Localization Report (MidCompetition)

> Student ID : A111137
>
> Name : 邱柏鎧

Abstract

In this report, I will introduce my attemptions and end up with my choice for each subproblem (that is competition 1, 2, 3). Notice that i didn't use "robot localization" pkg, so the final score might not good enougth ?! The score can idealy be improved by EKF, or other novel methods.

ICP part

The first thing is construct an ICP algorithm to run basic condition (competition 1). By the PCL library, we can implement ICP easily. The whole part of ICP (align map) need several steps as below.

1. **Down Sampling** is the first step of all procedure. Two siginificant purposes are reducing heavy load of calculation and eliminate outlier.

```
// Downsampling scan_points
downsampling_filter("voxel", scan_range, scan_points,
filtered_scan_ptr);

/* Function for downsampling */
pcl::PointCloud<pcl::PointXYZI>::Ptr downsampling_filter(...){
  ...

  if(type == "voxel"){
    voxel_filter.setInputCloud(input_points);
    voxel_filter.setLeafSize(range[0], range[1], range[2]);
    voxel_filter.filter(*filtered_ptr);
  }
  else if(type == "passthrough_z"){
    ...
  }
  else if(type == "passthrough_xy"){
    pass_filter.setInputCloud(input_points);
    pass_filter.setFilterFieldName("x");
    pass_filter.setFilterLimits(init_guess(0,3) - range[0],
init_guess(0,3) + range[0]);
    pass_filter.filter (*filtered_ptr);
    pass_filter.setInputCloud(filtered_ptr);
    pass_filter.setFilterFieldName("y");
    pass_filter.setFilterLimits(init_guess(1,3) - range[1],
init_guess(1,3) + range[1]);
    pass_filter.filter (*filtered_ptr);
  }
  else{
    filtered_ptr = input_points;
  }
  return filtered_ptr;
}
```

After some attemptions, I found that passthrough and voxel filter are easy to design than other exist filter method. In my setting, Voxel filter is used for reduce high dense of scanning points, and passthrough filter is used for eliminate outlier and point cloud on the floor.

2. **ICP initial guess** is used at the first time to find the best yaw of the scanning points. Iteratively rotate the car, and calculate ICP score at each yaw, the highest score represent the best initial guessing pose. To accelerate working efficiency, i only check best position at first try, then i will set this result as known.

3. **ICP calculation** is the main part for calculating the score, and related transformation between lidar and map.

```
//==== ICP ====//
std::cout << "Start ICP " << std::endl;
icp.setInputSource(filtered_scan_ptr);
icp.setInputTarget(filtered_map_ptr);
icp.setMaxCorrespondenceDistance(10);
icp.setMaximumIterations(1000);
icp.setTransformationEpsilon(1e-12);
icp.setEuclideanFitnessEpsilon(1e-4);
icp.setRANSACOutlierRejectionThreshold(0.05);
icp.align(*transformed_scan_ptr, init_guess);
result = icp.getFinalTransformation();
```

Testing part

It is important for different try, even the parameters modify can change the fitting result. However, i won't introduce how to train these parameters, cause it mostly depends on intuition. Also i didn't spend much of time on these modification, instead, i will talk about my attempts of using the sensor messages.

To check wich message is available, i write a code ploting all message and find their performances. GPS works roughly well, but not precise; wheel odometry can use position and velocity for checking the path; IMU is good for checking the facing angle of the vehicle, however, i encounter serious problem when using secondary integral of linear accelearation.

1. Pure GPS update

GPS is one of the realiable message, and I just used it for update the initial guess. Because GPS will oscillate among a scope, even it is not precise enough, it is still useful when the point cloud feature is not clear. The final result can ensure car still moving, instead of restrict by icp result. Of course, GPS just barely reach the baseline(in competition 2 and 3).

2. Pure Odometry update

Using velocity of wheel odometry, update the initial guess before the ICP, ensuring the car will not stuck at origin position. The final result is better than GPS, cause GPS sometimes go backs (oscillation). The tendency of velocity is accurate enough, even cornering, the performence still well in the competition 3.

```
void pc_callback(const sensor_msgs::PointCloud2::ConstPtr& msg){
  ...

  ros::Duration pc_dt = msg->header.stamp - pc_lastTime;
  record_pc = msg;

  pure_odom_predict(odom_vx, odom_vy, pc_dt.toSec());
  pointcloud_process(record_pc);

  pc_lastTime = msg->header.stamp;
}

void pure_odom_predict(float v_x, float v_y, float dt){
  std::cout << "========Pure odom predict========" << std::endl;
  float global_vx = v_x*cos(imu_yaw) - v_y*sin(imu_yaw);
  float global_vy = v_x*sin(imu_yaw) + v_y*cos(imu_yaw);

  odom_x += global_vx*dt;
  odom_y += global_vy*dt;
  init_guess(0, 3) = odom_x;
  init_guess(1, 3) = odom_y;
  return;
}
```

3. KF for fusion odometry, GPS, ICP result

I try two combinations, one is fusion odometry and GPS. In my case, odometry is used for prediction, and GPS is used for update the position. ICP won't directly change the covariance, but will alter the position guess. The second method is fusion three of the information, that is ICP will also update the position, and the covariance is setting as below. The final result seems that the former works better, maybe the covariance of ICP need to be modified. The main problem of KF is setting the covariances, because we don't know the ground truth, so the accuracy of each sensor data can't be easily dicided.

```
gps_cov << 0.3, 0.0,
           0.0, 0.3; // 0.1
odom_cov << 0.05, 0.0,
            0.0 , 0.05;
state_cov << 1.0, 0.0,
             0.0, 1.0; // initial state cov
// ICP change covaraince of state_cov
state_cov(0, 0) += (float)score;
state_cov(1, 1) += (float)score;
// or just set as KF update
if(pc_frame>1){
  icp_cov << (float)score, 0.0,
            0.0,           (float)score;
  KF_update(result(0, 3), result(1, 3), icp_cov);
}
```

Code of KF part.

```
void KF_predict(float v_x, float v_y, float dt, Eigen::Matrix2f
predict_cov){
  std::cout << "==========KF predict==========" << std::endl;

  Eigen::Vector2f input_velocity(v_x*cos(imu_yaw) - v_y*sin(imu_yaw),
v_x*sin(imu_yaw) + v_y*cos(imu_yaw));
  Eigen::Vector2f current_state(init_guess(0, 3), init_guess(1, 3));

  Eigen::Vector2f miu = At*current_state + input_velocity*dt;
  Eigen::Matrix2f sigma = At*state_cov*At.transpose() + predict_cov;
  init_guess(0, 3) = miu(0);
  init_guess(1, 3) = miu(1);
  state_cov = sigma;
  return;
}

void KF_update(float update_x, float update_y, Eigen::Matrix2f update_cov){
  std::cout << "==========KF update==========" << std::endl;

  Eigen::Vector2f update_state(update_x, update_y);
  Eigen::Vector2f current_state(init_guess(0, 3), init_guess(1, 3));
  Eigen::Matrix2f identity_matrix;
  identity_matrix << 1.0, 0.0,
                     0.0, 1.0;

  Eigen::Matrix2f Kt = state_cov*update_matrix.transpose()*
(update_matrix*state_cov*update_matrix.transpose() + update_cov).inverse();
  Eigen::Vector2f miu = current_state + Kt*(update_state -
update_matrix*current_state);
  Eigen::Matrix2f sigma = (identity_matrix - Kt*update_matrix)*state_cov;

  init_guess(0, 3) = miu(0);
  init_guess(1, 3) = miu(1);
  state_cov = sigma;
  return;
}
```

4. Odom position difference

This method reachs same accuracy as KF method, however, this method is more easy to adjust, so I choose it as final submition. The concept of this method is same as the method 2, but there are some modification, that is using difference of wheel odometry position instead of velocity integral. And the most siginificant change is using **interpolation** to modify the distance bwtween two point cloud steps. If the ICP score become smaller, then the fix_rate will reduce confidence of odometry message, on the contrary, we will make the difference larger if the score become larger.

```
//==== Get odom data ====//
void odom_callback(const nav_msgs::Odometry::ConstPtr& msg){
  std::cout << "======== Odom msg ========" << std::endl;

  double pose_x = msg->pose.pose.position.x;
  double pose_y = msg->pose.pose.position.y;

  diff_x = pose_x*cos(imu_yaw)-pose_y*sin(imu_yaw) - odom_x_relate;
  diff_y = pose_x*sin(imu_yaw)+pose_y*cos(imu_yaw) - odom_y_relate;
  odom_x_relate = pose_x*cos(imu_yaw)-pose_y*sin(imu_yaw);
  odom_y_relate = pose_x*sin(imu_yaw)+pose_y*cos(imu_yaw);

  odom_ready = true;
  std::cout << "=================================" << std::endl;
  return;
}

//==== predtict next time position by odom ====//
init_guess(0, 3) += diff_x / freq_ratio;
init_guess(1, 3) += diff_y / freq_ratio;

if (score > last_score || !icp.hasConverged())
    freq_ratio * fix_rate;
else
    freq_ratio / fix_rate;
last_score = score;
```

Other modification

Loading map pcd into code

At competition 2, 3. I got some problem while getting message of map, more over, the frame will drop out because of the latency, even i set the queue of point cloud message up to 40000. To easily solve this problem, i find the merged pcd file, and load it into ICP code.

```
//==== Read map and Pub map ====//
map_points = (new pcl::PointCloud<pcl::PointXYZI>)->makeShared();
if (pcl::io::loadPCDFile<pcl::PointXYZI>(map_path, *map_points) == -1){
  PCL_ERROR("Couldn't read that pcd file\n");
  exit(0);
}
else{
  ROS_INFO("Got map ~~");
  map_ready = true;
  sensor_msgs::PointCloud2::Ptr map_cloud(new sensor_msgs::PointCloud2);
  pcl::toROSMsg(*map_points, *map_cloud);
  map_cloud->header.frame_id = "world";
```

```
    pub_map.publish(*map_cloud);
    ROS_INFO("Pub Map");
  }
```