EE106A: Lab 8 - Zumy + Kalman Filter

Victor Shia, Jaime Fisac

October 18, 2015

Goals

By the end of this lab you should be able to:

- Track the Zumy using a Kalman filter
- Have ROS nodes communicate with each other using services
- ...

Relevant Tutorials and Documentation:

- ROS srv 1: `http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv`
- ROS srv 2: `http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28python%29`
- Kalman Filter: `http://wiki.ros.org/ar_track_alvar`

Contents

Introduction

In this lab, we will work on using different on-board and off-board sensors to track the position and orientation of a Zumy, using an extremely powerful mathematical tool for state estimation called a Kalman filter. You will learn to use ROS services to request and send information between the sensors and the state estimator and will also learn about the strengths and weaknesses of inertial sensors (accelerometers and gyroscopes) as compared to the vision-based sensing that we have used so far: more importantly, you will learn to use *sensor fusion* to combine the best of both worlds and get more accurate and reliable tracking than you could achieve by either method.

1 Learning about Services & Setting up the IMU service

The Zumies are equipped with an Inertial Measurement Unit (IMU) consisting of three accelerometers and three gyroscopes, measuring linear accelerations and angular velocities along the X, Y and Z axes of the body frame. When

running its basic ROS functionality, the Zumy publishes the raw readings from these sensors on the corresponding '/zumyX/imu' topic.

The Kalman filter (KF) uses the information about linear accelerations and angular velocities as the input for the stage known as the *time update*. The KF node that we will be running will need to get the most up to date measurement at regular intervals, but we don't want to be bothering it every time there is a new message published on the topic. Instead, we will have an 'IMU listener' node that will continually listen on the IMU topic and keep track of the latest IMU information available, so that the KF node can request it whenever it needs it.

As we did in the previous lab, start by launching the basic `odroid_machine` ROS package on the Zumy, which launches all the necessary ROS nodes on board the Zumy.

```
roslaunch odroid_machine remote_zumy.launch mname:=zumyX
```

Try running `rostopic list` and identifying the IMU topic. You can echo the topic to see the form of the message that is being published. This is also not very efficient because the message type allows for a lot of extra information fields that are not actually being used. Now, in the `resources` folder for this lab, you will find the file `imu_srv.py` with some skeleton code that you can modify to extract the information from these messages and have an up-to-date `last_imu` variable.

The code `imu_srv.py` creates the **imu_listener** node which subscribes to the `/imu` topic to update the last accelerometer reading and uses services to provide other nodes data. So far, you have mostly used messages to communicate between nodes. Services are another type of message allowed on topics which allow nodes to send a request and receive a specific response, much like 2-way radio communication (messages are just 1 way communication). Note that both requests and responses can be used to send data. To find the list of available services, type in `rossrv list`. You can explore the `rossrv` command by just typing in `rossrv` in the command prompt. For more information on services, look at the ROS tutorials.

To create a service type, look at the `service_example`. In the srv folder, you see the following:

```
int64 a
int64 b
---
int64 sum
```

This service contains the request and response message all in one file. The request contains the fields above the `---`, while the response contains the fields below the `---`. Look at `src/add_two_ints_server.py` for the server code. The service is setup with the command

```
rospy.Service( << topic name >>, << srv name >>, << callback function >>)
```

So the service setup with the command:

```
rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
```

receives a request over the topic `add_two_ints` of service type `AddTwoInts`. When the request is received, the function `handles_add_two_ints` is called and responds to the request.

Now look at `src/add_two_ints_client.py` for the client code. Look at the function `add_two_ints_client`. The function `rospy.wait_for_service('add_two_ints')` forces the code to wait until the service exists. This is useful in the case when the client is started before the server. Once the service is available, the program proceeds. A request message is generated with the command:

```
rospy.ServiceProxy(<< topic name >>, <<  srv name >>)
```

So the request setup with the command:

```
add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
```

creates service handle (on the left side) called `add_two_ints` over the topic `add_two_ints` of srv type `AddTwoInts`. The request is sent in the next line `add_two_ints(x,y)` and the response is set to `resp1`.

To add the service to the package, you will have to add the service to the file `CMakeLists.txt` Uncomment the `add_service_files` field and add the service type to it.

```
add_service_files(
  FILES
  AddTwoInts.srv
  )
```

Run `catkin_make` and test out the adder service.

Now let's try out the `imu_srv.py` node in the `kalman_zumy` package. In the command line, enter `rosrun kalman_zumy imu_srv.py zumyX`. If you have set up your server correctly, running `rosservice list` will now show a new service called `last_imu`. To check that this service works, try calling it from a terminal window running `rosservice call last_imu`[1]. Do you see any relevant accelerations when the Zumy is resting on the table/floor? How do the reported values change if you call the service while moving the Zumy?

2 Getting the Kalman filter node to call the IMU server

Now that you know how to read from the IMU, we want to use the IMU readings to help us track the position of the Zumy[2]. As we know, acceleration is the second derivative of position, so if we integrate the acceleration twice, we'll obtain the change in position. Look at the function `timeUpdate` in `kalman_filter.py` (irrelevant lines are removed for now).

```
def timeUpdate(self,u):
    # Define inputs
    u_lin = np.array([u.linear_acc.x,u.linear_acc.y]) - self.acc_bias
    u_ang = u.angular_velocity.z - self.gyro_bias
    # Determine orientation
    rot = np.array([[np.cos(self.psi),-np.sin(self.psi)],
                    [np.sin(self.psi), np.cos(self.psi)]])
    # Propagate dynamics
    self.x_lin += self.v_lin*self.dt + 0.5*rot.dot(u_lin)*pow(self.dt,2)
    self.v_lin += rot.dot(u_lin)*self.dt
    self.psi += u_ang*self.dt
```

The first two lines use the linear acceleration values and the rotational velocities from the srv `ImuSrv`. As the Zumy will probably not be facing the X direction, the next line computes the rotation matrix. The next 3 lines update the linear position and velocity of the Zumy and the heading angle, psi, of the Zumy.

Now, run your code by launching odroid, starting the imu_srv, and running the kalman filter via:

```
rosrun kalman_zumy kalman_filter.py zumy_name
```

[1] You need to start the Zumy first using the command: roslaunch odroid_machine remote_zumy.py mname:=zumyX

[2] Right now this alone is not the Kalman filter. This is part of the kalman filter, the time update part. We will build our way up to the Kalman Filter in the subsequent sections.

2.1 Testing your code

When running the program, open up Rviz and add the 'tf' package. You will see the estimated position and orientation of the Zumy. You can test to make sure the code works by physically moving the Zumy around and seeing if the corresponding Zumy in RViz moves too. Also try rotating the Zumy and making sure you see the Zumy rotate in Rviz. Does the code estimate the position of the Zumy perfectly? Why not? When does it work better and when does it work worse?

Now, comment out the line

```
Q, mu = self.calibrateSensors()
```

and repeat the subsection. Do you notice anything different? why?

Checkpoint 1

Get a TA to check your work. At this point you should be able to:

- Manually call the IMU service.
- Obtain the position estimate of the Zumy and see it on Rviz.
- Why the estimate works better and when it doesn't.
- Why the calibration improves the estimation of the Zumy.

---

3 Setting up the AR client node

As you can tell, the acceleration readings themselves are not perfect and simply integrating them twice will cause the estimate to drift.. However, as you also experienced from lab 6, the AR tags are not perfect either and drop quite frequently. What if we can combine the readings from both sensors to obtain a better estimate of the Zumy position? Ideally, we'd like to use the AR tags to give us absolute position when we have the data and when we don't have the data, we'd like to rely on the accelerometer readings.

The Kalman filter node doesn't know when new measurements will be available and so it doesn't make sense to have it permanently poll the AR node for information. Rather, we will have things work the other way around: the AR node can call the Kalman filter whenever it has new information available, and the Kalman filter will incorporate it into its estimation process. The AR fixes are used by the Kalman filter in the stage known as *measurement update*.

Modify your old AR tag tracker from Lab 6 to send the relative transformation of the origin to the zumy via the 'innovation' service. The service is started in `kalman_filter.py` via the command:

```
rospy.Service('innovation', NuSrv, self.triggerUpdate)
```

which means `kalman_filter` is the 'server' that receives a request from your AR tag code over the 'innovation' topic. Look at `NuSrv.srv` for the srv type. The origin field in `NuSrv.srv` denotes the AR tag or frame that should be considered the origin. Note that `NuSrv.srv` doesn't return any response but you will receive an error if you don't return something to the service call; therefore we need to return `[]` in your server code `kalman_filter.py`

4 Full Kalman filter implementation

Using the new position estimate of the Zumy via the Kalman filter, physically move the Zumy (either by hand or using zumy-teleop) around and watch its position in RViz. Are you able to track the Zumy better than before? Now run the keyboard code `roslaunch zumy_teleop zumy_teleop.launch zumyX` to control your zumy with the keyboard. In RViz, the estimated position of your zumy and the AR tag should be aligned. Try blocking camera's view of the Zumy, what happens to your estimated position of the Zumy?

Now read Appendix 1 - Kalman Filter and see if you can relate parts in the code to the equations in the appendix.

Checkpoint 2

Get a TA to check your work. At this point you should be able to:

- Show the KF working with the accelerometer and AR tags while controlling the Zumy with the keyboard
- Show the KF working when the camera is occluded while controlling the Zumy with the keyboard

---

Appendix 1 - Kalman Filter

The next step (contained in the function `kalman_filter.py`) is the meat of the tracking problem. The Kalman filter is an update rule that uses our measurement of the Zumy's current location to update our estimate of the Zumy's position while taking into account a simple physics model of the Zumy's motion. A Kalman filter iteration can be summarized by the following equations. (We will break down each of these equations in turn.)

$$\left.\begin{aligned} \hat{x} &= Ax + Bu \\ \hat{P} &= APA + Q \end{aligned}\right\} \quad \text{Predict Step} \tag{1}$$

$$\left.\begin{aligned} e &= z - C\hat{x} \\ S &= C\hat{P}C^T + R \\ K &= \hat{P}C^T S^{-1} \\ x^+ &= \hat{x} + Ke \\ P^+ &= (I - KC)\hat{P} \end{aligned}\right\} \quad \text{Update Step} \tag{2}$$

These equations are matrix equations and x is a vector that represents the state of the Zumy, both its position and velocity. $x := [x_1, \dot{x}_1, x_2, \dot{x}_2, \theta]^T$. The equation $\hat{x} = Ax + Bu$ is a called a state space model of the dynamics of the Zumy. $\hat{x}$ is x at the next time step. Thus, the matrix A tells us how x changes over time w.r.t. x and B tells us how the input u affects x. (Note that this type of linear equation can only represent very simple dynamics but it will be sufficient for our purposes.)

Actually, we really want to consider systems that have dynamics

$$\hat{x} = Ax + Bu + Bw \tag{3}$$

where $w \sim N(0, Q)$ is some 0-mean noise with covariance matrix Q. (Q is 4×4 positive definite matrix that describes the uncertainty in the state update.) In addition to our dynamics equation, we have a measurement equation that defines which states we are measuring.

$$y = Cx + v \tag{4}$$

y is our measurements and C defines how y relates to x. $v \sim N(0, R)$ represents the noise in our measurements. v is a 0-mean, random variable with covariance matrix R. R is a square, positive definite matrix but it has different dimensions than Q. It has the dimensions of y rather than x.

Let's now dig into the Kalman update equations. We'll focus on the update of x. We first go through a predictive step. Given the dynamics and our current estimate x, we would predict that at the next time step, the state would be equal to $\hat{x} = Ax$. If this were so, then our measurement would be equal to $C\hat{x}$. However, our real measurement z could be different giving us some error $e = z - C\hat{x}$. So instead of just assuming that our new state is $\hat{x} = Ax + Bu$, we will adjust for the error between our estimate and what we observe by updating our state to $x^+ = \hat{x} + Ke$. How do we determine this K matrix? It depends on our estimate of the uncertainty in the state which in turn depends on the noise in the state update and the noise in the measurements.

This is what the other equations in the Kalman filter update calculate. This K matrix is called the Kalman gain matrix. If you're interested in more of the theory, there are extensive resources online. The Kalman gain, as calculated in the update equations above, has been proven to be the best possible gain if the system is a linear system with Gaussian noise. This is not true for most systems but in practice the Kalman filter works well for many systems even nonlinear ones. In addition there are also extensions of the Kalman filter that apply to nonlinear systems. In our case, we will assume that the Zumy we're tracking is moving with constant velocity. This assumption will obviously not always be true but it is what we would expect if the Zumy is left to itself. Incorporating this assumption will help us to ignore measurements that indicate that the Zumy is jumping all over the place. In addition, it will allow us to maintain a more accurate estimate of the Zumy's position if the Zumy rolls out of view of the camera.