Kidnapped Vehicle Project

In this project, a 2-D particle filter implemented in C++ is used to localize the vehicle. The particle filter will be given a map and some initial localization information (analogous to what a GPS would provide). At each time step your filter will also get observation and control data.

At first, we should initialize the filter...

```
std::default_random_engine gen;
num_particles = 100;  // TODO: Set the number of particles

normal_distribution<double> dist_x(x, std[0]);
normal_distribution<double> dist_y(y, std[1]);
normal_distribution<double> dist_theta(theta, std[2]);

double sample_x, sample_y, sample_theta;

for(int i = 0;i<num_particles;i++)
{

  Particle particle;
  sample_x = dist_x(gen);
  sample_y = dist_y(gen);
  sample_theta = dist_theta(gen);

  particle.id = i;
  particle.x = sample_x;
  particle.y = sample_y;
  particle.theta = sample_theta;
  particle.weight = 1.0;
  particles.push_back(particle);
  weights.push_back(1.0);
}
is_initialized = true;
```

Then we should predict the new particle location and add random Gaussian noise...

```
for(int i =0;i<num_particles;i++)
{
  if(fabs(yaw_rate) < 0.0001)
  {

    particles[i].x =  particles[i].x + velocity*delta_t*cos(particles[i].theta);
    particles[i].y =  particles[i].y + velocity*delta_t*sin(particles[i].theta);
    particles[i].theta = particles[i].theta;
  }
  else
  {
   particles[i].x =  particles[i].x + (velocity/yaw_rate)*(sin(particles[i].theta+(yaw_rate*delta_t))-sin(particles[i].theta));
   particles[i].y =  particles[i].y + (velocity/yaw_rate)*(cos(particles[i].theta)-cos(particles[i].theta+(yaw_rate*delta_t)));
   particles[i].theta = particles[i].theta + yaw_rate*delta_t;

  }

 normal_distribution<double> dist_xf(particles[i].x, std_pos[0]);
 normal_distribution<double> dist_yf(particles[i].y, std_pos[1]);
 normal_distribution<double> dist_thetaf(particles[i].theta, std_pos[2]);
 particles[i].x =dist_xf(gen);
 particles[i].y = dist_yf(gen);
 particles[i].theta =dist_thetaf(gen);
}
```

In data association, I used nearest neighbor data association, and assign each sensor observation the map landmark ID associated with it .

```
double diff;
LandmarkObs Obs;
LandmarkObs pr;
double nearest = std::numeric_limits<double>::max();
int index;
for(unsigned int i =0;i<observations.size();i++)
{
  Obs = observations[i];
  for(unsigned int j =0;j<predicted.size();j++)
  {
    pr = predicted[j];
    
    diff = dist(Obs.x,Obs.y,pr.x,pr.y);
    
    if(diff<nearest)
    {
      nearest = diff;
      index= predicted[j].id;
    }
  }
  observations[i].id = index;
}
```

Next we will update the particle weight, first convert all observations from vehicle coordinate to map coordinate, then check the map landmarks that should be considered based on sensor range with respect to each particle location. In this function I will call SetAssociations and dataAssociation then update the weights of each particle using a mult-variate Gaussian distribution.

```
double W_S = 0.0;
gauss_norm = 1 / (2 * M_PI * std_landmark[0] * std_landmark[1]);

for(int i = 0;i<num_particles;i++)
{
   trans_observations.clear();
   for(unsigned int j=0;j<observations.size();j++)
   {
       LandmarkObs trans_obs;
       trans_obs.x = particles[i].x + (cos(particles[i].theta))*(observations[j].x) - (sin(particles[i].theta))*(observations[j].y);

       trans_obs.y = particles[i].y + (sin(particles[i].theta))*(observations[j].x) + (cos(particles[i].theta))*(observations[j].y);
       trans_obs.id = observations[j].id;
       trans_observations.push_back(trans_obs);

   }
   predictions.clear();
   for(unsigned int l = 0; l<map_landmarks.landmark_list.size(); l++)
   {

     double lm_x = map_landmarks.landmark_list[l].x_f;
     double lm_y = map_landmarks.landmark_list[l].y_f;
     int lm_id = map_landmarks.landmark_list[l].id_i;
     distance = dist(lm_x,lm_y,particles[i].x,particles[i].y);
     if(distance<=sensor_range)
     {
       predictions.push_back(LandmarkObs{ lm_id, lm_x, lm_y });
       associations.push_back(lm_id);
       sense_x.push_back(lm_x);
       sense_y.push_back(lm_y);
     }



    SetAssociations(particles[i], associations, sense_x, sense_y);
    dataAssociation(predictions, trans_observations);

    particles[i].weight =1.0;
    for(unsigned int k =0;k<trans_observations.size();k++){
      int id_search = trans_observations[k].id;
      obs_x = trans_observations[k].x;
      obs_y = trans_observations[k].y;
      auto itr = std::find_if(predictions.begin(),predictions.end(),[id_search](const LandmarkObs& landmark)
                              {return landmark.id==id_search;});
      if(itr!=predictions.cend()){
            idx = std::distance(predictions.begin(),itr);
       }
             mu_x = predictions[idx].x;
             mu_y = predictions[idx].y;


          exponent = (pow(obs_x - mu_x, 2) / (2 * pow(std_landmark[0], 2)))
                + (pow(obs_y - mu_y, 2) / (2 * pow(std_landmark[1], 2)));

          weight_perObs =gauss_norm*exp(-exponent);
          if(weight_perObs>0)
          {
            particles[i].weight*=weight_perObs;
          }
}
W_S+=particles[i].weight;

}
  for(unsigned int s = 0;s<particles.size();s++)
  {
    particles[s].weight = particles[s].weight/W_S;
    weights[s] = particles[s].weight;
```

Then Resample particles with replacement with probability proportional to weight.

```
    std::vector<Particle> particles_resampled;
    std::random_device rd;
    std::mt19937 gen(rd());
  

    std::discrete_distribution<> d(weights.begin(),weights.end());

    for(int n=0; n<num_particles; ++n) {
        particles_resampled.push_back(particles[d(gen)]);
    }
  particles = particles_resampled;
```