Technion – Israel Institute of Technology

HW2

Vision Aided Navigation

086761

Alon Spinner	305184335	alonspinner@gmail.com
Sher Hazan	308026467	sherhazan@campus.technion.ac.il

November 24, 2021

Basic Probability and Bayesian Inference

Question 1 : Consider a random vector $x \in \mathbb{R}^n$ with a Gaussian distribution, written in covariance form $x \sim N(\mu, \Sigma)$. Show the corresponding information form $x \sim N^{-1}(\eta, \Lambda)$ is:

$$p(x) = N^{-1}(\eta, \Lambda) = \frac{\exp\left(-\frac{1}{2}\eta^T \Lambda^{-1} \eta\right)}{\sqrt{\det(2\pi \Lambda^{-1})}} \exp\left(-\frac{1}{2} x^T \Lambda x + \eta^T x\right)$$

$$\Lambda = \Sigma^{-1}, \eta = \Lambda\mu \rightarrow \Sigma = \Lambda^{-1}, \mu = \Lambda^{-1}\eta$$

$$p(x) = \frac{1}{\sqrt{\det(2\pi\Sigma)}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right) =$$

$$= \frac{1}{\sqrt{\det(2\pi\Lambda^{-1})}} \exp\left(-\frac{1}{2}(x-\Lambda^{-1}\eta)^T \Lambda (x-\Lambda^{-1}\eta)\right) =$$

$$= \frac{1}{\sqrt{\det(2\pi\Lambda^{-1})}} \exp\left(-\frac{1}{2}(x-\Lambda^{-1}\eta)^T \Lambda (x-\Lambda^{-1}\eta)\right) =$$

$$= \frac{1}{\sqrt{\det(2\pi\Lambda^{-1})}} \exp\left(-\frac{1}{2}(x^{\mathrm{T}}\Lambda x - x^T \Lambda\Lambda^{-1}\eta - (\Lambda^{-1}\eta)^{\mathrm{T}} \Lambda x + (\Lambda^{-1}\eta)^T \Lambda\Lambda^{-1}\eta)\right) =$$

$$= \frac{1}{\sqrt{\det(2\pi\Lambda^{-1})}} \exp\left(-\frac{1}{2}(x^{\mathrm{T}}\Lambda x - x^T\eta - \eta^{\mathrm{T}}\Lambda^{-\mathrm{T}}\Lambda x + \eta^{\mathrm{T}}\Lambda^{-\mathrm{T}}\eta)\right) =$$

$$= \frac{\exp\left(-\frac{1}{2}\eta^T \Lambda^{-T}\eta\right)}{\sqrt{\det(2\pi\Lambda^{-1})}} \exp\left(-\frac{1}{2}(x^{\mathrm{T}}\Lambda x - x^T\eta - \eta^{\mathrm{T}}\Lambda^{-\mathrm{T}}\Lambda x)\right) =$$

since Σ is symmetric, Σ^{-1} is symmetric too, $\Sigma^{-1} = \Lambda$ so Λ, Λ^{-1} are symmetric.

$$= \frac{\exp\left(-\frac{1}{2}\eta^T \Lambda^{-1}\eta\right)}{\sqrt{\det(2\pi\Lambda^{-1})}} \exp\left(-\frac{1}{2}(x^{\mathrm{T}}\Lambda x - x^T\eta - \eta^{\mathrm{T}}\Lambda^{-1}\Lambda x)\right) =$$

$$= \frac{\exp\left(-\frac{1}{2}\eta^T \Lambda^{-1}\eta\right)}{\sqrt{\det(2\pi\Lambda^{-1})}} \exp\left(-\frac{1}{2}(x^{\mathrm{T}}\Lambda x - x^T\eta - \eta^{\mathrm{T}} x)\right) =$$

$$= \frac{\exp\left(-\frac{1}{2}\eta^T \Lambda^{-1}\eta\right)}{\sqrt{\det(2\pi\Lambda^{-1})}} \exp\left(-\frac{1}{2}x^{\mathrm{T}}\Lambda x + \frac{1}{2}x^T\eta + \frac{1}{2}\eta^{\mathrm{T}} x\right) =$$

$x^T\eta$ is scalar, there for it is equal to its transpose $(x^T\eta)^T = \eta^T x$.

$$= \boxed{\frac{\exp\left(-\frac{1}{2}\eta^T \Lambda^{-1}\eta\right)}{\sqrt{\det(2\pi\Lambda^{-1})}} \exp\left(-\frac{1}{2}x^{\mathrm{T}}\Lambda x + \eta^T x\right)}$$

Question 2 : Consider a standard observation model involving a random variable $x \in \mathbb{R}^n$.

$$z = h(x) + v\,, v \sim N(0, \Sigma_v)$$

and assume the initial belief regarding the state x is a Gaussian with mean $\hat{x}_0$ and covariance Σ_0.

(a) Write an expression for the prior $p(x)$ and the measurement likelihood $p(z|x)$.

$$p(x) = \boxed{\frac{1}{\sqrt{\det(2\pi\Sigma_0)}} \exp\left(-\frac{1}{2}(x - \hat{x}_0)^T \Sigma_0^{-1}(x - \hat{x}_0)\right)}$$

$$p(z|x) = \boxed{\frac{1}{\sqrt{(det(2\pi\Sigma_v))}} exp\left(-\frac{1}{2}\big(z - h(x)\big)^T \Sigma_v{}^{-1}\big(z - h(x)\big)\right)}$$

(b) A measurement z_1 is acquired. Assuming the measurement was generated by the measurement model (1), write an expression for the posterior probability $p\ (x|z_1)$ in terms of $p(x)$ and the measurement likelihood $p(z|x)$.

$$p(x|z_1) \underset{BR}{=} \frac{p(x)p(z_1|x)}{p(z_1)} = \boxed{\eta p(x)p(z_1|x)}$$

(c) Derive expressions for the posteriori mean $\hat{x}_1$ and covariance Σ_1 such that $p\ (x|z_1) = \ N\ (\hat{x}_1, \Sigma_1)$.

$$\hat{x}_1 = argmax\big(p(x|z_1)\big) = argmin\big(-\log\big(\eta p(x)p(z_1|x)\big)\big)$$

$$Plugging\ in\ values, we\ notice\ that:$$

$$\eta p(x)p(z_1|x) = \eta \frac{\exp\left(-\frac{1}{2}(x - \hat{x}_0)^T \Sigma_0^{-1}(x - \hat{x}_0)\right) exp\left(-\frac{1}{2}\big(z - h(x)\big)^T \Sigma_v{}^{-1}\big(z - h(x)\big)\right)}{\sqrt{\det(2\pi\Sigma_0)}\sqrt{(det(2\pi\Sigma_v))}}$$

$$-\log\big(\eta p(x)p(z_1|x)\big) = -Const\left(\left(-\frac{1}{2}(x - \hat{x}_0)^T \Sigma_0^{-1}(x - \hat{x}_0)\right) + \left(-\frac{1}{2}\big(z - h(x)\big)^T \Sigma_v{}^{-1}\big(z - h(x)\big)\right)\right)$$

$$where\ Const > 0$$

$$We\ define:$$

$$J(x,z) = -\frac{1}{2}||x - \hat{x}_0||_{\Sigma_0}^2 - \frac{1}{2}||z - h(x)||_{\Sigma_v}^2$$

$$such\ that:$$

$$-\log\big(\eta p(x)p(z_1|x)\big) = \ \mathrm{Const} \cdot \left(||x - \hat{x}_0||_{\Sigma_0}^2 + ||z - h(x)||_{\Sigma_v}^2\right) = Const \cdot J(x,z)$$

$$\hat{x}_1 = \ argmin\big(-\log\big(\eta p(x)p(z_1|x)\big)\big) = argmin\big(Const \cdot J(x,z)\big) = argmin\big(J(x,z)\big)$$

$$since\ h(x) is\ non\ linear, we\ will\ use\ iterative\ optimization\ as\ showed\ in\ the\ lecture (NG).$$

$$Linearization\ about\ \bar{x}:$$

$$x = \bar{x} + \Delta x$$

$$\hat{x}_1 = argmin\big(J(\bar{x} + \Delta x)\big)$$

$$x - \hat{x}_0 = \bar{x} + \Delta x - \hat{x}_0 = \Delta x + (\bar{x} - \hat{x}_0)$$

$$z - h(x) = z - h(\bar{x} - \Delta x) \approx z - h(\bar{x}) - H\Delta x$$

$$J(\bar{x} + \Delta x) = \big||\Delta x + (\bar{x} - \hat{x}_0)|\big|^2_{\Sigma_0} + \big||H\Delta x + h(\bar{x}) - z|\big|^2_{\Sigma_v} =$$

$$= \left\|\Sigma_0^{-\frac{1}{2}}\big(\Delta x + (\bar{x} - \hat{x}_0)\big)\right\|^2 + \left\|\Sigma_v^{-\frac{1}{2}}(H\Delta x + h(\bar{x}) - z)\right\|^2$$

$$A = \begin{pmatrix} \Sigma_0^{-\frac{1}{2}} \\ \Sigma_v^{-\frac{1}{2}} H \end{pmatrix}, b = \begin{pmatrix} \Sigma_0^{-\frac{1}{2}}(\bar{x} - \hat{x}_0) \\ \Sigma_v^{-\frac{1}{2}}(h(\bar{x}) - z) \end{pmatrix}$$

$$\hat{x}_1 = argmin\big||A\Delta x + b|\big|^2$$

$$\boxed{\begin{array}{c} \hat{x}_1 = (A^T A)^{-1} A^T b \\ \Sigma_1 = A^T A \\ p(x|z_1) = N(\hat{x}_1, \Sigma_1) \end{array}}$$

(d) A second measurement, z_2, is obtained. Assuming $p\,(x|z_1) = N\,(\hat{x}_1, \Sigma_1)$ from last clause is given, derive expressions for $p\,(x|z_1, z_2) = N\,(\hat{x}_2, \Sigma_2)$.

$$p(x|z_1, z_2) \underset{BR}{=} \frac{p(x|z_1)p(z_2|x, z_1)}{p(z_2|z_1)} = \frac{p(x|z_1)p(z_2|x)}{p(z_2)} = \eta p(x|z_1)p(z_2|x)$$

$$J(x) = \big||x - \hat{x}_1|\big|^2_{\Sigma_1} + \big||z_2 - h(x)|\big|^2_{\Sigma_v}$$

$$\hat{x}_2 = argmax\big(p(x|z_1, z_2)\big) = argmin\Big(-\log\big(\eta p(x|z_1)p(z_2|x)\big)\Big) = \mathrm{argmin}(J(x))$$

since $h(x)$ is non linear, we will use iterative optimization as showed in the lecture.
Linearization about $\bar{x}$:

$$x = \bar{x} + \Delta x$$

$$\hat{x}_2 = argmin\big(J(\bar{x} + \Delta x)\big)$$

$$x - \hat{x}_1 = \bar{x} + \Delta x - \hat{x}_1 = \Delta x + (\bar{x} - \hat{x}_1)$$

$$z_2 - h(x) = z_2 - h(\bar{x} - \Delta x) \approx z_2 - h(\bar{x}) - H\Delta x$$

$$J(\bar{x} + \Delta x) = \big||\Delta x + (\bar{x} - \hat{x}_1)|\big|^2_{\Sigma_1} + \big||H\Delta x + h(\bar{x}) - z_2|\big|^2_{\Sigma_v}$$

$$\left\|\Sigma_1^{-\frac{1}{2}}\big(\Delta x + (\bar{x} - \hat{x}_1)\big)\right\|^2 + \left\|\Sigma_v^{-\frac{1}{2}}(H\Delta x + h(\bar{x}) z_2)\right\|^2$$

$$A = \begin{pmatrix} \Sigma_1^{-\frac{1}{2}} \\ \Sigma_v^{-\frac{1}{2}} H \end{pmatrix}, b = \begin{pmatrix} \Sigma_1^{-\frac{1}{2}}(\bar{x} - \hat{x}_1) \\ \Sigma_v^{-\frac{1}{2}}(h(\bar{x}) - z_2) \end{pmatrix}$$

$$\hat{x}_2 = argmin\big||A\Delta x + b|\big|^2$$

$$\boxed{\begin{array}{c} \hat{x}_2 = (A^T A)^{-1} A^T b \\ \Sigma_2 = A^T A \\ p(x|z_1, z_2) = N(\hat{x}_2, \Sigma_2) \end{array}}$$

Question 3 : Consider a multivariate random variable $x_k \in \mathbb{R}^n$. with the following state transition mode and a standard observation model as in exercise 2.

$$x_{k+1} = f(x_k, u_k) + w_k \;, w_k \sim N(0, \Sigma_w)$$

(a) Write an expression for the motion mode $p(x_k | x_{k-1}, u_{k-1})$.

$$p(x_k | x_{k-1}, u_{k-1}) =$$

$$= \frac{1}{\sqrt{\det(2\pi\Sigma_w)}} \exp\left(-\frac{1}{2}\left(x_k - f(x_{k-1}, u_{k-1})\right)^T \Sigma_w \left(x_k - f(x_{k-1}, u_{k-1})\right)\right) =$$

$$= \boxed{\frac{1}{\sqrt{\det(2\pi\Sigma_w)}} \exp\left(-\frac{1}{2}||x_k - f(x_{k-1}, u_{k-1})||_{\Sigma_w}^2\right)}$$

(b) Assume the robot executes action u_0 and then acquires a measurement z_1. Write an expression for the a posteriori pdf $p(x_1 | z_1, u_0)$ in terms of the prior, motion and observation models.

$$The\ prior - p(x)$$

$$The\ motion\ modle - p(x_k | x_{k-1}, u_{k-1})$$

$$The\ observation\ modle - p(z_1 | x_1)$$

$$p(x_1 | z_1, u_0) \underset{BR}{=} \frac{p(x_1 | u_0) p(z_1 | x_1, u_0)}{p(z_1 | u_0)} \underset{Indep.}{=} \frac{p(x_1 | u_0) p(z_1 | x_1)}{p(z_1)} = \eta p(x_1 | u_0) p(z_1 | x_1)$$

$$p(x_1 | u_0) \underset{MR}{=} \int p(x_1, x_0 | u_0) dx_0 \underset{CR}{=} \int p(x_1 | x_0, u_0) p(x_0 | u_0) dx_0 \underset{indep.}{=} \int p(x_1 | x_0, u_0) p(x_0) dx_0$$

$$\boxed{p(x_1 | z_1, u_0) = \eta p(z_1 | x_1) \int p(x_1 | x_0, u_0) p(x_0) dx_0}$$

(c) In the same setting, consider the a posteriori pdf over the joint state

$$x = x_{0:1} = \binom{x_0}{x_1}, with\ x_i \in \mathbb{R}^n$$

Show that calculating the maximum a posteriori (MAP) estimate for x is equivalent to solving a non-linear least squares problem.

$$In\ this\ case\ we\ use\ smoothing\ instead\ of\ marginlization.$$

$$As\ showed\ in\ the\ lecture: p(x_{0:1} | u_{x:k}, z_{1:k}) = \eta p(x_0) \prod_{i=1}^{k} p(x_i | x_{i-1}, u_{i-1}) p(z_i | x_i)$$

$$p(x_{0:1} | u_0, z_1) = \eta p(x_0) p(x_1 | x_0, u_0) p(z_1 | x_1)$$

$$We\ will\ find\ the\ MAP\ as\ follows$$

$$x^* = argmax\left(p(x_{0:1} | u_0, z_1)\right) = argmax\left(\eta p(x_0) p(x_1 | x_0, u_0) p(z_1 | x_1)\right) =$$

$$= argmin\left(-\log\left(p(x_0) p(x_1 | x_0, u_0) p(z_1 | x_1)\right)\right) = argmin(J(x))$$

$$where\ J(x) = ||x_0 - \hat{x}_0||_{\Sigma_0}^2 + ||z_1 - h(x_1)||_{\Sigma_v}^2 + ||x - f(x_{k-1}, u_{k-1})||_{\Sigma_w}^2$$

$$Like\ in\ Q2\ we\ will\ use\ linearization$$

$$x_k = x_{k-1} + \Delta x$$

$$x^* = argmin\big(J(x_{k-1} + \Delta x)\big)$$

$$x_k - \hat{x}_0 = x_{k-1} + \Delta x - \hat{x}_0 = \Delta x + (x_{k-1} - \hat{x}_0)$$

$$z_1 - h(x_k) = z_1 - h(x_{k-1} - \Delta x) \approx z_1 - h(x_{k-1}) - H\Delta x$$

$$J(x_{k-1} + \Delta x) = ||\Delta x + (x_{k-1} - \hat{x}_0)||^2_{\Sigma_0} + ||H\Delta x + h(x_{k-1}) - z_1||^2_{\Sigma_v} + \left|\left|\Delta x + \big(x_{k-1} - f(x_{k-1}, u_{k-1})\big)\right|\right|^2_{\Sigma_w}$$

$$\left|\left|\Sigma_0^{-\frac{1}{2}}(\Delta x + (x_{k-1} - \hat{x}_0))\right|\right|^2 + \left|\left|\Sigma_v^{-\frac{1}{2}}(H\Delta x + h(x_{k-1})z_1)\right|\right|^2 + \left|\left|\Sigma_w^{-\frac{1}{2}}\Big(\Delta x + \big(x_{k-1} - f(x_{k-1}, u_{k-1})\big)\Big)\right|\right|^2$$

$$A = \begin{pmatrix} \Sigma_0^{-\frac{1}{2}} \\ \Sigma_v^{-\frac{1}{2}} H \\ \Sigma_w^{-\frac{1}{2}} \end{pmatrix}, b = \begin{pmatrix} \Sigma_0^{-\frac{1}{2}}(x_{k-1} - \hat{x}_0) \\ \Sigma_v^{-\frac{1}{2}}(h(x_{k-1}) - z_1) \\ \Sigma_w^{-\frac{1}{2}}\big(x_{k-1} - f(x_{k-1}, u_{k-1})\big) \end{pmatrix}$$

$$\boxed{x^* = argmin\left|\left|A\Delta x + b\right|\right|^2}$$

(d) Assume the a posteriori pdf over the joint state$x_{0:1}$ is given in covariance and information forms as

$$p(x_{0:1}|u_0, z_1) = N(\hat{x}_{0:1}, \Sigma_{0:1}) = N^{-1}(\hat{\eta}_{0:1}, I_{0:1})$$
$$\Sigma_{0:1} = \begin{bmatrix} \Sigma_{00} & \Sigma_{01} \\ \Sigma_{01}^T & \Sigma_{11} \end{bmatrix}, I_{1:0} = \begin{bmatrix} \mathrm{I}_{00} & \mathrm{I}_{01} \\ \mathrm{I}_{01}^T & \mathrm{I}_{11} \end{bmatrix}$$

Indicate the dimensionality of the covariance matrix $\Sigma_{0:1}$ and of its components. We are interested in the marginal pdf over the state x_1, while marginalizing out the past state x_0. Write expressions for the marginal covariance and information matrices, Σ_{01} and I_{01} , over the state x_1 such that:

$$p(x_{0:1}|u_0, z_1) = N(\times, \Sigma_1') = N^{-1}(\times, I_1')$$

$$x_k \in \mathbb{R}^n \rightarrow x_{0:1} \in \mathbb{R}^{2n} \rightarrow \Sigma_{0:1} \in \mathbb{R}^{2n\times 2n}$$

$$p(x_1|u_0, z_1) = N(\times, \Sigma_{11}) = N^{-1}(\times, I_{11} - I_{01}^T I_{00} I_{10})$$

hw2

November 24, 2021

1 Q1

Projection of a 3D point. Consider a camera with focal length of f = 480 pixels in each axis and principal point (320, 270) pixels. Camera pose is described by the following rotation and translation with respect to the global frame (as in Homework #1):

$$R_G^C = \begin{bmatrix} 0.5363 & -0.8440 & 0 \\ 0.8440 & 0.5363 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$t_{C->G}^G = (-451.2459, 257.0322, 400)$$

The camera observes a 3D point with known coordinates in a global frame: $X^G = (350, -250, -35)^T$. The corresponding image observation representing this 3D point is (241.5, 169).

[1]:
```python
import numpy as np

#formating printing with numpy arrays
float_formatter = "{:.2f}".format
np.set_printoptions(formatter={'float_kind':float_formatter})
```

1.1 (a)

Write an expression for the camera projection matrix P . Write also numerical values of this matrix according to the given data in this exercise.

$$P = K[R|t]$$

The projection matrix has two parts: The first, Extrinsic Matrix, $[R|t]$, takes a point represented in the world coordinate system and transforms the representation that such it is in the camera coordinate system

$R = R_G^C$ and $t = t_{C->G}^C$

The second, Intrinsic Matrix, K, projects the point onto the image plane

$$K = \begin{bmatrix} f & s & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix}$$

Here was assume s = 0

[2]:
```
f = 480
px = 320
py = 270
s = 0
K = np.array([
    [f, s, px],
    [0, f, py],
    [0, 0, 1]])

R_G2C = np.array([[0.5363, -0.8440, 0],
                  [0.8440, 0.5363, 0],
                  [0, 0 ,1]])
t_G_C2G = np.array([-451.2459,257.0322,400])
t_C_C2G = R_G2C@t_G_C2G


P = K @ np.hstack([R_G2C,t_C_C2G.reshape(3,1)])

print('for our system:')
print(f"P = \n{P}")
```

```
for our system:
P =
[[257.42 -405.12 320.00 -92290.41]
 [405.12 257.42 270.00 -8642.48]
 [0.00 0.00 1.00 400.00]]
```

1.2 (b)

Write the underlying equations for projecting a 3D point using the projection matrix P. Calculate the image coordinates by projecting the 3D X^G .

given an homogeneous point $X^G = [x, y, z, 1]^T$

$$\begin{bmatrix} \hat{u} \\ \hat{v} \\ \hat{w} \end{bmatrix} = PX$$

The point will be projected to pixels (u, v) where:

$$u = \hat{u}/\hat{w}$$

$$v = \hat{v}/\hat{w}$$

[3]:
```
X = np.array([350, -250, -35, 1])
Uz = P@X
uz = Uz[0]/Uz[2]
vz = Uz[1]/Uz[2]

print('measuerd pixel values:')
print(f'(u,v) = ({uz},{vz})')
```

```
measuerd pixel values:
(u,v) = (240.78901527232875,162.58498094465753)
```

1.3 (c)

Calculate the re-projection error.

The reprojection error is calculated as such:

$$e = ||(u_z, v_z) - (u, v)||_2$$

where the z stands is the measured pixels

[4]:
```
Uz = np.array([uz,vz])
U = np.array([241.5,169])
e = np.linalg.norm(Uz-U)

print('reprojection error in pixels:')
print(e)
```

```
reprojection error in pixels:
6.454298471823932
```

2 Q2

(a) Calibrate your own camera (any camera will do).

(b) Capture images of a calibration pattern and follow the described steps 2 to calibrate your own camera.

(c) Submit snapshots of the process.

(d) Write an expression for the estimated camera calibration matrix K. Indicate the principal point, and focal length in each axis.

[5]:
```
import cv2
import glob
import matplotlib.pyplot as plt
```

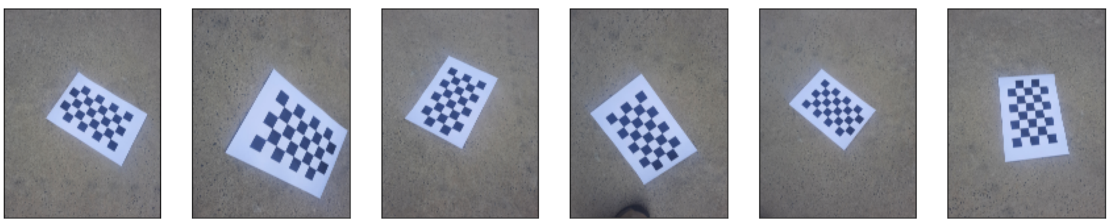
```
import numpy as np
%matplotlib inline

import numpy as np
#formating printing with numpy arrays
float_formatter = "{:.3f}".format
np.set_printoptions(formatter={'float_kind':float_formatter})
```

We decided on working with cv2 instead of Matlab

references: https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/

To start the process, we took 6 images with a Samsung Galaxy A51 phone:

```
[6]: files = glob.glob('./images/chessboard/*.jpg')
images = np.array([cv2.cvtColor(cv2.imread(f),cv2.COLOR_BGR2RGB) for f in␣
 ↪files])

fig, ax = plt.subplots(1, len(images),figsize=(15,15), sharex=True)
for idx in range(len(images)):
    ax[idx].imshow(images[idx])
    ax[idx].set_xticks([])
    ax[idx].set_yticks([])
```



We defined the checkboard parameters

```
[7]: CHECKERBOARD = (4,7)
dx = 30 #mm
```

The test is best with the checkboard defined static, and the camrea is moving. It also coorlates to the truth of events We define the world points for the checkerboard:

```
[8]: objp = np.zeros((1, CHECKERBOARD[0] * CHECKERBOARD[1], 3), np.float32)
gridp = np.mgrid[0:CHECKERBOARD[0], 0:CHECKERBOARD[1]].T.reshape(-1, 2)
objp[0,:,:2] = gridp/dx
```

We found the corners of checkerboards in the images. See the faint lines on the images provided below:

[9]:
```
#sub pixel optimization parameters
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# Arrays to store object points and image points from all the images.
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.
calibImages = []

for image in images:
    gray = cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)
    ret, corners = cv2.findChessboardCorners(gray, CHECKERBOARD, None)

    # If found, add object points, image points (after refining them)
    if ret == True:
        objpoints.append(objp)
        # refining pixel coordinates for given 2d points.
        corners2 = cv2.cornerSubPix(gray, corners, (11,11),(-1,-1), criteria)
        imgpoints.append(corners2)
        calibImages.append(image)
```

[10]:
```
m = len(calibImages)//2
n = 2
fig, ax = plt.subplots(m,n,figsize=(20,40),sharex=True, sharey=True)
for ii in range(m):
    for jj in range(n):
        idx = ii*(n-1)+jj
        img = cv2.drawChessboardCorners(calibImages[idx], CHECKERBOARD,␣
↪imgpoints[idx],True)
        ax[ii,jj].imshow(img)

for a in ax.ravel():
    a.set_axis_off()
fig.tight_layout(pad=0, w_pad=0, h_pad=0)
```

Performing camera calibration by passing the value of known 3D points (objpoints) and corresponding pixel coordinates of the detected corners (imgpoints)

[11]:
```
imageShape=images[0].shape
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints,␣
 ↪imageShape[:2][::-1], None, None)


print ("Resolution:")
print(imageShape[:2][::-1])
print("Camera matrix :")
print(mtx)
```

```
Resolution:
(3000, 4000)
Camera matrix :
[[3026.994 0.000 1540.323]
 [0.000 3042.652 2028.295]
 [0.000 0.000 1.000]]
```

That is to say, our K:

$$K = \begin{bmatrix} f_x & s & p_x \\ 0 & f_y & p_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 3026.994 & 0 & 1540.323 \\ 0 & 3042.652 & 2028.295 \\ 0 & 0 & 1 \end{bmatrix}$$

Sanity check:

Principle points are approx 1/2 of the resolution, which makes sense

Focal lengths are such that an object 1 meter long, standing 1 meter away, will take up most of the picture. sounds right

3 Q3

In this exercise we will perform basic image feature extraction and matching. You are free to choose what implementation and programming language to use - one recommended alternative for the latter is the vlfeat library 3 , which provides also interface to Matlab and useful tutorials.

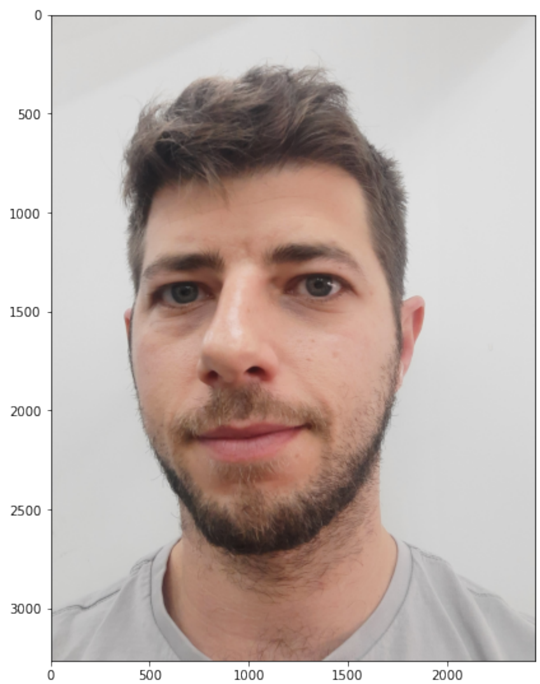
[12]:
```
import cv2
import glob
import matplotlib.pyplot as plt
import matplotlib
import numpy as np
from PIL import Image, ImageEnhance
```

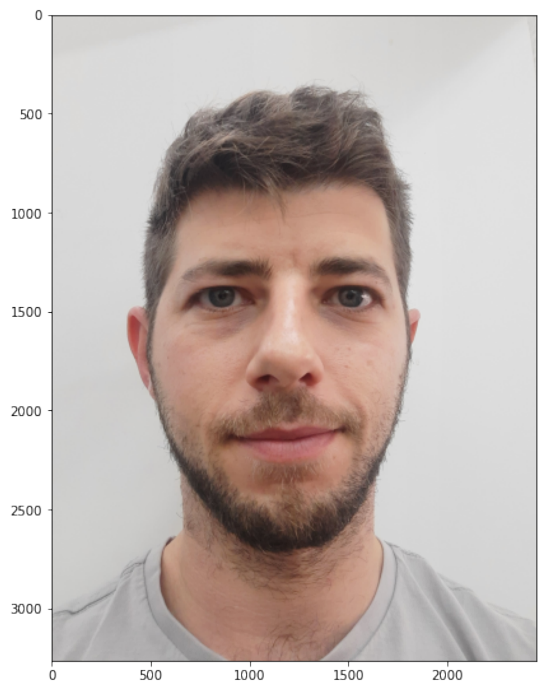
```
from scipy.spatial.distance import cdist
%matplotlib inline

import numpy as np
#formating printing with numpy arrays
float_formatter = "{:.3f}".format
np.set_printoptions(formatter={'float_kind':float_formatter})
```

3.1 (a)

Use a camera to capture 2 images of yourself from relatively close viewpoints (not more than 30 degrees change in yaw angle)

[13]:
```
files = glob.glob('./images/sift/*.jpg')
images = np.array([cv2.cvtColor(cv2.imread(f),cv2.COLOR_BGR2RGB) for f in␣
 ↪files])
szImages = [cv2.resize(image.copy(),None,fx=1,fy=1) for image in images]

fig, ax = plt.subplots(1, len(szImages),figsize=(15,15))
for idx,image in enumerate(szImages):
    ax[idx].imshow(image)
```



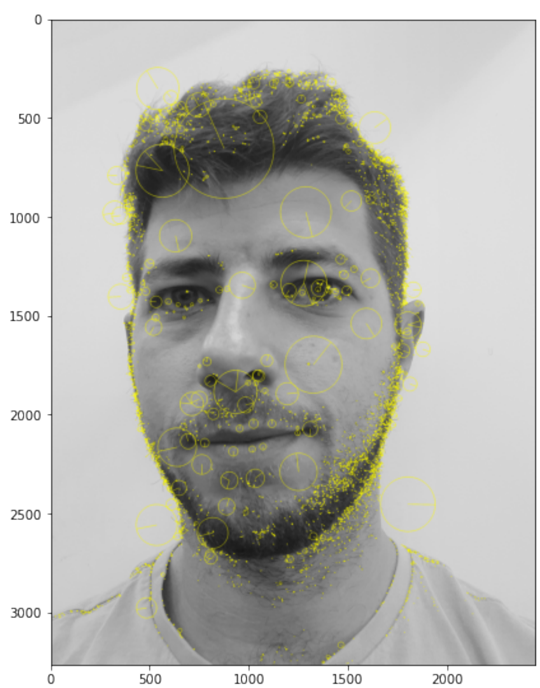


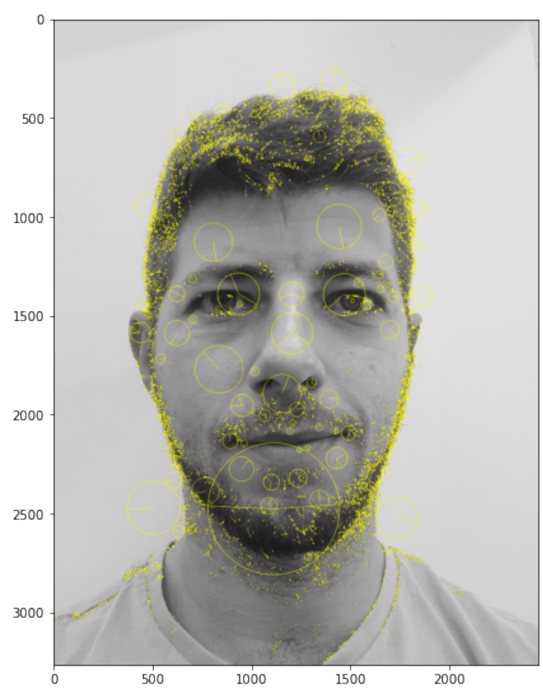
3.2 (b)

Extract SIFT features in each image. Please attach the two images with the extracted features. Indicate feature scale and orientation for a representative feature.

[14]:
```
grayImages = [cv2.cvtColor(image.copy(), cv2.COLOR_BGR2GRAY) for image in␣
↪szImages]
sift = cv2.SIFT_create()

kpsList = []
descsList = []
for gray in grayImages:
    kps, descs = (sift.detectAndCompute(gray,None))
    kpsList.append(kps)
    descsList.append(descs)
```

[15]:
```
fig, ax = plt.subplots(1, len(images),figsize=(15,15))
for idx,(image,gray,kps) in enumerate(zip(szImages,grayImages,kpsList)):
    img = cv2.drawKeypoints(gray, kps, outImage=np.array([]), color=(255, 255,␣
↪0),
        flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    img = Image.fromarray(img)
    filter = ImageEnhance.Color(img)
    img = filter.enhance(10)
    ax[idx].imshow(img)
```





3.3 (c)

Calculate putative matches by matching SIFT descriptors as explained in class. Please show, on a separate figure, the calculated matches between the two images. Among these matches, indicate representative inlier and outlier matches.

It is important to note that we do not have inliers and outliers in this problem as we do not have a mathematical model to fit! What we do have, is a way to determine a good match from a bad one. We decided that a good match is one that is reciprocal (both side's minimal distance to neighbor points to each other), and a bad match is one that isn't A worst match is where both side's maximal distance to neighbor points to each other.

[16]:

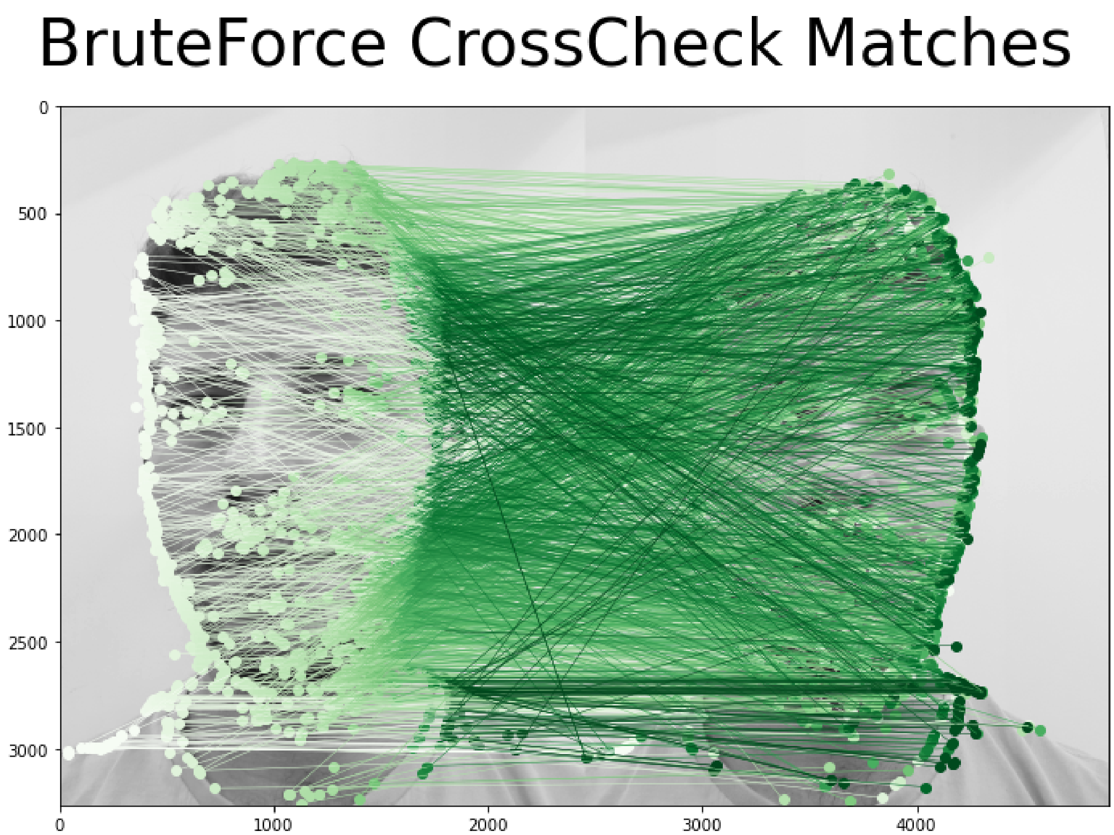
```
# Some helping functions
def findMatches(desc1,desc2):
    #desc1,desc2 - mxn numpy array matrix, m - amount of features, n -␣
↪descriptor length
    #returns matches, nx2 numpy array matrix containing indcies of good matches.
                    #first column correlates to desc1, and second to desc2

    #brute force, crossheck

    # Will be equivalent to, but returns numpy array instead of wierd objects:
    # bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
    # matches = bf.match(descs[0],descs[1])
    # matches = sorted(matches, key = lambda x:x.distance)

    matches = []

    D = cdist(np.float32(desc1), np.float32(desc2), metric='euclidean') #create␣
↪a matrix of mxn with distance values
    m,n = D.shape

    minIdx0 = np.argmin(D, axis=0) #returns array of length n, argmin of each␣
↪column
    minIdx1 = np.argmin(D, axis=1) #returns array of length m, argmin of each␣
↪row

    for ii in range(m):
        for jj in range(n):
            if ii == minIdx0[jj] and jj == minIdx1[ii]:
                matches.append([ii,jj])

    return np.array(matches)
def findBadMatches(desc1,desc2):
    #desc1,desc2 - mxn numpy array matrix, m - amount of features, n -␣
↪descriptor length
    #returns badMatches, nx2 numpy array matrix containing indcies of good␣
↪matches.
```

```
                #first column correlates to desc1, and second to desc2

    #brute force, crossheck - these matches are the absolute worst, agreed from␣
↪both sides

    badMatches = []

    D = cdist(np.float32(desc1), np.float32(desc2), metric='euclidean') #create␣
↪a matrix of mxn with distance values
    m,n = D.shape

    maxIdx0 = np.argmax(D, axis=0) #returns array of length n, argmax of each␣
↪column
    maxIdx1 = np.argmax(D, axis=1) #returns array of length m, argmax of each␣
↪row

    for ii in range(m):
        for jj in range(n):
            if ii == maxIdx0[jj] and jj == maxIdx1[ii]:
                badMatches.append([ii,jj])

    return np.array(badMatches)
def plotMatches(im1, im2, matches, locs1, locs2, colormap ,title):
    #im1,im2 - gray images
    #matches - nx2 numpy array matrix containing indcies of good matches.
                #first column correlates to desc1, and second to desc2
    #locs1,locs2 - 2xn numpy array containing location of features
    #title - string

    fig, ax = plt.subplots(figsize=(20,8))
    fig.suptitle(title,fontsize=40)

    # draw two images side by side
    imH = max(im1.shape[0], im2.shape[0])
    im = np.zeros((imH, im1.shape[1]+im2.shape[1]), dtype='uint8')
    im[0:im1.shape[0], 0:im1.shape[1]] = im1
    im[0:im2.shape[0], im1.shape[1]:] = im2
    ax.imshow(im, cmap='gray')

    colors = colormap(np.linspace(0, 1, matches.shape[0]))

    for i, c in zip(range(matches.shape[0]), colors):
        pt1 = locs1[matches[i,0], 0:2]
        pt2 = locs2[matches[i,1], 0:2].copy()
        pt2[0] += im1.shape[1]
        x = np.asarray([pt1[0], pt2[0]])
        y = np.asarray([pt1[1], pt2[1]])
```

```
        ax.plot(x,y,color=c,linewidth = 0.5)
        ax.scatter(x,y,color=c)
def kp2locs(kps):
    locs = []
    for kp in kps:
        locs.append(kp.pt)
    return np.array(locs)
```

[17]:
```
#reformat data for our needs
locs0 = kp2locs(kpsList[0])
locs1 = kp2locs(kpsList[1])
```

[36]:
```
#good matches
matches = findMatches(descsList[0],descsList[1])
plotMatches(grayImages[0], grayImages[1], matches, locs0, locs1, matplotlib.cm.
 ↪Greens,'BruteForce CrossCheck Matches')
```



Nice! note how feature of each side of the face match correctly

[19]:
```
#bad matches
badMatches = findBadMatches(descsList[0],descsList[1])
plotMatches(grayImages[0], grayImages[1], badMatches, locs0, locs1, matplotlib.
↪cm.Reds,'Absolutely Worst Matches')
```