ROB 535 HW 3

Prof. Johnson-Roberson and Prof. Vasudevan

Due: 07 Nov 2019 2:00 PM EDT

Submission Details

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

Problem 1: Iterative Closest Point (ICP) [30 points]

Let p_s and p_t be the source and target point clouds, respectively.

$$p_s = \begin{bmatrix} x_{s1} & \dots \\ y_{s1} & \dots \\ z_{s1} & \dots \end{bmatrix} \in \mathbb{R}^{3\times n_s}, \; p_t = \begin{bmatrix} x_{t1} & \dots \\ y_{t1} & \dots \\ z_{t1} & \dots \end{bmatrix} \in \mathbb{R}^{3\times n_t}$$

where n_s and n_t are number of points in each point cloud. The goal of Iterative Closest Point (ICP) is to find a rigid body transformation $(R, t) \in \mathrm{SE}(3)$ such that the two point clouds are aligned.

$$p_t \sim R \cdot p_s + t$$

The transformation (R, t) can be used in graph SLAM as odometry measurement, or be used for 3D reconstruction.

1.1 Fit Rigid Body Transformations [10 points]

Let $p_1,\ p_2 \in \mathbb{R}^{m\times n}$ be two point clouds with n points in $m-$D. Assuming that the $i-$th point of p_1 corresponds to the $i-$th point of p_2. Write a function `rigid_fit` to calculate the optimal rigid body motion (R, t) to align p_1 and p_2, by solving the following optimization problem:

$$\min_{R\in\mathrm{SO}(m),\ t\in\mathbb{R}^m} \quad \sum_{i=1}^{N} w_i \ \|R \cdot p_{1i} + t - p_{2i}\|_2^2$$

where $w_i \geq 0$ is the weight of the $i-$th point, and p_{ki} is the $i-$th point (i.e. column) of $p_k,\ k \in \{1, 2\}$.

Notes:

- The output $(R, t) \in \mathrm{SE}(m)$ is a rigid body transformation in $m-$D.
- The analytical solution (https://igl.ethz.ch/projects/ARAP/svd_rot.pdf) can be found using Singular Value Decomposition (SVD).
- Use the following code to test your implementation:

```
[p1, p2, R, t] = gen_points(1000, 0.01);
disp([R, t]);   % ground truth
[R1, t1] = rigid_fit(p1, p2);
disp([R1, t1]); % should be pretty close to the ground truth
```

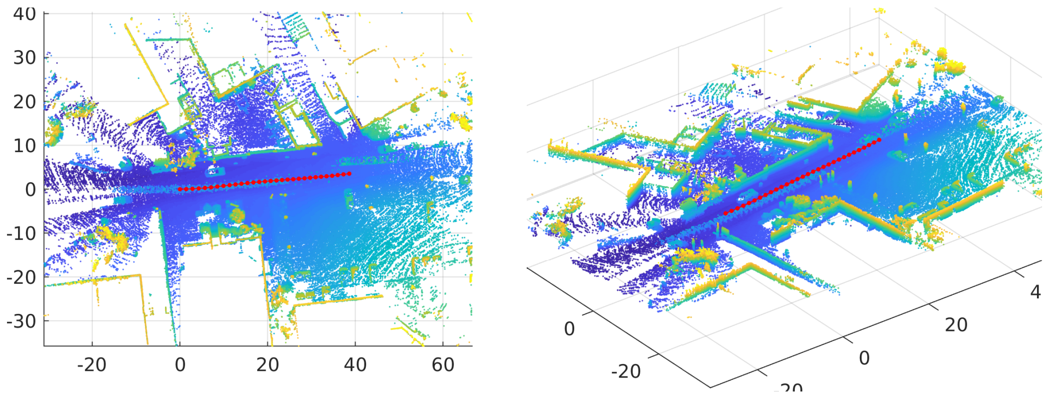


Figure 1: Top-down (left) and isometric (right) view of a reconstructed scene generated by `velodyne_straight.mat`. The trajectory (red) shows that the ego vehicle traveled 38.75 meters.

1.2 Naive ICP [5 points] Assuming that $w_i = 1\ \forall i \in \{1, \ldots, n_s\}$. Write a function `icp` to compute the transformation which aligns p_t and p_s.

Initialize the final transformation T_{final} with an initial value T_{init}. At each iteration, first update p_1 using the final transformation

$$p_1 = R_{final} \cdot p_s + t_{final}$$

and then find the nearest point in p_t for each point in p_1 using `knnsearch`. This gives you both correspondences and distances between corresponding points. Find p_2 and use the function from Problem 1.1 to compute an incremental transformation $(\Delta R, \Delta t)$. Update the final transformation with $(\Delta R, \Delta t)$, and iterate until the the incremental transformation is sufficiently small.

1.3 Weighted ICP [5 points] Let d_i be the Euclidean distance between p_{1i} and p_{2i}. Let σ_d be the standard deviation of values in $\{d_i\}_{i=1}^{n_s}$. Modify your code so that when `use_naive` is `false`, weights are assigned by

$$w_i = \exp\left(-\frac{d_i^2}{\sigma_d^2}\right)$$

By doing so, the weighted version of ICP becomes more robust to outliers, since bad correspondences gain less weight.

1.4 Odometry and 3D Reconstruction [10 points] Write a function `kitti_icp` to estimate a trajectory (i.e. sequence of transformations) in the world frame from a sequence of point clouds in the body frame. Use the T_{init} in the template and the weighted version of ICP to calculate the relative motion between two consecutive point clouds, and obtain the trajectory. Use the trajectory to transform each point cloud to the world frame and plot it in the same map, as shown in Figure 1.

Problem 2: Compute Disparity Map [10 points]

Given a pair of rectified stereo images, the problem of stereo matching is defined as follows: for a point at pixel coordinate (x_l, y_l) in the left stereo image, find the pixel coordinate (x_r, y_r) of the corresponding pixel in the same row in the right image. The difference $d = x_r - x_l$ is called the disparity at that pixel, and if we perform this correspondence matching for all pixels in the left image, we will have computed a disparity map for the stereo pair of images.

For further clarification: suppose a particular 3D point in the physical world is located at the position (x_l, y_l) in the left image, and the same point is located on $(x_l + d, y_r)$ in the right image, then the location (x_l, y_l) on the disparity map holds the value d.

As discussed in class, image rectification makes two stereo images 'parallel', i.e, the rows in each image of a stereo image pair are aligned, which means the y coordinates of corresponding points in each image are the same. Hence, we only need to perform matching along the rows of each image as opposed to matching along the rows and columns of each image. Thus, we can define disparity at each point in the image plane as: $d = x_l - x_r$.

You will need to write a function

`[D] = genDisparityMap(I1, I2, min_d, max_d, w_radius)`

that takes as input the left and right images of a stereo pair, the minimum disparity value, the maximum disparity value, and the window radius (window width). The output `D` is the disparity map.

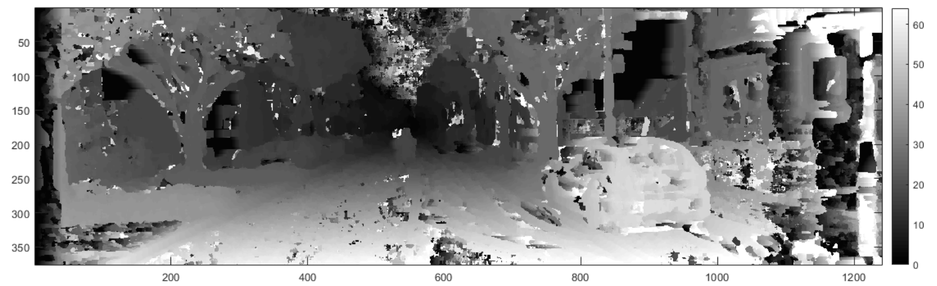
For your implementation, adhere to the following specifications:

- Your stereo pair image input will be from KITTI.
- For each pixel in the left image, extract the 9 by 9 pixel window around it (`w_radius=4.0` input parameter) and use it as a template to search along the same row in the right image for the region of pixels that best matches it.
- Along the right image's row, search over all disparity values d in a disparity region of 0 to 64 pixels (the `min_d` and `max_d` input parameters, respectively) around the pixel's location in the right image.
- Use the sum of absolute differences (SAD) as the similarity metric to compare the left image template to the region in the right image.

PLEASE NOTE THE FOLLOWING:

1. Your disparity map for the two KITTI images should resemble as in Figure 2.
2. You are **NOT** allowed to use the Matlab function `disparity()`. It implements a more complicated variation of the block matching algorithm, and if you use it your disparity map will not match the autograder solution.
3. This page provides a nice conceptual explanation of pseudo code for this problem.

Figure 2: Disparity Map for the KITTI stereo pair at time t



Problem 3: Trajectory Synthesis [25 points]

In this problem you will use nonlinear optimization to generate a trajectory that satisfies a set of constraints. For this problem you will use the kinematic bicycle model

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} u\cos(\psi) - \frac{b}{L}u\tan(\delta)\sin(\psi) \\ u\sin(\psi) + \frac{b}{L}u\tan(\delta)\cos(\psi) \\ \frac{u}{L}\tan(\delta) \end{bmatrix} \tag{1}$$

where $b = 1.5$ and $L = 3$. Your initial condition will be $x_0 = y_0 = \psi_0 = 0$, and your goal is to reach $x_g = 7,\ y_g = \psi_g = 0$. The center of mass, located at (x, y), cannot leave the road i.e. $-3 < y < 3$ and $-1 < x < 8$. The center of mass of the vehicle must also avoid a square obstacle with side length 1 that is centered at $(3.5, -0.5)$. Input constraints of $0 < u < 1$ and $-0.5rad < \delta < 0.5rad$ must also be satisfied. The output of the trajectory synthesis will be a vector of discretized states with time step 0.05 seconds for 120 steps

$$z = \begin{bmatrix} x_0 & y_0 & \psi_0 \ldots x_{120} & y_{120} & \psi_{120} & u_0 & \delta_0 \ldots u_{119} & \delta_{119} \end{bmatrix} \tag{2}$$

Note that z is a 1-by-603 vector.

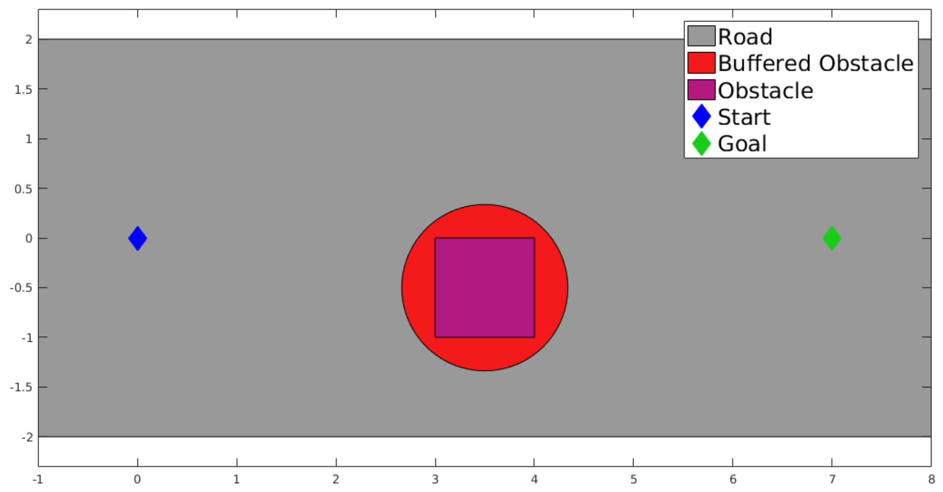


Figure 3: An illustration of the autonomous vehicle control design task

3.1 Upper and Lower Bounds [5 points] First you must create vectors for the upper and lower bounds `lb`$< z <$`ub` such that

$$-1 < x_i < 8 \tag{3}$$
$$-3 < y_i < 3 \tag{4}$$
$$-\frac{\pi}{2} < \psi_i < \frac{\pi}{2} \tag{5}$$
$$0 < u_j < 1 \tag{6}$$
$$-0.5 < \delta_j < 0.5 \tag{7}$$

for $i = 0, ..., 120$ and $j = 0, ..., 119$. Both `lb` and `ub` should be 603-by-1.

3.2 Nonlinear Constraint Function[12 points] Next you will write the nonlinear constraint function `[g,h,dg,dh] = nonlcon(z)` where g are the inequality constraints with gradient `dg` and `h` are the equality constraints with gradient `dh`. For the inequality constraints let `g` $= [g_0 \ \dots \ g_{120}]^T$ then

$$g_i = (0.7)^2 - (x_i - 3.5)^2 - (y_i - (-0.5))^2 \tag{8}$$

for $i = 0, ..., 120$. The resulting constraint boundary resulting from $g_i \leq 0$ provides a circular buffer of radius 0.7 around the obstacle as shown in Figure 3. The inequality constraints gradient `dg` is then the transpose of the Jacobian

$$\texttt{dg} = \left(\frac{\partial g}{\partial z}\bigg|_z \right)^T. \tag{9}$$

For the equality constraints let $\texttt{h} = [h_0 \ \dots \ h_{363}]$ then

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \\ \psi_0 \end{bmatrix} \tag{10}$$

$$\begin{bmatrix} h_{3i+1} \\ h_{3i+2} \\ h_{3i+3} \end{bmatrix} = \begin{bmatrix} x_i \\ y_i \\ \psi_i \end{bmatrix} - \begin{bmatrix} x_{i-1} \\ y_{i-1} \\ \psi_{i-1} \end{bmatrix} - 0.05 \begin{bmatrix} \dot{x}_{i-1} \\ \dot{y}_{i-1} \\ \dot{\psi}_{i-1} \end{bmatrix} \qquad i = 1, ..., 120. \tag{11}$$

Here (10) enforces the initial condition and (11) enforces the the dynamics based on Euler integration. The equality constraint gradient `dh` is then the transpose of the Jacobian

$$\texttt{dh} = \left(\frac{\partial h}{\partial z}\bigg|_z \right)^T. \tag{12}$$

3.3 Cost Function [8 points]

Add a cost function `[J,dJ] = costfun(z)` where the cost `J` is

$$\texttt{J} = \left(\sum_{i=0}^{120} (x_i - 7)^2 + y_i^2 + \psi_i^2 \right) + \left(\sum_{j=0}^{119} u_j^2 + \delta_j^2 \right). \tag{13}$$

This applies a quadratic penalty based on the distance to the goal, and to the inputs. The cost function gradient `dJ` is the transpose of the Jacobian

$$\texttt{dJ} = \left(\frac{\partial \texttt{J}}{\partial z}\bigg|_z \right)^T. \tag{14}$$

3.4 Solve Using fmincon [0 points]

(This is already coded for you) Run fmincon using the constraints you have created in previous steps. Use the inputs returned from fmincon and ode45 to find the true trajectory. Change the initial condition to $x_0 = y_0 = 0, \psi_0 = -0.01$ and run ode45 again to find the true trajectory for this slightly different initial condition. Plot all the trajectory returned from fmincon and the two trajectories you found using ode45.

(Note: The trajectory from ode45 is slightly different from the fmincon result. Furthermore, for slight deviations in the initial condition the resulting trajectory is quite different. This motivates the need for MPC in addition to trajectory synthesis.)

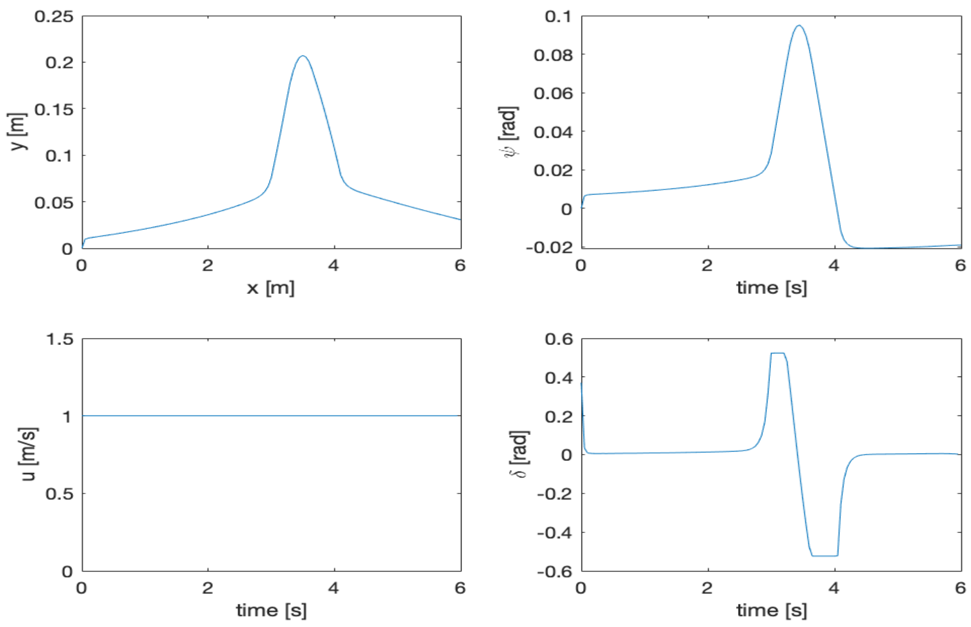
Problem 4: MPC Control [50 points]

In this problem, you will use quadratic programming based MPC to design a controller to follow a given trajectory. You may refer to the code posted on Canvas > Files > code > MPC to get a better understanding if needed.

You will use the following dynamics (kinematic bicycle model) for the MPC controller and to simulate the vehicle:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} u\cos(\psi) - \frac{b}{L}u\tan(\delta)\sin(\psi) \\ u\sin(\psi) + \frac{b}{L}u\tan(\delta)\cos(\psi) \\ \frac{u}{L}\tan(\delta) \end{bmatrix} \tag{15}$$

where (x, y, ψ) are the states (pose) of the vehicle, and (u, δ) are the inputs (longitudinal velocity and wheel angle). The MPC controller will track the reference trajectory (plotted below), that Matt the Eager Junior has already computed



The reference trajectory is saved as `part1_traj_05_timestep.mat` and is available to download on canvas. The file contains variables `U` and `Y` that contain the reference input and states at a time discretization of 0.05 s. The learner template in Matlab grader provides the code to load and interpolate the reference trajectory to the correct time discretization for the MPC controller. Unfortunately, we did not do a good job at state-estimation, so our initial condition will be different from the reference trajectory.

4.1 Euler Discretization [10 points] You decide to represent the linearized dynamics about the trajectory using Euler discretization with a time step $\Delta t = 0.01$. Compute the discrete-time LTV system representation of the A and B matrices and save them as function handles named `A` and `B`. The function handles should take in an index $i \in [1, 2, ...601]$ and return A_i and B_i matrices for that timestep along the reference trajectory. Note that the discrete time matrices are similar to, but not the same as the Jacobians of a continuous-time linearized system. The discrete time matrices satisfy the equation:

$$\begin{bmatrix} \tilde{x}_{i+1} \\ \tilde{y}_{i+1} \\ \tilde{\psi}_{i+1} \end{bmatrix} = A_i \begin{bmatrix} \tilde{x}_i \\ \tilde{y}_i \\ \tilde{\psi}_i \end{bmatrix} + B_i \begin{bmatrix} \tilde{u}_i \\ \tilde{\delta}_i \end{bmatrix} \tag{16}$$

Note, the $\tilde{}$ symbol indicates the states are linearized about the reference trajectory.

4.2 Decision Variables [5 points] Next, you decide to use a planning horizon of 10 time steps, i.e. 0.1 seconds. You will use MPC at each discrete time step i to solve for 10 inputs. At each time step, you decide to let the first portion of the decision variable, z, represent the discrete states of the system from time step i to time step $i + 10$ in order. You let the second portion of the decision variable space represent the input of the system from time step i to time step $i+9$ in order. Compute the total size of your decision variable space and save them in a double named `Ndec`.

4.3 Equality Constraints [15 points] Next, you decide to solve the MPC using the MATLAB function '`quadprog`,' which represents the problem using an equality and inequality constraint (hint use '`help quadprog`' in MATLAB). You decide to represent the Euler integration using an equality constraint $A_{eq}z = b_{eq}$. The first 3 rows of A_{eq} represents satisfying the initial condition (for the mpc controller) at time step i in the linearized dynamics about the reference trajectory. Each subsequent row of A_{eq} represents the Euler integration steps in order from time step i to time step $i + 10$.

Write a function to generate the equality constraint matrices, A_{eq} and b_{eq}, when given the time-step and initial condition.

For the autograder, The order of your decision variable, z, should be

$$z = [\tilde{x}_i,\ \tilde{y}_i,\ \tilde{\psi}_i, \tilde{x}_{i+1},\ \tilde{y}_{i+1},\ \tilde{\psi}_{i+1}, \cdots, \tilde{x}_{i+10},\ \tilde{y}_{i+10},\ \tilde{\psi}_{i+10},\ \tilde{u}_i, \tilde{\delta}_i,\ \tilde{u}_{i+1}\ \tilde{\delta}_{i+1}, \cdots,\ \tilde{u}_{i+9}\ \tilde{\delta}_{i+9}]^T \tag{17}$$

and each equality constraint for the dynamics should be written as:

$$A_i \begin{bmatrix} \tilde{x}_i \\ \tilde{y}_i \\ \tilde{\psi}_i \end{bmatrix} + B_i \begin{bmatrix} \tilde{u}_i \\ \tilde{\delta}_i \end{bmatrix} - \begin{bmatrix} \tilde{x}_{i+1} \\ \tilde{y}_{i+1} \\ \tilde{\psi}_{i+1} \end{bmatrix} = 0 \tag{18}$$

To test your function compute the matrices A_{eq} and b_{eq} starting at time $i = 1$ ($t = 0$) with an initial condition: $[\tilde{x};\ \tilde{y};\ \tilde{\psi}] = [0.25;\ -0.25;\ -0.1]$.
Store your answer in variables named `Aeq_test1` and `beq_test1`.

4.4 Boundary Constraints [10 points] Next, you want to constrain the input at each MPC time step to the bounds $u(t) \in [0, 1]$ and $\delta(t) \in [-0.5,\ 0.5]$. To ensure that this is satisfied, make use of the available upper and lower bound inputs in function `quadprog`. Write a function that returns the upper and lower bound vectors Lb and Ub starting from a given a time-step along the reference trajectory. To test your function compute the matrices Lb and Ub starting at time $i = 1$ ($t = 0$).
Store your answer in variables named `Lb_test1` and `Ub_test1`.

4.5 Generating a Solution [10 points] Next you decide to run the quadratic programming-based MPC with a quadratic penalty of $Q = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0.5 \end{bmatrix}$ on the state of the linearized dynamics, and $R = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.01 \end{bmatrix}$ on the inputs of the linearized dynamics. Use and initial condition of: $[\tilde{x};\ \tilde{y};\ \tilde{\psi}] = [0.25;\ -0.25;\ -0.1]$. For each $i > 1$, the state $Y(:, i)$ of the system should be generated by using **ode45** and (15) for the time discretization (0.01 s), from the previous state $Y(:, i-1)$ with the input computed using the quadratic programming-based MPC at the previous time step.
Run the quadratic programming-based MPC loop and plot the reference and actual trajectories.
Calculate the maximum distance error between the reference and actual trajectories when the x position of the actual state is crossing the obstacle (3≤x≤ 4):

$$\texttt{max\_dist\_error} = \max_{3 \le x \le 4} \sqrt{(x_i - x_{i,\mathrm{ref}})^2 + (y_i - y_{i,\mathrm{ref}})^2}$$

Note: just check the points in the trajectory you generated (with the time discretization of 0.01 s) you do not need to interpolate to check any more points.