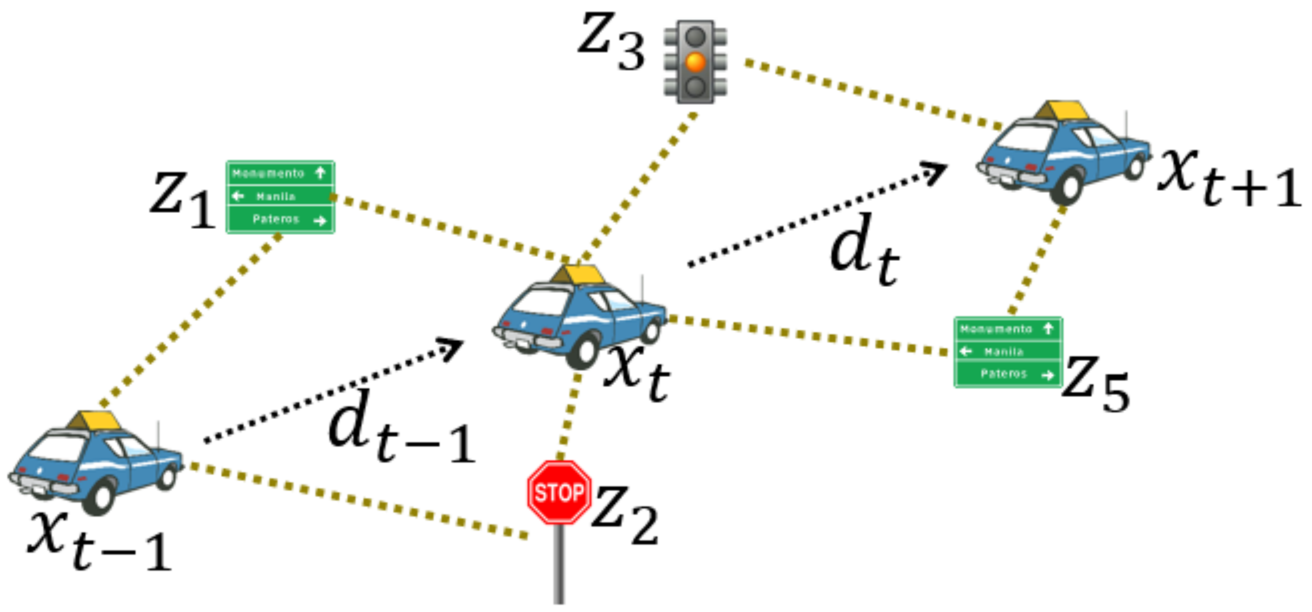
A simple Localization Problem



Given

1. Map: Global position of landmarks as z_i
2. Motion model of the robot, e.g. $x_{t+1} = x_t + d_t + \epsilon_t$, $\epsilon_t \sim \mathcal{N}(0, R_t)$.
3. Observations model as the bearing of the landmarks $f_{i,t} = H(z_i - x_t) + \eta_{i,t}$, $\eta_{i,t} \sim \mathcal{N}(0, Q_t)$.
4. Local observations at each time step for some of the landmarks $f_{i,t}$.
5. Initial Location $x_0 \sim \mathcal{N}(\mu_0, \Sigma_0)$.

Find

The position and orientation of the robot, x_t

```
!pip install -q requests scipy ipywidgets svgpath2mpl
from scipy.stats import multivariate_normal
import requests
from io import BytesIO
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
from IPython.display import display

from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
```

```
from svgpath2mpl import parse_path

# 🚗
mplcar = parse_path('M 9880,3580 C 8932,3481 8224,2669 8263,1725 8305,731 9137,-38
mplcar.vertices -= mplcar.vertices.mean(axis=0)
# 🚦
mpltraffic = parse_path('M384 192h-64v-37.88c37.2-13.22 64-48.38 64-90.12h-64V32c0-
mpltraffic.vertices -= mpltraffic.vertices.mean(axis=0)
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 11.8 MB/s eta 0:00:0
```

Input data

```
#@title Input data
url = 'http://www-personal.umich.edu/~dh
response = requests.get( url)
data = np.load(BytesIO(response.content)
f_i_t = [data['obs_vals_%d' % k] for k i
visible_all_t = [data['obs_ids_%d' % k]
d = data['d']
z = data['z']
#@markdown ### Scale Model noise by a sc
scale_q = 1 #@param {type: "slider", min
Q = data['Q'] * scale_q

#@markdown ### Scale Observation noise b
scale_r = 2 #@param {type: "slider", min
R = data['R'] * scale_r
H = data['H']
μ0 = data['mu0']
Σ0 = data['sigma0']
```

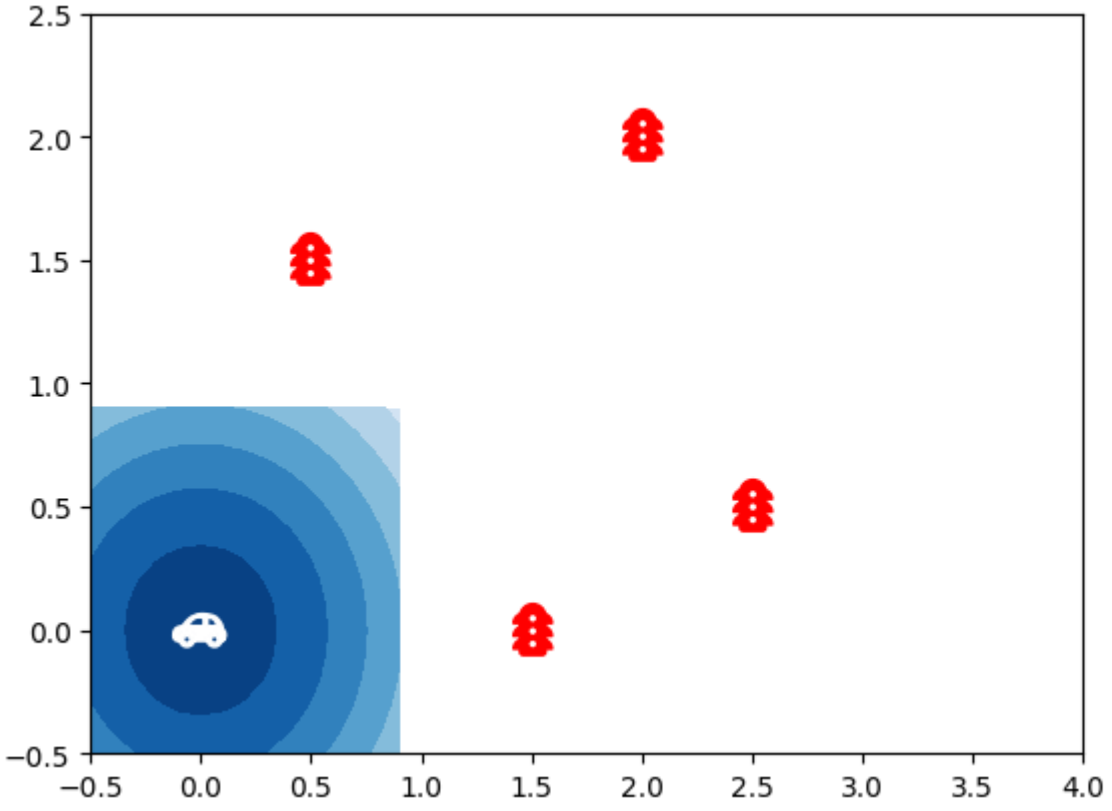
Scale Model noise by a scalar value scale_q

scale_q: 1

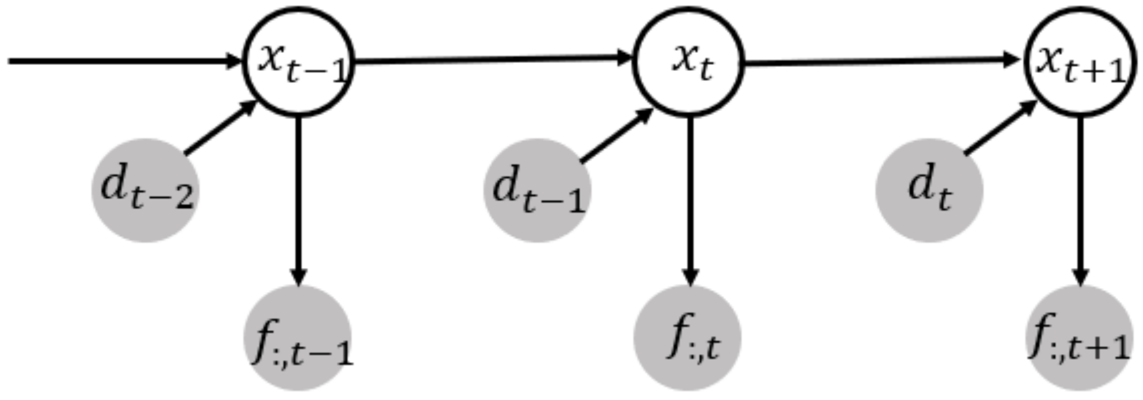
Scale Observation noise by a scalar value scale_r

scale_r: 2

```
def plot_truncated_gaussian(mean, cov, s=1,n=10):
  xsigma = np.sqrt(cov[0, 0])
  ysigma = np.sqrt(cov[1, 1])
  xslice = slice(mean[0]-s*xsigma, mean[0]+s*xsigma, xsigma/n)
  yslice = slice(mean[0]-s*ysigma, mean[0]+s*ysigma, ysigma/n)
  x, y = np.mgrid[xslice, yslice]
  pos = np.empty(x.shape + (2,))
  pos[:, :, 0] = x; pos[:, :, 1] = y
  rv = multivariate_normal(mean, cov)
  plt.contourf(x, y, rv.pdf(pos), cmap='Blues')
  plt.plot(mean[0], mean[1], color='w', marker=mplcar, markersize=20, linestyle='

def plot_particles(w_particles, x_particles):
  plt.scatter(x_particles[:, 0], x_particles[:, 1], c=w_particles.flatten(), cmap
  mean = (w_particles * x_particles).sum(axis=0)
  plt.plot(mean[0], mean[1], color='y', marker=mplcar, markersize=20, linestyle='
def plot_landmarks(z):
  # Let's plot our global markers
  plt.plot(z[:, 0], z[:, 1], color='r', marker=mpltraffic, markersize=20, linesty
  plt.xlim(-0.5, 4.0)
  plt.ylim(-0.5, 2.5)

plot_landmarks(z)
plot_truncated_gaussian(μ0, Σ0)
```



Bayes filter



[1]

For our problem, $u_t = d_t$.

Let all the observation upto time t be represented by $f_{1:t}$ and all the odometry inputs be denoted by $d_{1:t-1}$. The localization problem is the estimation of the probability $p(x_t|f_{1:t}, d_{1:t-1})$. We assume the process generating the data to be a Markovian process with state x_t. This means that given x_{t-1}, future states $x_t, x_{t+1}, \ldots$ are independent of all past states $x_{t-2}, x_{t-3}, \ldots$. This allows us to write the localization problem as a recursive equation

$$p(x_t|f_{1:t}, d_{1:t-1}) = \int_{\mathcal{X}_{t-1}} p(x_t|f_t, d_{t-1}, x_{t-1}) p(x_{t-1}|f_{1:t-1}, d_{1:t-2}) dx_{t-1}$$

Note that the second term inside the integral is the solution of previous localization problem. The first term is broken down into two steps, (1) the motion update or prediction step (2) measurement update step. Due to Bayes rule, we can write

$$p(x_t|f_t, d_{t-1}, x_{t-1}) = \frac{p(f_t, d_{t-1}|x_t, x_{t-1}) p(x_t|x_{t-1})}{p(f_t, d_{t-1}|x_{t-1})}$$

Using the independence of $f_t \perp d_{t-1}|x_t$ and $f_t \perp x_{t-1}|x_t$, and $p(f_t, d_{t-1}|x_t, x_{t-1}) = p(f_t|x_t)p(d_{t-1}|x_t, x_{t-1})$. Overall with some rearrangement, we get

$$p(x_t|f_t, d_{t-1}, x_{t-1}) = \frac{p(f_t|x_t) p(x_t|d_{t-1}, x_{t-1})}{p(f_t|x_{t-1})}$$

This is convinient since we know the measurement model $p(f_t|x_t)$ and the motion model $p(x_t|d_{t-1}, x_{t-1})$. We assume that all observations are equally likely and hence only act as a normalizing factor:

$$p(x_t|f_t, d_{t-1}, x_{t-1}) \propto p(f_t|x_t) p(x_t|d_{t-1}, x_{t-1})$$

Putting this back in the Bayes recursive formulation, we get

$$p(x_t|f_{1:t}, d_{1:t-1}) \propto p(f_t|x_t) \int_{\mathcal{X}_{t-1}} p(x_t|d_{t-1}, x_{t-1}) p(x_{t-1}|f_{1:t-1}, d_{1:t-2}) dx_{t-1}$$

Motion update or prediction step

Computing the term $\bar{\text{bel}}(x_{t-1}) = \int_{\mathcal{X}_{t-1}} p(x_t|d_{t-1}, x_{t-1}) p(x_{t-1}|f_{1:t-1}, d_{1:t-2}) dx_{t-1}$

Measurement update step

Computing the term $p(x_t|f_{1:t}, d_{1:t-1}) \propto p(f_t|x_t)\bar{\text{bel}}(x_{t-1})$

The bayes filter can be implemented in many ways, depending upon how the probability distribution is represented:

1. Particle filter

2. Histogram fitler
3. Kalman filter

Particle filter or Sequential Importance resampling

A probability distribution can be computationally represented as samples and weights.

Given a bunch of samples $\{x_i\}_{i=1}^N$ samples sampled from a *proposal distribution* $q(x) > 0$, if we want to compute the expected value of a function $g(x)$ over target distribution $p(x)$ 2

$$\mathbb{E}_{p(x)}[g(x)] = \sum_{i=1}^{N} \frac{g(x)p(x)}{q(x)}$$

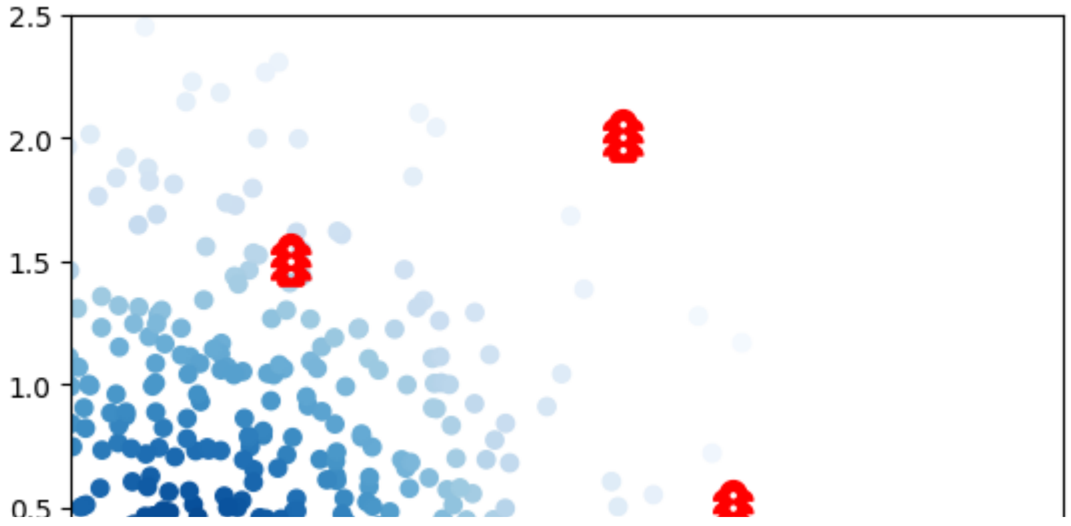
Representing $p(x_0)$ as particles

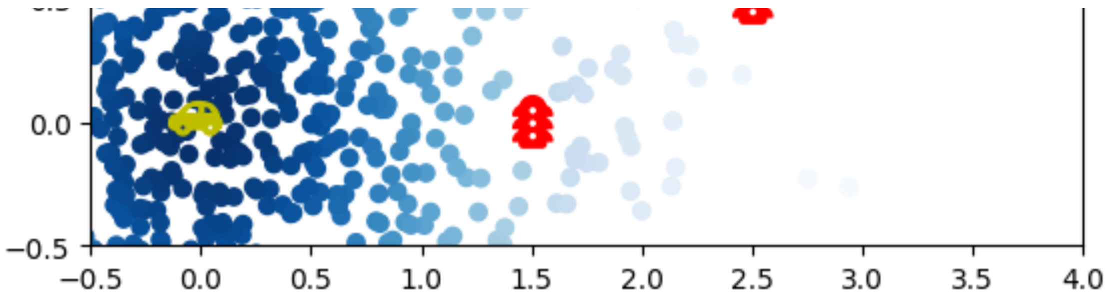
First step is to convert $x_0 \sim \mathcal{N}(\mu_0, \Sigma_0)$ as particles, $(w_t^{(j)}, x_t^{(j)}) : w_t^{(j)} \approx p(x_t^{(j)} | f_{1:t}, d_{1:t-1})$.

```
#@markdown Enter number of particles as
logM = 3 #@param {type: "slider", min: 1
M = 10**logM
def initialize_particles(M, μ0, Σ0):
    x0_particles = np.random.multivariat
    w_particles_unnormalized = multivari
    w0_particles = w_particles_unnormali
    return w0_particles, x0_particles

w0_particles, x0_particles = initialize_
plot_landmarks(z)
plot_particles(w0_particles, x0_particle
```

Enter number of particles as 10^{logM}

logM: 3

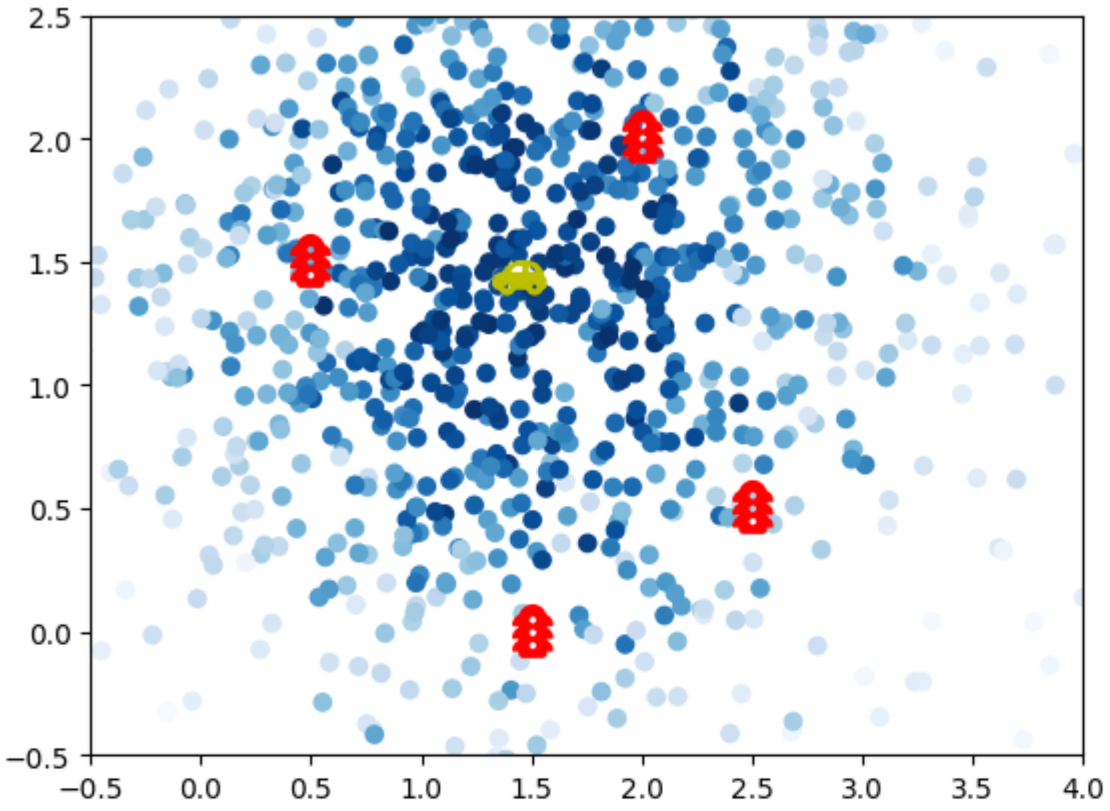




Particle filter motion update or prediction step

$$x_{t+1} \sim \mathcal{N}(x_t + d_t, R_t)$$

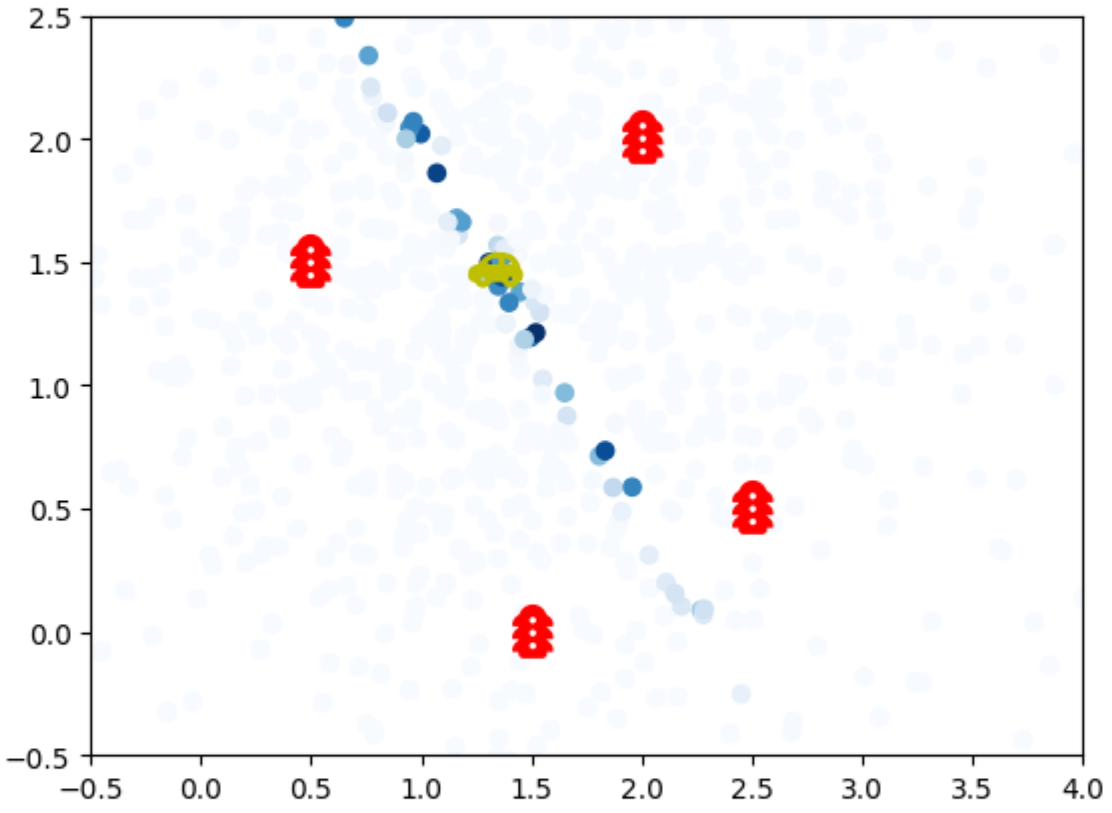
```
mean_0 = (w0_particles * x0_particles).sum(axis=0)
def motion_update(w0_particles, x0_particles, d_t):
    x1_particles = np.empty_like(x0_particles)
    for i, x0 in enumerate(x0_particles):
      x1_particles[i, :] = np.random.multivariate_normal(x0 + d_t, R)
    return w0_particles, x1_particles

w1_particles_motion, x1_particles_motion = motion_update(w0_particles, x0_particl
plot_landmarks(z)
plot_particles(w1_particles_motion, x1_particles_motion)
```



Particle filter measurement update

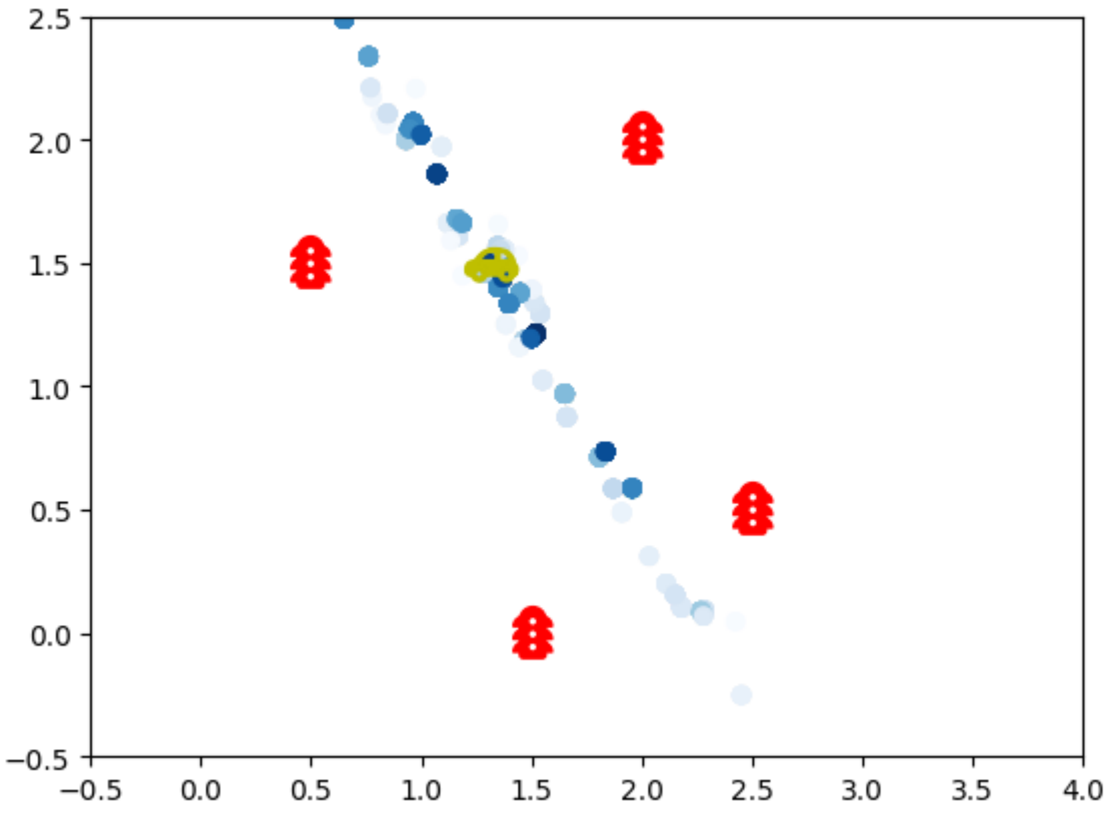
$$f_{i,t} | x_t \sim \mathcal{N}(H(z_i - x_t), Q)$$
$$p(f_t | x_t) = \prod_i p(f_t | x_t)$$
$$w_{t+1}^{(j)} \propto w_t^{(j)} p(f_t | x_t)$$

```
def measurement_update(w1_particles_motion, x1_particles_motion, visible_t, ft):
    diff_i_t = z[visible_t, np.newaxis, :] - x1_particles_motion[np.newaxis, :, :
    VMD = diff_i_t.shape
    obs_mean = (diff_i_t.reshape(-1, VMD[-1]) @ H.T).reshape(*VMD[:2],-1)
    p_f_t = 1
    for i, fit in enumerate(ft):
        p_f_t = p_f_t * multivariate_normal(fit, Q).pdf(obs_mean[i])
        assert p_f_t.shape[0] == VMD[1]
    w1_particles_unnormalized = w0_particles * p_f_t.reshape(-1,1)
    w1_particles = w1_particles_unnormalized / w1_particles_unnormalized.sum()
    return w1_particles, x1_particles_motion

w1_particles, x1_particles = measurement_update(w1_particles_motion, x1_particles
plot_landmarks(z)
plot_particles(w1_particles, x1_particles)
```



Particle resampling step

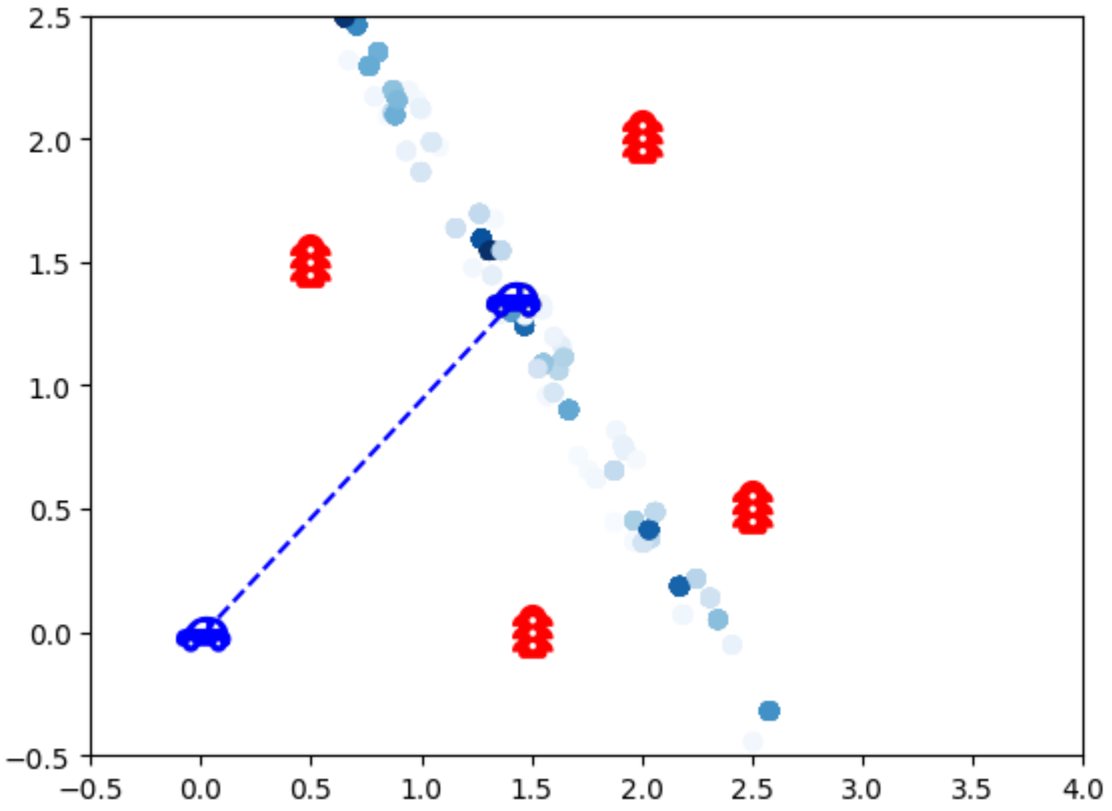
```
def resample_particles(w1_particles, x1_particles):
    effective_number = 1 / (w1_particles**2).sum()
    if effective_number > w1_particles.shape[0]/2:
      return w1_particles, x1_particles
    x1_indices = np.random.choice(w1_particles.shape[0], w1_particles.shape[0], p
    x1_particles_resampled = x1_particles[x1_indices]
    w1_particles_resampled = w1_particles[x1_indices] / w1_particles[x1_indices].
    return w1_particles_resampled, x1_particles_resampled

w1_particles_resampled, x1_particles_resampled = resample_particles(w1_particles,
plot_landmarks(z)
plot_particles(w1_particles_resampled, x1_particles_resampled)
```



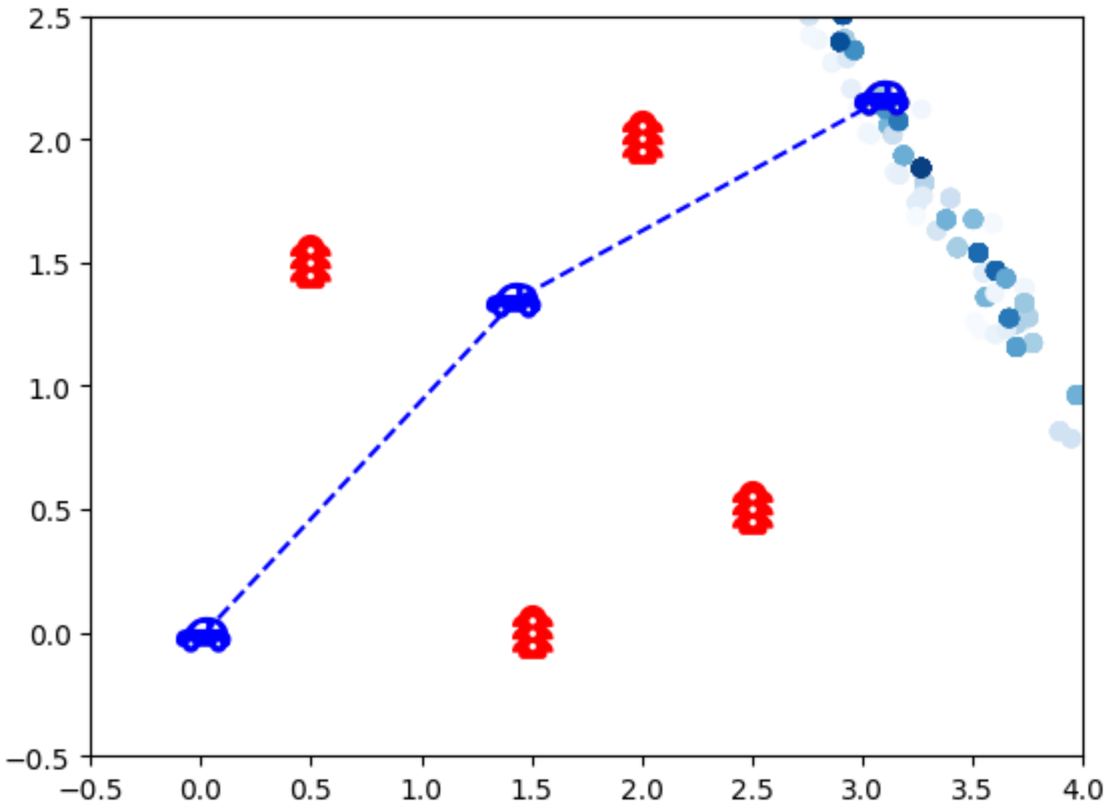
Full particle particle filter algorithm

```
wt_particles, xt_particles = initialize_particles(M, μ0, Σ0)
means = [(wt_particles * xt_particles).sum(axis=0)]
for t in range(d.shape[0]):
  w1m, x1m = motion_update(wt_particles, xt_particles, d[t])
  w1, x1 = measurement_update(w1m, x1m, visible_all_t[t+1], f_i_t[t+1])
  wt_particles, xt_particles = resample_particles(w1, x1)
  plot_landmarks(z)
  plot_particles(wt_particles, xt_particles)
  means.append((wt_particles * xt_particles).sum(axis=0))
  mean_np = np.array(means)
```

```
plt.plot(mean_np[:, 0], mean_np[:, 1], markersize=20, marker=mplcar, linestyle=
display(plt.show())
```



```
None
```



```
None
```

Kalman Filter

Kalman filter assumes Gaussian distributions and linear measurement and motion models.

Assume motion model

$$x_{t+1} = A_t x_t + B_t d_t + \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, R_t)$$

and measurement model

$$f_t = H_t x_t + \eta_t, \quad \eta_t \sim \mathcal{N}(0, Q_t)$$

Kalman filter model for the simple Localization problem

The measurement model in simple localization problem is linear when written as $f_{i,t} = H(z_i - x_t) + \eta_t$. It is affine. Moreover, it needs to be stacked to compute f_t for all i in a single matrix multiplication. Let the new state be defined as $\bar{x}_t = [x_t^\top, 1]^\top$. Then the motion model becomes,

$$\bar{x}_{t+1} = \bar{x}_t + \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} d_t + \bar{\epsilon}_t, \quad \bar{\epsilon}_t \sim \mathcal{N}(0, \bar{R}_t)$$

where

$$\bar{R}_t = \begin{bmatrix} R_t & 0 \\ 0 & \delta \end{bmatrix}$$

. $\delta = 10^{-6}$.

The measurement model becomes,

$$\begin{bmatrix} f_{1,t} \\ f_{2,t} \\ \vdots \\ f_{N,t} \end{bmatrix} = \underbrace{\begin{bmatrix} -H & Hz_1 \\ -H & Hz_2 \\ \vdots & \vdots \\ -H & Hz_N \end{bmatrix}}_{\bar{H}_t} \bar{x}_t + \underbrace{\begin{bmatrix} \eta_1 \\ \eta_2 \\ \vdots \\ \eta_N \end{bmatrix}}_{\bar{\eta}}$$

,

$$\bar{\eta} \sim \mathcal{N}\left(0, \begin{bmatrix} Q & 0 & \dots & 0 \\ 0 & Q & \dots & 0 \\ \vdots & \vdots & \dots & \vdots \\ 0 & 0 & \dots & Q \end{bmatrix}\right)$$

Hence, the Kalman filter constants for the simple localization problem are:

$$\begin{bmatrix} -H & Hz_1 \end{bmatrix} \qquad \begin{bmatrix} Q & 0 & \dots & 0 \end{bmatrix}$$

$$\bar{A}_t = I_{3\times3}, \bar{B}_t = I_{3\times2}, \bar{H}_t = \begin{bmatrix} -H & Hz_2 \\ \vdots & \vdots \\ -H & Hz_N \end{bmatrix}, \bar{Q}_t = \begin{bmatrix} 0 & Q & \dots & 0 \\ \vdots & \vdots & \dots & \vdots \\ 0 & 0 & \dots & Q \end{bmatrix}, \bar{R}_t = \begin{bmatrix} R & 0 \\ 0 & \delta \end{bmatrix}$$

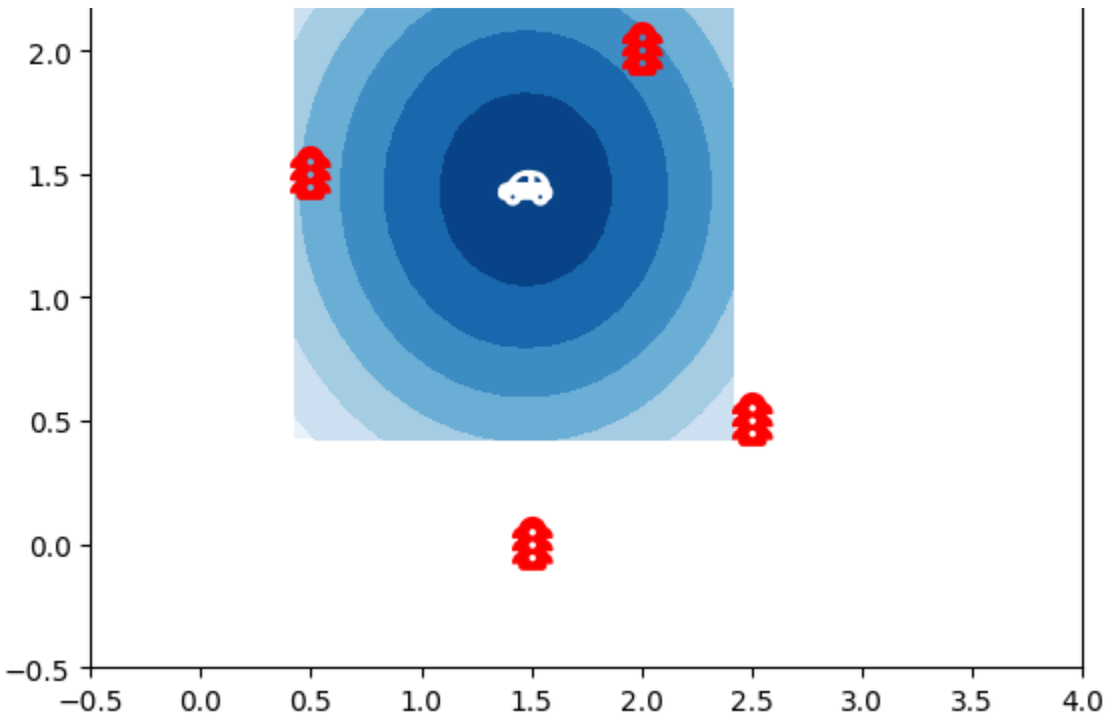
```
mu_0 = np.hstack((μ0, 1))
sigma_0 = np.eye(mu_0.shape[0])
sigma_0[:-1, :-1] = Σ0

D = μ0.shape[0]
Abart = np.eye(D+1)
Bbart = np.zeros((D+1,D))
Bbart[:D,:D] = np.eye(D)

def Hbart_create(H, z, visible_t):
  Hstacked = np.repeat(H, len(visible_t), axis=0)
  Hzis = []
  for i, lid in enumerate(visible_t):
    Hzis.append(H @ z[lid])
  return np.hstack((-Hstacked, np.vstack(Hzis)))

Rbart = np.eye(D+1) * 1e-6
Rbart[:D, :D] = R

def Qbart_create(Q, visible_t):
  from scipy.linalg import block_diag
  return block_diag(*([Q]*len(visible_t)))
```

Motion or prediction step

Thrun et al. 2005 marginal of a gaussian marginal of a gaussian without schur complement kalman filter using marginal

$$\bar{\mu}_{t+1} = A\mu_t + Bd_t$$
$$\bar{\Sigma}_{t+1} = A_t\Sigma_t A_t^\top + R_t$$

```
def motion_update_kf(mu_t, sigma_t, d_t, At, Bt, Rt):
  mu_tp1 = At @ mu_t + Bt @ d_t
  sigma_tp1 = At @ sigma_t @ At.T + Rt
  return mu_tp1, sigma_tp1

mu_1_motion, sigma_1_motion = motion_update_kf(mu_0, sigma_0, d[0], Abart, Bbart,
plot_truncated_gaussian(mu_1_motion[:D], sigma_1_motion[:D, :D])
plot_landmarks(z)
```

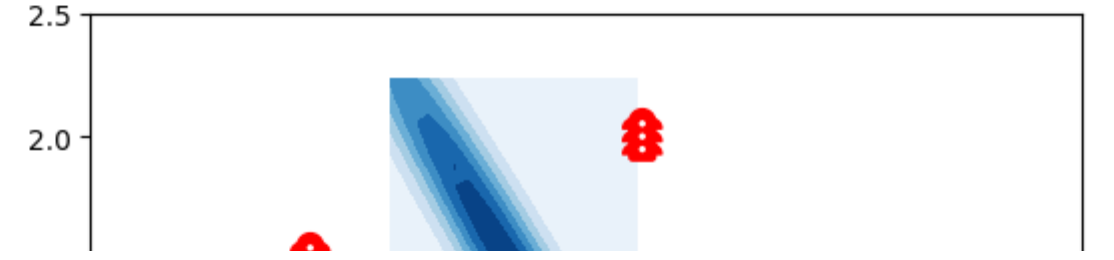


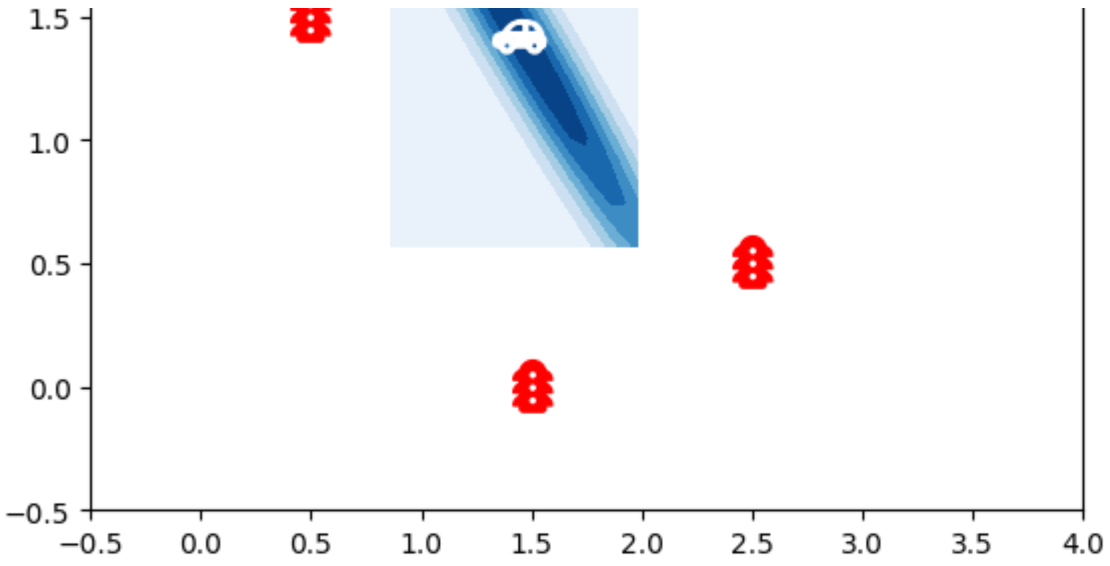
Measurement update step

Thrun et al. 2005

$$
\begin{aligned}
K_{t+1} &= \bar{\Sigma}_{t+1} H_t^\top (H_t \bar{\Sigma}_{t+1} H_t^\top + Q_t)^{-1} \\
\mu_{t+1} &= \bar{\mu}_{t+1} + K_{t+1}(f_t - H_t \mu_{t+1}) \\
\Sigma_{t+1} &= (I - K_{t+1} H_t) \bar{\Sigma}_{t+1}
\end{aligned}
$$

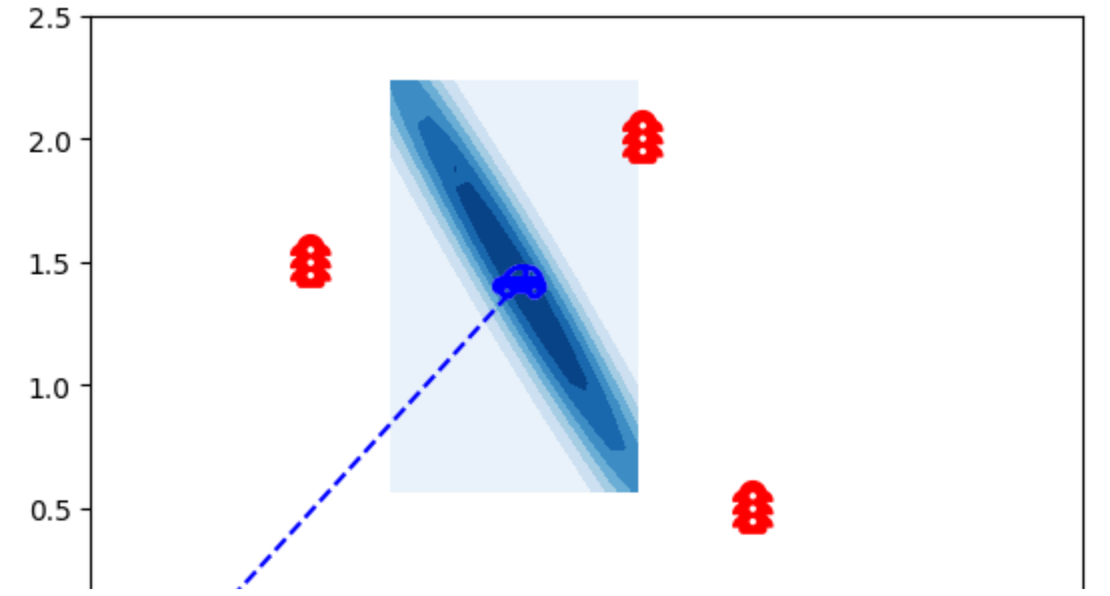
```
def measurement_update_kf(mu_t, sigma_t, f_is, Hbart, Qbart):
  diff = f_is - Hbart @ mu_t
  HSigmaH_minus_Q = Hbart @ sigma_t @ Hbart.T + Qbart
  mu_tp1 = mu_t + sigma_t @ Hbart.T @ np.linalg.solve(HSigmaH_minus_Q, diff)
  I = np.eye(mu_t.shape[0])
  sigma_tp1 = (I - sigma_t @ Hbart.T @ np.linalg.solve(HSigmaH_minus_Q, Hbart)) @
  return mu_tp1, sigma_tp1

f_is = f_i_t[1].flatten()
mu_1, sigma_1 = measurement_update_kf(mu_1_motion, sigma_1_motion, f_is, Hbart_cr

plot_truncated_gaussian(mu_1[:D], sigma_1[:D, :D])
plot_landmarks(z)
```

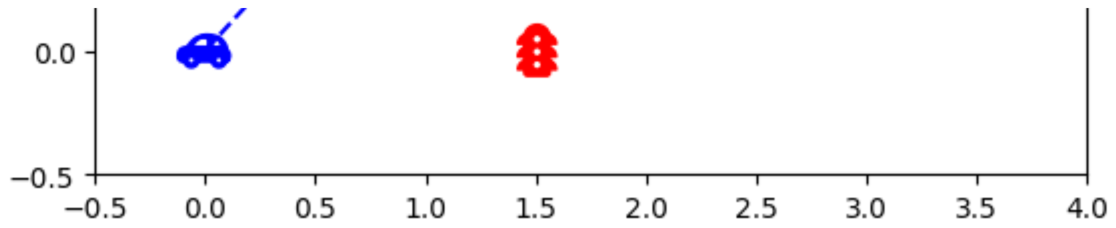




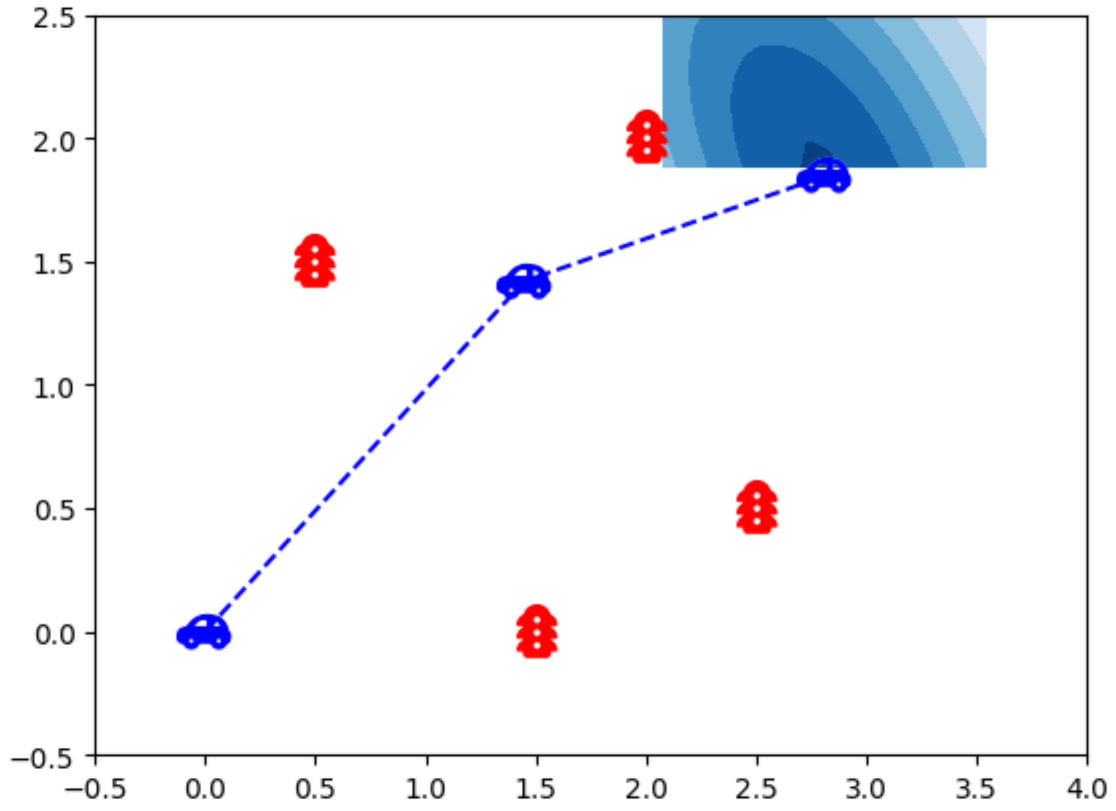
Full Kalman Filter algorithm

```
mu_t, sigma_t = μ0, Σ0
means = [mu_t]
for t in range(d.shape[0]):
  mu_1_motion, sigma_1_motion = motion_update_kf(mu_0, sigma_0, d[t], Abart, Bbar
  f_is = f_i_t[t+1].flatten()
  mu_t, sigma_t = measurement_update_kf(mu_1_motion, sigma_1_motion, f_is,
                                    Hbart_create(H, z, visible_all_t[t+1]),
                                     Qbart_create(Q, visible_all_t[t+1]))
  plot_truncated_gaussian(mu_t[:D], sigma_t[:D, :D])
  plot_landmarks(z)
  means.append(mu_t[:D])
  mean_np = np.hstack(means).reshape(-1, D)
  plt.plot(mean_np[:, 0], mean_np[:, 1], markersize=20, marker=mplcar, linestyle=
  display(plt.show())
```





None



None

[MacKay1998]: MacKay, David JC. "Introduction to monte carlo methods." Learning in graphical models. Springer, Dordrecht, 1998. 175-204

[Thrun2005]: Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. Probabilistic robotics. MIT press, 2005