AdvML Lab3

Andreas Stasinakis(andst745)

07 October, 2019

Contents

Question 1: STATE SPACE MODELS - PARTICLE FILTER **2**
1.1 Simulate from a SSM. Robot Localization via Particle Filter 2
1.2 Repeat the procedure with different emission standard deviation 12
sd = 5 12
sd = 50 14
1.3 Weights of Filter equal to 1 16

Question 1: STATE SPACE MODELS - PARTICLE FILTER

The purpose of the lab is to put in practice some of the concepts covered in the lectures. To do so, you are asked to implement the particle filter for robot localization. For the particle filter algorithm, please check Section 13.3.4 of Bishop's book and/or the slides for the last lecture on state space models (SSMs). The robot moves along the horizontal axis according to the following SSM:

$$p(z_t \mid z_{t-1}) = \Big(\mathcal{N}(z_t \mid z_{t-1}, 1) + \mathcal{N}(z_t \mid z_{t-1} + 1, 1) + \mathcal{N}(z_t \mid z_{t-1} + 1, 1)\Big)/3 \quad \text{Transition model}$$

$$p(x_t \mid z_t) = \Big(\mathcal{N}(x_t \mid z_t, 1) + \mathcal{N}(x_t \mid z_t - 1, 1) + \mathcal{N}(x_t \mid z_t + 1, 1)\Big)/3 \quad \text{Emission model}$$

$$p(z_1) = Uniform(0, 100) \quad \text{Initial model}$$

1.1 Simulate from a SSM. Robot Localization via Particle Filter

Implement the SSM above. Simulate it for T = 100 time steps to obtain z_(1:100) (i.e., states) and x1:100 (i.e., observations). Use the observations (i.e., sensor readings) to identify the state (i.e., robot location) via particle filtering. Use 100 particles. Show the particles, the expected location and the true location for the first and last time steps, as well as for two intermediate time steps of your choice.

```
library(gridExtra)
library(ggplot2)
```

```
###################### IMPLEMENTATION OF THE FUNCTIONS ##########################

#The transition model: NOTE - It is not the average of three normal!
#It is a mixture model. Each distribution has the same coeficient
#That means that it is equal likely to generate from one distribution between 3
#For that reason, we sample an index between in order to know from which
#distribution we are going to generate
#After that we generate from that one
#The difference between the distr is that we add a different constant in the mean
#For that reason we sample that constant with equal probability

TransModel = function(size,mu , sd_trans, coef){

  #sample equally from 3 constants
  index = sample(x = c(0,1,2),size = size,replace = TRUE,prob = coef)

  #Sample from the normal adding each time the constant
  #In this way, we sample from the micture distribution
  smple = rnorm(n = size,mean = mu + index, sd = 1)
  return(smple)

}

#The emission Model
#Every we say above holds also for the emission model
EmissModel = function(size, mu , sd_emi,coef){

  #Here the constant we add is different though
```

```
  index = sample(x = c(0,-1,1),size = size,replace = TRUE,prob = coef)

  smple = rnorm(n = size, mean = mu + index, sd = sd_emi)
  return(smple)
}


#The initial model: Just a uniform distribution: No clue where the stupid robot is
InitModel = function(init,size,...){

  res = init(size,...)
  return(res)

}

#Function to simulate the SSM
#In this function
SSM_simulation = function(steps,size,InitMl,sd_trans,sd_emi,...){

  #generate from the initial model
  #Use the function IniModel in order to use it for different models
  initValues = InitModel(init = InitMl,size, ...)

  True_state = rep(0,steps)
  True_state[1] = initValues

  Observations = rep(0,steps)
  Observations[1] = EmissModel(size = 1,True_state[1],sd_emi = sd_emi,
                               coef = rep(1/3,3))
  #a for loop to generate from the transition and emission model
  for (i in 2:steps) {

    True_state[i] = TransModel(size = size,mu = True_state[i-1],
                               sd_trans = sd_trans,coef = rep(1/3,3))

    Observations[i] = EmissModel(size = size,True_state[i],sd_emi = sd_emi,
                                 coef = rep(1/3,3))
    }

  return(list(True_state,Observations))


}

#function to evaluate the particles( compute the weights)
#Here we can caclulate the densities of the three normal and average them
#We also have to normalize the weights
evalParticles = function(Observation,particls,s){

  dens = (dnorm(x = Observation,mean = particls,sd = s)+
  dnorm(x = Observation,mean = particls -1 ,sd = s)+
  dnorm(x = Observation,mean = particls +1 ,sd = s))/3
```

```
  norm_dens = dens/sum(dens)
  #return the average of the densities
  return(norm_dens)

}

########Particle filter##########

PartFIlter = function(steps,#How many steps we will do
                      partSize,#How many particles
                      initMdl, #what is my initial model
                      sd_trans, #sd of transition model
                      sd_emi,#sd of emition model
                      correction = TRUE,#True if we want to calculate weights
                      method, #which method the user wants
                      observ, #send the observations
                      ...)#... for some extra parametres
  {

  #method first
  if(method == 1){

    #Initially generate partSize of particles
    #The first step is to generate from the initial model Z0
    particles  = InitModel(init = initMdl,size = partSize,...)
    N = length(particles)

    #Store all the particles in each step to plot them
    all_particles = matrix(0,nrow = steps,ncol = partSize)

    #vector to store the expected positions
    predictions = rep(0,N)
    Weights = rep(0,N) #store the weights


    for (i in 1:(steps)) {

      #sample from transition model one step ahead using the previous particles
      #So we think If i am in this position now, where i will be in the next
      #Time step. Of course this is just my belief.
      # I am doing that for ALL particles without using the weights

      belief = TransModel(size = N,mu = particles,sd_trans = sd_trans,
                          coef = rep(1/3,3))

      #I calculate the weights now. The weights are how likely is to have
      #an observation with that position.
      #We know Z_t(observation) and we condition to the belief we did before.

      weights = evalParticles(Observation = observ[i],
                              particls = belief,
                              s = sd_emi)
```

```
    #If we do not want a correction, we convert all weights into 1/N
    if(correction == FALSE){
      weights = rep(1/N,N)
    }

    #Now the Resampling starts. We have the belief, but we know that not all
    #the particles are needed, in the sense that some of them are closer to
    #the real state. Therefore we have to weight them and use the weights
    #In this step we sample from our belief but we also use the weights.
    #More specific, we sample the same number of particles BUT using the
    #weights will give us many times the same particles.
    particles = sample(x = belief,size = N,replace = TRUE,prob = weights)
    all_particles[i,] = particles

    predictions[i] = mean(particles) #make the prediction

  }
  return(list(predictions,all_particles))

#second method
}else{

  #We store the particles at each step
  particles = matrix(0,nrow = steps,ncol = partSize)

  #Initially generate partSize of particles
  init_particles  = InitModel(init = initMdl,size = partSize,...)

  N = ncol(particles)

  #vector to store the expected positions
  predictions = rep(0,N)

  Weights = matrix(0,nrow = steps,ncol = partSize) #store the weights

  #In this method, we first evaluate the weights, using the function evalParticles
  #We calculate the weights for each particle in every step
  #This will give us the probability of my predictions


  #Now we use the trainsition model
  #First we sample from the particles that we already have,
  #But we use the weights in order to sample the most probable
  #In the initial step, all weights are equal

  temp_particles = sample(x = init_particles,size = N,replace = TRUE,
                          prob = rep(1,N))

      #After that we sent them to the transition model in order to make a prediction
  particles[1,] = TransModel(size = N,mu = temp_particles,
                             sd_trans = sd_trans, coef = rep(1/3,3))

  #finally we update the weights
```

```
     Weights[1,] = evalParticles(Observation = observ[1],
                                 particls = particles[1,],
                                 s = sd_emi)
    if(correction == FALSE){
         Weights[1,] = rep(1/N,N)


    }

    #Store the prediction
    predictions[1] = sum(Weights[1,]*particles[1,])

    #We now do the same for ALL steps, each time the same procedure
    for (i in 2:(steps)) {


      #So first we have to propagate the particles one step ahead
      #We sample using the weights N particles in order to use the most likely
      temp_particles = sample(x = particles[i-1,],size = N,replace = TRUE,
                              prob = Weights[i-1,] )

      #We propagate those particles using the transition model
      particles[i,] = TransModel(size = N,mu = temp_particles,
                                 sd_trans = sd_trans, coef = rep(1/3,3))

      #Now we update the weights for each particle
      Weights[i,] = evalParticles(Observation = observ[i],
                                  particls = particles[i,],
                                  s = sd_emi)


       #If we do not need a correction, we just assign  equal weights
      if(correction == FALSE){
        Weights[i,] = rep(1/N,N)
      }


      #Here our prediction is the sum of the multiplication between the
      #current weight and the particles

      predictions[i] = sum(Weights[i,]*particles[i,])

      }
  }

    #The function returns the predictions and the particles for each time step
    return(list(predictions,particles))
}



# Function to take the legend of a plot
g_legend <- function(a.gplot){
```

```
  tmp <- ggplot_gtable(ggplot_build(a.gplot))
  leg <- which(sapply(tmp$grobs, function(x) x$name) == "guide-box")
  legend <- tmp$grobs[[leg]]

  return(legend)

}



#A function to plot the particles in different times steps
#Also we plot the true and the expected location
plot_function = function(particles,true_pos,exp_pos,plots){

  plot_list = list()
  plt_N = 1
  legend_taken = FALSE

  for(i in plots){
    p = eval(substitute(

      ggplot() +
        geom_histogram(mapping = aes(x = particles[i,], y = ..density..),
                       bins = 50,fill = "lightblue",color = "blue")+
        geom_density(mapping = aes(x = particles[i,]),
                     color = "yellow",size = 0.5, lty = 2)+
        geom_point(mapping = aes(x = exp_pos[i], y = 0,
                                 col = "Expected\nPosition"), size = 2.5 )+
        geom_point(mapping = aes(x = true_pos[i], y = 0,
                                 col = "True\nPosition") ,size = 2.5)+
        scale_color_manual(values = c("True\nPosition" = "red",
                                      "Expected\nPosition" = "blue"),
                           name = "Position")),list(i = i))+
      labs(subtitle = paste("Time step = ",i),x = "Position",
           y = "Density")+
      theme_bw()



    # Make a common legend for every plot
    if(!legend_taken) {

      # Take the legend
      p =  p +
      theme(legend.text = element_text(margin = margin(b = 7.5, unit = "pt")))
      my_legend <- g_legend(p)
      legend_taken <- T
    }

    # Remove the legend from the individual plot and add it to the plot list
    p = p + theme(legend.position = "none",
    plot.margin = unit(c(5.5,22,5.5,5.5), "pt"))
```

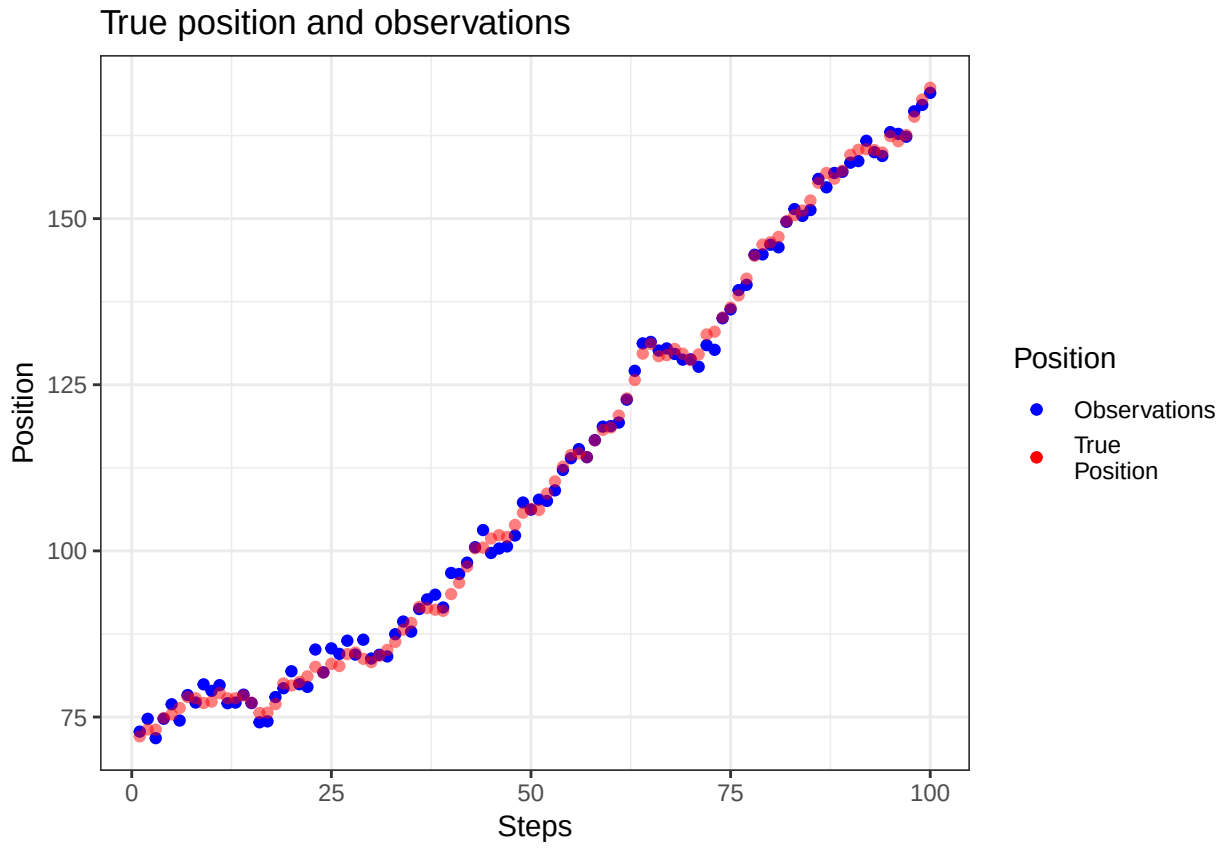
```
    #plot_list = rlist::list.append(plot_list, p)
    plot_list[[plt_N]] = p
    plt_N = plt_N +1


    }



  grid.arrange(do.call("arrangeGrob", c(plot_list, ncol=2)), my_legend,
               ncol = 2, widths = c(8, 1))

}

#Plot the residuals
plot_res = function(resid,#vector of the residuals
                    sd) # For what sd we run the filter in order to print it
  {

  #plot the residuals
  res_plot = ggplot()+
  #geom_histogram(mapping = aes(resid, y = ..density..),fill = "green",bins = 50)+
  geom_density(mapping = aes(resid), color= "black", alpha = 0.5,
               fill = "red", lty = 2) +
  geom_vline(xintercept = mean(resid), color = "yellow", lty = 2, size = 1)+
  labs(subtitle = paste("sd =",sd),
       x = "Residuals", y = "Density")+
  theme_light()
  return(res_plot)

}


#plot the predictions in the last step
plt_predict = function(exp_pos,true_pos, sd){

  steps = seq(1,nrow(exp_pos),1)
  plot_list = list()
  plt_N = 1
  legend_taken = FALSE

  for(i in sd){
    p = eval(substitute(

   ggplot()+
    geom_point(mapping = aes(x = steps, y = exp_pos[,plt_N],
                             col = "Expected\nPosition"), alpha = 0.5)+
    geom_line(mapping = aes(x = steps, y = true_pos[,plt_N],
                             col = "True\nPosition"), lty = 2)+
     scale_color_manual(values = c("True\nPosition" = "red",
                                   "Expected\nPosition" = "blue"),
                             name = "Position")),list(i = i))+
      labs(subtitle = paste("sd = ", sd[plt_N]),
```

```
            x = "Position",
            y = "Density")+
      theme_bw()

     # Make a common legend for every plot
    if(!legend_taken) {

      # Take the legend
      p =  p +
      theme(legend.text = element_text(margin = margin(b = 7.5, unit = "pt")))
      my_legend <- g_legend(p)
      legend_taken <- T
    }

    # Remove the legend from the individual plot and add it to the plot list
    p = p + theme(legend.position = "none",
    plot.margin = unit(c(5.5,22,5.5,5.5), "pt"))

    #plot_list = rlist::list.append(plot_list, p)
    plot_list[[plt_N]] = p
    plt_N = plt_N +1


    }


   grid.arrange(do.call("arrangeGrob", c(plot_list, ncol=2)), my_legend,
                ncol = 2, widths = c(8, 1), top = "Filtering distributions")

}
```

```
############################## RUN THE ALGORITHS ##################################

#Simulate the data
set.seed(12345) #set the seed for same results

#Simulate 100 steps
simulation_SSM = SSM_simulation(steps = 100,size = 1,InitMl = runif,1,1,0,100)

true_states = simulation_SSM[[1]] #store the true positions
Obs_states = simulation_SSM[[2]]  #store the observations

#plot true positions and the observations of the robot in order to get an idea
plot_true =  ggplot()+
    geom_point(mapping = aes(x = c(1:100), y = Obs_states,
                             col = "Observations"))+
    geom_point(mapping = aes(x = c(1:100), y = true_states,
                             col = "True\nPosition"), alpha = 0.5)+
     scale_color_manual(values = c("True\nPosition" = "red",
                                   "Observations" = "blue"),
                             name = "Position")+
    labs(title = "True position and observations",
                         x = "Steps",
                          y = "Position")+
```

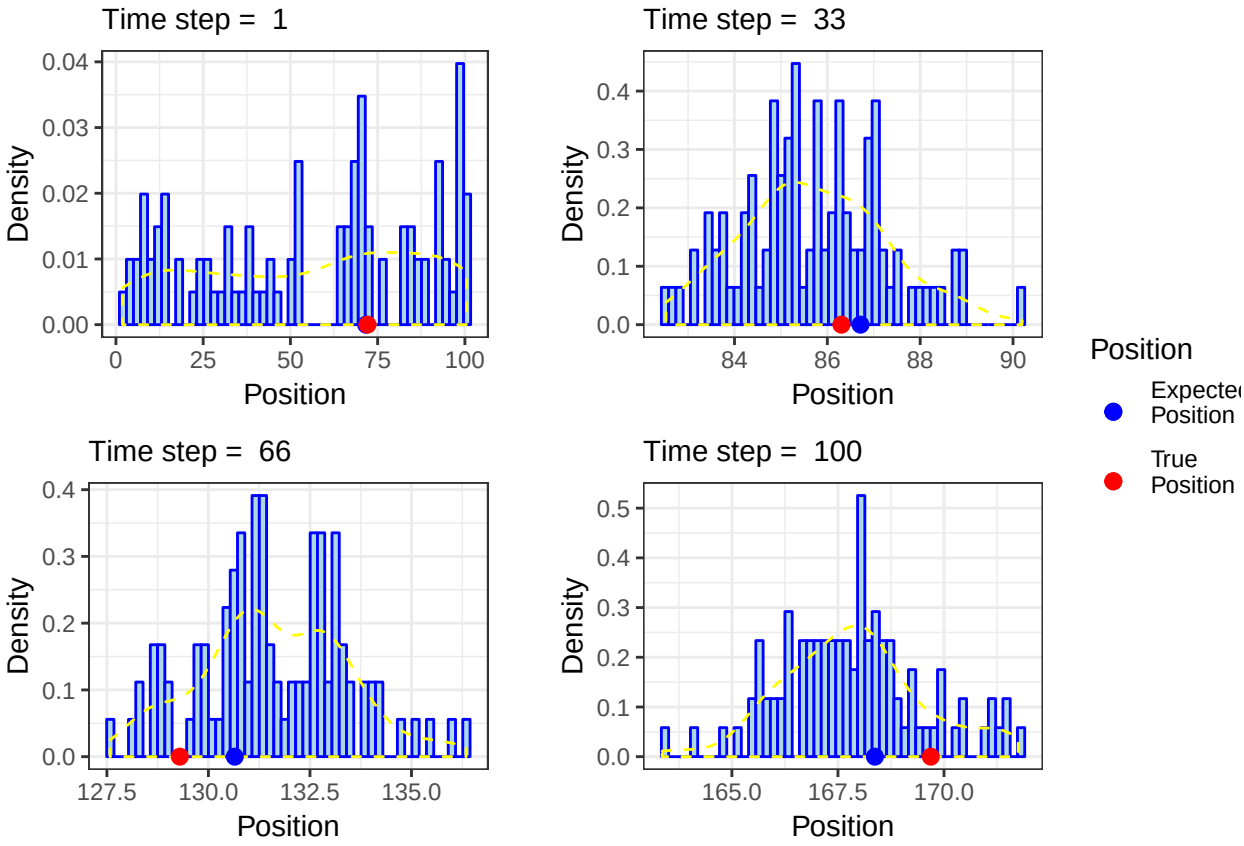
```
      theme_bw()

#Run the filter for emission standard deviaton equal to 1

part_Filt_sd1 = PartFIlter(steps = 100,
                           partSize = 100,
                           initMdl = runif,
                           sd_trans = 1,
                           sd_emi = 1,
                           correction = TRUE,
                           method = 2,
                           observ = Obs_states,
                           0,100)

predict_sd1 = part_Filt_sd1[[1]] #Store the predictions
particles = part_Filt_sd1[[2]]
residuals = predict_sd1 - true_states #calculate the residuals
abs_error = mean(abs(residuals)) #calculate the mean of the  resid error
```

We first plot the observations and the true position of the robot. It is obvious that the sensor gives as pretty accurate results, which means that the proposal distribution(emission model) seems reasonable. Of course we could comment that from just looking to the two models(transition and emission), which are both a mixture of three Normal distributions with equal linear coefficients.



The plots below present the filtering distribution $P(Z_t|X_{1:t})$ for different time steps. They also show the true location and the expected location of the robot. As we can see the predictions seem both reasonable and

accurate.


Time step = 1
Time step = 33
Time step = 66
Time step = 100
Density
Position
Position
Expecte
Position
True
Position

1.2 Repeat the procedure with different emission standard deviation

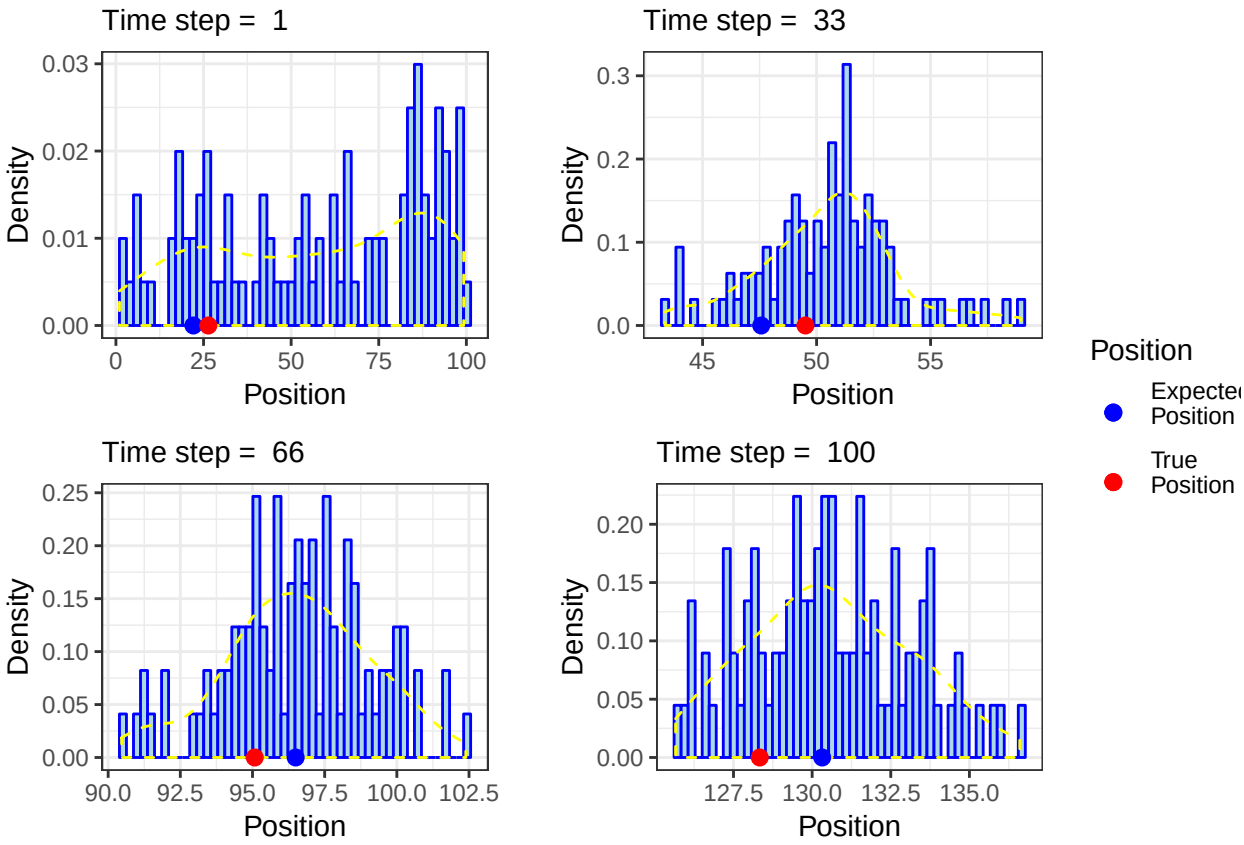
Repeat the exercise above replacing the standard deviation of the emission model with 5 and then with 50. Comment on how this affects the results.

In this task we run the particle filter again but this time we change the standard deviation of emission model

sd = 5

As we can see the results are slighly worse than the run with sd = 1. In this case the probability that a value outside of the target distribution will be selected is higher than before, because the proposal distribution is wider. Therefore, the particles that are close to the mean of will have a huge probability to be selected while other particles may never selected. As a result, the predictions will follow a linear form. We also plot the predictions in the task 1.3 below for each time step in order to comment on that assumption.

```
###### change the variance
#So we generate again the data
simulation_sd5 = SSM_simulation(steps = 100,size = 1,InitMl = runif,1,
                                sd_emi = 5,0,100)

true_states_sd5 = simulation_sd5[[1]] #store the true positions
Obs_states_sd5 = simulation_sd5[[2]]


#Here we run the particle filter with sd = 5
part_Filt_sd5 = PartFIlter(steps = 100,partSize = 100,initMdl = runif,
                           sd_trans = 1,sd_emi = 5,correction = TRUE,method = 2,
                           observ = Obs_states_sd5,
                           0,100)


predict_sd5 = part_Filt_sd5[[1]] #Store the predictions
particles_sd5 = part_Filt_sd5[[2]]
residuals_sd5 = predict_sd5 - true_states_sd5 #calculate the residuals
abs_error_sd5 = mean(abs(residuals_sd5)) #calculate the abs_error


plt_sd5 = plot_function(particles = particles_sd5,true_pos = true_states_sd5,
                        exp_pos = predict_sd5,plots = c(1,33,66,100))
```

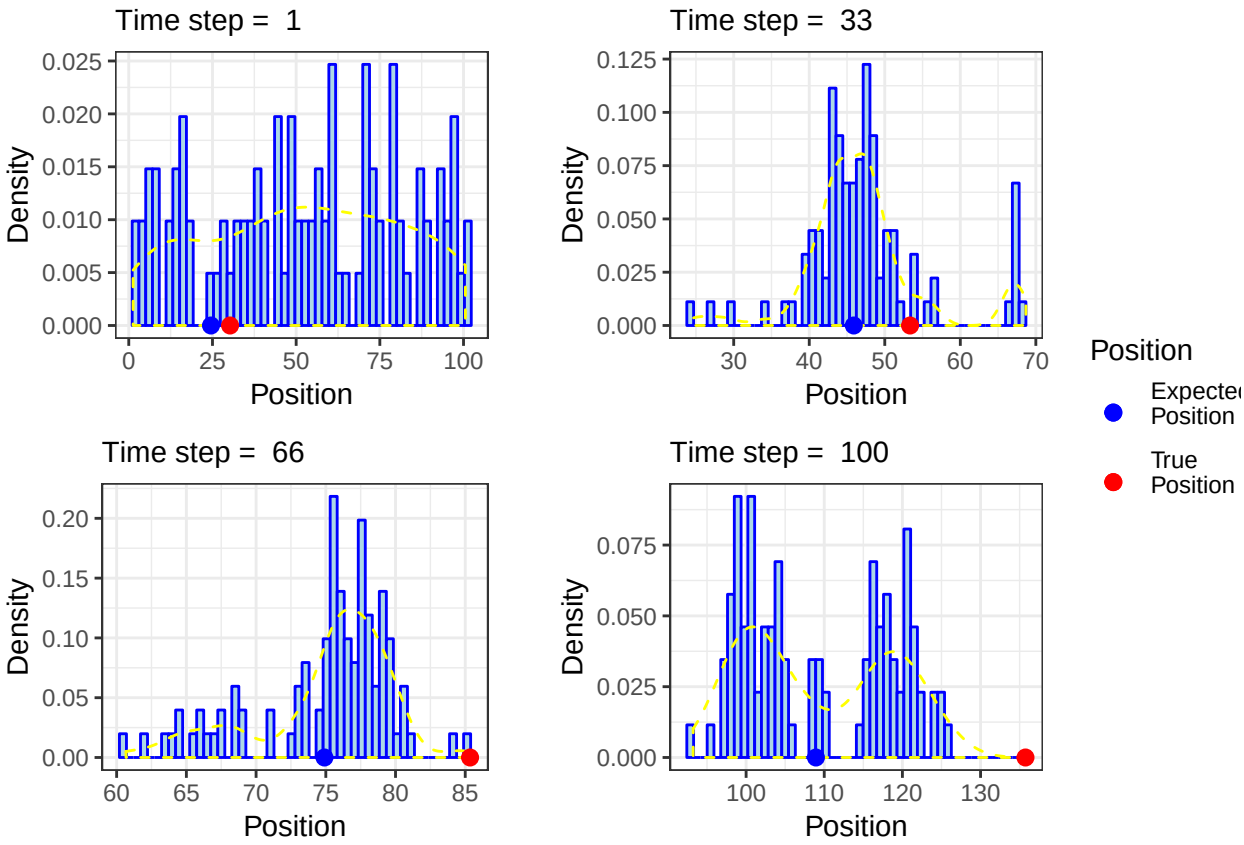



```
#plot the residuals
plt_Residuals_sd5 = plot_res(resid = residuals_sd5,sd = 5)
```

sd = 50

It is obvious that when we run the filter for sd = 50, we do not have good results. The reason for that is because we generate the particles using a really huge range. That means that we do not have much information for the relationship between the obserations and the true position. Therefore, the sensor is not giving valiable information about the position of the robot. One solution to that could be to increase the number of particles. By the law of large numbers we know that, while N $\to \infty$, the expected value of the estimator will converge to the real value.

```
#So we generate again the data
simulation_sd50 = SSM_simulation(steps = 100,size = 1,InitMl = runif,1,sd_emi = 50,0,100)

true_states_sd50 = simulation_sd50[[1]] #store the true positions
Obs_states_sd50 = simulation_sd50[[2]]



part_Filt_sd50 = PartFIlter(steps = 100,partSize = 100,initMdl = runif,
                            sd_trans = 1,sd_emi = 50,correction = TRUE,
                            observ = Obs_states_sd50,
                            method = 2,0,100)


predict_sd50 = part_Filt_sd50[[1]] #Store the predictions
particles_sd50 = part_Filt_sd50[[2]]
residuals_sd50 = predict_sd50 - true_states_sd50 #calculate the residuals
abs_error_sd50 = mean(abs(residuals_sd50)) #calculate the abs_error


plt_sd50 = plot_function(particles = particles_sd50,true_pos = true_states_sd50,
                         exp_pos = predict_sd50,plots = c(1,33,66,100))
```



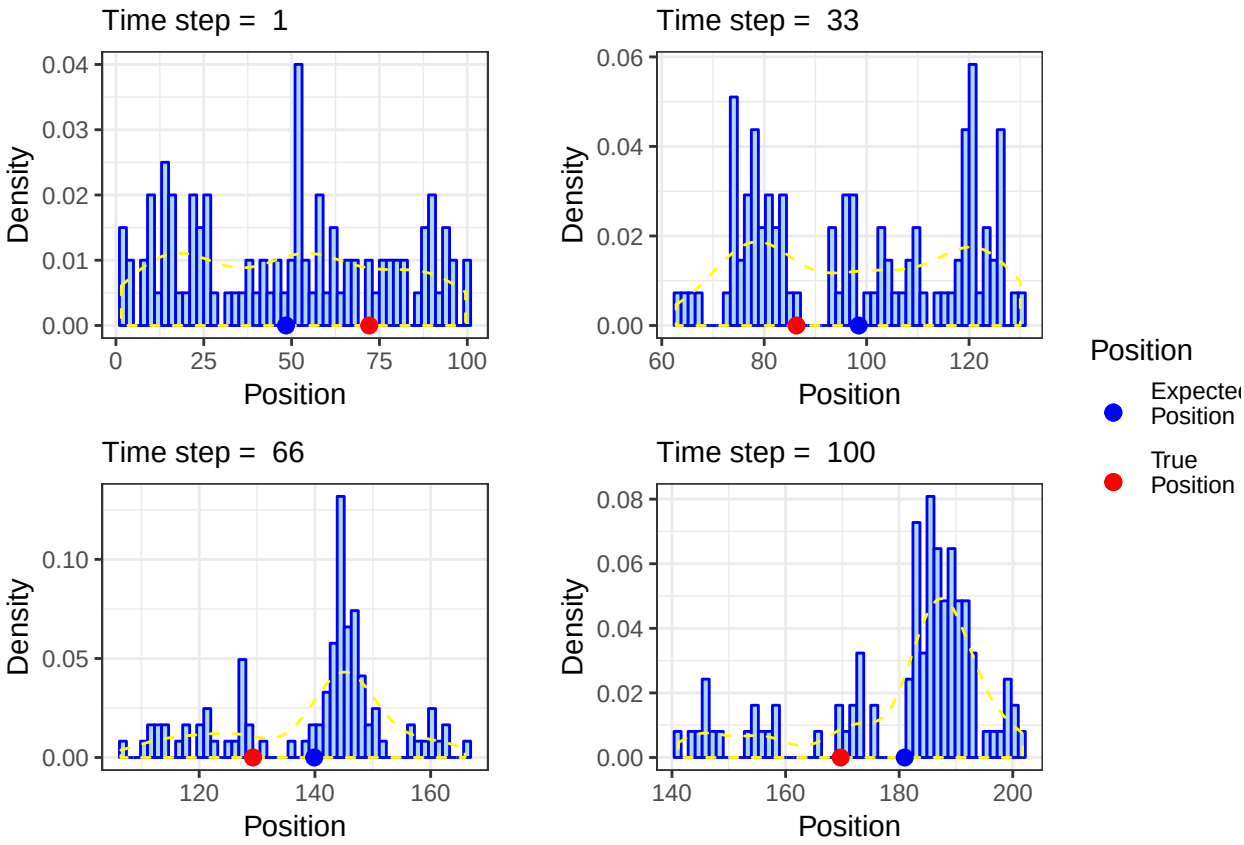

```
#plot the residuals
plt_Residuals_sd50 = plot_res(resid = residuals_sd50,sd = 50)
```

1.3 Weights of Filter equal to 1

Finally, show and explain what happens when the weights in the particle filter are always equal to 1, i.e. there is no correction.

Finally, we run the algorithm without the correction part. That means that when we resample we do not use the emition model in order to compute the weights. It is obvious that this cause bad results. The entire idea of the *Particle Samlper* is that we weight the performance of each particle in order to avoid generating particles that are not possible to be found in the data. If you exclude this step, the algorithm is only using the transition model, which means that we do not actually use all information available. Therefore, this kind of sampler is not a particle filter but one regular Importance sampler. Again, as before, if we increase the number of particles, the expected value wil converge to the true value but it will be really computational expensive.

```
#Without Correction

part_Filt_noCOrr = PartFIlter(steps = 100,partSize = 100,
                              initMdl = runif,sd_trans = 1,
                              sd_emi = 1,correction = FALSE,
                              observ = Obs_states,
                              method = 2,0,100)


predict_noCorr = part_Filt_noCOrr[[1]] #Store the predictions
particles_noCorr = part_Filt_noCOrr[[2]]
residuals_noCorr = predict_noCorr - true_states #calculate the residuals
abs_error_noCorr = mean(abs(residuals_noCorr)) #calculate the abs_error


plt_noCorr = plot_function(particles = particles_noCorr,true_pos = true_states,
                       exp_pos = predict_noCorr,plots = c(1,33,66,100))
```

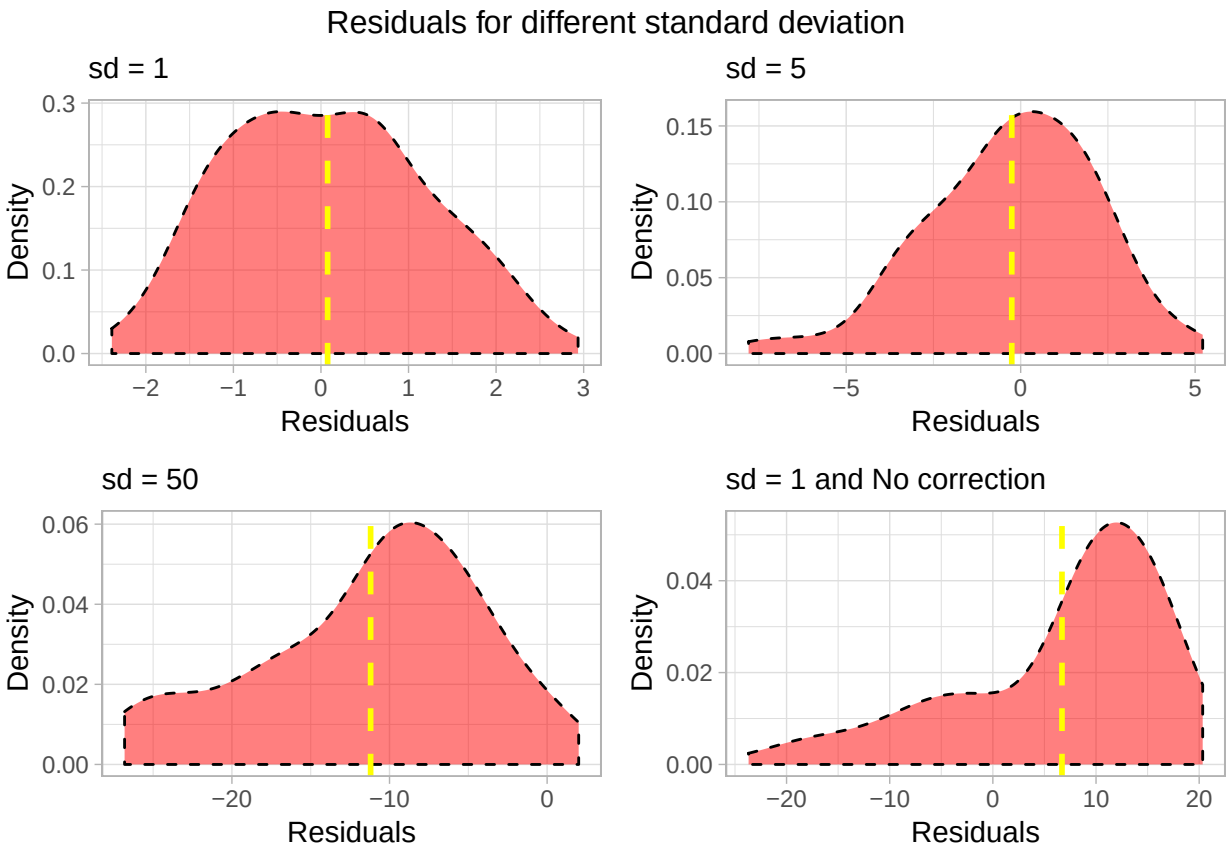


Finally, we also plot the residuals distribution and the prediction paths for all 4 methods above. For the residuals, we can see that will we increase the standard deviation, the distribution of the residuals is actually becoming wider and the mean of the residuals is of course increasing.

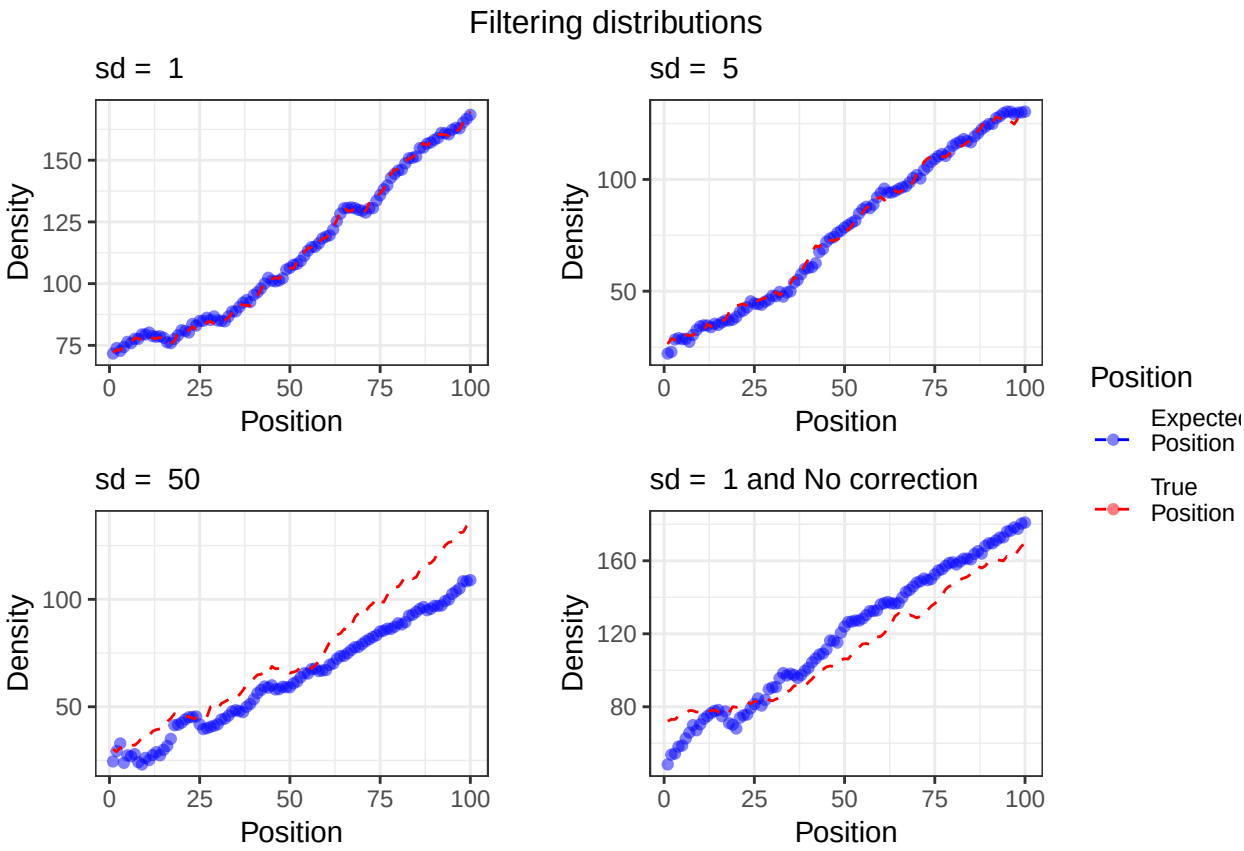
For the predictions, we can say that while sd is increasing, the predictions are not capture the noisy of the data. This is happening because, when we evaluate the weights at each step, the range of the posible particles is big. Therefore, many particles may have really low probabilities to be choosen. Then, when we sample using that weights only the particles that are close to the previous value will be choosen.

For the last method(no correction), it is clear that our predictions are fluctuating a lot because of the randomnes(the weights are not used here).

```
#plot the residuals
plt_Residuals_noCorr = plot_res(resid = residuals_noCorr,sd = "1 and No correction")

#Plot in the same figure all the residuals for diff sd
plt_list_res = list(plt_Residuals_sd1,plt_Residuals_sd5,
                    plt_Residuals_sd50,plt_Residuals_noCorr)
grid.arrange(grobs =plt_list_res, top = "Residuals for different standard deviation")
```




```
#Plot in the same figure all the predictions for diff sd
df_pred = data.frame(predict_sd1,predict_sd5,predict_sd50,predict_noCorr)
df_true_states = data.frame(true_states,true_states_sd5,true_states_sd50,true_states)
plt_all_predictions =  plt_predict(exp_pos = df_pred,df_true_states,
                                   sd = c(1,5,50,"1 and No correction"))
```


Filtering distributions
sd = 1
sd = 5
sd = 50
sd = 1 and No correction
Density
Position
0
25
50
75
100
150
125
100
75
100
50
160
120
80
Position
Expecte
Position
True
Position