all the other clouds to be aligned with `p{1}`.

2. In very beginning of your `icp_3d.m`, use the following function to downsample both p_s and p_t.

```
function p_ = downsample(p, N)
idx = int64(linspace(1, size(p, 2), N));
p_ = p(:, idx);
end
```

You can first try to take only 500 points from each cloud, and make the number larger if the result looks reasonable. Using more points can be more accurate, but also takes more time. When submitting to Cody, use 500 points per cloud.

3. Use the script `kitti_icp.m` to check your results. Using 2000 points per cloud, I got

$$\texttt{[R\_total\{11\}, t\_total\{11\}]} = \begin{bmatrix} 0.9996 & -0.0299 & 0.0003 & -19.0213 \\ 0.0299 & 0.9996 & -0.0004 & -0.4340 \\ -0.0003 & 0.0004 & 1.0000 & -0.3893 \end{bmatrix}$$

   This indicates that the car traveled mainly along the $x-$axis for about +19 meters with nearly no rotation.

4. Use `scatter3()` to plot your results in 3D. You can actually see cars, tree trunks, and buildings, etc.

Problem 4: Bag-of-Words (BoW) [20 points]

In this problem, we will use Bag-of-Words (BoW) to query an image, `query.png`, within a dataset, `kitti/*.png`. The 102 images in the dataset was taken sequentially from a continuous ride, and every image has a GPS location associated to it. We want to find which image in the dataset is the best match to the query image.

4.1 Feature extraction [5 points] Write a function `SURF()` to extract features from an image.

```
function feature = SURF(image)
% Your code here ...
end
```

The output `feature` is a $n \times 64$ matrix, and each row is a SURF descriptor. The number of descriptors n varies depending on the input image. You should use the built-in functions, `detectSURFFeatures()` and `extractFeatures()`.

4.2 $k-$means clustering [5 points] Write a function `get_codeword()`, which takes *all* the features from the dataset (not including the query image), and uses $k-$means to cluster them into n_c clusters. The centroid of each cluster is a codeword.

```
function codeword = get_codeword(feature_all, n_c)
rng(0);
% Your code here ...
end
```

The first input `feature_all` is a $N \times 64$ matrix contains all the SURF features from the dataset. The output `codeword` is a $n_c \times 64$ matrix, and each row is a codeword. The function `rng` is used to eliminate randomness on the output.

4.3 BoW representation [5 points] Build a $k-d$ tree using codewords. Write a function `get_hist()` which takes a $k-d$ tree, features from *one* image, n_c, and returns the BoW representation $h \in \mathbb{R}^{1 \times n_c}$ for that image.

```
function h = get_hist(kdtree, feature, n_c)
% Your code here ...
end
```

4.4 Query [5 points] Calculate the BoW representation of the query image h_q using the same $k-d$ tree, and find the nearest one in $\{h_i\}_{i=1}^{102}$ using χ^2-distance.

```
function d = chi_sq_dist(h1, h2)
d = sum((h1 - h2).^2 ./ (h1 + h2 + eps)) / 2;
end
```

The image you found should look very similar to the query image.

On Cody, write a function `query()` which loads BoW representations for the dataset and a $k-d$ tree from provided `kitti_bow.mat`, and returns the distances between the query image and each image in the dataset $d \in \mathbb{R}^{102}$, and the index of the nearest image.

```
function [d, idx] = query()
kitti = load('kitti_bow.mat');
hist_train = kitti.hist_train;
kdtree = kitti.kdtree;
n_c = kitti.n_c;

img_q = rgb2gray(imread('query.png'));
% Your code here ...
end
```

Useful functions: **`detectSURFFeatures`**, **`extractFeatures`**, **`kmeans`**, **`KDTreeSearcher`**, **`knnsearch`**