PythonRobotics Documentation

Atsushi Sakai

Sep 22, 2019

Guide

1 Getting Started 3
- 1.1 What is this? 3
- 1.2 Requirements 3
- 1.3 How to use 4

2 Introduction 5
- 2.1 Ref 5

3 Localization 7
- 3.1 Extended Kalman Filter Localization 7
- 3.2 Unscented Kalman Filter localization 10
- 3.3 Particle filter localization 11
- 3.4 Histogram filter localization 11

4 Mapping 13
- 4.1 Gaussian grid map 13
- 4.2 Ray casting grid map 13
- 4.3 k-means object clustering 13
- 4.4 Object shape recognition using circle fitting 13

5 SLAM 15
- 5.1 Iterative Closest Point (ICP) Matching 15
- 5.2 EKF SLAM 15
- 5.3 FastSLAM1.0 16
- 5.4 FastSLAM 2.0 26
- 5.5 Graph based SLAM 26

6 Path Planning 29
- 6.1 Dynamic Window Approach 29
- 6.2 Grid based search 29
- 6.3 Model Predictive Trajectory Generator 30
- 6.4 State Lattice Planning 31
- 6.5 Probabilistic Road-Map (PRM) planning 31
- 6.6 Voronoi Road-Map planning 31
- 6.7 Rapidly-Exploring Random Trees (RRT) 32
- 6.8 Cubic spline planning 35
- 6.9 B-Spline planning 38

6.10 Eta^3 Spline path planning . . . 38
6.11 Bezier path planning . . . 39
6.12 Quintic polynomials planning . . . 40
6.13 Dubins path planning . . . 40
6.14 Reeds Shepp planning . . . 41
6.15 LQR based path planning . . . 41
6.16 Optimal Trajectory in a Frenet Frame . . . 41

7 Path tracking **43**
7.1 move to a pose control . . . 43
7.2 Pure pursuit tracking . . . 43
7.3 Stanley control . . . 43
7.4 Rear wheel feedback control . . . 44
7.5 Linear–quadratic regulator (LQR) steering control . . . 44
7.6 Linear–quadratic regulator (LQR) speed and steering control . . . 44
7.7 Model predictive speed and steering control . . . 44
7.8 Nonlinear Model Predictive Control with C-GMRES . . . 47

8 Arm Navigation **53**
8.1 Two joint arm to point control . . . 53
8.2 N joint arm to point control . . . 60
8.3 Arm navigation with obstacle avoidance . . . 61

9 Aerial Navigation **63**
9.1 Drone 3d trajectory following . . . 63
9.2 rocket powered landing . . . 63

10 Appendix **67**
10.1 KF Basics - Part I . . . 67
10.2 KF Basics - Part 2 . . . 77

11 Indices and tables **83**

Python codes for robotics algorithm. The project is on GitHub.

This is a Python code collection of robotics algorithms, especially for autonomous navigation.

Features:

1. Easy to read for understanding each algorithm's basic idea.
2. Widely used and practical algorithms are selected.
3. Minimum dependency.

See this paper for more details:

- [1808.10703] PythonRobotics: a Python code collection of robotics algorithms (BibTeX)

CHAPTER 1

Getting Started

1.1 What is this?

This is an Open Source Software (OSS) project: PythonRobotics, which is a Python code collection of robotics algorithms.

The focus of the project is on autonomous navigation, and the goal is for beginners in robotics to understand the basic ideas behind each algorithm.

In this project, the algorithms which are practical and widely used in both academia and industry are selected.

Each sample code is written in Python3 and only depends on some standard modules for readability and ease of use.

It includes intuitive animations to understand the behavior of the simulation.

See this paper for more details:

- PythonRobotics: a Python code collection of robotics algorithms: https://arxiv.org/abs/1808.10703

1.2 Requirements

- Python 3.6.x
- numpy
- scipy
- matplotlib
- pandas
- cvxpy

1.3 How to use

1. Install the required libraries. You can use environment.yml with conda command.
2. Clone this repo.
3. Execute python script in each directory.
4. Add star to this repo if you like it .

CHAPTER 2

Introduction

2.1 Ref

CHAPTER 3

Localization

3.1 Extended Kalman Filter Localization

```
from IPython.display import Image
Image(filename="ekf.png",width=600)
```

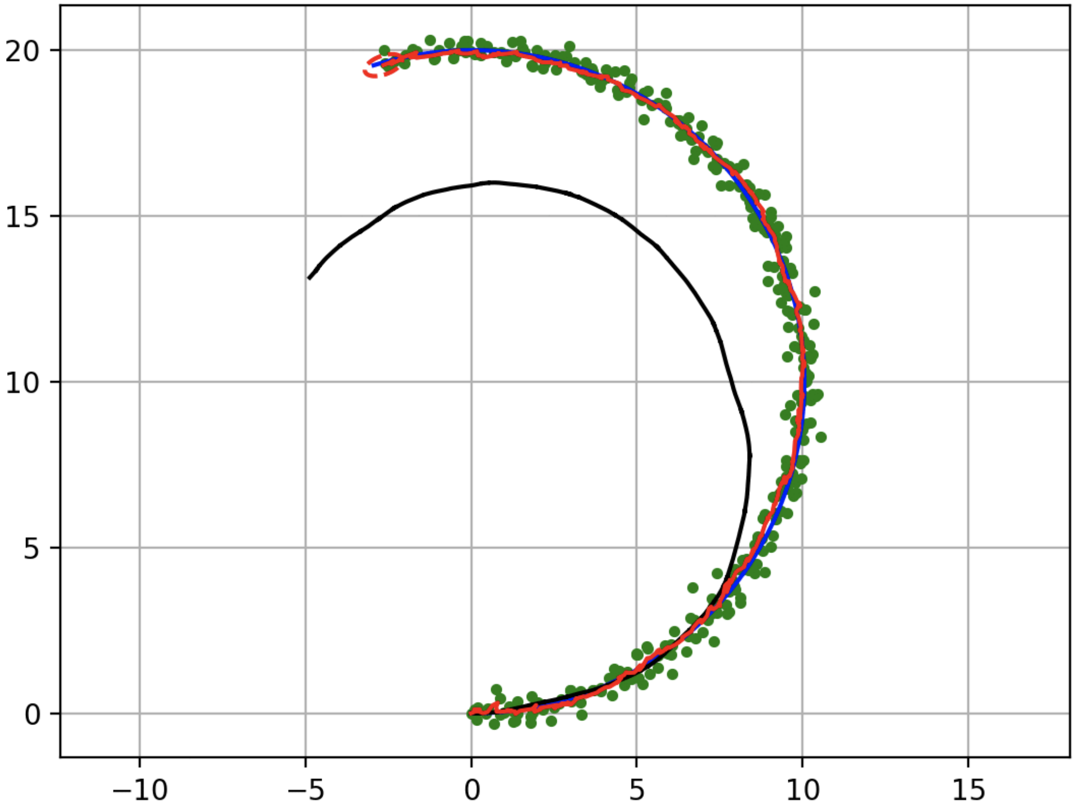


Fig. 1: EKF

This is a sensor fusion localization with Extended Kalman Filter(EKF).

The blue line is true trajectory, the black line is dead reckoning trajectory,

the green point is positioning observation (ex. GPS), and the red line is estimated trajectory with EKF.

The red ellipse is estimated covariance ellipse with EKF.

Code: PythonRobotics/extended_kalman_filter.py at master · AtsushiSakai/PythonRobotics

3.1.1 Filter design

In this simulation, the robot has a state vector includes 4 states at time t.

$$\mathbf{x}_t = [x_t, y_t, \phi_t, v_t]$$

x, y are a 2D x-y position, ϕ is orientation, and v is velocity.

In the code, "xEst" means the state vector. code

And, P_t is covariace matrix of the state,

Q is covariance matrix of process noise,

R is covariance matrix of observation noise at time t

The robot has a speed sensor and a gyro sensor.

So, the input vecor can be used as each time step

$$\mathbf{u}_t = [v_t, \omega_t]$$

Also, the robot has a GNSS sensor, it means that the robot can observe x-y position at each time.

$$\mathbf{z}_t = [x_t, y_t]$$

The input and observation vector includes sensor noise.

In the code, "observation" function generates the input and observation vector with noise code

3.1.2 Motion Model

The robot model is

$$\dot{x} = vcos(\phi)$$

$$\dot{y} = vsin((\phi)$$

$$\dot{\phi} = \omega$$

So, the motion model is

$$\mathbf{x}_{t+1} = F\mathbf{x}_t + B\mathbf{u}_t$$

where

$$F = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} cos(\phi)dt & 0 \\ sin(\phi)dt & 0 \\ 0 & dt \\ 1 & 0 \end{bmatrix}$$

dt is a time interval.

This is implemented at code

Its Jacobian matrix is

$$J_F = \begin{bmatrix} \frac{dx}{dx} & \frac{dx}{dy} & \frac{dx}{d\phi} & \frac{dx}{dv} \\ \frac{dy}{dx} & \frac{dy}{dy} & \frac{dy}{d\phi} & \frac{dy}{dv} \\ \frac{d\phi}{dx} & \frac{d\phi}{dy} & \frac{d\phi}{d\phi} & \frac{d\phi}{dv} \\ \frac{dv}{dx} & \frac{dv}{dy} & \frac{dv}{d\phi} & \frac{dv}{dv} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & -vsin(\phi)dt & cos(\phi)dt \\ 0 & 1 & vcos(\phi)dt & sin(\phi)dt \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.1.3 Observation Model

The robot can get x-y position infomation from GPS.

So GPS Observation model is

$$\mathbf{z}_t = H\mathbf{x}_t$$

where

$B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$

Its Jacobian matrix is

$J_H = \begin{bmatrix} \frac{dx}{dx} & \frac{dx}{dy} & \frac{dx}{d\phi} & \frac{dx}{dv} \\ \frac{dy}{dx} & \frac{dy}{dy} & \frac{dy}{d\phi} & \frac{dy}{dv} \end{bmatrix}$

$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$

3.1.4 Extented Kalman Filter

Localization process using Extendted Kalman Filter:EKF is

=== Predict ===

$x_{Pred} = Fx_t + Bu_t$

$P_{Pred} = J_F P_t J_F^T + Q$

=== Update ===

$z_{Pred} = Hx_{Pred}$

$y = z - z_{Pred}$

$S = J_H P_{Pred}.J_H^T + R$

$K = P_{Pred}.J_H^T S^{-1}$

$x_{t+1} = x_{Pred} + Ky$

$P_{t+1} = (I - KJ_H)P_{Pred}$

3.1.5 Ref:

- PROBABILISTIC-ROBOTICS.ORG

3.2 Unscented Kalman Filter localization

This is a sensor fusion localization with Unscented Kalman Filter(UKF).

The lines and points are same meaning of the EKF simulation.

Ref:

- Discriminatively Trained Unscented Kalman Filter for Mobile Robot Localization

3.3 Particle filter localization

This is a sensor fusion localization with Particle Filter(PF).

The blue line is true trajectory, the black line is dead reckoning trajectory,

and the red line is estimated trajectory with PF.

It is assumed that the robot can measure a distance from landmarks (RFID).

This measurements are used for PF localization.

Ref:

- PROBABILISTIC ROBOTICS

3.4 Histogram filter localization

This is a 2D localization example with Histogram filter.

The red cross is true position, black points are RFID positions.

The blue grid shows a position probability of histogram filter.

In this simulation, x,y are unknown, yaw is known.

The filter integrates speed input and range observations from RFID for localization.

Initial position is not needed.

Ref:

- PROBABILISTIC ROBOTICS

CHAPTER 4

Mapping

4.1 Gaussian grid map

This is a 2D Gaussian grid mapping example.

4.2 Ray casting grid map

This is a 2D ray casting grid mapping example.

4.3 k-means object clustering

This is a 2D object clustering with k-means algorithm.

4.4 Object shape recognition using circle fitting

This is an object shape recognition using circle fitting.

The blue circle is the true object shape.

The red crosses are observations from a ranging sensor.

The red circle is the estimated object shape using circle fitting.

CHAPTER 5

SLAM

Simultaneous Localization and Mapping(SLAM) examples

5.1 Iterative Closest Point (ICP) Matching

This is a 2D ICP matching example with singular value decomposition.

It can calculate a rotation matrix and a translation vector between points to points.

Ref:

- Introduction to Mobile Robotics: Iterative Closest Point Algorithm

5.2 EKF SLAM

This is an Extended Kalman Filter based SLAM example.

The blue line is ground truth, the black line is dead reckoning, the red line is the estimated trajectory with EKF SLAM.

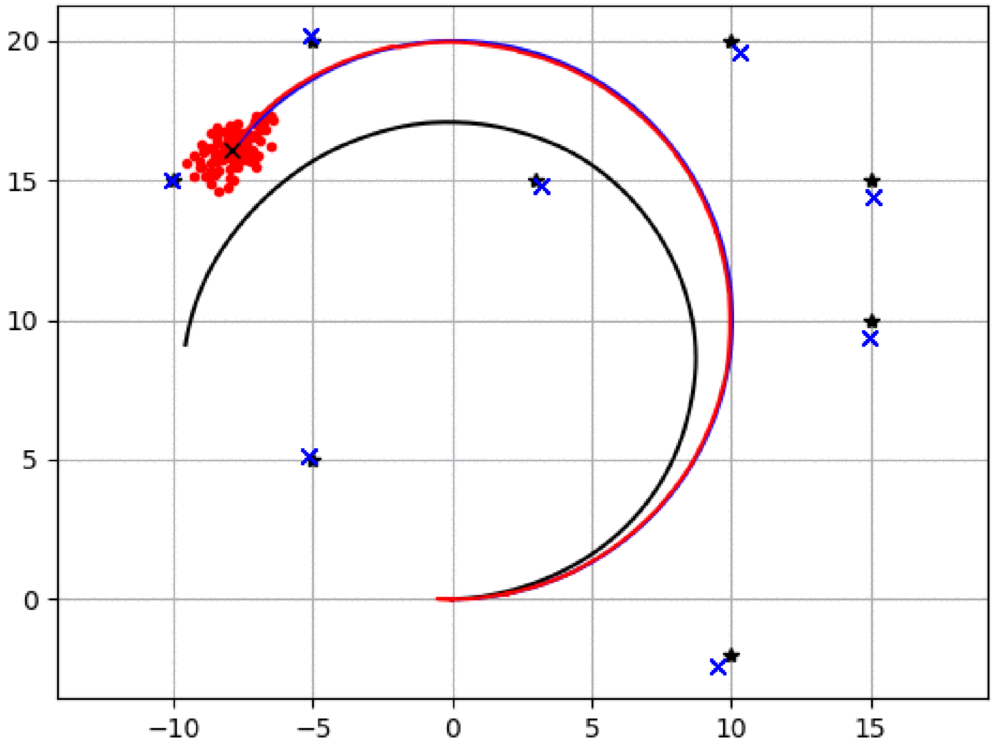
The green crosses are estimated landmarks.

Ref:

- PROBABILISTIC ROBOTICS

5.3 FastSLAM1.0

```
from IPython.display import Image
Image(filename="animation.png",width=600)
```



5.3.1 Simulation

This is a feature based SLAM example using FastSLAM 1.0.

The blue line is ground truth, the black line is dead reckoning, the red line is the estimated trajectory with FastSLAM.

The red points are particles of FastSLAM.

Black points are landmarks, blue crosses are estimated landmark positions by FastSLAM.

Fig. 1: gif

5.3.2 Introduction

FastSLAM algorithm implementation is based on particle filters and belongs to the family of probabilistic SLAM approaches. It is used with feature-based maps (see gif above) or with occupancy grid maps.

As it is shown, the particle filter differs from EKF by representing the robot's estimation through a set of particles. Each single particle has an independent belief, as it holds the pose (x, y, θ) and an array of landmark locations $[(x_1, y_1), (x_2, y_2), ...(x_n, y_n)]$ for n landmarks.

- The blue line is the true trajectory
- The red line is the estimated trajectory
- The red dots represent the distribution of particles
- The black line represent dead reckoning tracjectory
- The blue x is the observed and estimated landmarks
- The black x is the true landmark

I.e. Each particle maintains a deterministic pose and n-EKFs for each landmark and update it with each measurement.

5.3.3 Algorithm walkthrough

The particles are initially drawn from a uniform distribution the represent the initial uncertainty. At each time step we do:

- Predict the pose for each particle by using u and the motion model (the landmarks are not updated).
- Update the particles with observations z, where the weights are adjusted based on how likely the particle to have the correct pose given the sensor measurement
- Resampling such that the particles with the largest weights survive and the unlikely ones with the lowest weights die out.

5.3.4 1- Predict

The following equations and code snippets we can see how the particles distribution evolves in case we provide only the control (v, w), which are the linear and angular velocity repsectively.

$$F = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} \Delta t cos(\theta) & 0 \\ \Delta t sin(\theta) & 0 \\ 0 & \Delta t \end{bmatrix}$$

$$X = FX + BU$$

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \\ \theta_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} + \begin{bmatrix} \Delta t cos(\theta) & 0 \\ \Delta t sin(\theta) & 0 \\ 0 & \Delta t \end{bmatrix} \begin{bmatrix} v_t + \sigma_v \\ w_t + \sigma_w \end{bmatrix}$$

The following snippets playsback the recorded trajectory of each particle.

To get the insight of the motion model change the value of R and re-run the cells again. As R is the parameters that indicates how much we trust that the robot executed the motion commands.

It is interesting to notice also that only motion will increase the uncertainty in the system as the particles start to spread out more. If observations are included the uncertainty will decrease and particles will converge to the correct estimate.

```
# CODE SNIPPET #
import numpy as np
import math
from copy import deepcopy
# Fast SLAM covariance
Q = np.diag([3.0, np.deg2rad(10.0)])**2
R = np.diag([1.0, np.deg2rad(20.0)])**2

#  Simulation parameter
Qsim = np.diag([0.3, np.deg2rad(2.0)])**2
Rsim = np.diag([0.5, np.deg2rad(10.0)])**2
OFFSET_YAWRATE_NOISE = 0.01

DT = 0.1  # time tick [s]
SIM_TIME = 50.0  # simulation time [s]
MAX_RANGE = 20.0  # maximum observation range
M_DIST_TH = 2.0  # Threshold of Mahalanobis distance for data association.
STATE_SIZE = 3  # State size [x,y,yaw]
LM_SIZE = 2  # LM srate size [x,y]
N_PARTICLE = 100  # number of particle
NTH = N_PARTICLE / 1.5  # Number of particle for re-sampling

class Particle:

    def __init__(self, N_LM):
        self.w = 1.0 / N_PARTICLE
        self.x = 0.0
        self.y = 0.0
        self.yaw = 0.0
        # landmark x-y positions
        self.lm = np.zeros((N_LM, LM_SIZE))
        # landmark position covariance
        self.lmP = np.zeros((N_LM * LM_SIZE, LM_SIZE))

def motion_model(x, u):
    F = np.array([[1.0, 0, 0],
                  [0, 1.0, 0],
                  [0, 0, 1.0]])

    B = np.array([[DT * math.cos(x[2, 0]), 0],
                  [DT * math.sin(x[2, 0]), 0],
                  [0.0, DT]])
    x = F @ x + B @ u

    x[2, 0] = pi_2_pi(x[2, 0])
    return x

def predict_particles(particles, u):
    for i in range(N_PARTICLE):
        px = np.zeros((STATE_SIZE, 1))
        px[0, 0] = particles[i].x
        px[1, 0] = particles[i].y
        px[2, 0] = particles[i].yaw
        ud = u + (np.random.randn(1, 2) @ R).T  # add noise
        px = motion_model(px, ud)
        particles[i].x = px[0, 0]
        particles[i].y = px[1, 0]
```

(continues on next page)

(continued from previous page)

```
        particles[i].yaw = px[2, 0]

    return particles

def pi_2_pi(angle):
    return (angle + math.pi) % (2 * math.pi) - math.pi

# END OF SNIPPET

N_LM = 0
particles = [Particle(N_LM) for i in range(N_PARTICLE)]
time= 0.0
v = 1.0  # [m/s]
yawrate = 0.1  # [rad/s]
u = np.array([v, yawrate]).reshape(2, 1)
history = []
while SIM_TIME >= time:
    time += DT
    particles = predict_particles(particles, u)
    history.append(deepcopy(particles))
```

```
# from IPython.html.widgets import *
from ipywidgets import *
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# playback the recorded motion of the particles
def plot_particles(t=0):
    x = []
    y = []
    for i in range(len(history[t])):
        x.append(history[t][i].x)
        y.append(history[t][i].y)
    plt.figtext(0.15,0.82,'t = ' + str(t))
    plt.plot(x, y, '.r')
    plt.axis([-20,20, -5,25])

interact(plot_particles, t=(0,len(history)-1,1));
```

```
interactive(children=(IntSlider(value=0, description='t', max=499), Output()), _dom_
↪classes=('widget-interact'...
```

5.3.5 2- Update

For the update step it is useful to observe a single particle and the effect of getting a new measurements on the weight of the particle.

As mentioned earlier, each particle maintains N $2x2$ EKFs to estimate the landmarks, which includes the EKF process described in the EKF notebook. The difference is the change in the weight of the particle according to how likely the measurement is.

The weight is updated according to the following equation:

$w_i = |2\pi Q|^{\frac{-1}{2}} exp\{\frac{-1}{2}(z_t - \hat{z}_i)^T Q^{-1}(z_t - \hat{z}_i)\}$

Where, w_i is the computed weight, Q is the measurement covariance, z_t is the actual measurment and $\hat{z}_i$ is the predicted measurement of particle i.

To experiment this, a single particle is initialized then passed an initial measurement, which results in a relatively average weight. However, setting the particle coordinate to a wrong value to simulate wrong estimation will result in a very low weight. The lower the weight the less likely that this particle will be drawn during resampling and probably will die out.

```
# CODE SNIPPET #
def observation(xTrue, xd, u, RFID):

    # calc true state
    xTrue = motion_model(xTrue, u)

    # add noise to range observation
    z = np.zeros((3, 0))
    for i in range(len(RFID[:, 0])):

        dx = RFID[i, 0] - xTrue[0, 0]
        dy = RFID[i, 1] - xTrue[1, 0]
        d = math.sqrt(dx**2 + dy**2)
        angle = pi_2_pi(math.atan2(dy, dx) - xTrue[2, 0])
        if d <= MAX_RANGE:
            dn = d + np.random.randn() * Qsim[0, 0]  # add noise
            anglen = angle + np.random.randn() * Qsim[1, 1]  # add noise
            zi = np.array([dn, pi_2_pi(anglen), i]).reshape(3, 1)
            z = np.hstack((z, zi))

    # add noise to input
    ud1 = u[0, 0] + np.random.randn() * Rsim[0, 0]
    ud2 = u[1, 0] + np.random.randn() * Rsim[1, 1] + OFFSET_YAWRATE_NOISE
    ud = np.array([ud1, ud2]).reshape(2, 1)

    xd = motion_model(xd, ud)

    return xTrue, z, xd, ud

def update_with_observation(particles, z):
    for iz in range(len(z[0, :])):

        lmid = int(z[2, iz])

        for ip in range(N_PARTICLE):
            # new landmark
            if abs(particles[ip].lm[lmid, 0]) <= 0.01:
                particles[ip] = add_new_lm(particles[ip], z[:, iz], Q)
            # known landmark
            else:
                w = compute_weight(particles[ip], z[:, iz], Q)
                particles[ip].w *= w
                particles[ip] = update_landmark(particles[ip], z[:, iz], Q)

    return particles

def compute_weight(particle, z, Q):
    lm_id = int(z[2])
    xf = np.array(particle.lm[lm_id, :]).reshape(2, 1)
    Pf = np.array(particle.lmP[2 * lm_id:2 * lm_id + 2])
```

(continues on next page)

(continued from previous page)

```
    zp, Hv, Hf, Sf = compute_jacobians(particle, xf, Pf, Q)
    dx = z[0:2].reshape(2, 1) - zp
    dx[1, 0] = pi_2_pi(dx[1, 0])

    try:
        invS = np.linalg.inv(Sf)
    except np.linalg.linalg.LinAlgError:
        print("singuler")
        return 1.0

    num = math.exp(-0.5 * dx.T @ invS @ dx)
    den = 2.0 * math.pi * math.sqrt(np.linalg.det(Sf))
    w = num / den

    return w

def compute_jacobians(particle, xf, Pf, Q):
    dx = xf[0, 0] - particle.x
    dy = xf[1, 0] - particle.y
    d2 = dx**2 + dy**2
    d = math.sqrt(d2)

    zp = np.array(
        [d, pi_2_pi(math.atan2(dy, dx) - particle.yaw)]).reshape(2, 1)

    Hv = np.array([[-dx / d, -dy / d, 0.0],
                   [dy / d2, -dx / d2, -1.0]])

    Hf = np.array([[dx / d, dy / d],
                   [-dy / d2, dx / d2]])

    Sf = Hf @ Pf @ Hf.T + Q

    return zp, Hv, Hf, Sf

def add_new_lm(particle, z, Q):

    r = z[0]
    b = z[1]
    lm_id = int(z[2])

    s = math.sin(pi_2_pi(particle.yaw + b))
    c = math.cos(pi_2_pi(particle.yaw + b))

    particle.lm[lm_id, 0] = particle.x + r * c
    particle.lm[lm_id, 1] = particle.y + r * s

    # covariance
    Gz = np.array([[c, -r * s],
                   [s, r * c]])

    particle.lmP[2 * lm_id:2 * lm_id + 2] = Gz @ Q @ Gz.T

    return particle

def update_KF_with_cholesky(xf, Pf, v, Q, Hf):
    PHt = Pf @ Hf.T
```

(continues on next page)

(continued from previous page)

```
    S = Hf @ PHt + Q

    S = (S + S.T) * 0.5
    SChol = np.linalg.cholesky(S).T
    ScholInv = np.linalg.inv(SChol)
    W1 = PHt @ ScholInv
    W = W1 @ ScholInv.T

    x = xf + W @ v
    P = Pf - W1 @ W1.T

    return x, P

def update_landmark(particle, z, Q):

    lm_id = int(z[2])
    xf = np.array(particle.lm[lm_id, :]).reshape(2, 1)
    Pf = np.array(particle.lmP[2 * lm_id:2 * lm_id + 2, :])

    zp, Hv, Hf, Sf = compute_jacobians(particle, xf, Pf, Q)

    dz = z[0:2].reshape(2, 1) - zp
    dz[1, 0] = pi_2_pi(dz[1, 0])

    xf, Pf = update_KF_with_cholesky(xf, Pf, dz, Q, Hf)

    particle.lm[lm_id, :] = xf.T
    particle.lmP[2 * lm_id:2 * lm_id + 2, :] = Pf

    return particle

# END OF CODE SNIPPET #



# Setting up the landmarks
RFID = np.array([[10.0, -2.0],
                [15.0, 10.0]])
N_LM = RFID.shape[0]

# Initialize 1 particle
N_PARTICLE = 1
particles = [Particle(N_LM) for i in range(N_PARTICLE)]

xTrue = np.zeros((STATE_SIZE, 1))
xDR = np.zeros((STATE_SIZE, 1))

print("initial weight", particles[0].w)

xTrue, z, _, ud = observation(xTrue, xDR, u, RFID)
# Initialize landmarks
particles = update_with_observation(particles, z)
print("weight after landmark intialization", particles[0].w)
particles = update_with_observation(particles, z)
print("weight after update ", particles[0].w)

particles[0].x = -10
```

(continues on next page)

(continued from previous page)

```
particles = update_with_observation(particles, z)
print("weight after wrong prediction", particles[0].w)
```

```
initial weight 1.0
weight after landmark intialization 1.0
weight after update  0.023098460073039763
weight after wrong prediction 7.951154575772496e-07
```

5.3.6 3- Resampling

In the reseampling steps a new set of particles are chosen from the old set. This is done according to the weight of each particle.

The figure shows 100 particles distributed uniformly between [-0.5, 0.5] with the weights of each particle distributed according to a Gaussian funciton.

The resampling picks

$i \in 1, ..., N$ particles with probability to pick particle with index $i\omega_i$, where ω_i is the weight of that particle

To get the intuition of the resampling step we will look at a set of particles which are initialized with a given x location and weight. After the resampling the particles are more concetrated in the location where they had the highest weights. This is also indicated by the indices

```
# CODE SNIPPET #
def normalize_weight(particles):

    sumw = sum([p.w for p in particles])

    try:
        for i in range(N_PARTICLE):
            particles[i].w /= sumw
    except ZeroDivisionError:
        for i in range(N_PARTICLE):
            particles[i].w = 1.0 / N_PARTICLE

        return particles

    return particles


def resampling(particles):
    """
    low variance re-sampling
    """

    particles = normalize_weight(particles)

    pw = []
    for i in range(N_PARTICLE):
        pw.append(particles[i].w)

    pw = np.array(pw)

    Neff = 1.0 / (pw @ pw.T)  # Effective particle number
    # print(Neff)
```

(continues on next page)

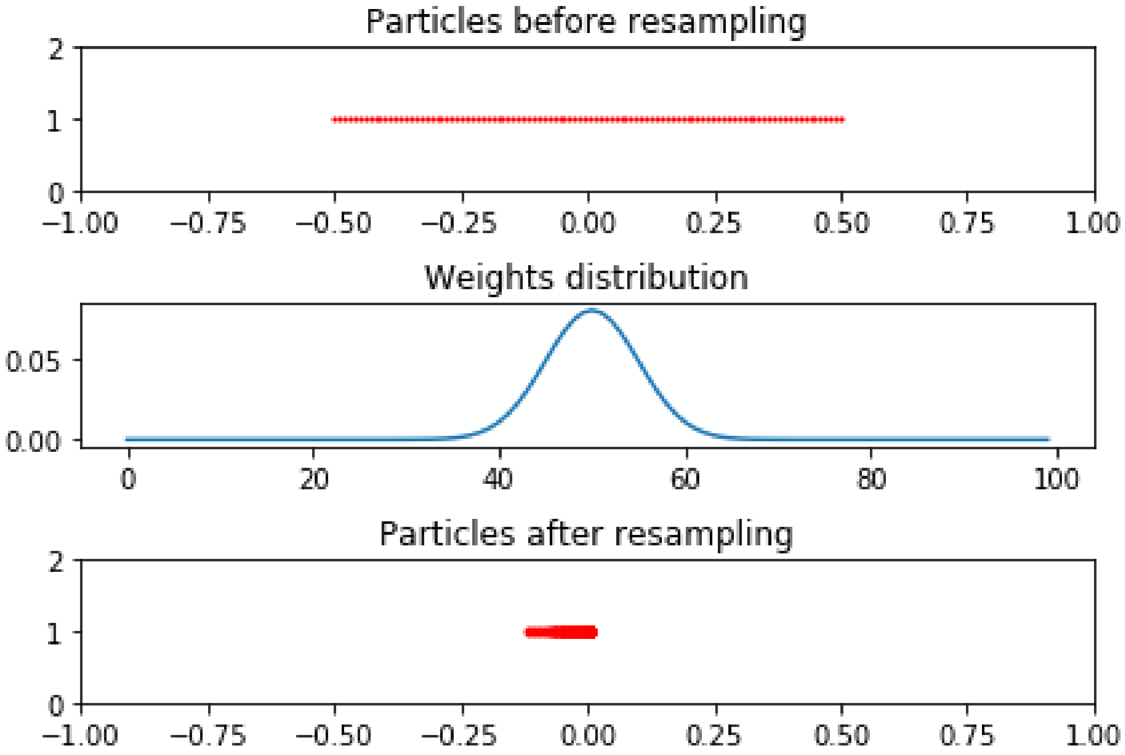
(continued from previous page)

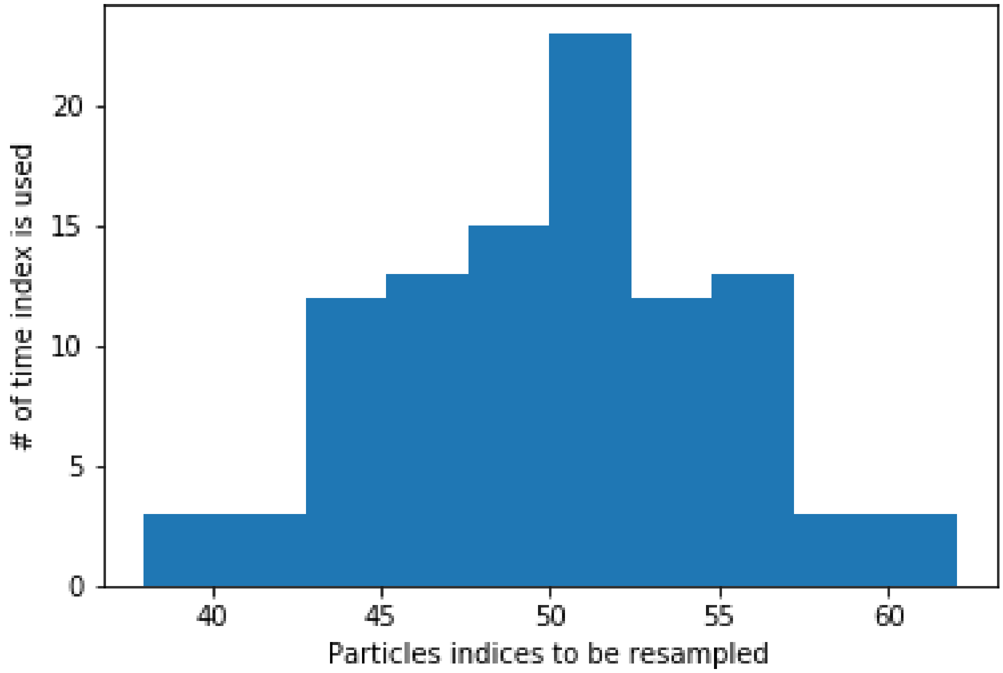
```
    if Neff < NTH:  # resampling
        wcum = np.cumsum(pw)
        base = np.cumsum(pw * 0.0 + 1 / N_PARTICLE) - 1 / N_PARTICLE
        resampleid = base + np.random.rand(base.shape[0]) / N_PARTICLE

        inds = []
        ind = 0
        for ip in range(N_PARTICLE):
            while ((ind < wcum.shape[0] - 1) and (resampleid[ip] > wcum[ind])):
                ind += 1
            inds.append(ind)

        tparticles = particles[:]
        for i in range(len(inds)):
            particles[i].x = tparticles[inds[i]].x
            particles[i].y = tparticles[inds[i]].y
            particles[i].yaw = tparticles[inds[i]].yaw
            particles[i].w = 1.0 / N_PARTICLE

    return particles, inds
# END OF SNIPPET #



def gaussian(x, mu, sig):
    return np.exp(-np.power(x - mu, 2.) / (2 * np.power(sig, 2.)))
N_PARTICLE = 100
particles = [Particle(N_LM) for i in range(N_PARTICLE)]
x_pos = []
w = []
for i in range(N_PARTICLE):
    particles[i].x = np.linspace(-0.5,0.5,N_PARTICLE)[i]
    x_pos.append(particles[i].x)
    particles[i].w = gaussian(i, N_PARTICLE/2, N_PARTICLE/20)
    w.append(particles[i].w)


# Normalize weights
sw = sum(w)
for i in range(N_PARTICLE):
    w[i] /= sw

particles, new_indices = resampling(particles)
x_pos2 = []
for i in range(N_PARTICLE):
    x_pos2.append(particles[i].x)

# Plot results
fig, ((ax1,ax2,ax3)) = plt.subplots(nrows=3, ncols=1)
fig.tight_layout()
ax1.plot(x_pos,np.ones((N_PARTICLE,1)), '.r', markersize=2)
ax1.set_title("Particles before resampling")
ax1.axis((-1, 1, 0, 2))
ax2.plot(w)
ax2.set_title("Weights distribution")
ax3.plot(x_pos2,np.ones((N_PARTICLE,1)), '.r')
```

(continues on next page)

(continued from previous page)

```
ax3.set_title("Particles after resampling")
ax3.axis((-1, 1, 0, 2))
fig.subplots_adjust(hspace=0.8)
plt.show()

plt.figure()
plt.hist(new_indices)
plt.xlabel("Particles indices to be resampled")
plt.ylabel("# of time index is used")
plt.show()
```





5.3.7 References

http://www.probabilistic-robotics.org/

http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/pdf/slam10-fastslam.pdf

Ref:

- PROBABILISTIC ROBOTICS
- SLAM simulations by Tim Bailey

5.4 FastSLAM 2.0

This is a feature based SLAM example using FastSLAM 2.0.

The animation has the same meanings as one of FastSLAM 1.0.

Ref:

- PROBABILISTIC ROBOTICS
- SLAM simulations by Tim Bailey

5.5 Graph based SLAM

This is a graph based SLAM example.

The blue line is ground truth.

The black line is dead reckoning.

The red line is the estimated trajectory with Graph based SLAM.

The black stars are landmarks for graph edge generation.

Ref:

- A Tutorial on Graph-Based SLAM

CHAPTER 6

Path Planning

6.1 Dynamic Window Approach

This is a 2D navigation sample code with Dynamic Window Approach.

- The Dynamic Window Approach to Collision Avoidance

6.2 Grid based search

6.2.1 Dijkstra algorithm

This is a 2D grid based shortest path planning with Dijkstra's algorithm.

In the animation, cyan points are searched nodes.

6.2.2 A* algorithm

This is a 2D grid based shortest path planning with A star algorithm.

In the animation, cyan points are searched nodes.

Its heuristic is 2D Euclid distance.

6.2.3 Potential Field algorithm

This is a 2D grid based path planning with Potential Field algorithm.

In the animation, the blue heat map shows potential value on each grid.

Ref:

- Robotic Motion Planning:Potential Functions

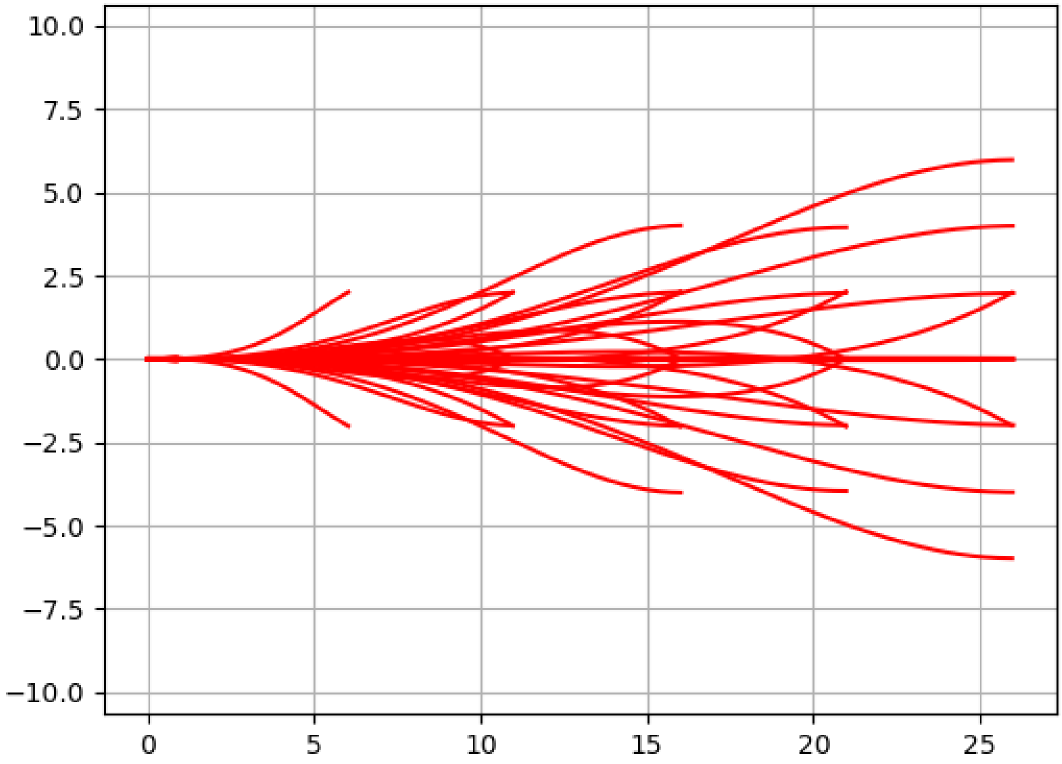
6.3 Model Predictive Trajectory Generator

This is a path optimization sample on model predictive trajectory generator.

This algorithm is used for state lattice planner.

6.3.1 Path optimization sample

6.3.2 Lookup table generation sample



Ref:

- Optimal rough terrain trajectory generation for wheeled mobile robots

6.4 State Lattice Planning

This script is a path planning code with state lattice planning.

This code uses the model predictive trajectory generator to solve boundary problem.

Ref:

- Optimal rough terrain trajectory generation for wheeled mobile robots
- State Space Sampling of Feasible Motions for High-Performance Mobile Robot Navigation in Complex Environments

6.4.1 Uniform polar sampling

6.4.2 Biased polar sampling

6.4.3 Lane sampling

6.5 Probabilistic Road-Map (PRM) planning

This PRM planner uses Dijkstra method for graph search.

In the animation, blue points are sampled points,

Cyan crosses means searched points with Dijkstra method,

The red line is the final path of PRM.

Ref:

- Probabilistic roadmap - Wikipedia

6.6 Voronoi Road-Map planning

This Voronoi road-map planner uses Dijkstra method for graph search.

In the animation, blue points are Voronoi points,

Cyan crosses mean searched points with Dijkstra method, The red line is the final path of Vornoi Road-Map.

Ref:

- Robotic Motion Planning

6.7 Rapidly-Exploring Random Trees (RRT)

6.7.1 Basic RRT

This is a simple path planning code with Rapidly-Exploring Random Trees (RRT)

Black circles are obstacles, green line is a searched tree, red crosses are start and goal positions.

6.7.2 RRT*

This is a path planning code with RRT*

Black circles are obstacles, green line is a searched tree, red crosses are start and goal positions.

Simulation

```
from IPython.display import Image
Image(filename="Figure_1.png",width=600)
```

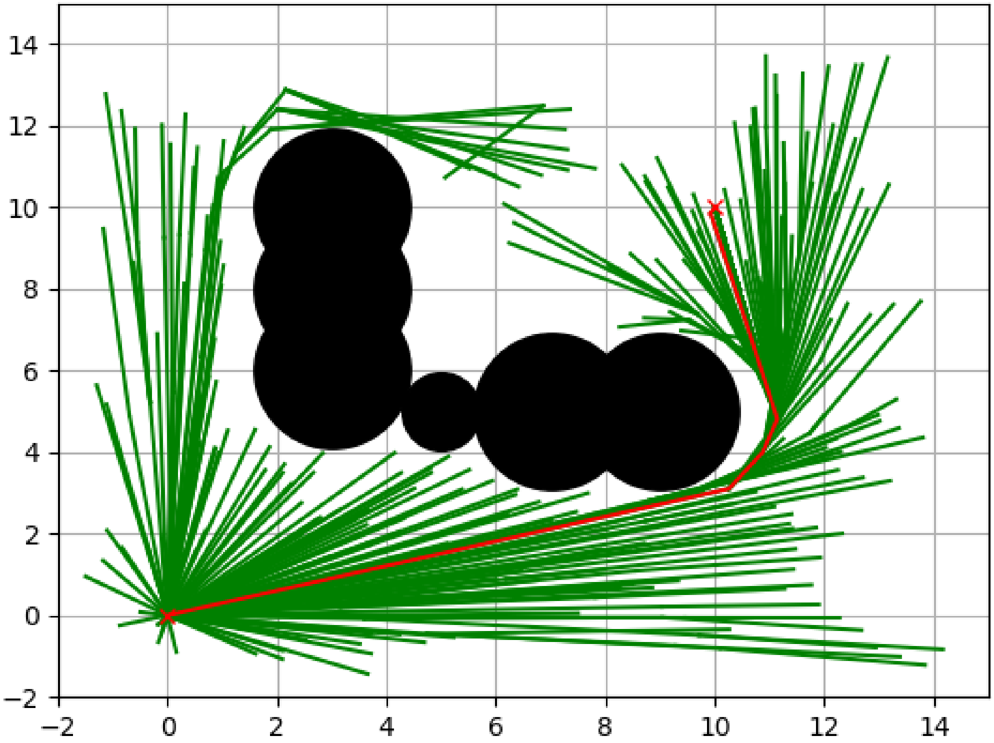


Fig. 1: gif

Ref

- Sampling-based Algorithms for Optimal Motion Planning

Ref:

- Incremental Sampling-based Algorithms for Optimal Motion Planning
- Sampling-based Algorithms for Optimal Motion Planning

6.7.3 RRT with dubins path

Path planning for a car robot with RRT and dubins path planner.

6.7.4 RRT* with dubins path

Path planning for a car robot with RRT* and dubins path planner.

6.7.5 RRT* with reeds-sheep path

Path planning for a car robot with RRT* and reeds sheep path planner.

6.7.6 Informed RRT*

This is a path planning code with Informed RRT*.

The cyan ellipse is the heuristic sampling domain of Informed RRT*.

Ref:

- Informed RRT*: Optimal Sampling-based Path Planning Focused via Direct Sampling of an Admissible Ellipsoidal Heuristic

6.7.7 Batch Informed RRT*

This is a path planning code with Batch Informed RRT*.

Ref:

- Batch Informed Trees (BIT*): Sampling-based Optimal Planning via the Heuristically Guided Search of Implicit Random Geometric Graphs

6.7.8 Closed Loop RRT*

A vehicle model based path planning with closed loop RRT*.

In this code, pure-pursuit algorithm is used for steering control,

PID is used for speed control.

Ref:

- Motion Planning in Complex Environments using Closed-loop Prediction
- Real-time Motion Planning with Applications to Autonomous Urban Driving
- [1601.06326] Sampling-based Algorithms for Optimal Motion Planning Using Closed-loop Prediction

6.7.9 LQR-RRT*

This is a path planning simulation with LQR-RRT*.

A double integrator motion model is used for LQR local planner.

Ref:

- LQR-RRT*: Optimal Sampling-Based Motion Planning with Automatically Derived Extension Heuristics

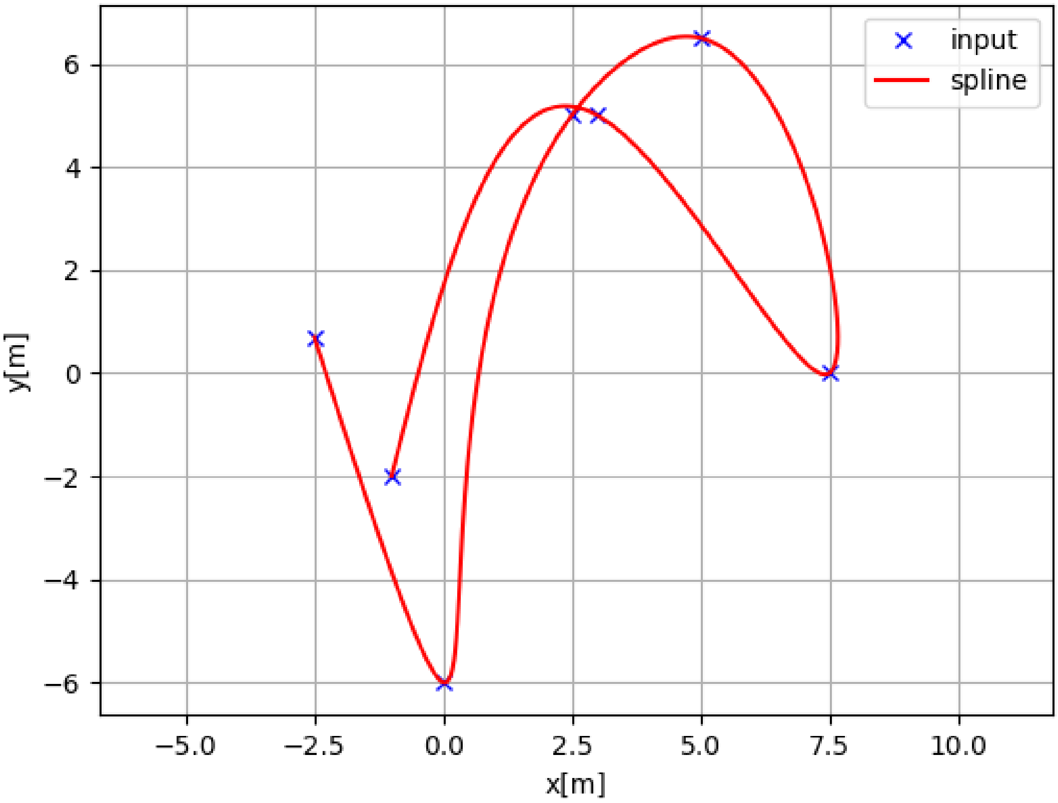
- MahanFathi/LQR-RRTstar: LQR-RRT* method is used for random motion planning of a simple pendulum in its phase plot

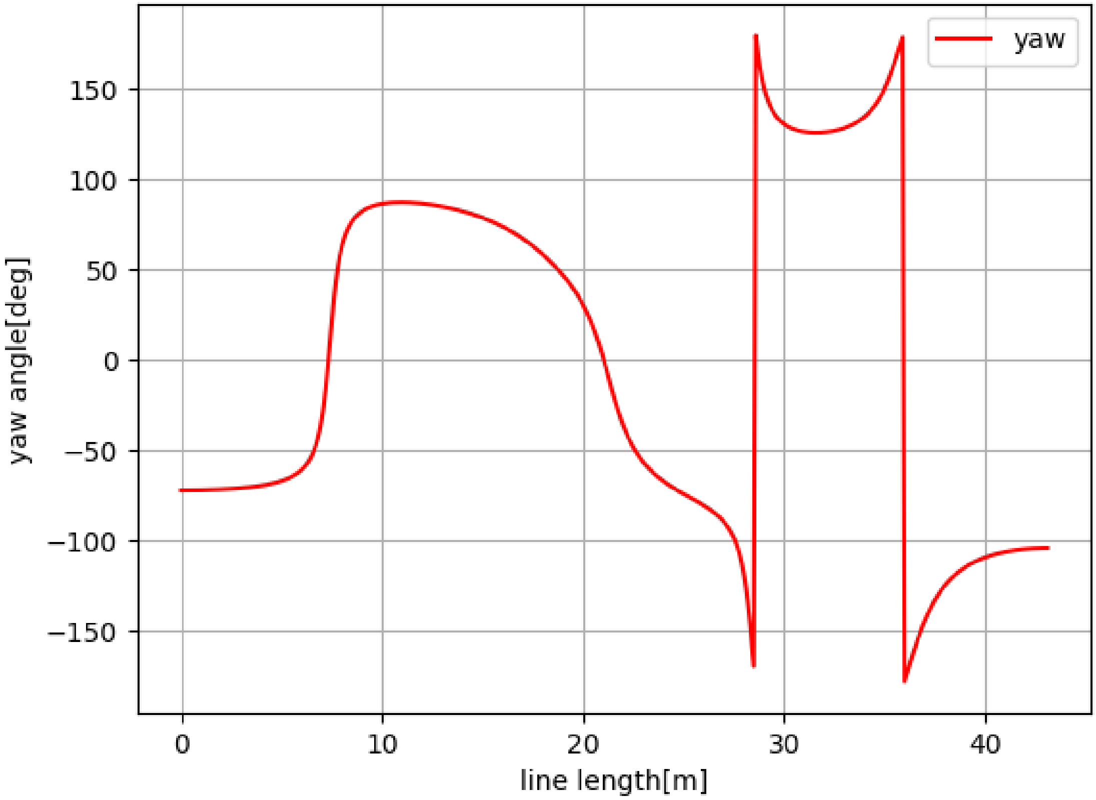
6.8 Cubic spline planning

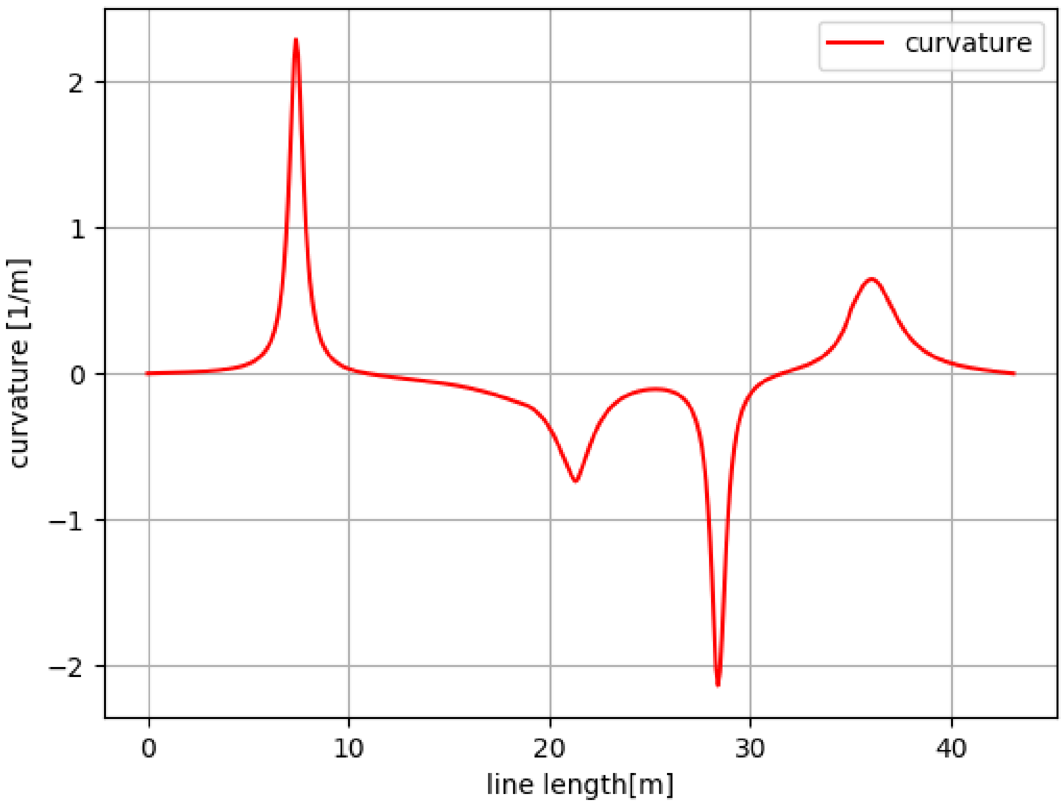
A sample code for cubic path planning.

This code generates a curvature continuous path based on x-y waypoints with cubic spline.

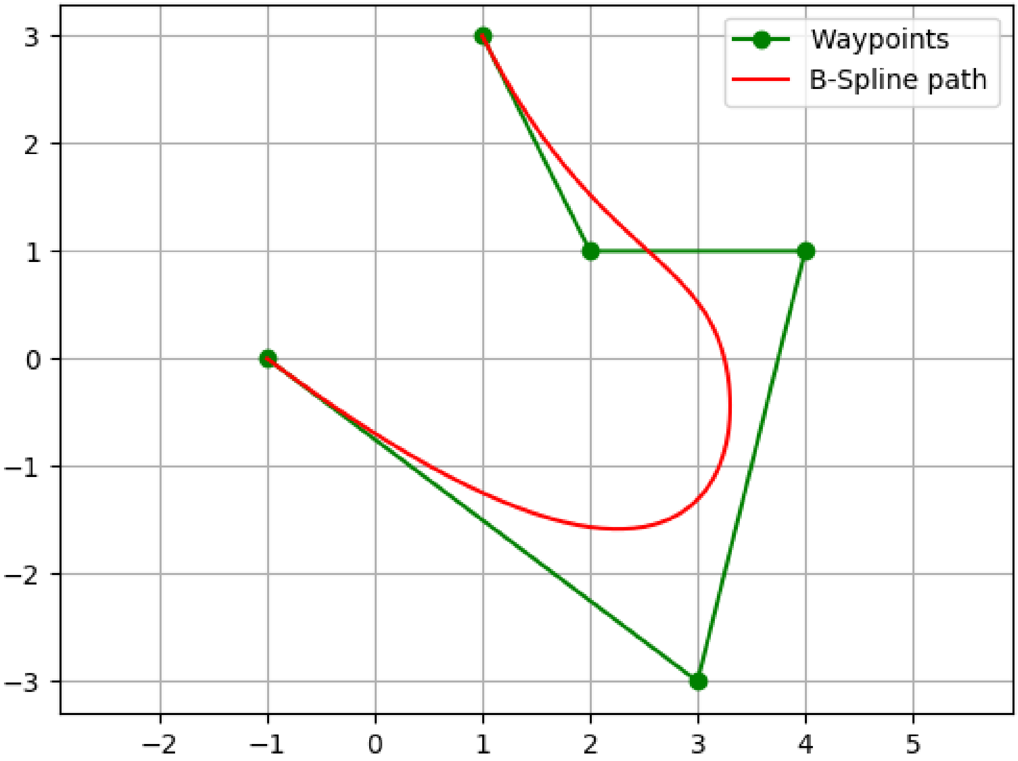
Heading angle of each point can be also calculated analytically.




yaw
150
100
50
0
−50
−100
−150
yaw angle[deg]
0
10
20
30
40
line length[m]


curvature
curvature [1/m]
line length[m]
2
1
0
−1
−2
0
10
20
30
40

6.9 B-Spline planning



This is a path planning with B-Spline curse.

If you input waypoints, it generates a smooth path with B-Spline curve.

The final course should be on the first and last waypoints.

Ref:

- B-spline - Wikipedia

6.10 Eta^3 Spline path planning

This is a path planning with Eta^3 spline.

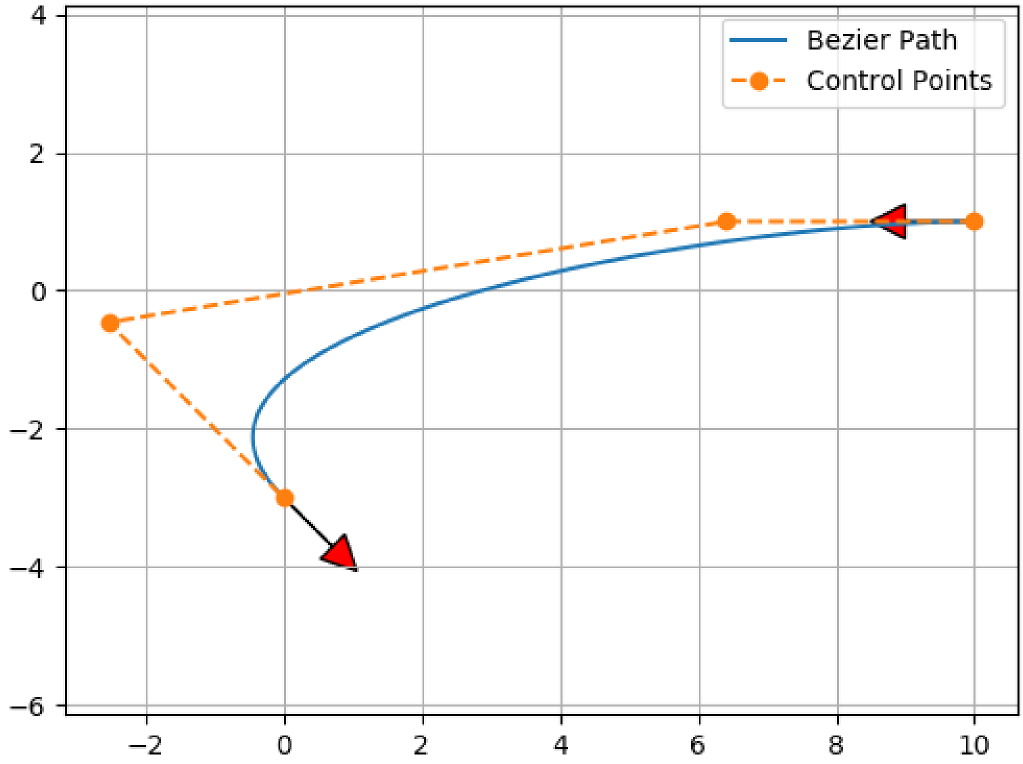
Ref:

- \eta^3-Splines for the Smooth Path Generation of Wheeled Mobile Robots

6.11 Bezier path planning

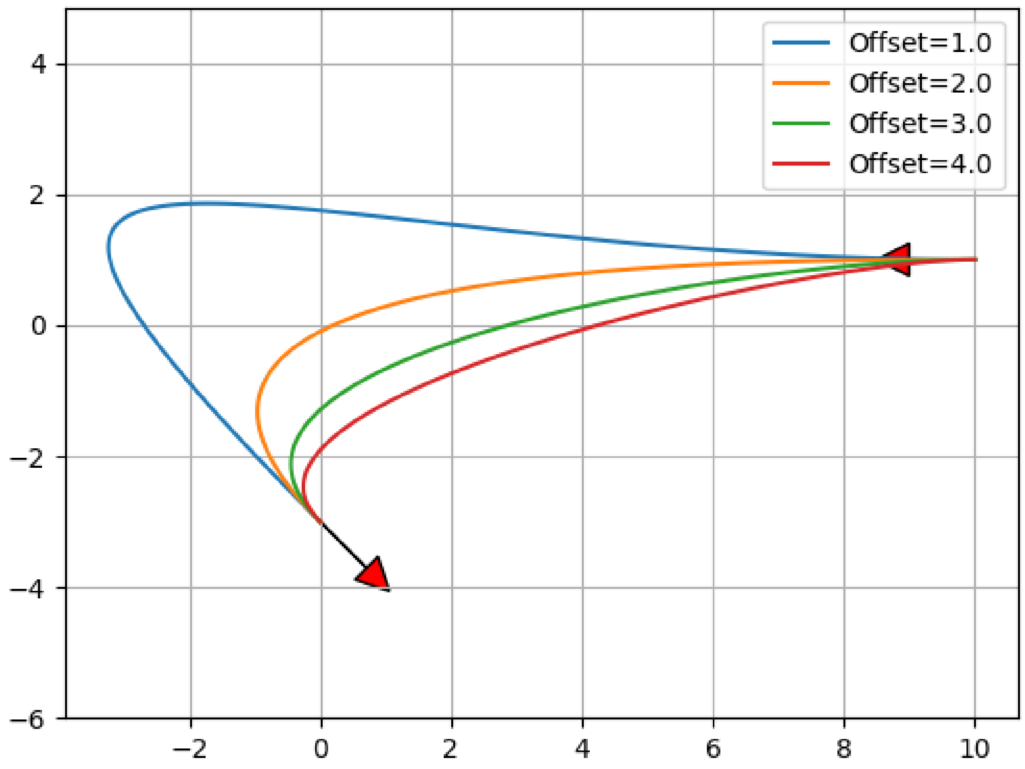
A sample code of Bezier path planning.

It is based on 4 control points Beier path.



If you change the offset distance from start and end point,

You can get different Beizer course:



Ref:

- Continuous Curvature Path Generation Based on Bezier Curves for Autonomous Vehicles

6.12 Quintic polynomials planning

Motion planning with quintic polynomials.

It can calculate 2D path, velocity, and acceleration profile based on quintic polynomials.

Ref:

- Local Path Planning And Motion Control For Agv In Positioning

6.13 Dubins path planning

A sample code for Dubins path planning.

Ref:

- Dubins path - Wikipedia

6.14 Reeds Shepp planning

A sample code with Reeds Shepp path planning.

Ref:

- 15.3.2 Reeds-Shepp Curves
- optimal paths for a car that goes both forwards and backwards
- ghliu/pyReedsShepp: Implementation of Reeds Shepp curve.

6.15 LQR based path planning

A sample code using LQR based path planning for double integrator model.

6.16 Optimal Trajectory in a Frenet Frame

This is optimal trajectory generation in a Frenet Frame.

The cyan line is the target course and black crosses are obstacles.

The red line is predicted path.

Ref:

- Optimal Trajectory Generation for Dynamic Street Scenarios in a Frenet Frame
- Optimal trajectory generation for dynamic street scenarios in a Frenet Frame

CHAPTER 7

Path tracking

7.1 move to a pose control

This is a simulation of moving to a pose control

Ref:

- P. I. Corke, "Robotics, Vision and Control" | SpringerLink p102

7.2 Pure pursuit tracking

Path tracking simulation with pure pursuit steering control and PID speed control.

The red line is a target course, the green cross means the target point for pure pursuit control, the blue line is the tracking.

Ref:

- A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles

7.3 Stanley control

Path tracking simulation with Stanley steering control and PID speed control.

Ref:

- Stanley: The robot that won the DARPA grand challenge

- Automatic Steering Methods for Autonomous Automobile Path Tracking

7.4 Rear wheel feedback control

Path tracking simulation with rear wheel feedback steering control and PID speed control.

Ref:

- A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles

7.5 Linear–quadratic regulator (LQR) steering control

Path tracking simulation with LQR steering control and PID speed control.

Ref:

- ApolloAuto/apollo: An open autonomous driving platform

7.6 Linear–quadratic regulator (LQR) speed and steering control

Path tracking simulation with LQR speed and steering control.

Ref:

- Towards fully autonomous driving: Systems and algorithms - IEEE Conference Publication

7.7 Model predictive speed and steering control

Fig. 1: Model predictive speed and steering control

code:

PythonRobotics/model_predictive_speed_and_steer_control.py at master · AtsushiSakai/PythonRobotics

This is a path tracking simulation using model predictive control (MPC).

The MPC controller controls vehicle speed and steering base on linealized model.

This code uses cvxpy as an optimization modeling tool.

- Welcome to CVXPY 1.0 — CVXPY 1.0.6 documentation

7.7.1 MPC modeling

State vector is:

$$z = [x, y, v, \phi]$$

x: x-position, y:y-position, v:velocity, ϕ: yaw angle

Input vector is:

$$u = [a, \delta]$$

a: accellation, δ: steering angle

The MPC cotroller minimize this cost function for path tracking:

$$min\ Q_f(z_{T,ref} - z_T)^2 + Q\Sigma(z_{t,ref} - z_t)^2 + R\Sigma {u_t}^2 + R_d\Sigma(u_{t+1} - u_t)^2$$

z_ref come from target path and speed.

subject to:

- Linearlied vehicle model

$$z_{t+1} = Az_t + Bu + C$$

- Maximum steering speed

$$|u_{t+1} - u_t| < du_{max}$$

- Maximum steering angle

$$|u_t| < u_{max}$$

- Initial state

$$z_0 = z_{0,ob}$$

- Maximum and minimum speed

$$v_{min} < v_t < v_{max}$$

- Maximum and minimum input

$$u_{min} < u_t < u_{max}$$

This is implemented at

PythonRobotics/model_predictive_speed_and_steer_control.py at f51a73f47cb922a12659f8ce2d544c347a2a8156 · AtsushiSakai/PythonRobotics

7.7.2 Vehicle model linearization

Vehicle model is

$$\dot{x} = vcos(\phi)$$

$$\dot{y} = vsin((\phi)$$

$$\dot{v} = a$$

$$\dot{\phi} = \frac{vtan(\delta)}{L}$$

ODE is

$$\dot{z} = \frac{\partial}{\partial z} z = f(z, u) = A'z + B'u$$

where

$$A' = \begin{bmatrix} \frac{\partial}{\partial x} vcos(\phi) & \frac{\partial}{\partial y} vcos(\phi) & \frac{\partial}{\partial v} vcos(\phi) & \frac{\partial}{\partial \phi} vcos(\phi) \\ \frac{\partial}{\partial x} vsin(\phi) & \frac{\partial}{\partial y} vsin(\phi) & \frac{\partial}{\partial v} vsin(\phi) & \frac{\partial}{\partial \phi} vsin(\phi) \\ \frac{\partial}{\partial x} a & \frac{\partial}{\partial y} a & \frac{\partial}{\partial v} a & \frac{\partial}{\partial \phi} a \\ \frac{\partial}{\partial x} \frac{vtan(\delta)}{L} & \frac{\partial}{\partial y} \frac{vtan(\delta)}{L} & \frac{\partial}{\partial v} \frac{vtan(\delta)}{L} & \frac{\partial}{\partial \phi} \frac{vtan(\delta)}{L} \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 0 & cos(\bar{\phi}) & -\bar{v}sin(\bar{\phi}) \\ 0 & 0 & sin(\bar{\phi}) & \bar{v}cos(\bar{\phi}) \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{tan(\bar{\delta})}{L} & 0 \end{bmatrix}$$

$$B' = \begin{bmatrix} \frac{\partial}{\partial a} vcos(\phi) & \frac{\partial}{\partial \delta} vcos(\phi) \\ \frac{\partial}{\partial a} vsin(\phi) & \frac{\partial}{\partial \delta} vsin(\phi) \\ \frac{\partial}{\partial a} a & \frac{\partial}{\partial \delta} a \\ \frac{\partial}{\partial a} \frac{vtan(\delta)}{L} & \frac{\partial}{\partial \delta} \frac{vtan(\delta)}{L} \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & \frac{\bar{v}}{Lcos^2(\bar{\delta})} \end{bmatrix}$$

You can get a discrete-time mode with Forward Euler Discretization with sampling time dt.

$$z_{k+1} = z_k + f(z_k, u_k)dt$$

Using first degree Tayer expantion around zbar and ubar

$$z_{k+1} = z_k + (f(\bar{z}, \bar{u}) + A'z_k + B'u_k - A'\bar{z} - B'\bar{u})dt$$

$$z_{k+1} = (I + dtA')z_k + (dtB')u_k + (f(\bar{z}, \bar{u}) - A'\bar{z} - B'\bar{u})dt$$

So,

$$z_{k+1} = Az_k + Bu_k + C$$

where,

$$A = (I + dtA')$$

$$= \begin{bmatrix} 1 & 0 & cos(\bar{\phi})dt & -\bar{v}sin(\bar{\phi})dt \\ 0 & 1 & sin(\bar{\phi})dt & \bar{v}cos(\bar{\phi})dt \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{tan(\bar{\delta})}{L}dt & 1 \end{bmatrix}$$

$$B = dtB'$$

$$= \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ dt & 0 \\ 0 & \frac{\bar{v}}{Lcos^2(\bar{\delta})}dt \end{bmatrix}$$

$$
\begin{aligned}
C &= (f(\bar{z},\bar{u}) - A'\bar{z} - B'\bar{u})dt \\
&= dt(\begin{bmatrix} \bar{v}cos(\bar{\phi}) \\ \bar{v}sin(\bar{\phi}) \\ \bar{a} \\ \frac{\bar{v}tan(\bar{\delta})}{L} \end{bmatrix} - \begin{bmatrix} \bar{v}cos(\bar{\phi}) - \bar{v}sin(\bar{\phi})\bar{\phi} \\ \bar{v}sin(\bar{\phi}) + \bar{v}cos(\bar{\phi})\bar{\phi} \\ 0 \\ \frac{\bar{v}tan(\bar{\delta})}{L} \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ \bar{a} \\ \frac{\bar{v}\bar{\delta}}{Lcos^2(\bar{\delta})} \end{bmatrix}) \\
&= \begin{bmatrix} \bar{v}sin(\bar{\phi})\bar{\phi}dt \\ -\bar{v}cos(\bar{\phi})\bar{\phi}dt \\ 0 \\ -\frac{\bar{v}\bar{\delta}}{Lcos^2(\bar{\delta})}dt \end{bmatrix}
\end{aligned}
$$

This equation is implemented at

PythonRobotics/model_predictive_speed_and_steer_control.py at eb6d1cbe6fc90c7be9210bf153b3a04f177cc138 · AtsushiSakai/PythonRobotics

7.7.3 Reference

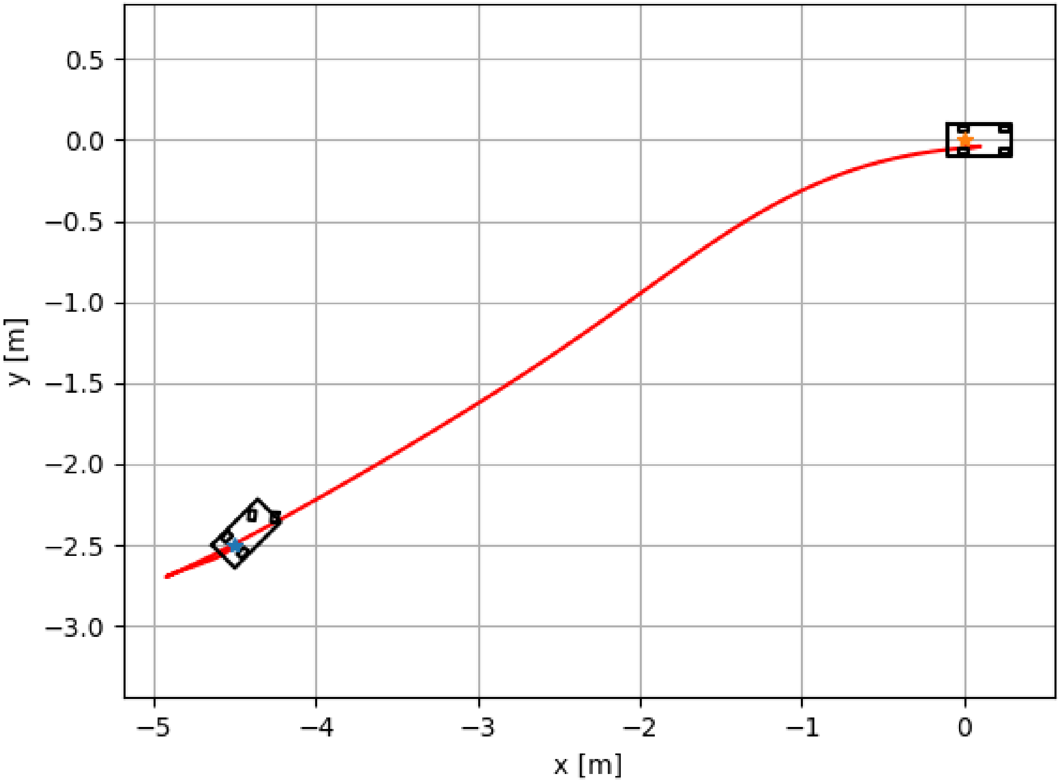
- Vehicle Dynamics and Control | Rajesh Rajamani | Springer
- MPC Course Material - MPC Lab @ UC-Berkeley

Ref:

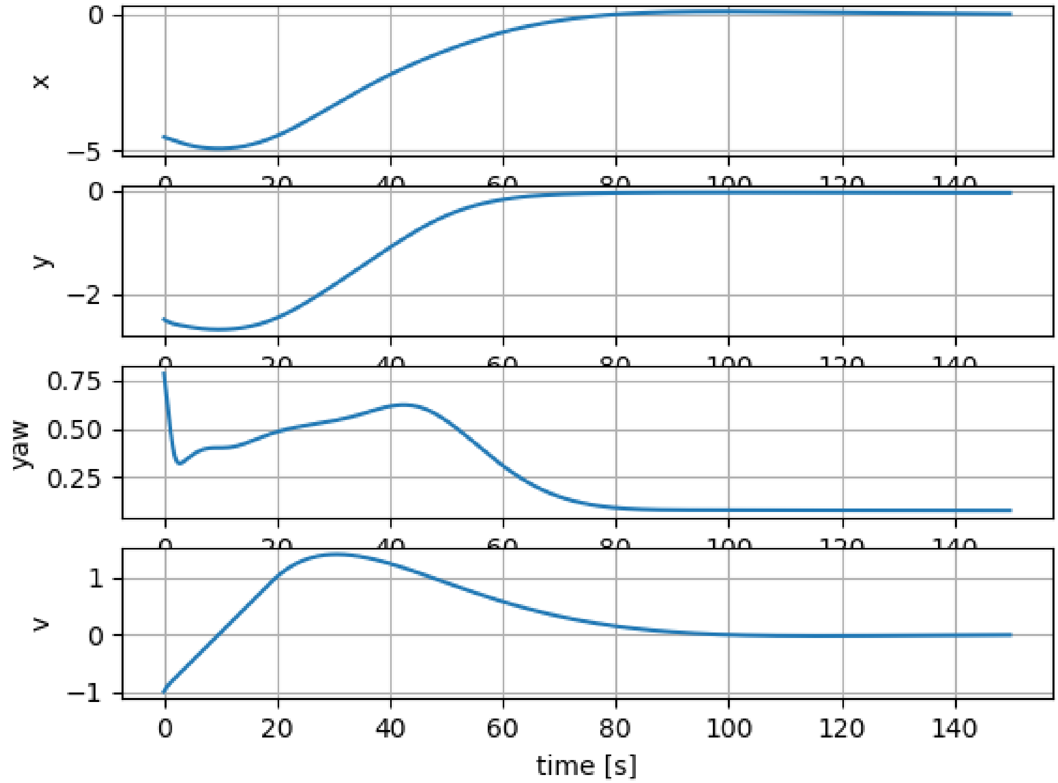
- notebook

7.8 Nonlinear Model Predictive Control with C-GMRES

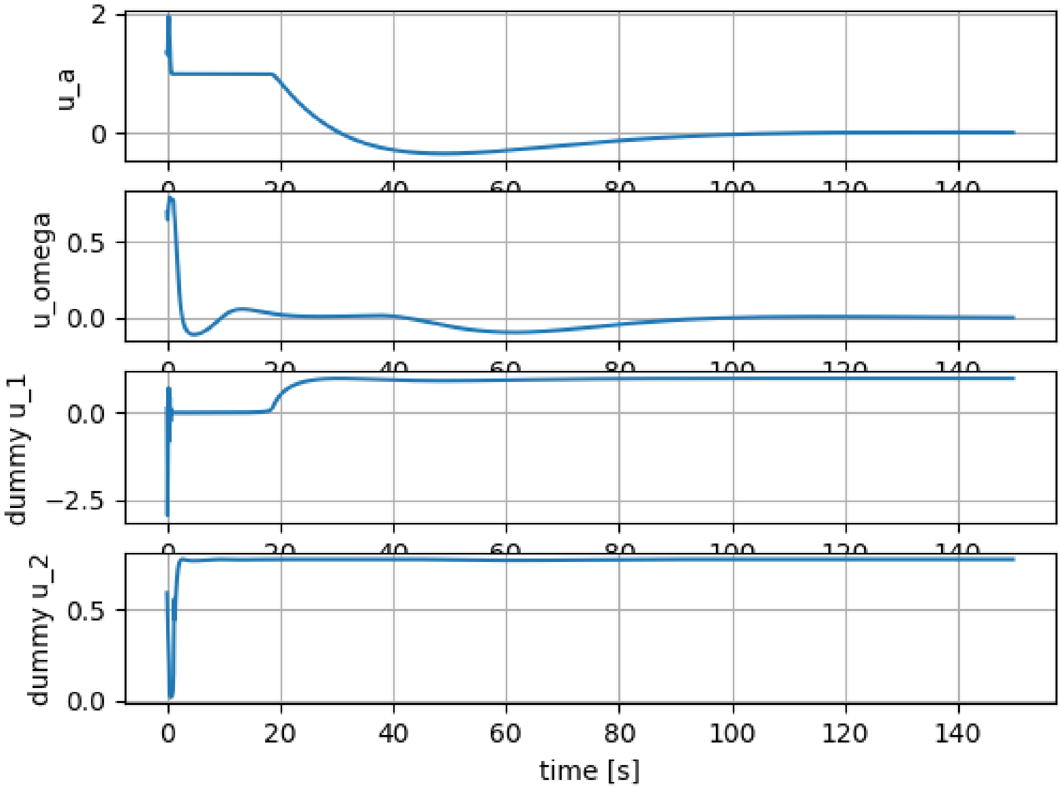
```
from IPython.display import Image
Image(filename="Figure_4.png",width=600)
```



```
Image(filename="Figure_1.png",width=600)
```



```
Image(filename="Figure_2.png",width=600)
```



```
Image(filename="Figure_3.png",width=600)
```

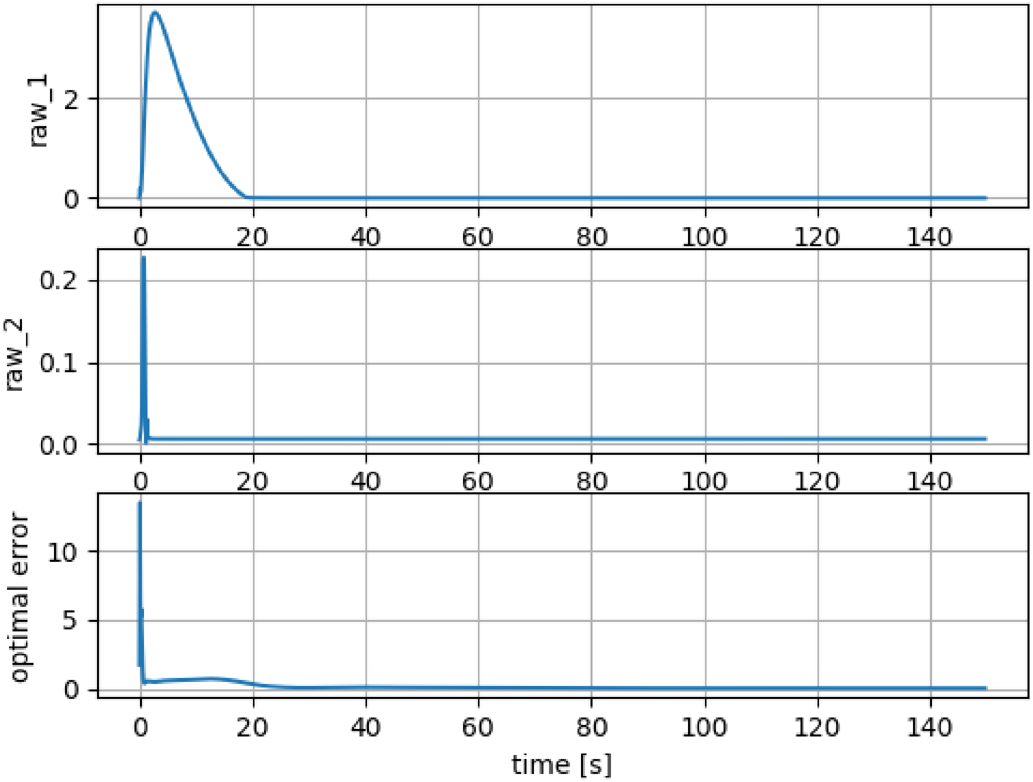


Fig. 2: gif

7.8.1 Mathematical Formulation

Motion model is

$$\dot{x} = v cos\theta$$

$$\dot{y} = v sin\theta$$

$$\dot{\theta} = \frac{v}{WB} sin(u_\delta)$$

(tan is not good for optimization)

$$\dot{v} = u_a$$

Cost function is

$$J = \frac{1}{2}(u_a^2 + u_\delta^2) - \phi_a d_a - \phi_\delta d_\delta$$

Input constraints are

$$|u_a| \leq u_{a,max}$$

$$|u_\delta| \leq u_{\delta,max}$$

So, Hamiltonianis

$$\begin{aligned} J = \frac{1}{2}(u_a^2 + u_\delta^2) - \phi_a d_a - \phi_\delta d_\delta \\ +\lambda_1 v cos\theta + \lambda_2 v sin\theta + \lambda_3 \frac{v}{WB} sin(u_\delta) + \lambda_4 u_a \\ +\rho_1(u_a^2 + d_a^2 + u_{a,max}^2) + \rho_2(u_\delta^2 + d_\delta^2 + u_{\delta,max}^2) \end{aligned}$$

Partial differential equations of the Hamiltonian are:

$$\frac{\partial H}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial H}{\partial x} = 0 \\ \frac{\partial H}{\partial y} = 0 \\ \frac{\partial H}{\partial \theta} = -\lambda_1 v sin\theta + \lambda_2 v cos\theta \\ \frac{\partial H}{\partial v} = -\lambda_1 cos\theta + \lambda_2 sin\theta + \lambda_3 \frac{1}{WB} sin(u_\delta) \end{bmatrix}$$

$$\frac{\partial H}{\partial \mathbf{u}} = \begin{bmatrix} \frac{\partial H}{\partial u_a} = u_a + \lambda_4 + 2\rho_1 u_a \\ \frac{\partial H}{\partial u_\delta} = u_\delta + \lambda_3 \frac{v}{WB} cos(u_\delta) + 2\rho_2 u_\delta \\ \frac{\partial H}{\partial d_a} = -\phi_a + 2\rho_1 d_a \\ \frac{\partial H}{\partial d_\delta} = -\phi_\delta + 2\rho_2 d_\delta \end{bmatrix}$$

7.8.2 Ref

- Shunichi09/nonlinear_control: Implementing the nonlinear model predictive control, sliding mode control
- CGMRESpython - Qiita

CHAPTER 8

Arm Navigation

8.1 Two joint arm to point control

Fig. 1: TwoJointArmToPointControl

This is two joint arm to a point control simulation.

This is a interactive simulation.

You can set the goal position of the end effector with left-click on the ploting area.

8.1.1 Inverse Kinematics for a Planar Two-Link Robotic Arm

A classic problem with robotic arms is getting the end-effector, the mechanism at the end of the arm responsible for manipulating the environment, to where you need it to be. Maybe the end-effector is a gripper and maybe you want to pick up an object and maybe you know where that object is relative to the robot - but you cannot tell the end-effector where to go directly. Instead, you have to determine the joint angles that get the end-effector to where you want it to be. This problem is known as inverse kinematics.

Credit for this solution goes to: https://robotacademy.net.au/lesson/inverse-kinematics-for-a-2-joint-robot-arm-using-geometry/

First, let's define a class to make plotting our arm easier.

```
%matplotlib inline
from math import cos, sin
import numpy as np
import matplotlib.pyplot as plt

class TwoLinkArm:
    def __init__(self, joint_angles=[0, 0]):
        self.shoulder = np.array([0, 0])
        self.link_lengths = [1, 1]
```

(continues on next page)

(continued from previous page)

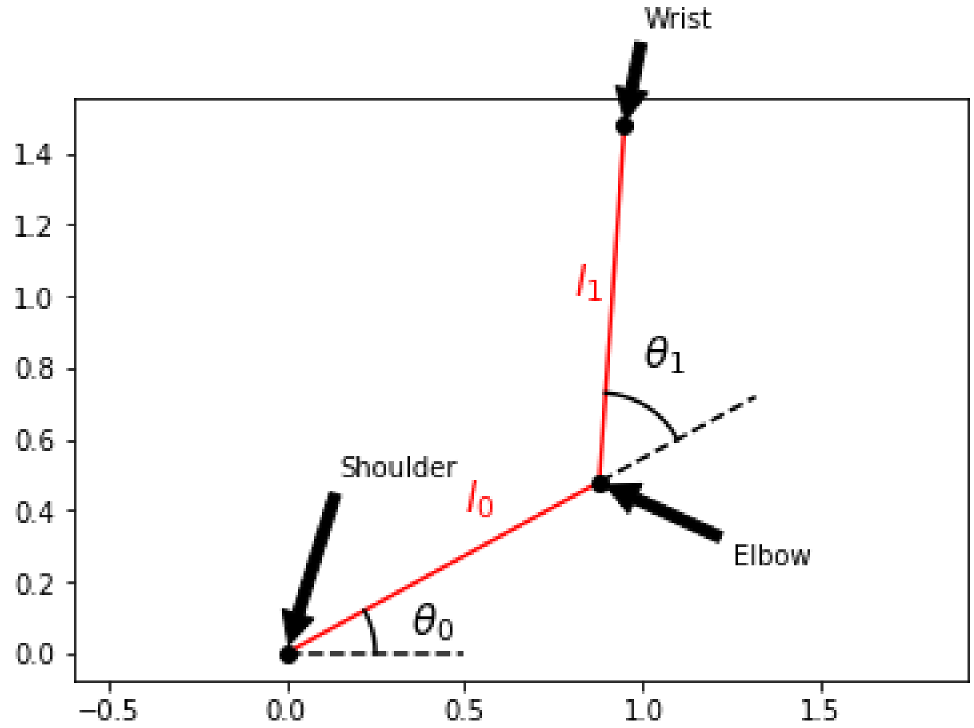
```
        self.update_joints(joint_angles)

    def update_joints(self, joint_angles):
        self.joint_angles = joint_angles
        self.forward_kinematics()

    def forward_kinematics(self):
        theta0 = self.joint_angles[0]
        theta1 = self.joint_angles[1]
        l0 = self.link_lengths[0]
        l1 = self.link_lengths[1]
        self.elbow = self.shoulder + np.array([l0*cos(theta0), l0*sin(theta0)])
        self.wrist = self.elbow + np.array([l1*cos(theta0 + theta1), l1*sin(theta0 +␣
↪theta1)])

    def plot(self):
        plt.plot([self.shoulder[0], self.elbow[0]],
                 [self.shoulder[1], self.elbow[1]],
                 'r-')
        plt.plot([self.elbow[0], self.wrist[0]],
                 [self.elbow[1], self.wrist[1]],
                 'r-')
        plt.plot(self.shoulder[0], self.shoulder[1], 'ko')
        plt.plot(self.elbow[0], self.elbow[1], 'ko')
        plt.plot(self.wrist[0], self.wrist[1], 'ko')
```

Let's also define a function to make it easier to draw an angle on our diagram.

```
from math import sqrt

def transform_points(points, theta, origin):
    T = np.array([[cos(theta), -sin(theta), origin[0]],
                  [sin(theta), cos(theta), origin[1]],
                  [0, 0, 1]])
    return np.matmul(T, np.array(points))

def draw_angle(angle, offset=0, origin=[0, 0], r=0.5, n_points=100):
        x_start = r*cos(angle)
        x_end = r
        dx = (x_end - x_start)/(n_points-1)
        coords = [[0 for _ in range(n_points)] for _ in range(3)]
        x = x_start
        for i in range(n_points-1):
            y = sqrt(r**2 - x**2)
            coords[0][i] = x
            coords[1][i] = y
            coords[2][i] = 1
            x += dx
        coords[0][-1] = r
        coords[2][-1] = 1
        coords = transform_points(coords, offset, origin)
        plt.plot(coords[0], coords[1], 'k-')
```

Okay, we now have a TwoLinkArm class to help us draw the arm, which we'll do several times during our derivation. Notice there is a method called forward_kinematics - forward kinematics specifies the end-effector position given the joint angles and link lengths. Forward kinematics is easier than inverse kinematics.

```
arm = TwoLinkArm()

theta0 = 0.5
theta1 = 1

arm.update_joints([theta0, theta1])
arm.plot()

def label_diagram():
    plt.plot([0, 0.5], [0, 0], 'k--')
    plt.plot([arm.elbow[0], arm.elbow[0]+0.5*cos(theta0)],
             [arm.elbow[1], arm.elbow[1]+0.5*sin(theta0)],
             'k--')

    draw_angle(theta0, r=0.25)
    draw_angle(theta1, offset=theta0, origin=[arm.elbow[0], arm.elbow[1]], r=0.25)

    plt.annotate("$l_0$", xy=(0.5, 0.4), size=15, color="r")
    plt.annotate("$l_1$", xy=(0.8, 1), size=15, color="r")

    plt.annotate(r"$\theta_0$", xy=(0.35, 0.05), size=15)
    plt.annotate(r"$\theta_1$", xy=(1, 0.8), size=15)

label_diagram()

plt.annotate("Shoulder", xy=(arm.shoulder[0], arm.shoulder[1]), xytext=(0.15, 0.5),
    arrowprops=dict(facecolor='black', shrink=0.05))
plt.annotate("Elbow", xy=(arm.elbow[0], arm.elbow[1]), xytext=(1.25, 0.25),
    arrowprops=dict(facecolor='black', shrink=0.05))
plt.annotate("Wrist", xy=(arm.wrist[0], arm.wrist[1]), xytext=(1, 1.75),
    arrowprops=dict(facecolor='black', shrink=0.05))

plt.axis("equal")

plt.show()
```



It's common to name arm joints anatomically, hence the names shoulder, elbow, and wrist. In this example, the wrist is not itself a joint, but we can consider it to be our end-effector. If we constrain the shoulder to the origin, we can write the forward kinematics for the elbow and the wrist.

$$elbow_x = l_0 \cos(\theta_0)$$
$$elbow_y = l_0 \sin(\theta_0)$$

$$wrist_x = elbow_x + l_1 \cos(\theta_0 + \theta_1) = l_0 \cos(\theta_0) + l_1 \cos(\theta_0 + \theta_1)$$
$$wrist_y = elbow_y + l_1 \sin(\theta_0 + \theta_1) = l_0 \sin(\theta_0) + l_1 \sin(\theta_0 + \theta_1)$$

Since the wrist is our end-effector, let's just call its coordinates x and y. The forward kinematics for our end-effector is then

$$x = l_0 \cos(\theta_0) + l_1 \cos(\theta_0 + \theta_1)$$
$$y = l_0 \sin(\theta_0) + l_1 \sin(\theta_0 + \theta_1)$$

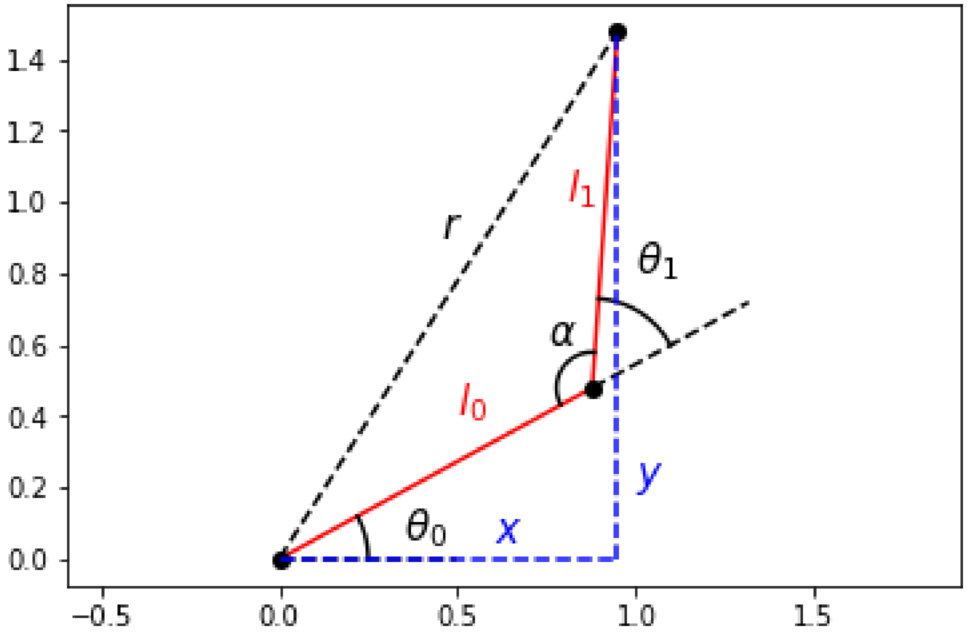
A first attempt to find the joint angles θ_0 and θ_1 that would get our end-effector to the desired coordinates x and y might be to try solving the forward kinematics for θ_0 and θ_1, but that would be the wrong move. An easier path involves going back to the geometry of the arm.

```
from math import pi

arm.plot()
label_diagram()
```

(continues on next page)

(continued from previous page)

```
plt.plot([0, arm.wrist[0]],
         [0, arm.wrist[1]],
         'k--')

plt.plot([arm.wrist[0], arm.wrist[0]],
         [0, arm.wrist[1]],
         'b--')
plt.plot([0, arm.wrist[0]],
         [0, 0],
         'b--')

plt.annotate("$x$", xy=(0.6, 0.05), size=15, color="b")
plt.annotate("$y$", xy=(1, 0.2), size=15, color="b")
plt.annotate("$r$", xy=(0.45, 0.9), size=15)
plt.annotate(r"$\alpha$", xy=(0.75, 0.6), size=15)

alpha = pi-theta1
draw_angle(alpha, offset=theta0+theta1, origin=[arm.elbow[0], arm.elbow[1]], r=0.1)

plt.axis("equal")
plt.show()
```



The distance from the end-effector to the robot base (shoulder joint) is r and can be written in terms of the end-effector position using the Pythagorean Theorem.

$r^2 = x^2 + y^2$

Then, by the law of cosines, $r2$ can also be written as:

$r^2 = l_0^2 + l_1^2 - 2l_0l_1\cos(\alpha)$

Because α can be written as $\pi - \theta_1$, we can relate the desired end-effector position to one of our joint angles, θ_1.

$x^2 + y^2 = l_0^2 + l_1^2 - 2l_0l_1\cos(\alpha)$

$x^2 + y^2 = l_0^2 + l_1^2 - 2l_0l_1\cos(\pi - \theta_1)$

$2l_0l_1\cos(\pi - \theta_1) = l_0^2 + l_1^2 - x^2 - y^2$

$\cos(\pi - \theta_1) = \frac{l_0^2+l_1^2-x^2-y^2}{2l_0l_1}$

$\cos(\pi - \theta_1) = -cos(\theta_1)$ is a trigonometric identity, so we can also write

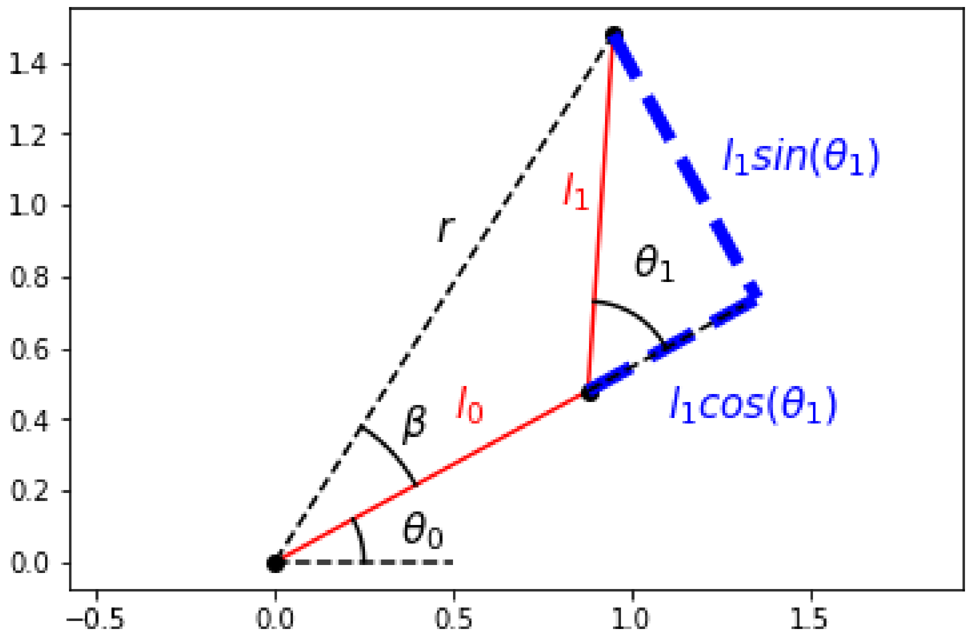
$\cos(\theta_1) = \frac{x^2+y^2-l_0^2-l_1^2}{2l_0l_1}$

which leads us to an equation for θ_1 in terms of the link lengths and the desired end-effector position!

$\theta_1 = \cos^{-1}(\frac{x^2+y^2-l_0^2-l_1^2}{2l_0l_1})$

This is actually one of two possible solutions for θ_1, but we'll ignore the other possibility for now. This solution will lead us to the "arm-down" configuration of the arm, which is what's shown in the diagram. Now we'll derive an equation for θ_0 that depends on this value of θ_1.

```
from math import atan2

arm.plot()
plt.plot([0, arm.wrist[0]],
         [0, arm.wrist[1]],
         'k--')

p = 1 + cos(theta1)
plt.plot([arm.elbow[0], p*cos(theta0)],
         [arm.elbow[1], p*sin(theta0)],
         'b--', linewidth=5)
plt.plot([arm.wrist[0], p*cos(theta0)],
         [arm.wrist[1], p*sin(theta0)],
         'b--', linewidth=5)

beta = atan2(arm.wrist[1], arm.wrist[0])-theta0
draw_angle(beta, offset=theta0, r=0.45)

plt.annotate(r"$\beta$", xy=(0.35, 0.35), size=15)
plt.annotate("$r$", xy=(0.45, 0.9), size=15)
plt.annotate(r"$l_1sin(\theta_1)$",xy=(1.25, 1.1), size=15, color="b")
plt.annotate(r"$l_1cos(\theta_1)$",xy=(1.1, 0.4), size=15, color="b")

label_diagram()

plt.axis("equal")

plt.show()
```

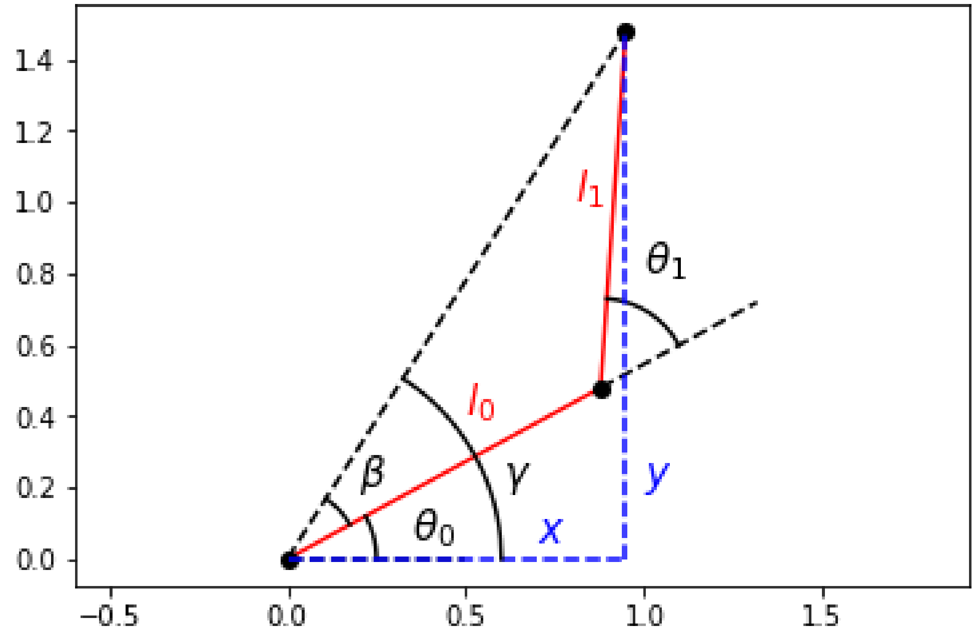


Consider the angle between the displacement vector r and the first link l_0; let's call it β. If we extend the first link to include the component of the second link in the same direction as the first, we form a right triangle with components $l_0 + l_1 cos(\theta_1)$ and $l_1 sin(\theta_1)$, allowing us to express β as

$\beta = \tan^{-1}(\frac{l_1 \sin(\theta_1)}{l_0 + l_1 \cos(\theta_1)})$

We now have an expression for this angle β in terms of one of our arm's joint angles. Now, can we relate β to θ_0? Yes!

```
arm.plot()
label_diagram()
plt.plot([0, arm.wrist[0]],
         [0, arm.wrist[1]],
         'k--')

plt.plot([arm.wrist[0], arm.wrist[0]],
         [0, arm.wrist[1]],
         'b--')
plt.plot([0, arm.wrist[0]],
         [0, 0],
         'b--')

gamma = atan2(arm.wrist[1], arm.wrist[0])
draw_angle(beta, offset=theta0, r=0.2)
draw_angle(gamma, r=0.6)

plt.annotate("$x$", xy=(0.7, 0.05), size=15, color="b")
plt.annotate("$y$", xy=(1, 0.2), size=15, color="b")
plt.annotate(r"$\beta$", xy=(0.2, 0.2), size=15)
plt.annotate(r"$\gamma$", xy=(0.6, 0.2), size=15)

plt.axis("equal")
plt.show()
```



Our first joint angle θ_0 added to β gives us the angle between the positive x-axis and the displacement vector r; let's call this angle γ.

$\gamma = \theta_0 + \beta$

θ_0, our remaining joint angle, can then be expressed as

$\theta_0 = \gamma - \beta$

We already know β. γ is simply the inverse tangent of $\frac{y}{x}$, so we have an equation of θ_0!

$\theta_0 = \tan^{-1}(\frac{y}{x}) - \tan^{-1}(\frac{l_1 \sin(\theta_1)}{l_0 + l_1 \cos(\theta_1)})$

We now have the inverse kinematics for a planar two-link robotic arm. If you're planning on implementing this in a programming language, it's best to use the atan2 function, which is included in most math libraries and correctly accounts for the signs of y and x. Notice that θ_1 must be calculated before θ_0.

$\theta_1 = \cos^{-1}(\frac{x^2 + y^2 - l_0^2 - l_1^2}{2 l_0 l_1})$
$\theta_0 = atan2(y, x) - atan2(l_1 \sin(\theta_1), l_0 + l_1 \cos(\theta_1))$

8.2 N joint arm to point control

N joint arm to a point control simulation.

This is a interactive simulation.

You can set the goal position of the end effector with left-click on the ploting area.

In this simulation N = 10, however, you can change it.

8.3 Arm navigation with obstacle avoidance

Arm navigation with obstacle avoidance simulation.

CHAPTER 9

Aerial Navigation

9.1 Drone 3d trajectory following

This is a 3d trajectory following simulation for a quadrotor.

9.2 rocket powered landing

9.2.1 Simulation

```
from IPython.display import Image
Image(filename="figure.png",width=600)
from IPython.display import display, HTML

display(HTML(data="""
<style>
    div#notebook-container    { width: 95%; }
    div#menubar-container     { width: 65%; }
    div#maintoolbar-container { width: 99%; }
</style>
"""))
```

Fig. 1: gif

9.2.2 Equation generation

```
import sympy as sp
import numpy as np
from IPython.display import display
sp.init_printing(use_latex='mathjax')
```

```
# parameters
# Angular moment of inertia
J_B = 1e-2 * np.diag([1., 1., 1.])

# Gravity
g_I = np.array((-1, 0., 0.))

# Fuel consumption
alpha_m = 0.01

# Vector from thrust point to CoM
r_T_B = np.array([-1e-2, 0., 0.])


def dir_cosine(q):
        return np.matrix([
            [1 - 2 * (q[2] ** 2 + q[3] ** 2), 2 * (q[1] * q[2] +
                                                   q[0] * q[3]), 2 * (q[1] * q[3] -␣
↪q[0] * q[2])],
            [2 * (q[1] * q[2] - q[0] * q[3]), 1 - 2 *
             (q[1] ** 2 + q[3] ** 2), 2 * (q[2] * q[3] + q[0] * q[1])],
            [2 * (q[1] * q[3] + q[0] * q[2]), 2 * (q[2] * q[3] -
                                                   q[0] * q[1]), 1 - 2 * (q[1] ** 2 +␣
↪q[2] ** 2)]
        ])

def omega(w):
        return np.matrix([
            [0, -w[0], -w[1], -w[2]],
            [w[0], 0, w[2], -w[1]],
            [w[1], -w[2], 0, w[0]],
            [w[2], w[1], -w[0], 0],
        ])

def skew(v):
    return np.matrix([
            [0, -v[2], v[1]],
            [v[2], 0, -v[0]],
            [-v[1], v[0], 0]
        ])
```

```
f = sp.zeros(14, 1)

x = sp.Matrix(sp.symbols(
    'm rx ry rz vx vy vz q0 q1 q2 q3 wx wy wz', real=True))
u = sp.Matrix(sp.symbols('ux uy uz', real=True))

g_I = sp.Matrix(g_I)
r_T_B = sp.Matrix(r_T_B)
J_B = sp.Matrix(J_B)
```

(continues on next page)

(continued from previous page)

```
C_B_I = dir_cosine(x[7:11, 0])
C_I_B = C_B_I.transpose()

f[0, 0] = - alpha_m * u.norm()
f[1:4, 0] = x[4:7, 0]
f[4:7, 0] = 1 / x[0, 0] * C_I_B * u + g_I
f[7:11, 0] = 1 / 2 * omega(x[11:14, 0]) * x[7: 11, 0]
f[11:14, 0] = J_B ** -1 * \
    (skew(r_T_B) * u - skew(x[11:14, 0]) * J_B * x[11:14, 0])
```

```
display(sp.simplify(f)) # f
```

$$\begin{bmatrix} -0.01\sqrt{ux^2+uy^2+uz^2} \\ vx \\ vy \\ vz \\ \frac{-1.0m-ux\left(2q_2^2+2q_3^2-1\right)-2uy(q_0q_3-q_1q_2)+2uz(q_0q_2+q_1q_3)}{m} \\ \frac{2ux(q_0q_3+q_1q_2)-uy\left(2q_1^2+2q_3^2-1\right)-2uz(q_0q_1-q_2q_3)}{m} \\ \frac{-2ux(q_0q_2-q_1q_3)+2uy(q_0q_1+q_2q_3)-uz\left(2q_1^2+2q_2^2-1\right)}{m} \\ -0.5q_1wx-0.5q_2wy-0.5q_3wz \\ 0.5q_0wx+0.5q_2wz-0.5q_3wy \\ 0.5q_0wy-0.5q_1wz+0.5q_3wx \\ 0.5q_0wz+0.5q_1wy-0.5q_2wx \\ 0 \\ 1.0uz \\ -1.0uy \end{bmatrix}$$

```
display(sp.simplify(f.jacobian(x)))# A
```

$$\left[\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \frac{ux\left(2q_2^2+2q_3^2-1\right)+2uy(q_0q_3-q_1q_2)-2uz(q_0q_2+q_1q_3)}{m^2} & 0 & 0 & 0 & 0 & 0 & 0 & \frac{2(q_2uz-q_3uy)}{m} & \frac{2(q_2uy+q_3uz)}{m} & \frac{2(q_0uz+q_1uy-2q_2ux}{m} \\ \frac{-2ux(q_0q_3+q_1q_2)+uy\left(2q_1^2+2q_3^2-1\right)+2uz(q_0q_1-q_2q_3)}{m^2} & 0 & 0 & 0 & 0 & 0 & 0 & \frac{2(-q_1uz+q_3ux)}{m} & \frac{2(-q_0uz-2q_1uy+q_2ux)}{m} & \frac{2(q_1ux+q_3uz)}{m} \\ \frac{2ux(q_0q_2-q_1q_3)-2uy(q_0q_1+q_2q_3)+uz\left(2q_1^2+2q_2^2-1\right)}{m^2} & 0 & 0 & 0 & 0 & 0 & 0 & \frac{2(q_1uy-q_2ux)}{m} & \frac{2(q_0uy-2q_1uz+q_3ux)}{m} & \frac{2(-q_0ux-2q_2uz+q_3u}{m} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.5wx & -0.5wy \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5wx & 0 & 0.5wz \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5wy & -0.5wz & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5wz & 0.5wy & -0.5wx \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}\right.$$

```
sp.simplify(f.jacobian(u)) # B
```

$$\left[\begin{array}{ccc}-\frac{0.01ux}{\sqrt{ux^{2}+uy^{2}+uz^{2}}} & -\frac{0.01uy}{\sqrt{ux^{2}+uy^{2}+uz^{2}}} & -\frac{0.01uz}{\sqrt{ux^{2}+uy^{2}+uz^{2}}}\\ 0 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\\ \frac{-2q_{2}^{2}-2q_{3}^{2}+1}{m} & \frac{2(-q_{0}q_{3}+q_{1}q_{2})}{m} & \frac{2(q_{0}q_{2}+q_{1}q_{3})}{m}\\ \frac{2(q_{0}q_{3}+q_{1}q_{2})}{m} & \frac{-2q_{1}^{2}-2q_{3}^{2}+1}{m} & \frac{2(-q_{0}q_{1}+q_{2}q_{3})}{m}\\ \frac{2(-q_{0}q_{2}+q_{1}q_{3})}{m} & \frac{2(q_{0}q_{1}+q_{2}q_{3})}{m} & \frac{-2q_{1}^{2}-2q_{2}^{2}+1}{m}\\ 0 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 1.0\\ 0 & -1.0 & 0\end{array}\right]$$

9.2.3 Ref

- Python implementation of ‘Successive Convexification for 6-DoF Mars Rocket Powered Landing with Free-Final-Time’ paper by Michael Szmuk and Behçet Açıkmeşe.
- inspired by EmbersArc/SuccessiveConvexificationFreeFinalTime: Implementation of “Successive Convexification for 6-DoF Mars Rocket Powered Landing with Free-Final-Time” https://github.com/EmbersArc/SuccessiveConvexificationFreeFinalTime

CHAPTER 10

Appendix

10.1 KF Basics - Part I

10.1.1 Introduction

What is the need to describe belief in terms of PDF's?

This is because robot environments are stochastic. A robot environment may have cows with Tesla by side. That is a robot and it's environment cannot be deterministically modelled(e.g as a function of something like time t). In the real world sensors are also error prone, and hence there'll be a set of values with a mean and variance that it can take. Hence, we always have to model around some mean and variances associated.

What is Expectation of a Random Variables?

Expectation is nothing but an average of the probabilites

$$\mathbb{E}[X] = \sum_{i=1}^{n} p_i x_i$$

In the continous form,

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x\, f(x)\, dx$$

```
import numpy as np
import random
x=[3,1,2]
p=[0.1,0.3,0.4]
E_x=np.sum(np.multiply(x,p))
print(E_x)
```

```
1.4000000000000001
```

What is the advantage of representing the belief as a unimodal as opposed to multimodal?

Obviously, it makes sense because we can't multiple probabilities to a car moving for two locations. This would be too confusing and the information will not be useful.

10.1.2 Variance, Covariance and Correlation

Variance

Variance is the spread of the data. The mean does'nt tell much **about** the data. Therefore the variance tells us about the **story** about the data meaning the spread of the data.

$$VAR(X) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

```
x=np.random.randn(10)
np.var(x)
```

```
1.0224618077401504
```

Covariance

This is for a multivariate distribution. For example, a robot in 2-D space can take values in both x and y. To describe them, a normal distribution with mean in both x and y is needed.

For a multivariate distribution, mean μ can be represented as a matrix,

$$\mu = \begin{bmatrix}\mu_1\\ \mu_2\\ \vdots\\ \mu_n\end{bmatrix}$$

Similarly, variance can also be represented.

But an important concept is that in the same way as every variable or dimension has a variation in its values, it is also possible that there will be values on how they **together vary**. This is also a measure of how two datasets are related to each other or **correlation**.

For example, as height increases weight also generally increases. These variables are correlated. They are positively correlated because as one variable gets larger so does the other.

We use a **covariance matrix** to denote covariances of a multivariate normal distribution:

$$\Sigma = \begin{bmatrix}\sigma_1^2 & \sigma_{12} & \cdots & \sigma_{1n}\\ \sigma_{21} & \sigma_2^2 & \cdots & \sigma_{2n}\\ \vdots & \vdots & \ddots & \vdots\\ \sigma_{n1} & \sigma_{n2} & \cdots & \sigma_n^2\end{bmatrix}$$

Diagonal - Variance of each variable associated.

Off-Diagonal - covariance between ith and jth variables.

$$VAR(X) = \sigma_x^2 = \frac{1}{n}\sum_{i=1}^{n}(X-\mu)^2$$

$$COV(X,Y) = \sigma_{xy} = \frac{1}{n}\sum_{i=1}^{n}[(X-\mu_x)(Y-\mu_y)]$$

```
x=np.random.random((3,3))
np.cov(x)
```

```
array([[0.08868895, 0.05064471, 0.08855629],
       [0.05064471, 0.06219243, 0.11555291],
       [0.08855629, 0.11555291, 0.21534324]])
```

Covariance taking the data as **sample** with $\frac{1}{N-1}$

```
x_cor=np.random.rand(1,10)
y_cor=np.random.rand(1,10)
np.cov(x_cor,y_cor)
```

```
array([[ 0.1571437 , -0.00766623],
       [-0.00766623,  0.13957621]])
```

Covariance taking the data as **population** with $\frac{1}{N}$

```
np.cov(x_cor,y_cor,bias=1)
```

```
array([[ 0.14142933, -0.0068996 ],
       [-0.0068996 ,  0.12561859]])
```

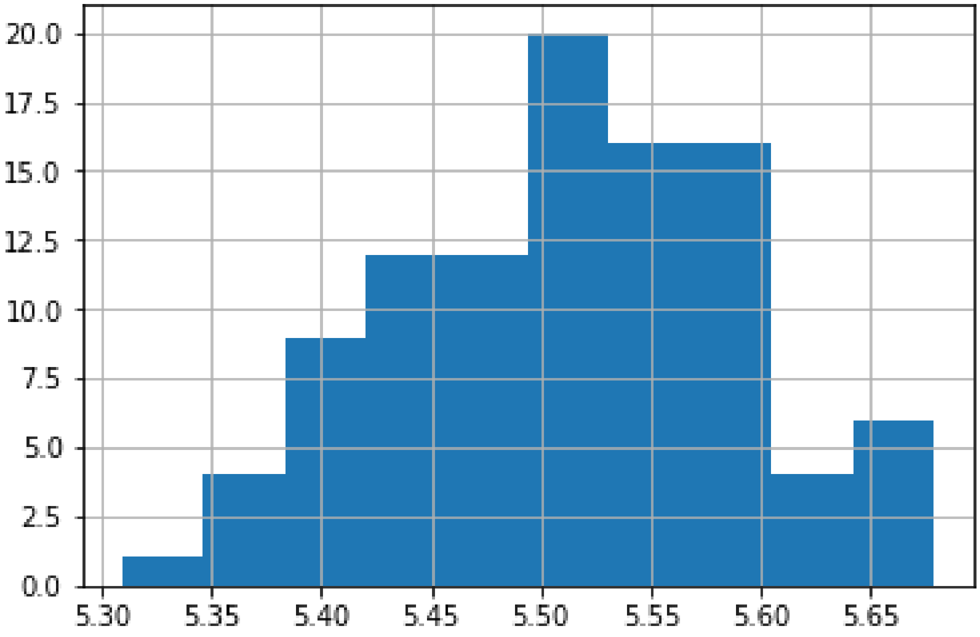
10.1.3 Gaussians

Central Limit Theorem

According to this theorem, the average of n samples of random and independant variables tends to follow a normal distribution as we increase the sample size.(Generally, for n>=30)

```
import matplotlib.pyplot as plt
import random
a=np.zeros((100,))
for i in range(100):
    x=[random.uniform(1,10) for _ in range(1000)]
    a[i]=np.sum(x,axis=0)/1000
plt.hist(a)
```

```
(array([ 1.,  4.,  9., 12., 12., 20., 16., 16.,  4.,  6.]),
 array([5.30943011, 5.34638597, 5.38334183, 5.42029769, 5.45725355,
        5.49420941, 5.53116527, 5.56812114, 5.605077  , 5.64203286,
        5.67898872]),
 <a list of 10 Patch objects>)
```



Gaussian Distribution

A Gaussian is a *continuous probability distribution* that is completely described with two parameters, the mean (μ) and the variance (σ^2). It is defined as:

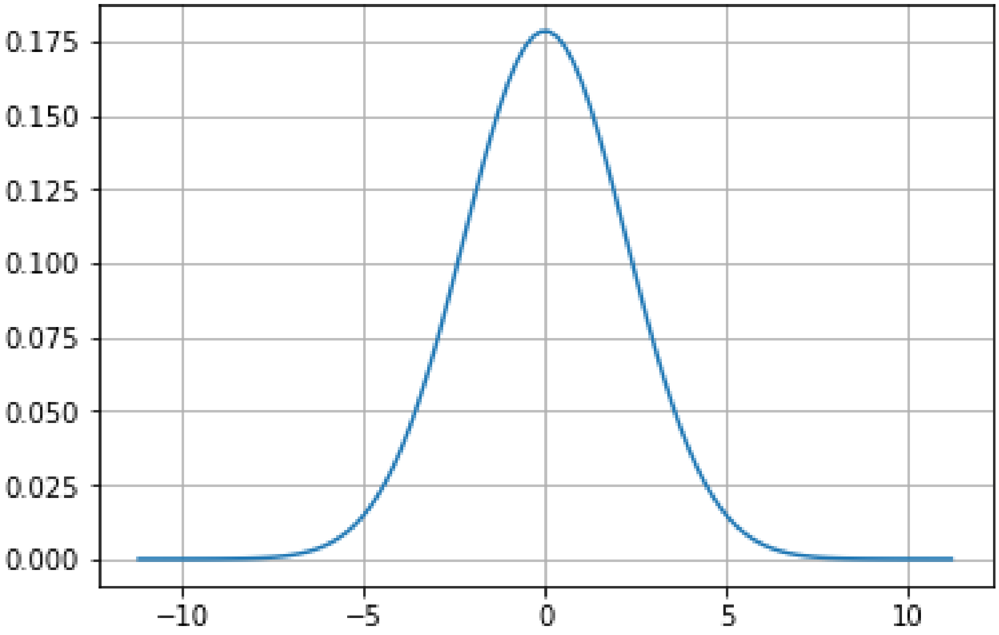
$$f(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\big[-\frac{(x-\mu)^2}{2\sigma^2}\big]$$

$$Rangeis$$

$$[-\inf, \inf]$$

This is just a function of mean(μ) and standard deviation (σ) and what gives the normal distribution the charecteristic **bell curve**.

```
import matplotlib.mlab as mlab
import math
import scipy.stats

mu = 0
variance = 5
sigma = math.sqrt(variance)
x = np.linspace(mu - 5*sigma, mu + 5*sigma, 100)
plt.plot(x,scipy.stats.norm.pdf(x, mu, sigma))
plt.show()
```



Why do we need Gaussian distributions?

Since it becomes really difficult in the real world to deal with multimodal distribution as we cannot put the belief in two seperate location of the robots. This becomes really confusing and in practice impossible to comprehend. Gaussian probability distribution allows us to drive the robots using only one mode with peak at the mean with some variance.

10.1.4 Gaussian Properties

Multiplication

For the measurement update in a Bayes Filter, the algorithm tells us to multiply the Prior P(X_t) and measurement P(Z_t|X_t) to calculate the posterior:

$$P(X \mid Z) = \frac{P(Z \mid X)P(X)}{P(Z)}$$

Here for the numerator, $P(Z \mid X), P(X)$ both are gaussian.

$N(\bar{\mu}, \bar{\sigma}^1)$ and $N(\bar{\mu}, \bar{\sigma}^2)$ are their mean and variances.

New mean is

$$\mu_{\texttt{new}} = \frac{\sigma_z^2 \bar{\mu} + \bar{\sigma}^2 z}{\bar{\sigma}^2 + \sigma_z^2}$$

$$New variance is$$

$$\sigma_{\texttt{new}} = \frac{\sigma_z^2 \bar{\sigma}^2}{\bar{\sigma}^2 + \sigma_z^2}$$

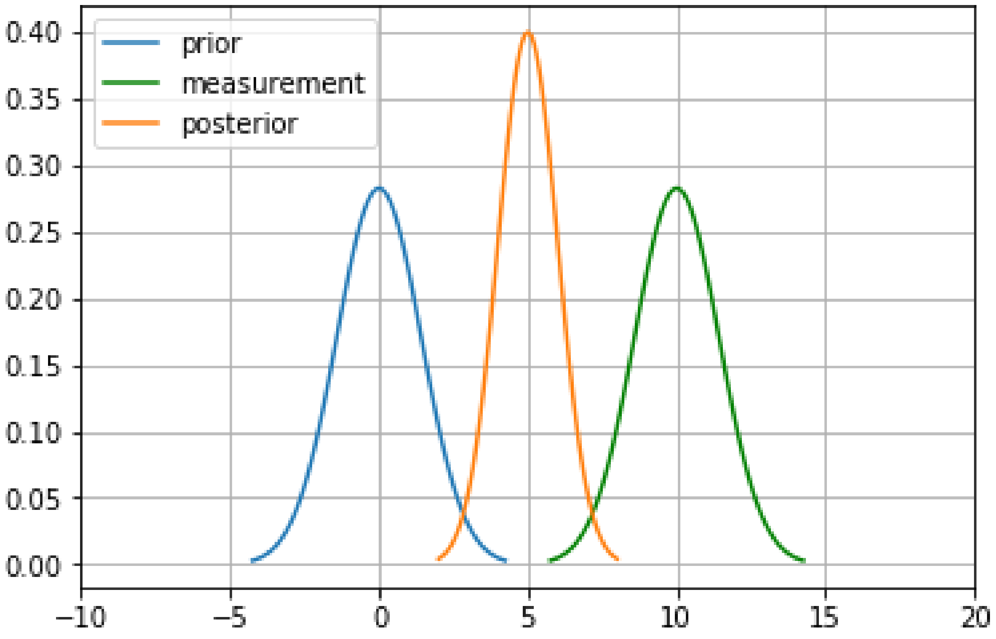
```
import matplotlib.mlab as mlab
import math
mu1 = 0
```

(continues on next page)

(continued from previous page)

```
variance1 = 2
sigma = math.sqrt(variance1)
x1 = np.linspace(mu1 - 3*sigma, mu1 + 3*sigma, 100)
plt.plot(x1,scipy.stats.norm.pdf(x1, mu1, sigma),label='prior')

mu2 = 10
variance2 = 2
sigma = math.sqrt(variance2)
x2 = np.linspace(mu2 - 3*sigma, mu2 + 3*sigma, 100)
plt.plot(x2,scipy.stats.norm.pdf(x2, mu2, sigma),"g-",label='measurement')


mu_new=(mu1*variance2+mu2*variance1)/(variance1+variance2)
print("New mean is at: ",mu_new)
var_new=(variance1*variance2)/(variance1+variance2)
print("New variance is: ",var_new)
sigma = math.sqrt(var_new)
x3 = np.linspace(mu_new - 3*sigma, mu_new + 3*sigma, 100)
plt.plot(x3,scipy.stats.norm.pdf(x3, mu_new, var_new),label="posterior")
plt.legend(loc='upper left')
plt.xlim(-10,20)
plt.show()
```

```
New mean is at:  5.0
New variance is:  1.0
```



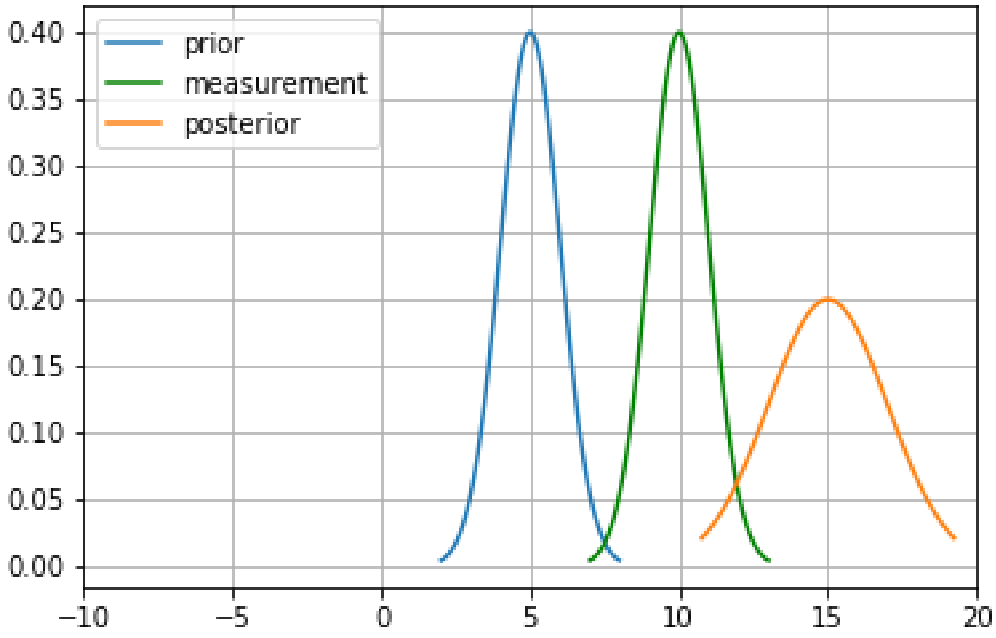
Addition

The motion step involves a case of adding up probability (Since it has to abide the Law of Total Probability). This means their beliefs are to be added and hence two gaussians. They are simply arithmetic additions of the two.

$$\mu_x = \mu_p + \mu_z$$
$$\sigma_x^2 = \sigma_z^2 + \sigma_p^2 \square$$

```
import matplotlib.mlab as mlab
import math
mu1 = 5
variance1 = 1
sigma = math.sqrt(variance1)
x1 = np.linspace(mu1 - 3*sigma, mu1 + 3*sigma, 100)
plt.plot(x1,scipy.stats.norm.pdf(x1, mu1, sigma),label='prior')

mu2 = 10
variance2 = 1
sigma = math.sqrt(variance2)
x2 = np.linspace(mu2 - 3*sigma, mu2 + 3*sigma, 100)
plt.plot(x2,scipy.stats.norm.pdf(x2, mu2, sigma),"g-",label='measurement')


mu_new=mu1+mu2
print("New mean is at: ",mu_new)
var_new=(variance1+variance2)
print("New variance is: ",var_new)
sigma = math.sqrt(var_new)
x3 = np.linspace(mu_new - 3*sigma, mu_new + 3*sigma, 100)
plt.plot(x3,scipy.stats.norm.pdf(x3, mu_new, var_new),label="posterior")
plt.legend(loc='upper left')
plt.xlim(-10,20)
plt.show()
```

```
New mean is at:  15
New variance is:  2
```

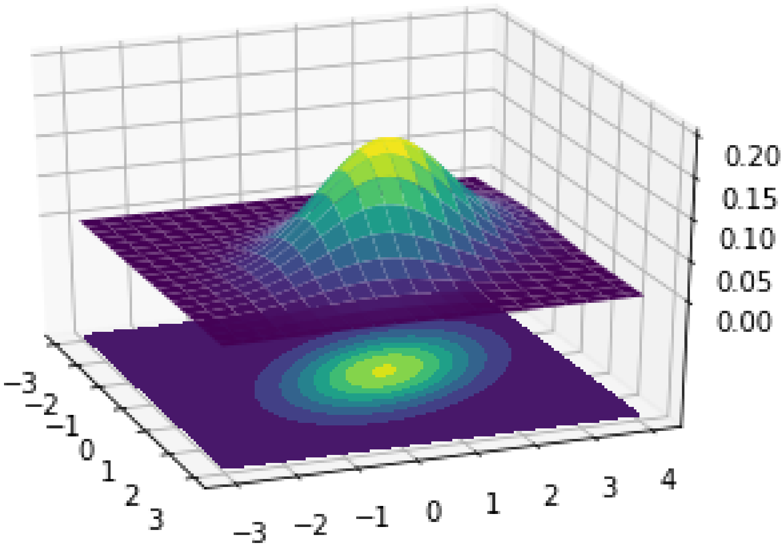


```
#Example from:
#https://scipython.com/blog/visualizing-the-bivariate-gaussian-distribution/
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D

# Our 2-dimensional distribution will be over variables X and Y
N = 60
X = np.linspace(-3, 3, N)
Y = np.linspace(-3, 4, N)
X, Y = np.meshgrid(X, Y)

# Mean vector and covariance matrix
mu = np.array([0., 1.])
Sigma = np.array([[ 1. , -0.5], [-0.5,  1.5]])

# Pack X and Y into a single 3-dimensional array
pos = np.empty(X.shape + (2,))
pos[:, :, 0] = X
pos[:, :, 1] = Y

def multivariate_gaussian(pos, mu, Sigma):
    """Return the multivariate Gaussian distribution on array pos.

    pos is an array constructed by packing the meshed arrays of variables
    x_1, x_2, x_3, ..., x_k into its _last_ dimension.

    """

    n = mu.shape[0]
    Sigma_det = np.linalg.det(Sigma)
    Sigma_inv = np.linalg.inv(Sigma)
    N = np.sqrt((2*np.pi)**n * Sigma_det)
    # This einsum call calculates (x-mu)T.Sigma-1.(x-mu) in a vectorized
    # way across all the input variables.
    fac = np.einsum('...k,kl,...l->...', pos-mu, Sigma_inv, pos-mu)

    return np.exp(-fac / 2) / N

# The distribution on the variables X, Y packed into pos.
Z = multivariate_gaussian(pos, mu, Sigma)

# Create a surface plot and projected filled contour plot under it.
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(X, Y, Z, rstride=3, cstride=3, linewidth=1, antialiased=True,
                cmap=cm.viridis)

cset = ax.contourf(X, Y, Z, zdir='z', offset=-0.15, cmap=cm.viridis)

# Adjust the limits, ticks and view angle
ax.set_zlim(-0.15,0.2)
ax.set_zticks(np.linspace(0,0.2,5))
ax.view_init(27, -21)

plt.show()
```



This is a 3D projection of the gaussians involved with the lower surface showing the 2D projection of the 3D projection above. The innermost ellipse represents the highest peak, that is the maximum probability for a given (X,Y) value.

** numpy einsum examples **

```
a = np.arange(25).reshape(5,5)
b = np.arange(5)
c = np.arange(6).reshape(2,3)
print(a)
print(b)
print(c)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
[0 1 2 3 4]
[[0 1 2]
 [3 4 5]]
```

```
#this is the diagonal sum, i repeated means the diagonal
np.einsum('ij', a)
#this takes the output ii which is the diagonal and outputs to a
np.einsum('ii->i',a)
#this takes in the array A represented by their axes 'ij' and  B by its only axes'j'
#and multiples them element wise
np.einsum('ij,j',a, b)
```

```
array([ 30,  80, 130, 180, 230])
```

```
A = np.arange(3).reshape(3,1)
B = np.array([[ 0,  1,  2,  3],
              [ 4,  5,  6,  7],
```

(continues on next page)

(continued from previous page)

```
              [ 8,  9, 10, 11]])
C=np.multiply(A,B)
np.sum(C,axis=1)
```

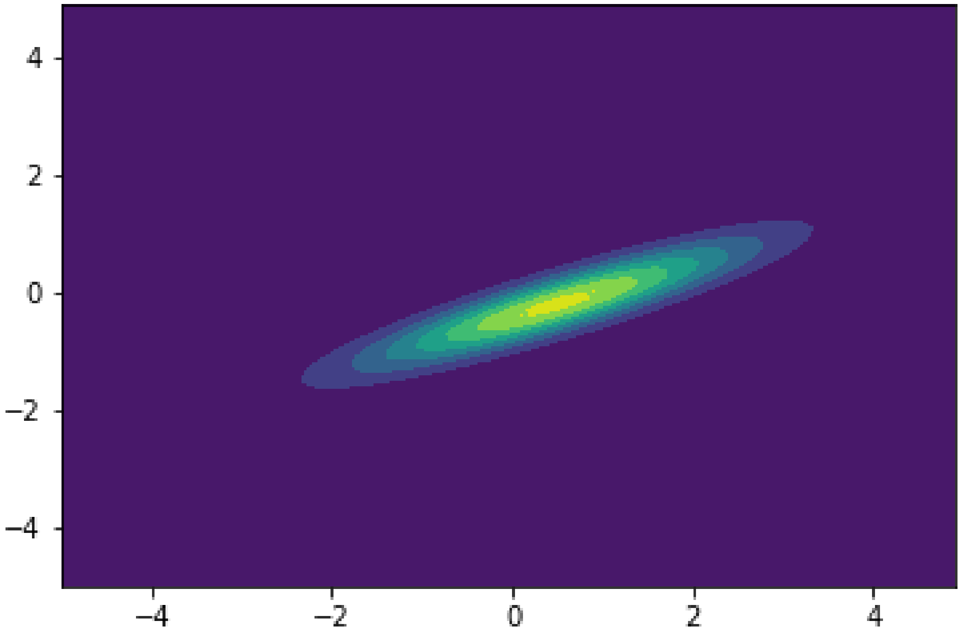
```
array([ 0, 22, 76])
```

```
D = np.array([0,1,2])
E = np.array([[ 0,  1,  2,  3],
              [ 4,  5,  6,  7],
              [ 8,  9, 10, 11]])

np.einsum('i,ij->i',D,E)
```

```
array([ 0, 22, 76])
```

```
from scipy.stats import multivariate_normal
x, y = np.mgrid[-5:5:.1, -5:5:.1]
pos = np.empty(x.shape + (2,))
pos[:, :, 0] = x; pos[:, :, 1] = y
rv = multivariate_normal([0.5, -0.2], [[2.0, 0.9], [0.9, 0.5]])
plt.contourf(x, y, rv.pdf(pos))
```

```
<matplotlib.contour.QuadContourSet at 0x139196438>
```



10.1.5 References:

1. Roger Labbe's repo on Kalman Filters. (Majority of the examples in the notes are from this)
2. Probabilistic Robotics by Sebastian Thrun, Wolfram Burgard and Dieter Fox, MIT Press.
3. Scipy Documentation

10.2 KF Basics - Part 2

10.2.1 Probabilistic Generative Laws

1st Law:

The belief representing the state x_t, is conditioned on all past states, measurements and controls. This can be shown mathematically by the conditional probability shown below:

$$p(x_t|x_{0:t-1},z_{1:t-1},u_{1:t})$$

1) z_t represents the **measurement**
2) u_t the **motion command**
3) x_t the **state** (can be the position, velocity, etc) of the robot or its environment at time t.

'If we know the state x_{t-1} and u_t, then knowing the states $x_{0:t-2}$, $z_{1:t-1}$ becomes immaterial through the property of **conditional independence**'. The state x_{t-1} introduces a conditional independence between x_t and $z_{1:t-1}$, $u_{1:t-1}$

Therefore the law now holds as:

$$p(x_t|x_{0:t-1},z_{1:t-1},u_{1:t})=p(x_t|x_{t-1},u_t)$$

2nd Law:

If x_t is complete, then:

$$p(z_t|x-_{0:t},z_{1:t-1},u_{1:t})=p(z_t|x_t)$$

x_t is **complete** means that the past states, controls or measurements carry no additional information to predict future.

$x_{0:t-1}$, $z_{1:t-1}$ and $u_{1:t}$ are **conditionally independent** of z_t given x_t of complete.

The filter works in two parts:

$p(x_t|x_{t-1},u_t)$ -> **State Transition Probability**

$p(z_t|x_t)$ -> **Measurement Probability**

10.2.2 Conditional dependence and independence example:

•Independent but conditionally dependent

Let's say you flip two fair coins

A - Your first coin flip is heads

B - Your second coin flip is heads

C - Your first two flips were the same

A and B here are independent. However, A and B are conditionally dependent given C, since if you know C then your first coin flip will inform the other one.

•Dependent but conditionally independent

A box contains two coins: a regular coin and one fake two-headed coin ((P(H)=1). I choose a coin at random and toss it twice. Define the following events.

A= First coin toss results in an H.

B= Second coin toss results in an H.

C= Coin 1 (regular) has been selected.

If we know A has occurred (i.e., the first coin toss has resulted in heads), we would guess that it is more likely that we have chosen Coin 2 than Coin 1. This in turn increases the conditional probability that B occurs. This suggests that A and B are not independent. On the other hand, given C (Coin 1 is selected), A and B are independent.

10.2.3 Bayes Rule:

Posterior =

$$\frac{Likelihood * Prior}{Marginal}$$

Here,

Posterior = Probability of an event occurring based on certain evidence.

Likelihood = How probable is the evidence given the event.

Prior = Probability of the just the event occurring without having any evidence.

Marginal = Probability of the evidence given all the instances of events possible.

Example:

1% of women have breast cancer (and therefore 99% do not). 80% of mammograms detect breast cancer when it is there (and therefore 20% miss it). 9.6% of mammograms detect breast cancer when its not there (and therefore 90.4% correctly return a negative result).

We can turn the process above into an equation, which is Bayes Theorem. Here is the equation:

$\mathrm{Pr}(\mathrm{A}|\mathrm{X}) = \frac{\mathrm{Pr}(\mathrm{X}|\mathrm{A})\,\mathrm{Pr}(\mathrm{A})}{\mathrm{Pr}(\mathrm{X}|\mathrm{A})\,\mathrm{Pr}(\mathrm{A}) + \mathrm{Pr}(\mathrm{X}|\mathrm{not\ A})\,\mathrm{Pr}(\mathrm{not\ A})}$

•Pr(A|X) = Chance of having cancer (A) given a positive test (X). This is what we want to know: How likely is it to have cancer with a positive result? In our case it was 7.8%.

•Pr(X|A) = Chance of a positive test (X) given that you had cancer (A). This is the chance of a true positive, 80% in our case.

•Pr(A) = Chance of having cancer (1%).

•Pr(not A) = Chance of not having cancer (99%).

•Pr(X|not A) = Chance of a positive test (X) given that you didn't have cancer (~A). This is a false positive, 9.6% in our case.

10.2.4 Bayes Filter Algorithm

The basic filter algorithm is:

for all x_t:

1. $\overline{bel}(x_t) = \int p(x_t|u_t, x_{t-1})bel(x_{t-1})dx$
2. $bel(x_t) = \eta p(z_t|x_t)\overline{bel}(x_t)$

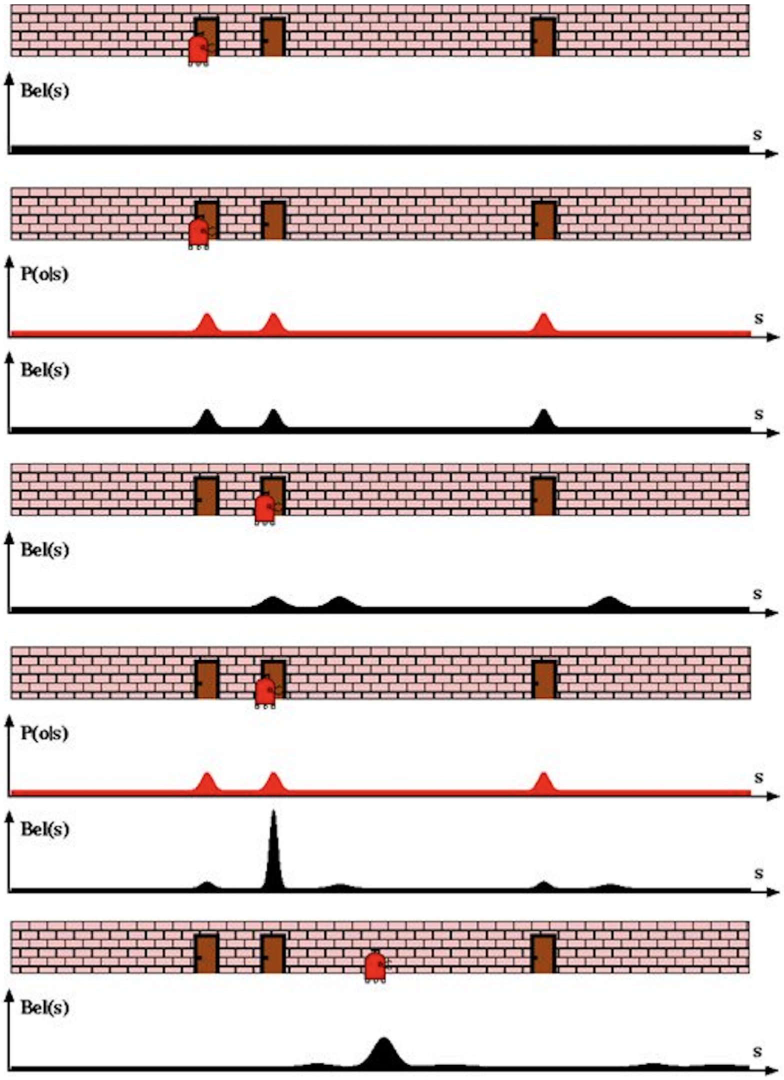
end.

→The first step in filter is to calculate the prior for the next step that uses the bayes theorem. This is the **Prediction** step. The belief, $\overline{bel}(x_t)$, is **before** incorporating measurement(z_t) at time t=t. This is the step where the motion occurs and information is lost because the means and covariances of the gaussians are added. The RHS of the equation incorporates the law of total probability for prior calculation.

→ This is the **Correction** or update step that calculates the belief of the robot **after** taking into account the measurement(z_t) at time t=t. This is where we incorporate the sensor information for the whereabouts of the robot. We gain information here as the gaussians get multiplied here. (Multiplication of gaussian values allows the resultant to lie in between these numbers and the resultant covariance is smaller.

10.2.5 Bayes filter localization example:

```
from IPython.display import Image
Image(filename="bayes_filter.png",width=400)
```



Given - A robot with a sensor to detect doorways along a hallway. Also, the robot knows how the hallway looks like but doesn't know where it is in the map.

1. Initially(first scenario), it doesn't know where it is with respect to the map and hence the belief assigns equal probability to each location in the map.
2. The first sensor reading is incorporated and it shows the presence of a door. Now the robot knows how the map looks like but cannot localize yet as map has 3 doors present. Therefore it assigns equal probability to each door present.
3. The robot now moves forward. This is the prediction step and the motion causes the robot to lose some of the information and hence the variance of the gaussians increase (diagram 4.). The final belief is **convolution** of posterior from previous step and the current state after motion. Also, the means shift on the right due to the motion.
4. Again, incorporating the measurement, the sensor senses a door and this time too the possibility of door is equal for the three door. This is where the filter's magic kicks in. For the final belief (diagram 5.), the posterior calculated after sensing is mixed or **convolution** of previous posterior and measurement. It improves the robot's belief at location near to the second door. The variance **decreases** and **peaks**.
5. Finally after series of iterations of motion and correction, the robot is able to localize itself with respect to the environment.(diagram 6.)

Do note that the robot knows the map but doesn't know where exactly it is on the map.

10.2.6 Bayes and Kalman filter structure

The basic structure and the concept remains the same as bayes filter for Kalman. The only key difference is the mathematical representation of Kalman filter. The Kalman filter is nothing but a bayesian filter that uses Gaussians.

For a bayes filter to be a Kalman filter, **each term of belief is now a gaussian**, unlike histograms. The basic formulation for the **bayes filter** algorithm is:

$$\begin{aligned} \bar{\mathbf{x}} &= \mathbf{x} * f_{\mathbf{x}}(\bullet) && \text{Prediction} \\ \mathbf{x} &= \mathcal{L} \cdot \bar{\mathbf{x}} && \text{Correction} \end{aligned}$$

$\bar{\mathbf{x}}$ is the *prior*

$\mathcal{L}$ is the *likelihood* of a measurement given the prior $\bar{\mathbf{x}}$

$f_{\mathbf{x}}(\bullet)$ is the *process model* or the gaussian term that helps predict the next state like velocity to track position or acceleration.

$*$ denotes *convolution*.

10.2.7 Kalman Gain

$$x = (\mathcal{L}\bar{x})$$

Where x is posterior and $\mathcal{L}$ and $\bar{x}$ are gaussians.

Therefore the mean of the posterior is given by:

$$\mu = \frac{\bar{\sigma}^2\, \mu_z + \sigma_z^2\, \bar{\mu}}{\bar{\sigma}^2 + \sigma_z^2}$$

$$\mu = \left(\frac{\bar{\sigma}^2}{\bar{\sigma}^2 + \sigma_z^2} \right) \mu_z + \left(\frac{\sigma_z^2}{\bar{\sigma}^2 + \sigma_z^2} \right) \bar{\mu}$$

In this form it is easy to see that we are scaling the measurement and the prior by weights:

$$\mu = W_1 \mu_z + W_2 \bar{\mu}$$

The weights sum to one because the denominator is a normalization term. We introduce a new term, $K = W_1$, giving us:

$$\begin{aligned}\mu &= K\mu_z + (1-K)\bar{\mu} \\ &= \bar{\mu} + K(\mu_z - \bar{\mu})\end{aligned}$$

where

$$K = \frac{\bar{\sigma}^2}{\bar{\sigma}^2 + \sigma_z^2}$$

The variance in terms of the Kalman gain:

$$\begin{aligned}\sigma^2 &= \frac{\bar{\sigma}^2\sigma_z^2}{\bar{\sigma}^2 + \sigma_z^2} \\ &= K\sigma_z^2 \\ &= (1-K)\bar{\sigma}^2\end{aligned}$$

K is the *Kalman gain*. It's the crux of the Kalman filter. It is a scaling term that chooses a value partway between μ_z and $\bar{\mu}$.

10.2.8 Kalman Filter - Univariate and Multivariate

Prediction

Univariate	Univariate (Kalman form)	Multivariate
$\bar{\mu} = \mu + \mu_{f_x}$	$\bar{x} = x + dx$	$\bar{\mathbf{x}} = \mathbf{Fx} + \mathbf{Bu}$
$\bar{\sigma}^2 = \sigma_x^2 + \sigma_{f_x}^2$	$\bar{P} = P + Q$	$\bar{\mathbf{P}} = \mathbf{FPF}^\mathsf{T} + \mathbf{Q}$

$\mathbf{x}$, $\mathbf{P}$ are the state mean and covariance. They correspond to x and σ^2.

$\mathbf{F}$ is the *state transition function*. When multiplied by $\mathbf{x}$ it computes the prior.

$\mathbf{Q}$ is the process covariance. It corresponds to $\sigma_{f_x}^2$.

$\mathbf{B}$ and $\mathbf{u}$ are model control inputs to the system.

Correction

Univariate	Univariate (Kalman form)	Multivariate
	$y = z - \bar{x}$	$\mathbf{y} = \mathbf{z} - \mathbf{H\bar{x}}$
	$K = \frac{\bar{P}}{\bar{P}+R}$	$\mathbf{K} = \mathbf{\bar{P}H}^\mathsf{T}(\mathbf{H\bar{P}H}^\mathsf{T} + \mathbf{R})^{-1}$
$\mu = \frac{\bar{\sigma}^2\,\mu_z + \sigma_z^2\,\bar{\mu}}{\bar{\sigma}^2 + \sigma_z^2}$	$x = \bar{x} + Ky$	$\mathbf{x} = \bar{\mathbf{x}} + \mathbf{Ky}$
$\sigma^2 = \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2 + \sigma_2^2}$	$P = (1-K)\bar{P}$	$\mathbf{P} = (\mathbf{I} - \mathbf{KH})\mathbf{\bar{P}}$

$\mathbf{H}$ is the measurement function.

$\mathbf{z}$, $\mathbf{R}$ are the measurement mean and noise covariance. They correspond to z and σ_z^2 in the univariate filter. $\mathbf{y}$ and $\mathbf{K}$ are the residual and Kalman gain.

The details will be different than the univariate filter because these are vectors and matrices, but the concepts are exactly the same:

- Use a Gaussian to represent our estimate of the state and error
- Use a Gaussian to represent the measurement and its error
- Use a Gaussian to represent the process model

- Use the process model to predict the next state (the prior)
- Form an estimate part way between the measurement and the prior

10.2.9 References:

1. Roger Labbe's repo on Kalman Filters. (Majority of text in the notes are from this)
2. Probabilistic Robotics by Sebastian Thrun, Wolfram Burgard and Dieter Fox, MIT Press.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search