732A96: Advance Machine Learning

LAB 4: STATE SPACE MODELS

Arian Barakat/ariba405

Background:

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 SSMs. The robot moves along the horizontal axis according to the following SSM:

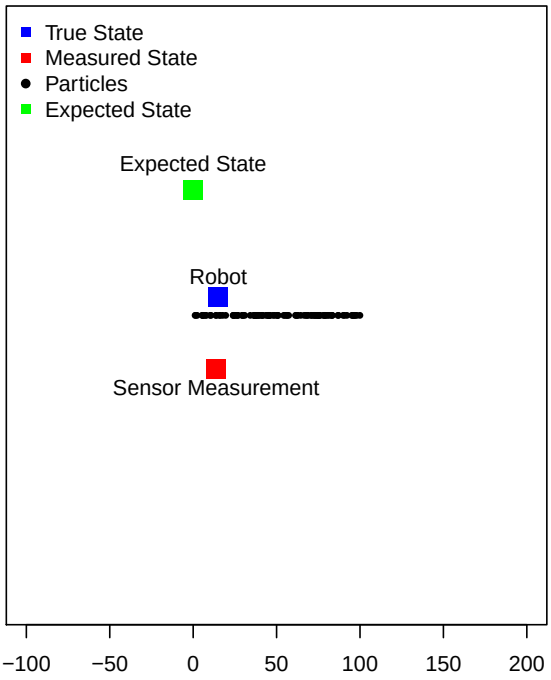
See instruction sheet for model specification

Implement the SSM above. Run it for $T = 100$ time steps to obtain $z_{1:100}$ (i.e. states) and $x_{1:100}$ (i.e. observations). Use the observations (i.e. sensor readings) to identify the state (i.e. robot location) via particle filtering. Use 100 particles. For each time step, show the particles, the expected location and the true location. Repeat the exercise after replacing the standard deviation of the emission model with 5 and then with 50. Comment on how this affects the results. Finally, show and explain what happens when the weights in the particle filter are always equal to 1, i.e. there is no correction.

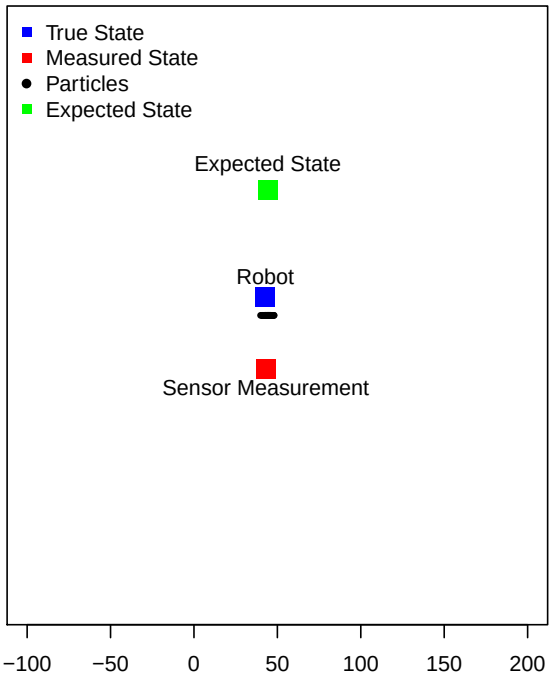
Standard Deviation: 1

Link To Video

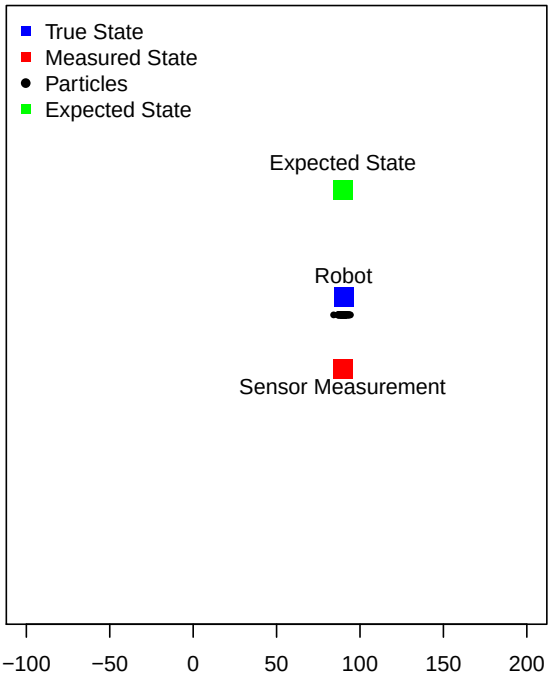
Time Step: 1



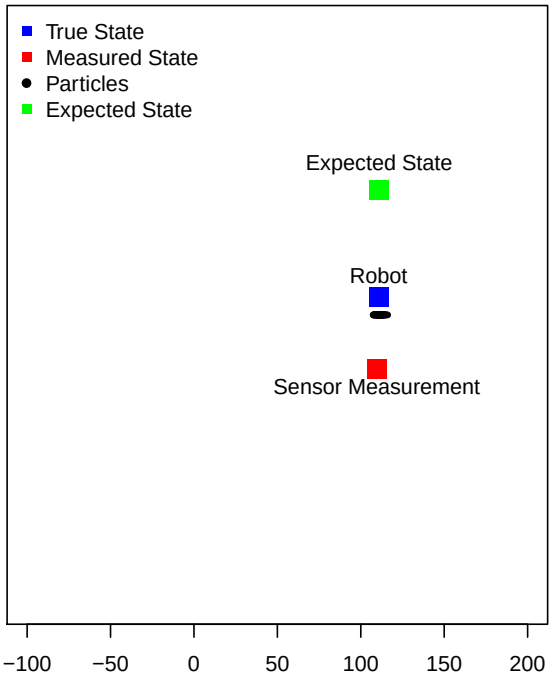
Time Step: 40



Time Step: 80



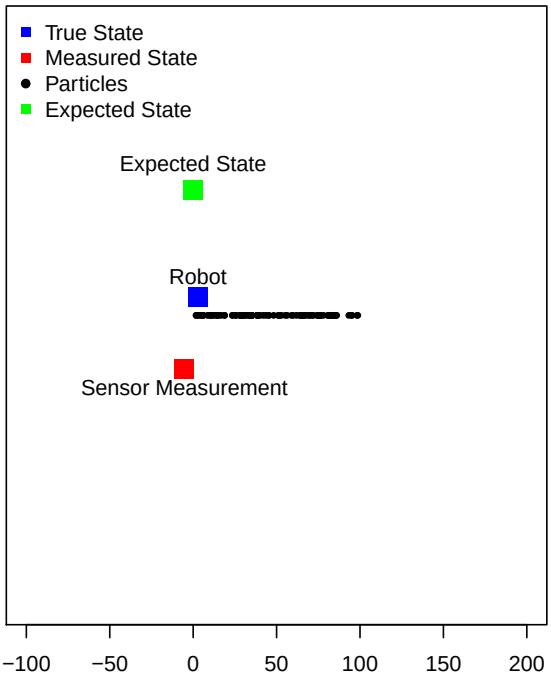
Time Step: 100



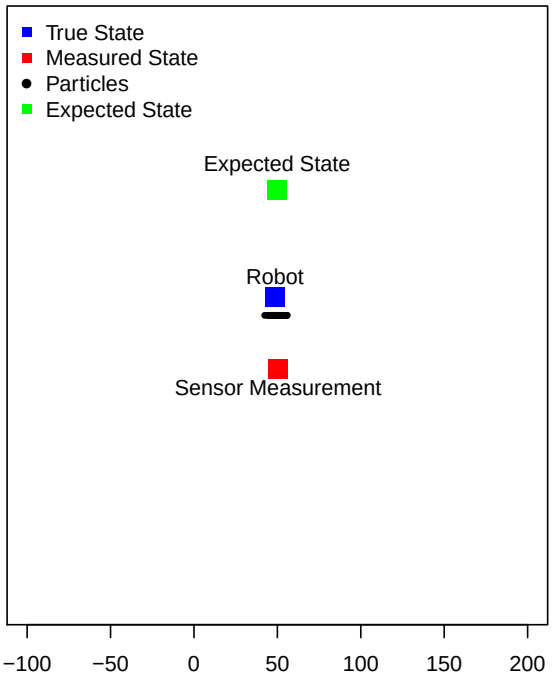
Standard Deviation: 5

Link To Video

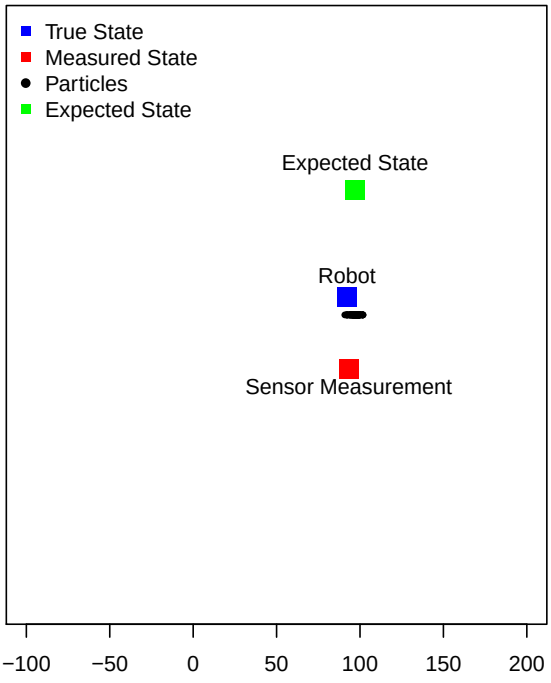
Time Step: 1



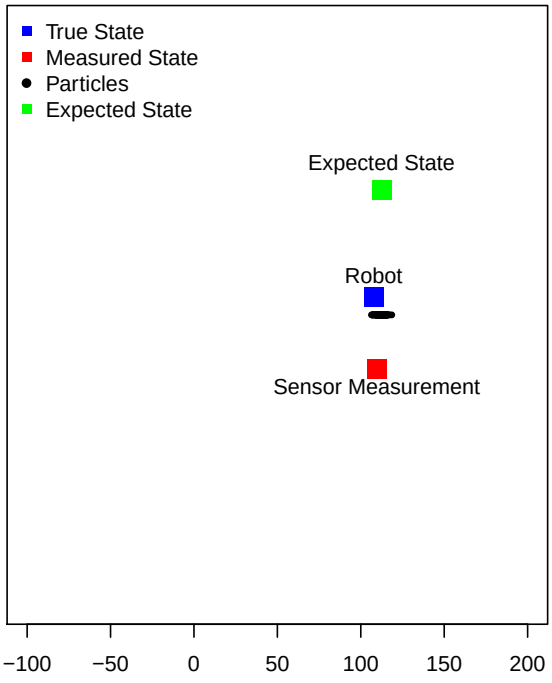
Time Step: 40



Time Step: 80



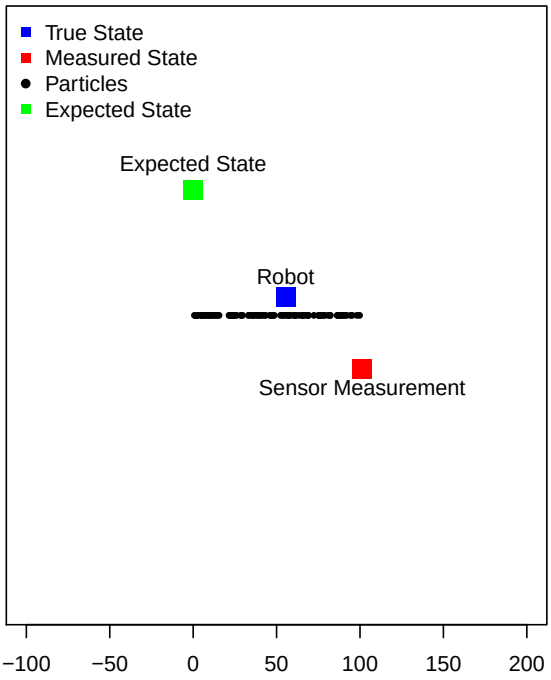
Time Step: 100



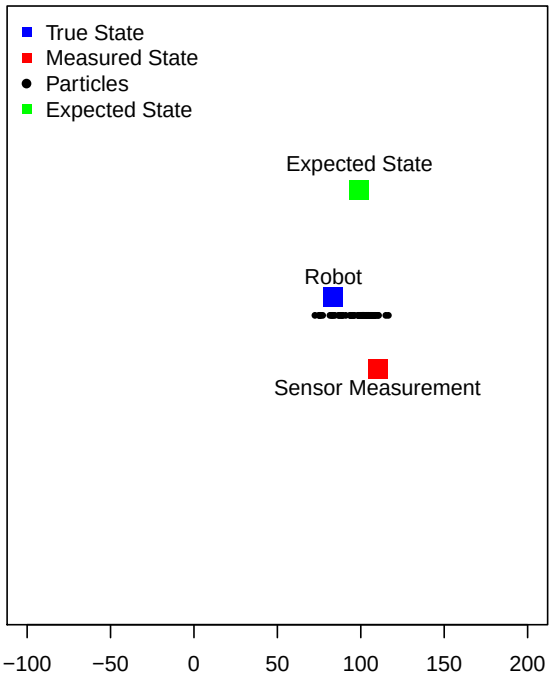
Standard Deviation: 50

Link To Video

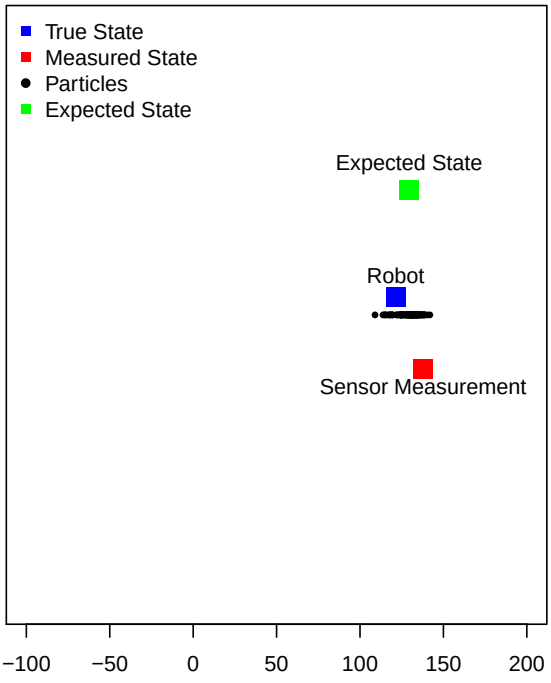
Time Step: 1



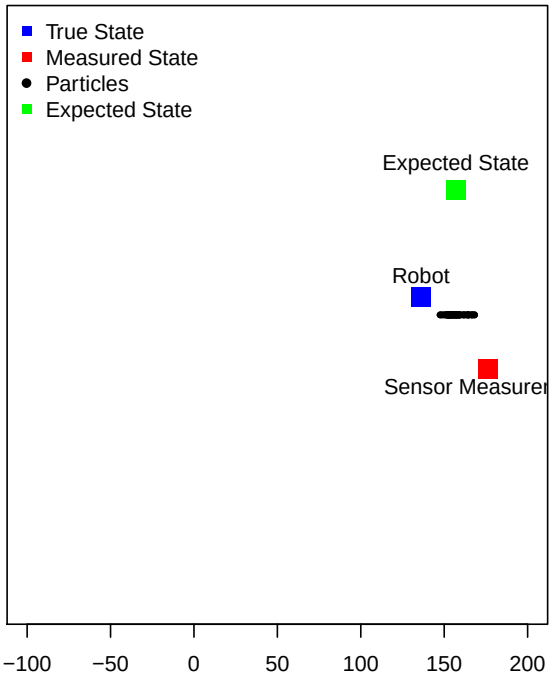
Time Step: 40



Time Step: 80



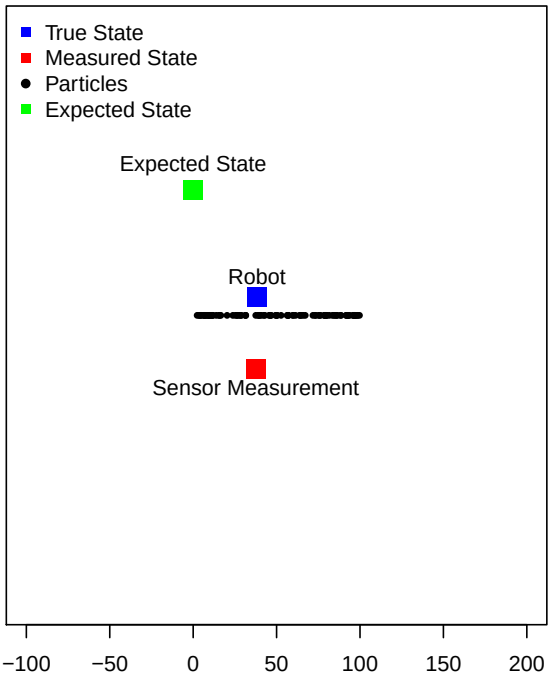
Time Step: 100



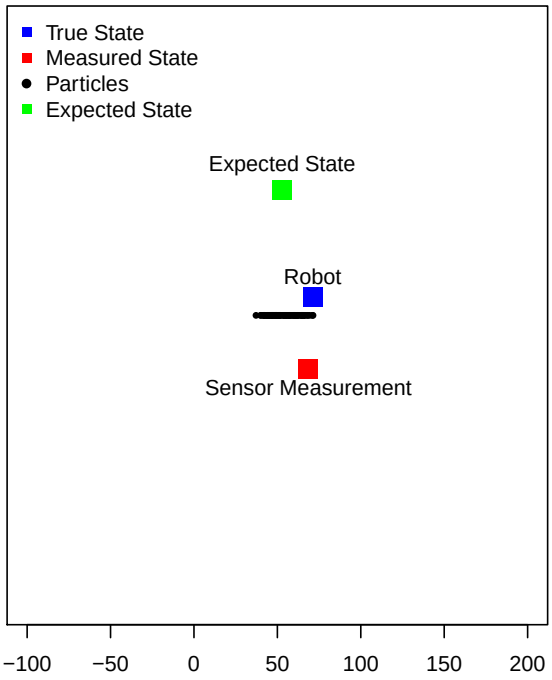
Equal Weight, Standard Deviation: 1

Link To Video

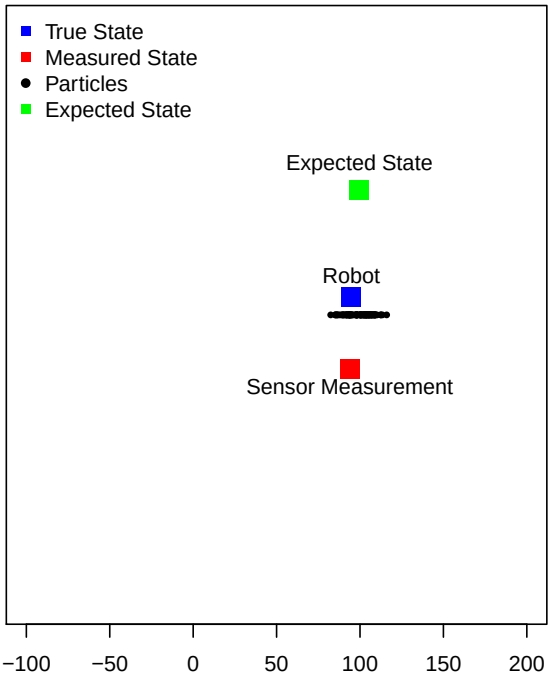
Time Step: 1



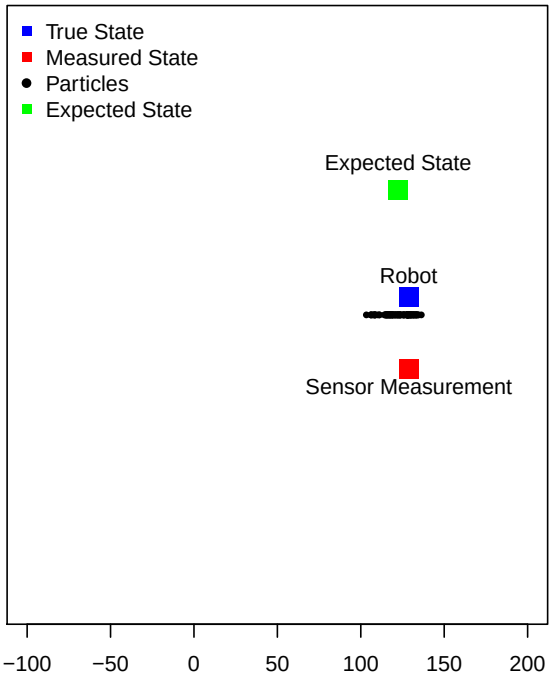
Time Step: 40



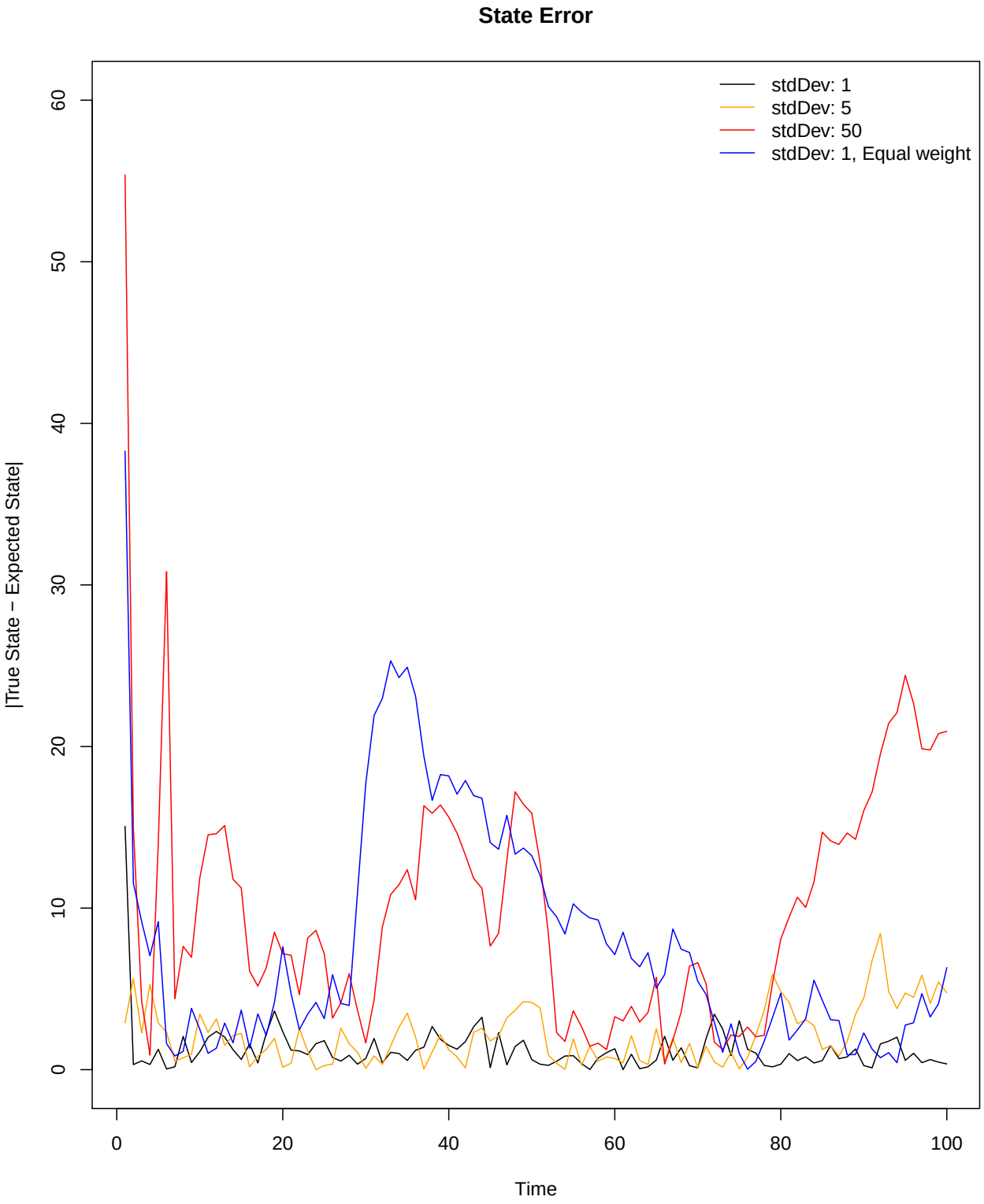
Time Step: 80



Time Step: 100



State Error


State Error
stdDev: 1
stdDev: 5
stdDev: 50
stdDev: 1, Equal weight
|True State – Expected State|
Time
0
10
20
30
40
50
60
0
20
40
60
80
100

Discussion

By increasing the standard deviation for the emission model we're implying that we're less certain about the sensor measurements. This, in turn, will affect the weight distribution of our particles (distant particles from the true state will be given a higher weight compared a to a simulation with a lower standard deviation), which affect the position of the expected state. The consequence of this can be viewed in the *state error* figure, where we can observe that the absolute difference between the true and expected state increases as we increase the standard deviation.

In the case when we're giving all particles equal weight the problem of absolute difference is similarly ill-behaved as in the case when the standard deviation is set to 50. Once again, the behavior can be linked to the fact that distant particles (from the true state) have a relatively high weight.

Appendix

```
knitr::opts_chunk$set(echo = FALSE,
                      warning = FALSE,
                      message = FALSE,
                      fig.height = 11,
                      fig.width = 9)

library(ggplot2)
library(dplyr)
library(grid)
library(gridExtra)
library(knitr)
library(png)
pathImg <- "../img"
pathData <- "../Data"
pathVid <- "/Users/arianbarakat/Dropbox/Public"



transModGeneric <- function(standDev){
  # Description:
  #     Function using closjure
  # Input:
  #     standDev: The Standard Deviation for the Normals
  # Returns:
  #     Model: Returns a function that calculates p(z_t+1 | z_t)
  #             linear combination of normals

  model <- function(z_t, z_tprev, sdDev = standDev){
    (dnorm(x =z_t , mean = z_tprev, sd = sdDev) +
      dnorm(x =z_t , mean = z_tprev + 1, sd = sdDev) +
      dnorm(x =z_t , mean = z_tprev + 2, sd = sdDev))/3
  }
  return(model)
}

emissModGeneric <- function(standDev){
  # Description:
  #     Function using closjure
  # Input:
  #     standDev: The Standard Deviation for the Normals
  # Returns:
  #     Model: Returns a function that calculates p(x_t | z_t),
  #             linear combination of normals

    model <- function(x_t, z_t, sdDev = standDev){
    (dnorm(x = x_t, mean = z_t, sd = sdDev) +
      dnorm(x = x_t, mean = z_t + 1, sd = sdDev) +
      dnorm(x = x_t, mean = z_t - 1, sd = sdDev))/3
  }
  return(model)
}
```

```r
calcWeight <- function(sample, state_t,  emissModel, case = c("normal", "equalWeight")){
  # Description:
  #     Function that calculates the weights of the particles
  #     in each time step
  # Input:
  #     sample:     x_t
  #     state_t:    z_t()
  #     emissModel: function that calculates p(x_t | z_t)
  #     case:       Normal particle filter algorithm or
  #                 special case when all weight are equal (w = 1)
  # Returns:
  #     w: Weights of the particles in each time step

  if(!is.character(case) && (case %in% c("normal", "equalWeight"))){
    stop("Supply case: 'normal' or 'equalWeight' in character form")
  }

  if(case == "normal"){
    emissVec <- do.call(emissModel, args = list(x_t = sample,
                                                z_t = state_t))

    w <- emissVec/sum(emissVec)
  }

  if(case == "equalWeight"){
    w <- rep(1, times = length(state_t))
    w <- w/sum(w)
  }

  return(w)
}

sampleObsStateGeneric <- function(standDev){
  # Description:
  #     Return a function (using closujure) that create a sample of x_t+1
  #     given z_t
  # Input:
  #     standDev: The Standard Deviation for the Normals
  # Returns:
  #     Model:

  model <- function(z_t, stDev = standDev, ...){
    mu <- c(z_t, z_t + 1, z_t - 1)

    comp <- sample(1:length(mu), size = 1, prob = rep(1/length(mu), times = 3))
    rnorm(n = 1, mean = mu[comp], sd = stDev)
  }

  return(model)
}

sampleHiddenStateGeneric <- function(standDev){
  # Description:
```

```
  #     Return a function (using closujure) that create a sample of z_t+1
  #     given z_t and weights. Function 13.119 in Bishop (p. 646)
  # Input:
  #     standDev: The Standard Deviation for the Normals
  # Returns:
  #     Model:

  model <- function(weights, statesPrev, stDev = standDev){

    if(length(weights) == 1){
      sampleMeans <- statesPrev
    } else {
      sampleMeans <- sample(statesPrev,
                            size = length(weights),
                            replace = TRUE,
                            prob = weights)

    }


    newStates <- vapply(sampleMeans, FUN = function(z_t, stdDev = stDev){

      mu <- c(z_t, z_t + 1, z_t + 2)
      comp <- sample(1:length(mu),
                     size = 1,
                     prob = rep(1/length(mu),
                                times = length(mu)))

      rnorm(n = 1, mean = mu[comp], sd = stDev)

    }, FUN.VALUE = numeric(1))

   return(newStates)
  }
  return(model)
}


initialModel <- function(minimum, maximum){
  model <- function(nSample, min = minimum, max = maximum){
    runif(n = nSample, min = min, max = max)
  }
  return(model)
}


particleFilter <- function(nStep, nParticles, standardDev, initMod, weightCase = "normal"){

  # Setup
  prEmiss <- emissModGeneric(standDev = standardDev)
  prTrans <- transModGeneric(standDev = 1)
  sampleXt <- sampleObsStateGeneric(standDev = standardDev)
  sampleZt <- sampleHiddenStateGeneric(standDev = 1)
```

```
  zParticles_tm <- matrix(nrow = nStep,
                ncol = nParticles)

  trueState <- vector("numeric", length = nStep)
  sensorRobot <- vector("numeric", length = nStep)
  expectedState <- vector("numeric", length = nStep)

  # Generate Data
  trueState[1] <- do.call(initMod, args = list(nSample = 1))
  sensorRobot[1] <- sampleXt(trueState[1])

  for(step in 2:nStep){
    trueState[step] <- sampleZt(weights = c(1),
                                statesPrev = trueState[step -1])
    sensorRobot[step] <- sampleXt(trueState[step])

  }



  # Init
  zParticles_tm[1,] <- z_t <- do.call(initMod, args = list(n = nParticles))

  for(step in 2:nStep){

    x_t <- sensorRobot[step]

    w_t <- calcWeight(sample = x_t,
                      state_t = z_t,
                      emissModel = prEmiss,
                      case = weightCase)

    zParticles_tm[step,] <- z_t <- sampleZt(weights = w_t,
                                            statesPrev = zParticles_tm[step -1,])
    expectedState[step] <- sum(w_t*z_t)
  }

  return(list("trueStates" = trueState,
              "sensorMeasurment" = sensorRobot,
              "statesParticles" = zParticles_tm,
              "exptectedState" = expectedState))
}



plotSSMsim <- function(object, plotToReport = FALSE,
                       plotToVideo = FALSE,
                       videoArgs = list("path" = NULL, "name" = NULL) ){
  suppressMessages(suppressWarnings(require(animation)))

  nIter <- length(object[["trueStates"]])

  if(plotToReport){
```

```
    stepLength <- c(1, round(c(nIter*0.4, nIter*0.8)),nIter)
  } else {
    stepLength <- seq(1, nIter, by = 1)
  }

  plotOut <- function(plotObject, plotSequence){
    for(step in plotSequence){

      plot(plotObject$statesParticles[step,],
           y = rep(2, 100),
           xlim = c(-100, 200), xlab ="", ylab = "",
           pch = 16, cex = .7, yaxt = "n",  main = paste("Time Step:", step))
      points(x = plotObject$trueStates[step], y = 2.05, pch = 15, col = "blue", cex = 2)
      text(x = plotObject$trueStates[step], y = 2.11, labels = "Robot", cex = 1)
      points(x = plotObject$sensorMeasurment[step], y = 1.85, pch = 15, col = "red", cex = 2)
      text(x = plotObject$sensorMeasurment[step], y = 1.80, labels = "Sensor Measurement", cex = 1)
      points(x = plotObject$exptectedState[step], y = 2.35, pch = 15, col = "green", cex = 2)
      text(x = plotObject$exptectedState[step], y = 2.42, labels = "Expected State", cex = 1)
      legend("topleft",
             legend = c("True State",
                        "Measured State",
                        "Particles",
                        "Expected State"),
             pch = c(15,15,16, 15),
             col = c("blue", "red", "black", "green"),
             bty = "n")
      Sys.sleep(0.1)
    }
  }


  if(plotToReport){
    par(mfrow = c(2,2))
    plotOut(plotObject = object, plotSequence = stepLength)
    par(mfrow = c(1,1))
  }

  if(plotToVideo){
    saveVideo({
      plotOut(plotObject = object, plotSequence = stepLength)
    },
    interval = 0.1,
    video.name = paste(videoArgs[[1]], videoArgs[[2]], sep = "/"),
    ani.width=800,
    ani.height=800)
  }

  if(!plotToReport){
    plotOut(plotObject = object, plotSequence = stepLength)
  }

}
```

```r
plotError <- function(object, add = FALSE, lineCol = NULL){
  steps <- length(object[["trueStates"]])
  if(!add){
    plot(x = 1:steps,
         y = abs(object$trueStates - object$exptectedState),
         type = "l",
         xlab = "Time",
         ylab = "|True State - Expected State|",
         main = "State Error",
         ylim = c(0,60))
  }

  if(add){
    if(is.null(lineCol)){
      stop("Provide line colour")
    }
    lines(x = 1:steps,
         y = abs(object$trueStates - object$exptectedState),
         col = lineCol)
  }

}



######## Running the Code


set.seed(1991)
initModd <- initialModel(minimum = 0, maximum = 100)
simSSMsd1 <- particleFilter(nStep = 100,
                            nParticles = 100,
                            standardDev = 1,
                            initMod = initModd)

simSSMsd5 <- particleFilter(nStep = 100,
                            nParticles = 100,
                            standardDev = 5,
                            initMod = initModd)

simSSMsd50 <- particleFilter(nStep = 100,
                            nParticles = 100,
                            standardDev = 50,
                            initMod = initModd)

simSSMsd1_equalWeight <- particleFilter(nStep = 100,
                                        nParticles = 100,
                                        standardDev = 1,
                                        initMod = initModd,
                                        weightCase =  "equalWeight")


# plotSSMsim(simSSMsd1, plotToVideo = TRUE, videoArgs = list(pathVid, "simSSMsd1.mp4"))
```

```r
plotSSMsim(simSSMsd1, plotToReport = TRUE)
```

```r
# plotSSMsim(simSSMsd5, plotToVideo = TRUE, videoArgs = list(pathVid, "simSSMsd5.mp4"))
plotSSMsim(simSSMsd5, plotToReport = TRUE)
```

```r
# plotSSMsim(simSSMsd50, plotToVideo = TRUE, videoArgs = list(pathVid, "simSSMsd50.mp4"))
plotSSMsim(simSSMsd50, plotToReport = TRUE)
```

```r
# plotSSMsim(simSSMsd1_equalWeight, plotToVideo = TRUE, videoArgs = list(pathVid, "simSSMsd1_equalWeigh
plotSSMsim(simSSMsd1_equalWeight, plotToReport = TRUE)
```

```r
plotError(simSSMsd1)
plotError(simSSMsd5, add = TRUE, lineCol = "orange")
plotError(simSSMsd50, add = TRUE, lineCol = "red")
plotError(simSSMsd1_equalWeight, lineCol = "blue", add = TRUE)
legend("topright",
       legend = c("stdDev: 1",
                  "stdDev: 5",
                  "stdDev: 50",
                  "stdDev: 1, Equal weight"),
       lty = rep(1, times = 4),
       col = c("black", "orange", "red", "blue"),
       bty = "n")
```

```r
# Own Implementations

# InitSSM
initSSM <- function(nIter, rTransModel, rEmissModel, initModel){

  z <- vector("numeric", length = nIter)
  x <- vector("numeric", length = nIter)

  if(is.function(initModel)){
    z[1] <- do.call(initModel, args = list(n = 1))
  }

  if(is.numeric(initModel)){
    z[1] <- initModel
  }

  for(step in 2:nIter){
    z[step] <- do.call(rTransModel,
                       args = list(1,
```

```
                                    zPrevious = z[step -1 ]))

    x[step] <- do.call(rEmissModel,
                       args = list(z[step]))
  }

  return(list(
    "hiddenState" = z,
    "observedState" = x
  ))
}

# Kalman

predictMu <- function(A, previousMu){
  return(A%*%previousMu)
}

predictSigma <- function(A, previousSigma, R){
  return(A%*%previousSigma%*%t(A) + R)
}

calcKalmanGain <- function(sigmaBar, C, Q){
  return(sigmaBar%*%t(C)%*%solve(C%*%sigmaBar%*%t(C) + Q))
}

correctionMu <- function(muBar, kalmanGain, x_t, C){
  return(muBar + kalmanGain%*%(x_t - C%*%muBar))
}

correctionSigma <- function(sigmaBar, kalmanGain, C){
  return( (diag(1, nrow=dim(kalmanGain)[1], ncol=dim(C)[2]) -
             kalmanGain%*%C)%*%sigmaBar)
}

kalmanFilter <- function(observed, A, C, R, Q, initVal = list()){
  nStep <- length(observed)
  mu_t <- list()
  sigma_t <- list()

  mu_t[[1]] <- initVal[[1]]
  sigma_t[[1]] <- initVal[[2]]

  for(t in 2:(nStep+1)){

    # Prediction
    muPred_t <- predictMu(A = A, previousMu = mu_t[[t-1]])
    sigmaPred_t <- predictSigma(A = A, previousSigma = sigma_t[[t-1]], R = R)

    # Kalman Gain
    kalmanGain_t <- calcKalmanGain(sigmaBar = sigmaPred_t, C = C, Q = Q)
```

```
    # Correction
    mu_t[[t]] <- correctionMu(muBar = muPred_t, kalmanGain = kalmanGain_t,
                              x_t = observed[t], C = C)

    sigma_t[[t]] <- correctionSigma(sigmaBar = sigmaPred_t,
                                    kalmanGain = kalmanGain_t,
                                    C = C)

  }

  return(list("mu_t" = mu_t,
              "sigma_t" = sigma_t))
}



# Particle

calcWeight <- function(observed, z_t,  dEmissModel, case = "normal"){
  # Description:
  #     Function that calculates the weights of the particles
  #     in each time step
  # Input:
  #     observed:     x_t
  #     z_t:          z_t()
  #     dEmissModel:   function that calculates p(x_t | z_t)
  #     case:         Normal particle filter algorithm or
  #                   special case when all weight are equal (w = 1)
  #                   possible inputs: c("normal", "equalWeight")
  # Returns:
  #     w: Weights of the particles in each time step

  if(!is.character(case) && (case %in% c("normal", "equalWeight"))){
    stop("Supply case: 'normal' or 'equalWeight' in character form")
  }

  if(case == "normal"){
    emissVec <- do.call(dEmissModel, args = list(x_t = observed,
                                                 z_t = z_t))

    w <- emissVec/sum(emissVec)
  }

  if(case == "equalWeight"){
    w <- rep(1, times = length(z_t))
    w <- w/sum(w)
  }

  return(w)
}


particleFilter <- function(observations, nParticles, rTransModel, dEmissModel, initModel, ...){
```

```
  # Setup
  nStep <- length(observations)
  zParticles_tm <- matrix(nrow = nStep,
                          ncol = nParticles)

  expectedState <- vector("numeric", length = nStep)

  # Init
  zParticles_tm[1,] <- z_t <- do.call(initModel, args = list(n = nParticles))

  for(step in 2:nStep){

    x_t <- observations[step]

    w_t <- calcWeight(observed = x_t,
                      z_t = z_t,
                      dEmissModel = dEmissModel,
                      ...)

    zParticles_tm[step,] <- z_t <- do.call(rTransModel,
                                           args = list(weights = w_t,
                                                       zParticles_tm[step -1,]))

    expectedState[step] <- sum(w_t*z_t)
  }

  return(list("statesParticles" = zParticles_tm,
              "exptectedState" = expectedState))
}
```