086761 - Homework 1

Yuri Feldman, 309467801 yurif@cs.technion.ac.il
Alexander Shender, 328626114 aka.sova@gmail.com

November 11, 2017

1

1.1

$$f_x(x) = \frac{1}{\sqrt{|2\pi\Sigma_x|}} \cdot e^{-\frac{1}{2}(x-\mu_x)^T\Sigma_x^{-1}(x-\mu_x)} \tag{1.1}$$

1.2

μ_y, Σ_y can be calculated using expectation properties (and do not depend on y being Gaussian):

$$\mu_y = \mathbb{E}Ax + b = A\mathbb{E}x + b = \boxed{A\mu_x + b} \tag{1.2}$$

$$\Sigma_y = \mathbb{E}\left[(Ax + b - \mu_y)(Ax + b - \mu_y)^T\right] = \mathbb{E}\left[A(x - \mu_x)(x - \mu_x)^T A^T\right] = \tag{1.3}$$

$$A\mathbb{E}\left[(x - \mu_x)(x - \mu_x)^T\right] A^T = \boxed{A\Sigma_x A^T} \tag{1.4}$$

Next, we show that y is Gaussian.

Note: In retrospective, an easier approach than the below is to consider that any projection of Ax is ultimately a projection of x and so is, by definition, Guassian (and hence so is $Ax + b$). Another alternative approach is to show that the characteristic function of $Ax + b$ has the same form as that of a Gaussian density. The direct approach below gets complicated (but still valid) when density of $Ax + b$ is degenerate.

We proceed to show that y is Gaussian. In the case where A is invertible this can be directly obtained from random vector transformation formula (1-to-1 transformation). Since $y = Ax + b$ the Jacobian matrix is A and since the mapping is one-to-one (A is invertible) according to the transformation theorem:

$$f_Y(y) = \frac{1}{|A|} f_X(A^{-1}(y - b)) \tag{1.5}$$

ignoring the constant normalization factors (since they don't depend on x and only guarantee the distributions' summing to 1) we calculate the exponent in the right hand side:

$$-\frac{1}{2}\left(A^{-1}(y - b) - \mu_x\right)^T \Sigma_x^{-1}\left(A^{-1}(y - b) - \mu_x\right) = \tag{1.6}$$

$$-\frac{1}{2}(y - b - A\mu_x)^T A^{-T}\Sigma_x^{-1}A^{-1}(y - b - A\mu_x) = \tag{1.7}$$

$$-\frac{1}{2}(y - (A\mu_x + b))^T \left(A\Sigma_x A^T\right)^{-1}(y - (A\mu_x + b)), \tag{1.8}$$

which exactly gives a Gaussian distribution (in particular, with the mean and variance we calculated beforehand).

We now deal with the case where $A \in \mathbb{R}^{m\times n}$ is not invertible - either rectangular or square with zero determinant. Consider the SVD decomposition of A:

$$A = U\Sigma V^T \tag{1.9}$$

Here, U and V are square invertible matrices of respective sizes $m \times m$ and $n \times n$. Note that $V^T = V^*$ as A is real. Σ is a rectangular matrix the same size as A with the main diagonal containing the singular values (and zero elsewhere). We can now write:

$$y = U\Sigma V^T x + b \tag{1.10}$$

Here, $V^T x$ is Gaussian, since V is invertible. Without loss of generality denote

$$\Sigma = \begin{pmatrix} \Sigma_1 & 0 \\ 0 & 0 \end{pmatrix} \qquad V^T x = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}, \tag{1.11}$$

where Σ_1 is (square) diagonal, with positive entries (and in particular, is invertible), and has the same number of columns as the length of vector v_1. We have then that

$$\Sigma V^T x = \begin{pmatrix} \Sigma_1 v_1 \\ 0 \end{pmatrix}, \tag{1.12}$$

where $\Sigma_1 v_1$ is Gaussian since v_1 is and Σ_1 is invertible, and hence $\Sigma V^T x = U^T(y - b)$ is (possibly degenerate) Gaussian (and hence, so is y).

2

2.1

$$p(x \mid z) = \frac{p(z \mid x)p(x)}{p(z)} = \frac{p(z \mid x)p(x)}{\int_x p(z \mid x)p(x) \cdot dx} \propto p(z \mid x) \cdot p(x) \tag{2.1}$$

2.2

We are interested in the MAP estimate:

$$x^*_{MAP} \doteq \arg\max_x p(x \mid z) = \arg\max_x p(z \mid x) \cdot p(x) = \arg\max_x \log p(z \mid x) + \log p(x) = \tag{2.2}$$

$$\arg\min_x ||z - Hx||^2_R + ||x - x_0||^2_{\Sigma_0} \tag{2.3}$$

Develop the latter:

$$||z - Hx||^2_R + ||x - x_0||^2_{\Sigma_0} = \left|\left|R^{-1/2}(Hx - z)\right|\right|^2 + \left|\left|\Sigma_0^{-1/2}(x - x_0)\right|\right|^2 = \tag{2.4}$$

$$\left|\left|\begin{pmatrix} R^{-1/2}(Hx - z) \\ \Sigma_0^{-1/2}(x - x_0) \end{pmatrix}\right|\right|^2 = \left|\left|\begin{pmatrix} R^{-1/2}H \\ \Sigma_0^{-1/2} \end{pmatrix} x - \begin{pmatrix} R^{-1/2}z \\ \Sigma_0^{-1/2}x_0 \end{pmatrix}\right|\right|^2, \tag{2.5}$$

from where x^*_{MAP} can be obtained as the least-squares solution using the pseudoinverse $A^\dagger b = (A^T A)^{-1} A^T b$, or equivalently proceed directly, noting that the function is convex and has a unique extremum which is the minimum. Develop the above into:

$$= x^T \left(H^T R^{-1} H + \Sigma_0^{-1}\right) x - 2\left(z^T R^{-1} H + x_0^T \Sigma_0^{-1}\right) x + \left(z^T R^{-1} z + x_0^T \Sigma_0^{-1} x_0\right) \tag{2.6}$$

find the zero of the gradient:

$$\nabla(\cdot) = 2\left(H^T R^{-1} H + \Sigma_0^{-1}\right) x^* - 2\left(\Sigma_0^{-1} x_0 + H^T R^{-1} z\right) = 0 \tag{2.7}$$

$$\implies \boxed{x^*_{MAP} = \left(H^T R^{-1} H + \Sigma_0^{-1}\right)^{-1} \left(\Sigma_0^{-1} x_0 + H^T R^{-1} z\right)} \tag{2.8}$$

The associated covariance is

$$\boxed{\Sigma = \left(H^T R^{-1} H + \Sigma_0^{-1}\right)^{-1}}, \tag{2.9}$$

as can be seen from the quadratic term in Eq.(2.6).

3 Code

```
In [2]: import numpy as np
        from numpy.linalg import norm
        from math import sin, cos, pi, asin, atan2, degrees
        from tabulate import tabulate

        np.set_printoptions(suppress=True,precision=11)
```

3.1 Rotations

```
In [19]: def wedge(v):
            return np.matrix([[0, -v[2], v[1]],
            [v[2], 0,  -v[0]],
            [-v[1], v[0], 0]]);

        def get_rotation_rodriguez(v, theta=None):
            if theta is None:
            theta = norm(v);
            n = v/theta;
            else:
            n = v;

            cM = wedge(n);
            return np.identity(3) + sin(theta)*cM + (1-cos(theta))*cM*cM
```

3.1.1

```
In [14]: def rotx(phi):
             v = phi*np.array([1, 0, 0]);
             return get_rotation_rodriguez(v)

        def roty(theta):
             v = theta*np.array([0, 1, 0]);
             return get_rotation_rodriguez(v)
```

```
def rotz(psi):
    v = psi*np.array([0, 0, 1]);
    return get_rotation_rodriguez(v)

def angles2rot(phi, theta, psi):
    return rotz(psi)*roty(theta)*rotx(phi)
```

3.1.2

```
In [16]: print('R= \n' + str(angles2rot(phi=pi/4, theta=pi/5, psi=pi/7)))
```

```
R=
[[ 0.72889912554  0.06766479742  0.68126906578]
[ 0.35101931853  0.81741496597 -0.45674742629]
[-0.58778525229  0.57206140282  0.57206140282]]
```

3.1.3

Development here is based on [1].

We first write the rotation matrix as the product of individual axis rotation matrices:

$$\begin{pmatrix} \cos(\psi)\cos(\theta) & \cos(\psi)\sin(\phi)\sin(\theta)-\cos(\phi)\sin(\psi) & \sin(\phi)\sin(\psi)+\cos(\phi)\cos(\psi)\sin(\theta) \\ \cos(\theta)\sin(\psi) & \cos(\phi)\cos(\psi)+\sin(\phi)\sin(\psi)\sin(\theta) & \cos(\phi)\sin(\psi)\sin(\theta)-\cos(\psi)\sin(\phi) \\ -\sin(\theta) & \cos(\theta)\sin(\phi) & \cos(\phi)\cos(\theta) \end{pmatrix} \tag{3.1}$$

Given rotation matrix R, we will want to extract angles by demanding per-element equality. We treat separately the case where $\cos(\theta)=0$. For the case $\theta=\frac{\pi}{2}$ we obtain the matrix:

$$\begin{pmatrix} 0 & \cos(\psi)\sin(\phi)-\cos(\phi)\sin(\psi) & \sin(\phi)\sin(\psi)+\cos(\phi)\cos(\psi) \\ 0 & \cos(\phi)\cos(\psi)+\sin(\phi)\sin(\psi) & \cos(\phi)\sin(\psi)-\cos(\psi)\sin(\phi) \\ -1 & 0 & 0 \end{pmatrix} \tag{3.2}$$

$$= \begin{pmatrix} 0 & \sin(\phi-\psi) & \cos(\phi-\psi) \\ 0 & \cos(\phi-\psi) & -\sin(\phi-\psi) \\ -1 & 0 & 0 \end{pmatrix} \tag{3.3}$$

For the case $\theta=-\frac{\pi}{2}$:

$$\begin{pmatrix} 0 & -\cos(\psi)\sin(\phi)-\cos(\phi)\sin(\psi) & \sin(\phi)\sin(\psi)-\cos(\phi)\cos(\psi) \\ 0 & \cos(\phi)\cos(\psi)-\sin(\phi)\sin(\psi) & -\cos(\phi)\sin(\psi)-\cos(\psi)\sin(\phi) \\ 1 & 0 & 0 \end{pmatrix} \tag{3.4}$$

$$= \begin{pmatrix} 0 & -\sin(\phi+\psi) & -\cos(\phi+\psi) \\ 0 & \cos(\phi+\psi) & -\sin(\phi+\psi) \\ 1 & 0 & 0 \end{pmatrix} \tag{3.5}$$

From here we proceed extracting the angles (see code and [1]).

```
In [17]: def rot2angles(M):
             sin_theta = -M[2, 0];
             theta = asin(sin_theta);

             if sin_theta==1:
                psi = 0; # arbitrary
                phi = atan2(M[0,1],M[0,2]);
             elif    sin_theta==-1:
                psi = 0; # arbitrary
                phi = atan2(-M[0,1],-M[0,2]);
             else:
                # another valid solution: pi-theta.
                cos_theta = cos(theta);
                phi = atan2(M[2,1]/cos_theta, M[2, 2]/cos_theta);
                psi = atan2(M[1,0]/cos_theta, M[0, 0]/cos_theta);

             return phi, theta, psi
```

3.1.4

```
In [22]: phi, theta, psi = (degrees(x) for x in rot2angles(
np.mat(' 0.813797681 -0.440969611 0.378522306;  \
                           0.46984631   0.882564119 0.0180283112; \
                          -0.342020143  0.163175911 0.925416578')))
print('phi=' + str(phi), 'theta=' + str(theta), 'psi=' + str(psi))
```

```
phi=9.999999994217536 theta=19.999999980143038 psi=29.99999998990158
```

3.2 3D Rigid Transformation

$$l^C = R_G^C l^G + t_{C\to G}^C = R_G^C(l^G + t_{C\to G}^G) \tag{3.6}$$

```
In [23]: trans = np.mat([-451.2459, 257.0322, 400])
rot = np.mat('0.5363 -0.8440 0; 0.8440 0.5363 0; 0 0 1');

pt = np.mat([450, 400, 50])

Tgl = rot.dot((pt + trans).T)
print(Tgl)
```

```
[[-555.20335297]
 [ 351.31482926]
 [ 450.        ]]
```

3.3 Pose Composition

Commanded:

$$x^c_{k+1} = x_k + \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{3.7}$$

$$R^c_{k+1} = R^c_k \tag{3.8}$$

$${}^{k+1}_{k}T^{(commanded)} = \begin{pmatrix} I & \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{pmatrix} \tag{3.9}$$

Actual:

$$x^a_{k+1} = x_k + \begin{pmatrix} 1.01 \\ 0 \end{pmatrix} \tag{3.10}$$

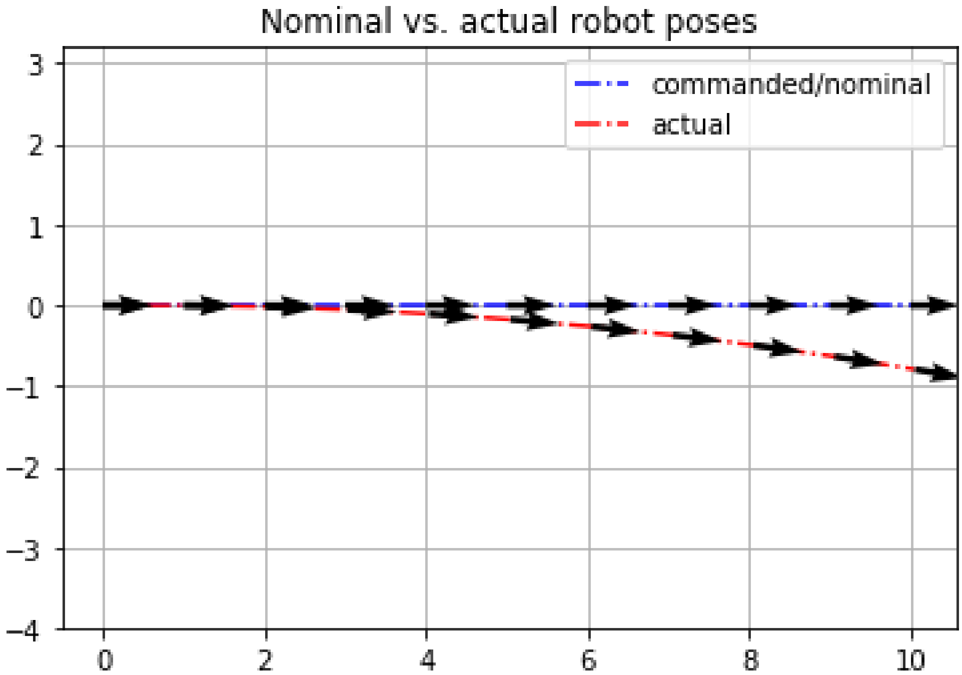
$$R^a_{k+1} = R(\theta = 1^\circ) \cdot R^c_k \tag{3.11}$$

$${}^{k+1}_{k}T^{(actual)} = \begin{pmatrix} R(\theta^\circ) & \begin{pmatrix} 1.01 \\ 0 \end{pmatrix} \end{pmatrix} \tag{3.12}$$

```
In [121]: %matplotlib inline

import matplotlib.pyplot as plt
from copy import deepcopy

def rotation2(theta):
return np.mat([[cos(theta),sin(theta)],
[-sin(theta),cos(theta)]]);

def rot2angle(R):
return atan2(R[0, 1], R[1, 1]);

Tc = np.concatenate((np.identity(2), np.mat([[1],[0]])), axis=1);

dtheta = pi/180;
Ta = np.concatenate((rotation2(dtheta), np.mat([[1.01],[0]])), axis=1);

# Initial pose
origin = np.mat([[0,0]]).T;
orientation = np.mat([[1, 0]]).T;
Ti = np.concatenate((np.identity(2), np.mat([[0],[0]])), axis=1);

# Lists hold nominal and actual pose history
T1 = [{"T" : Ti, "origin" : origin, "orientation" : orientation}];
T2 = deepcopy(T1);
for ii in range(10):
# commanded
Tk = Tc.dot(np.vstack((T1[-1]['T'], [0, 0, 1])));
nominal_origin = Tk[:,-1];
Rk = Tk[0:2,0:2];
nominal_orientation = Rk.dot(orientation);
T1.append({"T" : Tk, "origin" : nominal_origin, "orientation" : nominal_orientation})

# actual
```

```
Tk = Ta.dot(np.vstack((T2[-1]['T'], [0, 0, 1])));
actual_origin = Tk[:,-1];
Rk = Tk[0:2,0:2];
actual_orientation = Rk.dot(orientation);
T2.append({"T" : Tk, "origin" : actual_origin, "orientation" : actual_orientation})

def disp_track(locations, orientations, color='b'):
x = [o[0].item(0) for o in locations]
y = [o[1].item(0) for o in locations]
u = [o[0].item(0) for o in orientations]
v = [o[1].item(0) for o in orientations]
plt.plot(x, y, '-.', color=color)
plt.quiver(x, y, u, v, color='k',zorder=10)

plt.figure()

Pc = [P['origin'] for P in T1]
Uc = [P['orientation'] for P in T1]
disp_track(Pc, Uc)

Pa = [P['origin'] for P in T2]
Ua = [P['orientation'] for P in T2]
disp_track(Pa, Ua, 'r')

plt.grid(True)
plt.axis('equal');
plt.legend(['commanded/nominal', 'actual']);
plt.title('Nominal vs. actual robot poses');

print('Location difference: ' + str(norm(Pc[-1]-Pa[-1])) + ' m')
print('Orientation difference: ' + str(degrees(acos(Ua[-1].T.dot(Uc[-1])))) + u'\N{DEGREE SIG
```

```
Location difference: 0.793435624902 m
Orientation difference: 9.999999999999902°
```



Bibliography

[1] Gregory G Slabaugh. Computing euler angles from a rotation matrix. *Retrieved on August*, 6(2000):39–63, 1999.