Final Exam: Event B

Branch Vincent

December 19, 2016

1 Summary

This system is designed to catch falling objects using a robot with a scoop as its end effector. This system includes three main components: perception, planning, and control. These components, and associated subcomponents, are illustrated below.

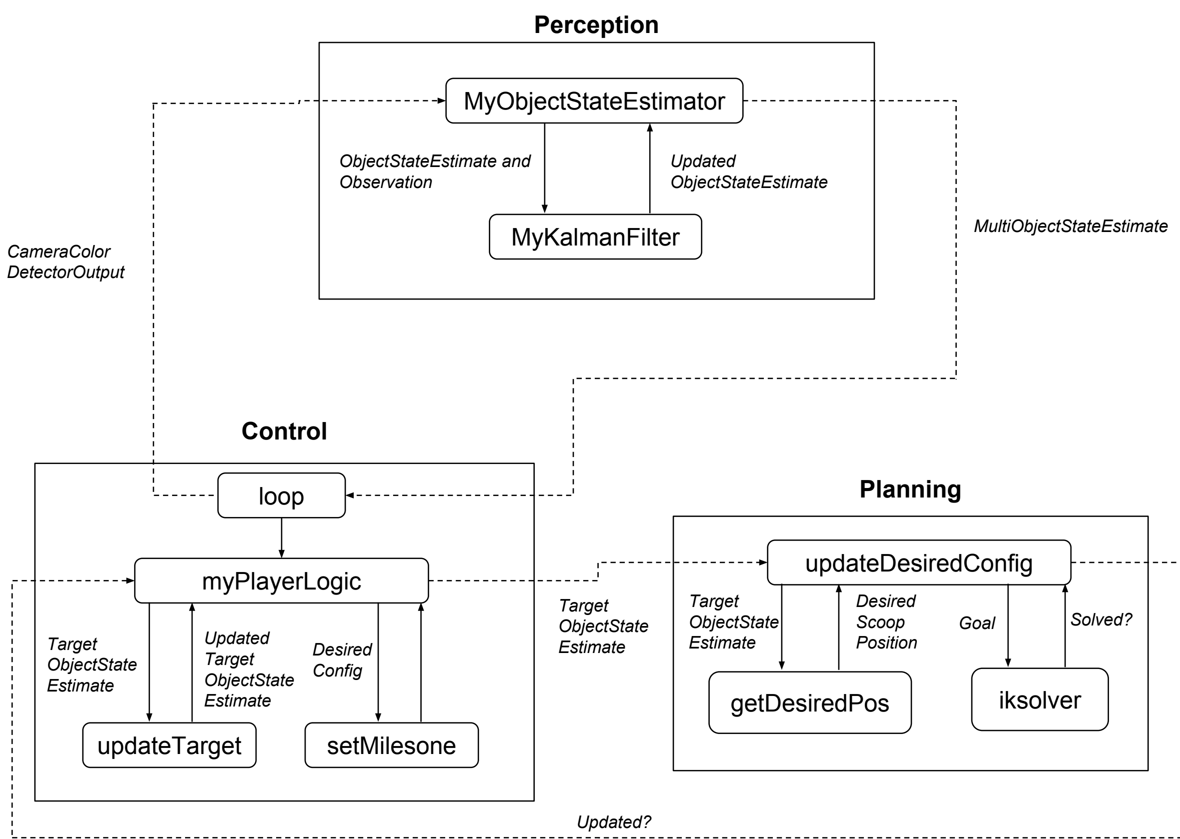

Figure 1: System Diagram

1.1 Perception

The perception component tracks the mean and covariance of object estimates and processes noisy camera sensor readings via a Kalman filter. More specifically, `MyObjectStateEstimator` monitors the current object estimates via a `MultiObjectStateEstimate` instance and filters noisy readings via a `MyKalmanFilter` instance. When updated with a new `CameraColorDetectorOutput` instance, the state estimator converts each `CameraBlob` from pixel coordinates to world coordinates. Then, the corresponding `ObjectStateEstimate` is either initialized with an initial belief state or updated with the new observation via the Kalman filter. Finally, any unobserved objects in the `MultiObjectStateEstimate` are updated with a prediction.

1.2 Planning

The planning component, housed within `updateDesiredConfig`, calculates the robot configuration at which to catch the current target object. More specifically, `getDesiredPos()` determines the world position at which the current target `ObjectStateEstimate` instance will intersect the desired catching plane, specified by the world z-coordinate $z_{catching}$. This calculation is performed using the object's mean position and velocity, which can be a considerable source of error. It also determines the target object's world position 0.5 seconds before this intersection to calculate the desired scoop orientation. This information is passed back to `updateDesiredConfig()`. If a solution was found and the robot's current configuration cannot catch the target object, then these two points, along with the local coordinates of the scoop's midpoint and axis corresponding with the desired orientation, are used to create an `ik.objective`. This objective is then passed to Klamp't's `ik.solve_nearby()` to calculate a desired configuration. This module was chosen because it does not allow the solution to deviate too far from the initial configuration, meaning that the geodesic interpolation problem from the low-level controller (see below) is minimized.

1.3 Control

The control component tracks the robot's current state and target object. The two states in the machine are `waiting`, meaning that the robot is completely idle, or `catching`, meaning that it is attempting to catch the target object. More specifically, `MyController` updates the target object each time step via `updateTarget()`. If the target changes, then the controller changes the state to `waiting`. Then, if the robot is `waiting`, the controller calls the planner to plane the movement to the target object's predicted location. The planner returns if it updated the desired configuration and then calls `SimRobotController.setMilestone()` for low-level control. This function was chosen because it is time-optimal with respect to the velocity and acceleration bounds. However, it does not perform geodesic interpolations, which can cause inefficiencies.

2 Components

The major components of the system are now described in more detail, along with each associated subcomponent.

2.1 Perception

The perception component includes a Kalman filter, implemented via the `MyKalmanFilter` class, and an object state estimator, implemented via the `MyObjectStateEstimator` class.

2.1.1 Kalman Filter

The `MyKalmanFilter` class includes the following three functions.

Function: `createEstimate(name,z)`

(a) *Purpose*: To create a new `ObjectStateEstimate`, first initialized with the initial belief state, and then updated with the observation.

(b) *Input*: Object name, `name`, and observation vector, `z`. The object name is a list representing the object's RGBA color value. The observation vector is a 6D array in world coordiantes representing position and velocity, or $\begin{bmatrix} p_{x_t} & p_{y_t} & p_{z_t} & \dot{p}_{x_t} & \dot{p}_{y_t} & \dot{p}_{z_t} \end{bmatrix}$.

(c) *Output*: New `ObjectStateEstimate`

(d) *Method of Invocation*: Invoked on request by `MyObjectStateEstimator`, when encountering a new object that has not yet been estimated.

(e) *Implementation*: Creates a new `ObjectStateEstimate` with the Kalman filter's initial beliefs for mean, μ_0, and covariance, Σ_0. Calls and returns `updateWithObservation()`.

(f) *Failure Cases*: Incorrect input data types.

Function: `updateWithObservation(obj,z)`

(a) *Purpose*: To update the object's estimated state with the new observation.

(b) *Input*: Object, `obj`, and observation vector, `z`. The object is an `ObjectStateEstimate`. The observation vector is a 6D array in world coordiantes representing position and velocity, or $\begin{bmatrix} p_{x_t} & p_{y_t} & p_{z_t} & \dot{p}_{x_t} & \dot{p}_{y_t} & \dot{p}_{z_t} \end{bmatrix}$.

(c) *Output*: Updated `ObjectStateEstimate`

(d) *Method of Invocation*: Invoked on request by `MyObjectStateEstimator` when encountering a previously estimated object.

(e) *Implementation*: Calls `common.kalman_filter_update()`, storing the returned mean and covariance.

(f) *Failure Cases*: Incorrect input data types or incorrect vector size.

Function: `updateWithPrediction(obj)`

(a) *Purpose*: To update the object's estimated state with the predicted state, in the absence of an observation.

(b) *Input*: Object, `obj`. The object is an `ObjectStateEstimate`.

(c) *Output*: Updated `ObjectStateEstimate`.

(d) *Method of Invocation*: Invoked on request by `MyObjectStateEstimator`, when encountering a previously estimated object that was not observed at the current time step.

(e) *Implementation*: Calls `common.kalman_filter_predict()`, storing the returned mean and covariance.

(f) *Failure Cases*: Incorrect input data type.

2.1.2 Object State Estimator

The `MyObjectStateEstimator` class includes the following two functions.

Function: `update(observation)`

(a) *Purpose*: To update all object state estimates with the camera's sensor readings or, if absent from the camera, with the predicted state.

(b) *Input*: The `CameraColorDetectorOutput` sensor reading, `observation`, in pixel coordinates.

(c) *Output*: Updated `MultiObjectStateEstimate`.

(d) *Method of Invocation*: Invoked at a constant rate each control loop by `myPlayerLogic()`.

(e) *Implementation*: Each blob is first converted to world coordinates, via `blobToWorld()`. Then, the object state estimate is either created via `createEstimate()` or updated via `updateWithObservation()`, as needed. Next, each unobserved but previously estimated state is updated via `updateWithPrediction()`.

(f) *Failure Cases*: None

Function: `blobToWorld(blob)`

(a) *Purpose*: To convert a `CameraBlob` image to world coordinates.

(b) *Input*: A `CameraBlob`, `blob`.

(c) *Output*: The `blob`'s central position as a 3D vector, in world coordinates.

(d) *Method of Invocation*: Invoked on request by `update()`, for any detected blob.

(e) *Implementation*: Converts pixel coordinates to camera local coordinates via trigonometric relationships and known camera parameters. Then converts camera local coordinates to world coordinates via `Tsensor`.

(f) *Failure Cases*: Due to the camera's noise, the calculated world position can be quite inaccurate.

2.2 Planning and Control

The planning and control components include the `MyController`.

2.2.1 MyController

Function: `updateTarget()`

(a) *Purpose*: To update the target object.

(b) *Input*: None

(c) *Output*: The updated `ObjectStateEstimate` target object.

(d) *Method of Invocation*: Invoked at a constant rate each control loop by `myPlayerLogic()`.

(e) *Implementation*: The current target's z position, p_z is compared to the catching plane, $z_{catching}$. If $p_z < z_{catching}$, the target is advanced to the next object in `MultiObjectStateEstimate`, if applicable, and the robot's state is set to `waiting`.

(f) *Failure Cases*: Since the estimated p_z is subject to noise, this may incorrectly discard the target object or fail to do so.

Function: `getDesiredPos(obj,zdes)`

(a) *Purpose*: To calculate the anticipated position at which the target object will intersect the catching plane, z_{des}.

(b) *Input*: The `ObjectStateEstimate obj` and the desired catching plane, `zdes`. The catching plane is a float in world coordinates representing the position on the z-axis at which the scope should catch the ball.

(c) *Output*: A list `(wp1,wp2)`, where `wp1` is the desired position of the scoop's midpoint and `wp2` is the desired axis of the scoop's z-axis. Both `wp1` and `wp2` are 3D vectors in world coordinates.

(d) *Method of Invocation*: Invoked at a constant rate each control loop by `updateDesiredConfiguration()`.

(e) *Implementation*: Uses the object's estimated position and velocity to calculate the time at which the object will intersect `zdes`. The quadratic is solved via `numpy.roots()`.

(f) *Failure Cases*: May return a null list if the quadratic could not be solved.

Function: `updateDesiredConfig(obj)`

(a) *Purpose*: To update the desired robot configuration.

(b) *Input*: The `ObjectStateEstimate obj`.

(c) *Output*: A Boolean representing if the configuration was updated.

(d) *Method of Invocation*: Invoked on request by `myPlayerLogic()`.

(e) *Implementation*: Finds the desired scoop position and axis via `getDesiredPosition()`. If a solution was found and the robot's current configuration cannot catch the target object, then it sets up an `ik.objective` to match the world coordinates to the corresponding local scoop coordinates. This objective is then passed and solved the to `ik.solve_nearby` and updates the resulting configuration.

(f) *Failure Cases*: This can fail if the desired position is outside the robot's workspace. Furthermore, `ik.solve_nearby` may fail to converge since it is a numerical method.

3 Planning and Control Strategy

The time at which the object will intersect $p_{z_{des}}$ is

$$p_z + v_z t + \frac{1}{2} a_z t^2 = p_{z_{des}} \tag{1}$$

The solution, t, and $t - 0.5$, is used to determine the object's position and orientation such that the robot can catch the object with its scoop perpendicular to the trajectory. If t exists and the robot cannot catch the object in its current configuration, then the corresponding positions are passed to `ik.solve_nearby`. A planning failure will return false. The `SimRobotController.setMilestone()` function is used for low-level control. However, it does not perform geodesic interpolations.

4 Perception Strategy

4.1 Converting Camera Readings

The first step of perception is converting the camera sensor readings into world coordinates. This is achieved via `MyObjectStateEstimator.blobToWorld()`. This function receives a `CameraBlob` instance `blob`, providing the image's midpoint, (x_p, y_p), width, w_p, and height, h_p, all in pixel coordinates. We know the camera's horizontal field of view, α, and the image frame's width, w_{max}, and height, h_{max}. Given the object's actual width w and height h, we can convert pixel coordinates to local camera coordinates using the following equations derived from similar triangles:

$$\frac{w}{w_p} = \frac{x_l}{x_p - \frac{1}{2} w_{max}} \tag{2}$$

$$\frac{h}{h_p} = \frac{y_l}{y_p - \frac{1}{2} h_{max}} \tag{3}$$

$$\frac{w}{w_p} = \frac{z_l \tan\left(\frac{\alpha}{2}\right)}{\frac{1}{2} w_{max}} \tag{4}$$

$$\tag{5}$$

Since the object in this case is a ball, the width and height are equal. Let $s \equiv w = h$ and $s_p \equiv w_p = h_p$. Thus, we can solve for the local position.

$$x_l = \frac{s}{s_p}\left(x_p - \frac{1}{2}w_{max}\right) \tag{6}$$

$$y_l = \frac{s}{s_p}\left(y_p - \frac{1}{2}h_{max}\right) \tag{7}$$

$$z_l = \frac{s}{2s_p}\cot\left(\frac{\alpha}{2}\right)w_{max} \tag{8}$$

$$\tag{9}$$

The last step is converting from local to world coordinates, which is achieved via applying the transform `MyObjectStateEstimator.Tsensor`.

4.2 Filtering Camera Readings

The camera sensor provides noisy readings, which can be challenging to filter. However, a priori knowledge as well as a filtering technique helps mitigate this error.

4.2.1 Pre-Processed Filtering

The camera sensor has a pixel error of $\epsilon_p \sim U(-0.5, 0.5)$. Thus, the corrupted values are:

$$\widetilde{x}_p \in [x_p - 0.5, x_p + 0.5) \tag{10}$$

$$\widetilde{y}_p \in [y_p - 0.5, y_p + 0.5) \tag{11}$$

$$\widetilde{w}_p \in (w_p - 1, w_p + 1) \tag{12}$$

$$\widetilde{h}_p \in (h_p - 1, h_p + 1) \tag{13}$$

Note that this ignores the error introduced when the image begins to move outside the frame. But we know that this occurs when $|\widetilde{w}_p - \widetilde{h}_p| > 2$. Combining this with the fact that $w_p = s_p$, we can pre-process s_p as follows:

$$\widetilde{s}_p = \begin{cases} \frac{1}{2}(\widetilde{w}_p + \widetilde{h}_p) & |\widetilde{w}_p - \widetilde{h}_p| < 2 \\ max(\widetilde{w}_p, \widetilde{h}_p) & |\widetilde{w}_p - \widetilde{h}_p| \geq 2 \end{cases} \tag{14}$$

$$\tag{15}$$

Then

$$\widetilde{s}_p \in (s_p - 1, s_p + 1) \tag{16}$$

$$\widetilde{x}_l = \frac{s}{\widetilde{s}_p}\left(\widetilde{x}_p - \frac{1}{2}w_{max}\right) \tag{17}$$

$$\widetilde{y}_l = \frac{s}{\widetilde{s}_p}\left(\widetilde{y}_p - \frac{1}{2}h_{max}\right) \tag{18}$$

$$\widetilde{z}_l = \frac{s}{2\widetilde{s}_p}\cot\left(\frac{\alpha}{2}\right)w_{max} \tag{19}$$

$$\tag{20}$$

4.2.2 Post-Processed Filtering

Post-process filtering is achieved via a Kalman filter. The state estimate of each object, $x = \begin{bmatrix} p & \dot{p} \end{bmatrix}^T$, includes both 3D position, p, and 3D velocity, $\dot{p}$ or v. Note that for simplicity, these two vectors are not expanded when describing the Kalman matrices below. For the transition model, we have

$$x_{t+1} = \begin{bmatrix} p_t + v_t\Delta t + \frac{a_t}{2}\Delta t^2 \\ v_t + a_t\Delta t \end{bmatrix} + \epsilon_x \tag{21}$$

$$= \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} x_t + \begin{bmatrix} \frac{\Delta t^2}{2} \\ \Delta t \end{bmatrix} a_t + \epsilon_x, \qquad \epsilon_x \sim \mathcal{N}(0, \Sigma_x) \tag{22}$$

$$\Sigma_x = E[\epsilon_x \epsilon_x^T] \tag{23}$$

$$\tag{24}$$

For the observation model, we have

$$z_t = p_t \tag{25}$$

$$= \begin{bmatrix} 1 & 0 \end{bmatrix} x_t + \epsilon_z, \qquad \epsilon_z \sim \mathcal{N}(0, \Sigma_z) \tag{26}$$

$$\Sigma_z = \sigma_p^2 \tag{27}$$

Each state estimation is initialized with the belief state

$$x_{bel} \sim \mathcal{N}(\mu_0, \Sigma_0) \tag{28}$$

$$\mu_0 = \begin{bmatrix} p_0 \\ v_0 \end{bmatrix}, \Sigma_0 = \begin{bmatrix} \sigma_{p_0}^2 & 0 \\ 0 & \sigma_{v_0}^2 \end{bmatrix} \tag{29}$$

The covariance matrices Σ_x and Σ_z were tuned empirically until convergence was achieved in an acceptable time. The belief initialization was similarly tuned, using empirical averages of x_0 via the `omniscient_sensor`.

5 Reflection

The system struggled with the noisy sensor. This can be improved by continuing to tune the Kalman filter's covariance matrices. The system also can spend too much time planning, if the estimated trajectory moves around a lot. In a real world scenario, this system would face additional challenges such as external forces changing the trajectory of the ball and difficulties in combining the hardware and software. For example, special tuning may be needed for the software's commanded configuration to match the hardware's actual configuration.